NAG Library Function Document

nag_ztpqrt (f08bpc)

1 Purpose

nag_ztpqrt (f08bpc) computes the QR factorization of a complex $(m+n)$ by n triangular-pentagonal matrix.

2 Specification

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

```
void nag_ztpqrt (Nag_OrderType order, Integer m, Integer n, Integer l,
     Integer nb, Complex a[], Integer pda, Complex b[], Integer pdb,
     Complex t[], Integer pdt, NagError *fail)
```

3 Description

nag_ztpqrt (f08bpc) forms the QR factorization of a complex $(m+n)$ by n triangular-pentagonal matrix C,

$$C = \begin{pmatrix} A \\ B \end{pmatrix}$$

where A is an upper triangular n by n matrix and B is an m by n pentagonal matrix consisting of an $(m-l)$ by n rectangular matrix B_1 on top of an l by n upper trapezoidal matrix B_2:

$$B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}.$$

The upper trapezoidal matrix B_2 consists of the first l rows of an n by n upper triangular matrix, where $0 \leq l \leq \min(m,n)$. If $l=0$, B is m by n rectangular; if $l=n$ and $m=n$, B is upper triangular.

A recursive, explicitly blocked, QR factorization (see nag_zgeqrt (f08apc)) is performed on the matrix C. The upper triangular matrix R, details of the unitary matrix Q, and further details (the block reflector factors) of Q are returned.

Typically the matrix A or B_2 contains the matrix R from the QR factorization of a subproblem and nag_ztpqrt (f08bpc) performs the QR update operation from the inclusion of matrix B_1.

For example, consider the QR factorization of an l by n matrix $\hat{B}$ with $l<n$: $\hat{B} = \hat{Q}\hat{R}$, $\hat{R} = \begin{pmatrix} \hat{R}_1 & \hat{R}_2 \end{pmatrix}$, where $\hat{R}_1$ is l by l upper triangular and $\hat{R}_2$ is $(n-l)$ by n rectangular (this can be performed by nag_zgeqrt (f08apc)). Given an initial least-squares problem $\hat{B}\hat{X} = \hat{Y}$ where X and Y are l by $nrhs$ matrices, we have $\hat{R}\hat{X} = \hat{Q}^{\mathrm{H}}\hat{Y}$.

Now, adding an additional $m-l$ rows to the original system gives the augmented least squares problem

$$BX = Y$$

where B is an m by n matrix formed by adding $m-l$ rows on top of $\hat{R}$ and Y is an m by $nrhs$ matrix formed by adding $m-l$ rows on top of $\hat{Q}^{\mathrm{H}}\hat{Y}$.

nag_ztpqrt (f08bpc) can then be used to perform the QR factorization of the pentagonal matrix B; the n by n matrix A will be zero on input and contain R on output.

In the case where $\hat{B}$ is r by n, $r \geq n$, $\hat{R}$ is n by n upper triangular (forming A) on top of $r-n$ rows of zeros (forming first $r-n$ rows of B). Augmentation is then performed by adding rows to the bottom of B with $l=0$.

4 References

Elmroth E and Gustavson F (2000) Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance *IBM Journal of Research and Development. (Volume 44)* **4** 605–624

Golub G H and Van Loan C F (2012) *Matrix Computations* (4th 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: **m** – Integer *Input*

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

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

3: **n** – Integer *Input*

On entry: n, the number of columns of the matrix B and the order of the upper triangular matrix A.

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

4: **l** – Integer *Input*

On entry: l, the number of rows of the trapezoidal part of B (i.e., B_2).

Constraint: $0 \leq \mathbf{l} \leq \min(\mathbf{m}, \mathbf{n})$.

5: **nb** – Integer *Input*

On entry: the explicitly chosen block-size to be used in the algorithm for computing the QR factorization. See Section 9 for details.

Constraints:

$\mathbf{nb} \geq 1$;
if $\mathbf{n} > 0$, $\mathbf{nb} \leq \mathbf{n}$.

