# NAG Library Function Document

# nag_dtgexc (f08yfc)

## 1 Purpose

nag_dtgexc (f08yfc) reorders the generalized Schur factorization of a matrix pair in real generalized Schur form.

## 2 Specification

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

```
void nag_dtgexc (Nag_OrderType order, Nag_Boolean wantq, Nag_Boolean wantz,
     Integer n, double a[], Integer pda, double b[], Integer pdb, double q[],
     Integer pdq, double z[], Integer pdz, Integer *ifst, Integer *ilst,
     NagError *fail)
```

## 3 Description

nag_dtgexc (f08yfc) reorders the generalized real $n$ by $n$ matrix pair $(S,T)$ in real generalized Schur form, so that the diagonal element or block of $(S,T)$ with row index $i_1$ is moved to row $i_2$, using an orthogonal equivalence transformation. That is, $S$ and $T$ are factorized as

$$S = \hat{Q}\hat{S}\hat{Z}^{\mathrm{T}}, \quad T = \hat{Q}\hat{T}\hat{Z}^{\mathrm{T}},$$

where $\left(\hat{S},\hat{T}\right)$ are also in real generalized Schur form.

The pair $(S,T)$ are in real generalized Schur form if $S$ is block upper triangular with 1 by 1 and 2 by 2 diagonal blocks and $T$ is upper triangular as returned, for example, by nag_dgges (f08xac), or nag_dhgeqz (f08xec) with **job** $=$ Nag_Schur.

If $S$ and $T$ are the result of a generalized Schur factorization of a matrix pair $(A,B)$

$$A = QSZ^{\mathrm{T}}, \quad B = QTZ^{\mathrm{T}}$$

then, optionally, the matrices $Q$ and $Z$ can be updated as $Q\hat{Q}$ and $Z\hat{Z}$.

## 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

## 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: **wantq** – Nag_Boolean *Input*

*On entry*: if **wantq** $=$ Nag_TRUE, update the left transformation matrix $Q$.

If **wantq** $=$ Nag_FALSE, do not update $Q$.

3: **wantz** – Nag_Boolean *Input*

*On entry*: if **wantz** $=$ Nag_TRUE, update the right transformation matrix $Z$.

If **wantz** $=$ Nag_FALSE, do not update $Z$.

4: **n** – Integer *Input*

*On entry*: $n$, the order of the matrices $S$ and $T$.

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

5: **a**$[dim]$ – double *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 matrix $S$ in the pair $(S,T)$.

*On exit*: the updated matrix $\hat{S}$.

6: **pda** – Integer *Input*

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

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

7: **b**$[dim]$ – double *Input/Output*

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

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 matrix $T$, in the pair $(S,T)$.

*On exit*: the updated matrix $\hat{T}$

8: **pdb** – Integer *Input*

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

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

9: **q**$[dim]$ – double *Input/Output*

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

$\max(1, \mathbf{pdq} \times \mathbf{n})$ when **wantq** $=$ Nag_TRUE;
1 otherwise.

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

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

*On entry*: if **wantq** $=$ Nag_TRUE, the orthogonal matrix $Q$.

*On exit*: if **wantq** $=$ Nag_TRUE, the updated matrix $Q\hat{Q}$.

If **wantq** = Nag_FALSE, **q** is not referenced.

10: **pdq** – Integer *Input*

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

*Constraints*:

if **wantq** = Nag_TRUE, **pdq** $\geq \max(1, \mathbf{n})$;
otherwise **pdq** $\geq 1$.

11: **z**[$dim$] – double *Input/Output*

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

$\max(1, \mathbf{pdz} \times \mathbf{n})$ when **wantz** = Nag_TRUE;
1 otherwise.

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

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

*On entry*: if **wantz** = Nag_TRUE, the orthogonal matrix $Z$.

*On exit*: if **wantz** = Nag_TRUE, the updated matrix $Z\hat{Z}$.

If **wantz** = Nag_FALSE, **z** is not referenced.

12: **pdz** – Integer *Input*

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

*Constraints*:

if **wantz** = Nag_TRUE, **pdz** $\geq \max(1, \mathbf{n})$;
otherwise **pdz** $\geq 1$.

