

# NAG Library Function Document

## nag\_zhemm (f16zcc)

### 1 Purpose

nag\_zhemm (f16zcc) performs matrix-matrix multiplication for a complex Hermitian matrix.

### 2 Specification

```
#include <nag.h>
#include <nagf16.h>

void nag_zhemm (Nag_OrderType order, Nag_SideType side, Nag_UploType uplo,
               Integer m, Integer n, Complex alpha, const Complex a[], Integer pda,
               const Complex b[], Integer pdb, Complex beta, Complex c[], Integer pdc,
               NagError *fail)
```

### 3 Description

nag\_zhemm (f16zcc) performs one of the matrix-matrix operations

$$C \leftarrow \alpha AB + \beta C \quad \text{or} \quad C \leftarrow \alpha BA + \beta C,$$

where  $A$  is a complex Hermitian matrix,  $B$  and  $C$  are  $m$  by  $n$  complex matrices, and  $\alpha$  and  $\beta$  are complex scalars.

### 4 References

Basic Linear Algebra Subprograms Technical (BLAST) Forum (2001) *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard* University of Tennessee, Knoxville, Tennessee <http://www.netlib.org/blas/blast-forum/blas-report.pdf>

### 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: **side** – Nag\_SideType *Input*  
*On entry:* specifies whether  $B$  is operated on from the left or the right.  
**side** = Nag\_LeftSide  
 $B$  is pre-multiplied from the left.  
**side** = Nag\_RightSide  
 $B$  is post-multiplied from the right.  
*Constraint:* **side** = Nag\_LeftSide or Nag\_RightSide.

- 3: **uplo** – Nag\_UploType *Input*  
*On entry:* specifies whether the upper or lower triangular part of  $A$  is stored.  
**uplo** = Nag\_Upper  
 The upper triangular part of  $A$  is stored.  
**uplo** = Nag\_Lower  
 The lower triangular part of  $A$  is stored.  
*Constraint:* **uplo** = Nag\_Upper or Nag\_Lower.
- 4: **m** – Integer *Input*  
*On entry:*  $m$ , the number of rows of the matrices  $B$  and  $C$ ; the order of  $A$  if **side** = Nag\_LeftSide.  
*Constraint:* **m**  $\geq$  0.
- 5: **n** – Integer *Input*  
*On entry:*  $n$ , the number of columns of the matrices  $B$  and  $C$ ; the order of  $A$  if **side** = Nag\_RightSide.  
*Constraint:* **n**  $\geq$  0.
- 6: **alpha** – Complex *Input*  
*On entry:* the scalar  $\alpha$ .
- 7: **a**[*dim*] – const Complex *Input*  
**Note:** the dimension, *dim*, of the array **a** must be at least  
 $\max(1, \mathbf{pda} \times \mathbf{m})$  when **side** = Nag\_LeftSide;  
 $\max(1, \mathbf{pda} \times \mathbf{n})$  when **side** = Nag\_RightSide.  
*On entry:* the Hermitian matrix  $A$ ;  $A$  is  $m$  by  $m$  if **side** = Nag\_LeftSide, or  $n$  by  $n$  if **side** = Nag\_RightSide.  
 If **order** = 'Nag\_ColMajor',  $A_{ij}$  is stored in **a**[( $j - 1$ )  $\times$  **pda** +  $i - 1$ ].  
 If **order** = 'Nag\_RowMajor',  $A_{ij}$  is stored in **a**[( $i - 1$ )  $\times$  **pda** +  $j - 1$ ].  
 If **uplo** = 'Nag\_Upper', the upper triangular part of  $A$  must be stored and the elements of the array below the diagonal are not referenced.  
 If **uplo** = 'Nag\_Lower', the lower triangular part of  $A$  must be stored and the elements of the array above the diagonal are not referenced.
- 8: **pda** – Integer *Input*  
*On entry:* the stride separating row or column elements (depending on the value of **order**) of the matrix  $A$  in the array **a**.  
*Constraints:*  
 if **side** = Nag\_LeftSide, **pda**  $\geq$   $\max(1, \mathbf{m})$ ;  
 if **side** = Nag\_RightSide, **pda**  $\geq$   $\max(1, \mathbf{n})$ .
- 9: **b**[*dim*] – const Complex *Input*  
**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.  
 If **order** = 'Nag\_ColMajor',  $B_{ij}$  is stored in **b**[( $j - 1$ )  $\times$  **pdb** +  $i - 1$ ].  
 If **order** = 'Nag\_RowMajor',  $B_{ij}$  is stored in **b**[( $i - 1$ )  $\times$  **pdb** +  $j - 1$ ].