6: $\mathbf{a}[\mathit{dim}]$ – Complex *Input/Output*

Note: the dimension, *dim*, of the array **a** must be at least $\max(1, \mathbf{pda} \times \mathbf{n})$.

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

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

On entry: the n by n upper triangular matrix A.

On exit: the upper triangle is overwritten by the corresponding elements of the n by n upper triangular matrix R.

7: **pda** – Integer *Input*

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

Constraint: **pda** $\geq \max(1, \mathbf{n})$.

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

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

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

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

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

On entry: the m by n pentagonal matrix B composed of an $(m-l)$ by n rectangular matrix B_1 above an l by n upper trapezoidal matrix B_2.

On exit: details of the unitary matrix Q.

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

10: **t**[*dim*] – Complex *Output*

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

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

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

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

On exit: further details of the unitary matrix Q. The number of blocks is $b = \left\lceil \frac{k}{\mathbf{nb}} \right\rceil$, where $k = \min(m, n)$ and each block is of order **nb** except for the last block, which is of order $k - (b-1) \times \mathbf{nb}$. For each of the blocks, an upper triangular block reflector factor is computed: $\boldsymbol{T}_1, \boldsymbol{T}_2, \ldots, \boldsymbol{T}_b$. These are stored in the **nb** by n matrix T as $\boldsymbol{T} = [\boldsymbol{T}_1 | \boldsymbol{T}_2 | \ldots | \boldsymbol{T}_b]$.

11: **pdt** – Integer *Input*

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

Constraints:

if **order** = Nag_ColMajor, **pdt** $\geq$ **nb**;
if **order** = Nag_RowMajor, **pdt** $\geq$ **n**.

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

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

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.

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.

NE_INT_2

On entry, **nb** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **nb** ≥ 1 and
if **n** > 0, **nb** $\leq$ **n**.

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

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

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

On entry, **pdt** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: **pdt** $\geq$ **n**.

On entry, **pdt** = ⟨*value*⟩ and **nb** = ⟨*value*⟩.
Constraint: **pdt** $\geq$ **nb**.

NE_INT_3

On entry, **l** = ⟨*value*⟩, **m** = ⟨*value*⟩ and **n** = ⟨*value*⟩.
Constraint: $0 \leq \mathbf{l} \leq \min(\mathbf{m}, \mathbf{n})$.

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.

7 Accuracy

The computed factorization is the exact factorization of a nearby matrix $(A + E)$, where

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

and ϵ is the ***machine precision***.

8 Parallelism and Performance

nag_ztpqrt (f08bpc) is not threaded by NAG in any implementation.

nag_ztpqrt (f08bpc) 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 total number of floating-point operations is approximately $\frac{2}{3}n^2(3m-n)$ if $m \geq n$ or $\frac{2}{3}m^2(3n-m)$ if $m < n$.

The block size, **nb**, used by nag_ztpqrt (f08bpc) is supplied explicitly through the interface. For moderate and large sizes of matrix, the block size can have a marked effect on the efficiency of the algorithm with the optimal value being dependent on problem size and platform. A value of $\mathbf{nb} = 64 \ll \min(m, n)$ is likely to achieve good efficiency and it is unlikely that an optimal value would exceed 340.

To apply Q to an arbitrary complex rectangular matrix C, nag_ztpqrt (f08bpc) may be followed by a call to nag_ztpmqrt (f08bqc). For example,

```
nag_ztpmqrt(Nag_ColMajor,Nag_LeftSide,Nag_Trans,m,p,n,l,nb,b,pdb,
t,pdt,c,pdc,&c(n+1,1),ldc,&fail)
```

forms $C = Q^{\mathrm{H}}C$, where C is $(m+n)$ by p.

To form the unitary matrix Q explicitly set $p = m + n$, initialize C to the identity matrix and make a call to nag_ztpmqrt (f08bqc) as above.