13: **ifst** – Integer * *Input/Output*
14: **ilst** – Integer * *Input/Output*

*On entry*: the indices $i_1$ and $i_2$ that specify the reordering of the diagonal blocks of $(S, T)$. The block with row index **ifst** is moved to row **ilst**, by a sequence of swapping between adjacent blocks.

*On exit*: if **ifst** pointed on entry to the second row of a 2 by 2 block, it is changed to point to the first row; **ilst** always points to the first row of the block in its final position (which may differ from its input value by $+1$ or $-1$).

*Constraint*: $1 \leq \mathbf{ifst} \leq \mathbf{n}$ and $1 \leq \mathbf{ilst} \leq \mathbf{n}$.

15: **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_CONSTRAINT**

On entry, **wantq** $= \langle value \rangle$, **pdq** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: if **wantq** $=$ Nag_TRUE, **pdq** $\geq \max(1, \mathbf{n})$;
otherwise **pdq** $\geq 1$.

On entry, **wantz** $= \langle value \rangle$, **pdz** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: if **wantz** $=$ Nag_TRUE, **pdz** $\geq \max(1, \mathbf{n})$;
otherwise **pdz** $\geq 1$.

**NE_INT**

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

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

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

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

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

**NE_INT_2**

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

**NE_INT_3**

On entry, **ifst** $= \langle value \rangle$, **ilst** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: $1 \leq$ **ifst** $\leq$ **n** and $1 \leq$ **ilst** $\leq$ **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.

**NE_SCHUR**

The transformed matrix pair would be too far from generalized Schur form; the problem is ill-conditioned. $(S, T)$ may have been partially reordered, and **ilst** points to the first row of the current position of the block being moved.

## 7 Accuracy

The computed generalized Schur form is nearly the exact generalized Schur form for nearby matrices $(S + E)$ and $(T + F)$, where

$$\|E\|_2 = O\,\epsilon\|S\|_2 \quad \text{and} \quad \|F\|_2 = O\,\epsilon\|T\|_2,$$

and $\epsilon$ is the ***machine precision***. See Section 4.11 of Anderson *et al.* (1999) for further details of error bounds for the generalized nonsymmetric eigenproblem.

## 8 Parallelism and Performance

Not applicable.

## 9 Further Comments

The complex analogue of this function is nag_ztgexc (f08ytc).

## 10 Example

This example exchanges blocks 2 and 1 of the matrix pair $(S, T)$, where

$$S = \begin{pmatrix} 4.0 & 1.0 & 1.0 & 2.0 \\ 0 & 3.0 & 4.0 & 1.0 \\ 0 & 1.0 & 3.0 & 1.0 \\ 0 & 0 & 0 & 6.0 \end{pmatrix} \quad \text{and} \quad T = \begin{pmatrix} 2.0 & 1.0 & 1.0 & 3.0 \\ 0 & 1.0 & 2.0 & 1.0 \\ 0 & 0 & 1.0 & 1.0 \\ 0 & 0 & 0 & 2.0 \end{pmatrix}.$$

### 10.1 Program Text

```
/* nag_dtgexc (f08yfc) Example Program.
 *
 * Copyright 2011 Numerical Algorithms Group.
 *
 * Mark 23, 2011.
 */

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

int main(void)
{
  /* Scalars */
  double        alpha, beta, eps, norma, normb, norms, normt;
  Integer       i, ifst, ilst, j, n, pda, pdb, pdc, pdq, pds;
  Integer       pdt, pdz, exit_status = 0;
  /* Arrays */
  double        *a = 0, *b = 0, *c = 0, *q = 0, *s = 0, *t = 0, *z = 0;
  char          nag_enum_arg[40];

  /* Nag Types */
  NagError      fail;
  Nag_OrderType order;
  Nag_Boolean   wantq, wantz;

#ifdef NAG_COLUMN_MAJOR
#define S(I, J) s[(J-1)*pds + I - 1]
#define T(I, J) t[(J-1)*pdt + I - 1]
  order = Nag_ColMajor;
#else
#define S(I, J) s[(I-1)*pds + J - 1]
#define T(I, J) t[(I-1)*pdt + J - 1]
  order = Nag_RowMajor;
#endif

  INIT_FAIL(fail);

  printf("nag_dtgexc (f08yfc) Example Program Results\n\n");

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