On entry: the  $m$  by  $n$  matrix  $B$ .

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

11: **beta** – Complex *Input*

On entry: the scalar  $\beta$ .

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

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

$\max(1, \mathbf{pdc} \times \mathbf{n})$  when **order** = Nag\_ColMajor;  
 $\max(1, \mathbf{m} \times \mathbf{pdc})$  when **order** = Nag\_RowMajor.

If **order** = 'Nag\_ColMajor',  $C_{ij}$  is stored in **c**[( $j - 1$ )  $\times$  **pdc** +  $i - 1$ ].

If **order** = 'Nag\_RowMajor',  $C_{ij}$  is stored in **c**[( $i - 1$ )  $\times$  **pdc** +  $j - 1$ ].

On entry: the  $m$  by  $n$  matrix  $C$ .

If **beta** = 0, **c** need not be set.

On exit: the updated matrix  $C$ .

13: **pdc** – Integer *Input*

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

Constraints:

if **order** = Nag\_ColMajor, **pdc**  $\geq$   $\max(1, \mathbf{m})$ ;  
if **order** = Nag\_RowMajor, **pdc**  $\geq$   $\max(1, \mathbf{n})$ .

14: **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  $\langle value \rangle$  had an illegal value.

### NE\_ENUM\_INT\_2

On entry, **side** =  $\langle value \rangle$ , **m** =  $\langle value \rangle$ , **pda** =  $\langle value \rangle$ .  
Constraint: if **side** = Nag\_LeftSide, **pda**  $\geq$   $\max(1, \mathbf{m})$ .

On entry, **side** =  $\langle value \rangle$ , **n** =  $\langle value \rangle$ , **pda** =  $\langle value \rangle$ .  
Constraint: if **side** = Nag\_RightSide, **pda**  $\geq$   $\max(1, \mathbf{n})$ .

**NE\_INT**

On entry, **m** =  $\langle value \rangle$ .

Constraint: **m**  $\geq 0$ .

On entry, **n** =  $\langle value \rangle$ .

Constraint: **n**  $\geq 0$ .

**NE\_INT\_2**

On entry, **pdb** =  $\langle value \rangle$ , **m** =  $\langle value \rangle$ .

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

On entry, **pdb** =  $\langle value \rangle$  and **n** =  $\langle value \rangle$ .

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

On entry, **pdc** =  $\langle value \rangle$ , **m** =  $\langle value \rangle$ .

Constraint: **pdc**  $\geq \max(1, \mathbf{m})$ .

On entry, **pdc** =  $\langle value \rangle$  and **n** =  $\langle value \rangle$ .

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

**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.

**7 Accuracy**

The BLAS standard requires accurate implementations which avoid unnecessary over/underflow (see Section 2.7 of Basic Linear Algebra Subprograms Technical (BLAST) Forum (2001)).

**8 Parallelism and Performance**

Not applicable.

**9 Further Comments**

None.