10 Example

This example finds the basic solutions for the linear least squares problems

$$\text{minimize } \|Ax_i - b_i\|_2, \quad i = 1, 2$$

where b_1 and b_2 are the columns of the matrix B,

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

$$B = \begin{pmatrix} -2.09 + 1.93i & 3.26 - 2.70i \\ 3.34 - 3.53i & -6.22 + 1.16i \\ -4.94 - 2.04i & 7.94 - 3.13i \\ 0.17 + 4.23i & 1.04 - 4.26i \\ -5.19 + 3.63i & -2.31 - 2.12i \\ 0.98 + 2.53i & -1.39 - 4.05i \end{pmatrix}.$$

A QR factorization is performed on the first 4 rows of A using nag_zgeqrt (f08apc) after which the first 4 rows of B are updated by applying Q^T using nag_zgemqrt (f08aqc). The remaining row is added by performing a QR update using nag_ztpqrt (f08bpc); B is updated by applying the new Q^T using nag_ztpmqrt (f08bqc); the solution is finally obtained by triangular solve using R from the updated QR.

10.1 Program Text

```
/* nag_ztpqrt (f08bpc) Example Program.
 *
 * Copyright 2013, Numerical Algorithms Group.
 *
 * Mark 24, 2013.
 */

#include <nag.h>
#include <nag_stdlib.h>
#include <nagf07.h>
```

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

int main(void)
{
  /* Scalars */
  double  rnorm;
  Integer exit_status = 0;
  Integer pda, pdb, pdt;
  Integer i, j, m, n, nb, nrhs;
  /* Arrays */
  Complex  *a = 0, *b = 0, *c = 0, *t = 0;
  /* Nag Types */
  Nag_OrderType order;
  NagError      fail;

#ifdef NAG_COLUMN_MAJOR
#define A(I,J) a[(J-1)*pda + I-1]
#define B(I,J) b[(J-1)*pdb + I-1]
#define C(I,J) c[(J-1)*pdb + I-1]
#define T(I,J) t[(J-1)*pdt + 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]
#define C(I,J) c[(I-1)*pdb + J-1]
#define T(I,J) t[(I-1)*pdt + J-1]
  order = Nag_RowMajor;
#endif

  INIT_FAIL(fail);

  printf("nag_ztpqrt (f08bpc) Example Program Results\n\n");
  fflush(stdout);

  /* Skip heading in data file*/
  scanf("%*[^\n]");
  scanf("%ld%ld%ld%*[^\n]", &m, &n, &nrhs);
  nb = MIN(m, n);
  if (!(a = NAG_ALLOC(m*n, Complex))||
      !(b = NAG_ALLOC(m*nrhs, Complex))||
      !(c = NAG_ALLOC(m*nrhs, Complex))||
      !(t = NAG_ALLOC(nb*MIN(m, n), Complex)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }
#ifdef NAG_COLUMN_MAJOR
  pda = m;
  pdb = m;
  pdt = nb;
#else
  pda = n;
  pdb = nrhs;
  pdt = MIN(m, n);
#endif

  /* Read A and B from data file */
  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 <= m; ++i)
    for (j = 1; j <= nrhs; ++j)
      scanf(" ( %lf , %lf )", &B(i, j).re, &B(i, j).im);
  scanf("%*[^\n]");

  for (i = 1; i <= m; ++i)
```

```
    for (j = 1; j <= nrhs; ++j)
      C(i, j) = B(i, j);

  /* nag_zgeqrt (f08apc).
   * Compute the QR factorization of first n rows of A by recursive algorithm.
   */
  nag_zgeqrt(order, n, n, nb, a, pda, t, pdt, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zgeqrt (f08apc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* nag_zgemqrt (f08aqc).
   * Compute C = (C1) = (Q^H)*B, storing the result in C
   *             (C2)
   * by applying Q^H from left.
   */
  nag_zgemqrt(order, Nag_LeftSide, Nag_ConjTrans, n, nrhs, n, nb, a, pda, t,
              pdt, c, pdb, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_zgemqrt (f08aqc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  for (i = 1; i <= n; ++i)
    for (j = 1; j <= nrhs; ++j)
      B(i, j) = C(i, j);