```
  }
scanf(" %39s%*[^\n]", nag_enum_arg);
/* nag_enum_name_to_value (x04nac).
 * Converts NAG enum member name to value
 */
wantq = (Nag_Boolean) nag_enum_name_to_value(nag_enum_arg);
scanf(" %39s%*[^\n]", nag_enum_arg);
wantz = (Nag_Boolean) nag_enum_name_to_value(nag_enum_arg);

pds = n;
pdt = n;

pdq = (wantq?n:1);
pdz = (wantz?n:1);
pda = (wantq && wantz?n:1);
pdb = pda;
pdc = pda;

/* Allocate memory */
if (!(s = NAG_ALLOC(n*n, double)) ||
    !(t = NAG_ALLOC(n*n, double)) ||
    !(a = NAG_ALLOC(pda*pda, double)) ||
    !(b = NAG_ALLOC(pdb*pdb, double)) ||
    !(c = NAG_ALLOC(pdc*pdc, double)) ||
    !(q = NAG_ALLOC(pdq*pdq, double)) ||
    !(z = NAG_ALLOC(pdz*pdz, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

/* Read S and T from data file */
for (i = 1; i <= n; ++i)
  for (j = 1; j <= n; ++j) scanf("%lf", &S(i, j));
scanf("%*[^\n]");
for (i = 1; i <= n; ++i)
  for (j = 1; j <= n; ++j) scanf("%lf", &T(i, j));
scanf("%*[^\n]");

/* Compute norm of matrices S and T using nag_dge_norm (f16rac). */
nag_dge_norm(order, Nag_OneNorm, n, n, s, pds, &norms, &fail);
nag_dge_norm(order, Nag_OneNorm, n, n, t, pdt, &normt, &fail);
if (fail.code != NE_NOERROR)
  {
    printf("Error from nag_dge_norm (f16rac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
norms = sqrt(norms*norms + normt*normt);

/* Copy matrices S and T to matrices A and B using nag_dge_copy (f16qfc),
 * real valued general matrix copy.
 * The copies will be used as comparison against reconstructed matrices.
 */
if (wantq && wantz) {
  nag_dge_copy(order, Nag_NoTrans, n, n, s, pds, a, pda, &fail);
  nag_dge_copy(order, Nag_NoTrans, n, n, t, pdt, b, pdb, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_dge_copy (f16qfc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
}
/* Initialize Q an Z to identity matrices using nag_dge_load (f16qhc). */
alpha = 0.0;
beta = 1.0;
if (wantq) nag_dge_load(order, n, n, alpha, beta, q, pdq, &fail);
if (wantz) nag_dge_load(order, n, n, alpha, beta, z, pdz, &fail);
if (fail.code != NE_NOERROR)
```

```
    {
      printf("Error from nag_dge_load (f16qhc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Read the row indices of diagonal elements or blocks to be swapped. */
  scanf("%ld%ld%*[^\n]", &ifst, &ilst);

  /* nag_gen_real_mat_print (x04cac): Print Matrix S and Matrix T. */
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n,
                         s, pds, "Matrix S", 0, &fail);
  printf("\n");
  if (fail.code != NE_NOERROR) goto PRERR;
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n,
                         t, pdt, "Matrix T", 0, &fail);
  printf("\n");
  if (fail.code != NE_NOERROR) goto PRERR;

  /* Reorder S and T */
  nag_dtgexc(order, wantq, wantz, n, s, pds, t, pdt, q, pdq, z, pdz, &ifst,
             &ilst, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_dtgexc (f08yfc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* nag_gen_real_mat_print (x04cac): Print reordered S and T. */
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n,
                         s, pds, "Reordered matrix S", 0, &fail);
  printf("\n");
  if (fail.code != NE_NOERROR) goto PRERR;
  fflush(stdout);
  nag_gen_real_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag, n, n,
                         t, pdt, "Reordered matrix T", 0, &fail);
  printf("\n");
 PRERR:
  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 (wantq && wantz) {
    /* Reconstruct original S and T by applying orthogonal transformations:
     * e.g. S = Q^T S' Z, and subtract from original S and T using
     * nag_dgemm (f16yac), twice each.
     */
    alpha = 1.0;
    beta = 0.0;
    nag_dgemm(order, Nag_NoTrans, Nag_NoTrans, n, n, n, alpha, q, pdq, s, pds,
              beta, c, pdc, &fail);
    if (fail.code != NE_NOERROR) goto DGEMMERR;
    beta = -1.0;
    nag_dgemm(order, Nag_NoTrans, Nag_Trans, n, n, n, alpha, c, pdc, z, pdz,
              beta, a, pda, &fail);
    if (fail.code != NE_NOERROR) goto DGEMMERR;
    /* nag_dgemm (f16yac): Compute B - Qt*Tt*Zt^T */
    alpha = 1.0;
    beta = 0.0;
    nag_dgemm(order, Nag_NoTrans, Nag_NoTrans, n, n, n, alpha, q, pdq, t, pdt,
            beta, c, pdc, &fail);
    if (fail.code != NE_NOERROR) goto DGEMMERR;
    beta = -1.0;
```

```
    nag_dgemm(order, Nag_NoTrans, Nag_Trans, n, n, n, alpha, c, pdc, z, pdz,
              beta, b, pdb, &fail);
  DGEMMERR:
    if (fail.code != NE_NOERROR)
      {
        printf("Error from nag_dgemm (f16yac).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
    /* Compute norm of difference matrices using nag_dge_norm (f16rac). */
    nag_dge_norm(order, Nag_OneNorm, n, n, a, pda, &norma, &fail);
    nag_dge_norm(order, Nag_OneNorm, n, n, b, pdb, &normb, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from nag_dge_norm (f16rac).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
      }
    norma = sqrt(norma*norma + normb*normb);

    /* nag_machine_precision (x02ajc) */
    eps = nag_machine_precision;
    if (norma > pow(eps,0.8)*norms)
      {
        printf("The norm of the error in the reconstructed matrices is greater "
               "than expected.\nThe Schur factorization has failed.\n");
        exit_status = 1;
        goto END;
      }
  }

 END:
  NAG_FREE(a);
  NAG_FREE(b);
  NAG_FREE(c);
  NAG_FREE(q);
  NAG_FREE(s);
  NAG_FREE(t);
  NAG_FREE(z);

  return exit_status;
}
```

## 10.2 Program Data

```
nag_dtgexc (f08yfc) Example Program Data

  4                     : n

  Nag_TRUE              : wantp
  Nag_TRUE              : wantz


  4.0  1.0  1.0  2.0
  0.0  3.0  4.0  1.0
  0.0  1.0  3.0  1.0
  0.0  0.0  0.0  6.0    : matrix S

  2.0  1.0  1.0  3.0
  0.0  1.0  2.0  1.0
  0.0  0.0  1.0  1.0
  0.0  0.0  0.0  2.0    : matrix T

  2  1                  : ifst and ilst
```

### 10.3 Program Results

```
nag_dtgexc (f08yfc) Example Program Results

 Matrix S
             1          2          3          4
 1      4.0000     1.0000     1.0000     2.0000
 2      0.0000     3.0000     4.0000     1.0000
 3      0.0000     1.0000     3.0000     1.0000
 4      0.0000     0.0000     0.0000     6.0000

 Matrix T
             1          2          3          4
 1      2.0000     1.0000     1.0000     3.0000
 2      0.0000     1.0000     2.0000     1.0000
 3      0.0000     0.0000     1.0000     1.0000
 4      0.0000     0.0000     0.0000     2.0000

 Reordered matrix S
             1          2          3          4
 1      4.1926     1.2591     2.5578     0.4520
 2      0.8712    -0.8627    -2.7912    -1.1383
 3      0.0000     0.0000     4.2426     2.1213
 4      0.0000     0.0000     0.0000     6.0000

 Reordered matrix T
             1          2          3          4
 1      1.7439     0.0000     0.7533     0.0661
 2      0.0000    -0.5406    -1.8972    -1.7308
 3      0.0000     0.0000     2.1213     2.8284
 4      0.0000     0.0000     0.0000     2.0000
```