  /* nag_ztrtrs (f07tsc).
   * Compute least-squares solutions for first n rows
   * by backsubstitution in R*X = C1.
   */
  nag_ztrtrs(order, Nag_Upper, Nag_NoTrans, Nag_NonUnitDiag, n, nrhs, a, pda,
             c, pdb, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ztrtrs (f07tsc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* nag_gen_complx_mat_print_comp (x04dbc).
   * Print least-squares solutions using first n rows.
   */
  nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n,
                                nrhs, c, pdb, Nag_BracketForm, "%7.4f",
                                "Solution(s) for n rows", 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;
  }

  /* nag_ztpqrt (f08bpc).
   * Now add the remaining rows and perform QR update.
   */
  nag_ztpqrt(order, m - n, n, 0, nb, a, pda, &A(n + 1, 1), pda, t, pdt, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ztpqrt (f08bpc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* nag_ztpmqrt (f08bqc).
   * Apply orthogonal transformations to C.
   */
  nag_ztpmqrt(order, Nag_LeftSide, Nag_ConjTrans, m - n, nrhs, n, 0, nb,
              &A(n + 1, 1), pda, t, pdt, b, pdb, &B(5, 1),pdb, &fail);
```

```
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ztpmqrt (f08bqc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* nag_ztrtrs (f07tsc).
   * Compute least-squares solutions for first n rows
   * by backsubstitution in R*X = C1.
   */
  nag_ztrtrs(order, Nag_Upper, Nag_NoTrans, Nag_NonUnitDiag, n, nrhs, a, pda,
             b, pdb, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_ztrtrs (f07tsc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* nag_gen_complx_mat_print_comp (x04dbc).
   * Print least-squares solutions.
   */
  printf("\n");
  nag_gen_complx_mat_print_comp(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n,
                                nrhs, b, pdb, Nag_BracketForm, "%7.4f",
                                "Least-squares solution(s) for all rows",
                                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 Square root(s) of the residual sum(s) of squares\n");
  for ( j=1; j<=nrhs; j++) {
    /* nag_zge_norm (f16uac).
     * Compute and print estimate of the square root of the residual
     * sum of squares.
     */
    nag_zge_norm(order, Nag_FrobeniusNorm, m - n, 1, &B(n + 1,j), pdb, &rnorm,
                 &fail);
    if (fail.code != NE_NOERROR) {
      printf("\nError from nag_zge_norm (f16uac).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
    printf("  %11.2e ", rnorm);
  }
  printf("\n");

 END:
  NAG_FREE(a);
  NAG_FREE(b);
  NAG_FREE(c);
  NAG_FREE(t);

return exit_status;
}
```

10.2 Program Data

```
nag_ztpqrt (f08bpc) Example Program Data

  6      4      2                                        : m, n and nrhs

 ( 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

(-2.09, 1.93) ( 3.26,-2.70)
( 3.34,-3.53) (-6.22, 1.16)
(-4.94,-2.04) ( 7.94,-3.13)
( 0.17, 4.23) ( 1.04,-4.26)
(-5.19, 3.63) (-2.31,-2.12)
( 0.98, 2.53) (-1.39,-4.05)                               : matrix B
```

10.3 Program Results

```
nag_ztpqrt (f08bpc) Example Program Results

 Solution(s) for n rows
                    1                  2
 1  (-0.5091,-1.2428)  ( 0.7569, 1.4384)
 2  (-2.3789, 2.8651)  ( 5.1727,-3.6193)
 3  ( 1.4634,-2.2064)  (-2.6613, 2.1339)
 4  ( 0.4701, 2.6964)  (-2.6933, 0.2724)

 Least-squares solution(s) for all rows
                    1                  2
 1  (-0.5044,-1.2179)  ( 0.7629, 1.4529)
 2  (-2.4281, 2.8574)  ( 5.1570,-3.6089)
 3  ( 1.4872,-2.1955)  (-2.6518, 2.1203)
 4  ( 0.4537, 2.6904)  (-2.7606, 0.3318)

 Square root(s) of the residual sum(s) of squares
     6.88e-02      1.87e-01
```