**10 Example**

This example computes the matrix-matrix product

$$C = \alpha AB + \beta C$$

where

$$A = \begin{pmatrix} 1.0 + 0.0i & 1.0 + 2.0i & -2.0 + 3.0i \\ 1.0 - 2.0i & 2.0 + 0.0i & 1.0 + 2.0i \\ -2.0 - 3.0i & 1.0 - 2.0i & 3.0 + 0.0i \end{pmatrix},$$

$$B = \begin{pmatrix} 1.0 - 1.0i & 1.0 + 2.0i \\ -2.0 + 1.0i & 2.0 - 2.0i \\ 3.0 - 1.0i & -3.0 + 1.0i \end{pmatrix},$$

$$C = \begin{pmatrix} -3.5 - 0.5i & 1.5 + 2.0i \\ -4.5 + 1.5i & -2.0 + 3.5i \\ -5.5 + 3.5i & 3.0 - 1.5i \end{pmatrix},$$

$$\alpha = 1.0 + 0.0i \quad \text{and} \quad \beta = 2.0 + 0.0i.$$

## 10.1 Program Text

```

/* nag_zhemm (f16zcc) Example Program.
 *
 * Copyright 2005 Numerical Algorithms Group.
 *
 * Mark 8, 2005.
 */

#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagf16.h>
#include <nagx04.h>

int main(void)
{
    /* Scalars */
    Complex    alpha, beta;
    Integer     exit_status, i, j, m, n, pda, pdb, pdc;

    /* Arrays */
    Complex    *a = 0, *b = 0, *c = 0;
    char       nag_enum_arg[40];

    /* Nag Types */
    NagError    fail;
    Nag_OrderType order;
    Nag_SideType side;
    Nag_UploType uplo;

#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)*pdc + 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)*pdc + J - 1]
    order = Nag_RowMajor;
#endif

    exit_status = 0;
    INIT_FAIL(fail);

    printf("nag_zhemm (f16zcc) Example Program Results\n\n");

    /* Skip heading in data file */
    scanf("%*[\n] ");

    /* Read the problem dimensions */
    scanf("%ld%ld%*[\n] ",
          &m, &n);

    /* Read the side parameter */
    scanf("%39s%*[\n] ", nag_enum_arg);
    /* nag_enum_name_to_value (x04nac).
     * Converts NAG enum member name to value
     */
    side = (Nag_SideType) nag_enum_name_to_value(nag_enum_arg);
    /* Read uplo */
    scanf("%39s%*[\n] ", nag_enum_arg);
    /* nag_enum_name_to_value (x04nac).
     * Converts NAG enum member name to value
     */
    uplo = (Nag_UploType) nag_enum_name_to_value(nag_enum_arg);
    /* Read scalar parameters */
    scanf(" ( %lf , %lf ) ( %lf , %lf )%*[\n] ",

```

```

        &alpha.re, &alpha.im, &beta.re, &beta.im);

    if (side == Nag_LeftSide)
        pda = m;
    else
        pda = n;
#ifdef NAG_COLUMN_MAJOR
    pdb = m;
    pdc = m;
#else
    pdb = n;
    pdc = n;
#endif

    if (m > 0 && n > 0)
    {
        /* Allocate memory */
        if (side == Nag_LeftSide)
        {
            if (!(a = NAG_ALLOC(m*m, Complex)))
            {
                printf("Allocation failure\n");
                exit_status = -1;
                goto END;
            }
        }
        else
        {
            if (!(a = NAG_ALLOC(n*n, Complex)))
            {
                printf("Allocation failure\n");
                exit_status = -1;
                goto END;
            }
        }
        if (!(b = NAG_ALLOC(m*n, Complex)) ||
            !(c = NAG_ALLOC(m*n, Complex)))
        {
            printf("Allocation failure\n");
            exit_status = -1;
            goto END;
        }
    }
    else
    {
        printf("Invalid m or n\n");
        exit_status = 1;
        return exit_status;
    }

    /* Input matrix A */
    if (uplo == Nag_Upper)
    {
        for (i = 1; i <= pda; ++i)
        {
            for (j = i; j <= pda; ++j)
                scanf(" ( %lf , %lf )", &A(i, j).re, &A(i, j).im);
            scanf("%*[\n] ");
        }
    }
    else
    {
        for (i = 1; i <= pda; ++i)
        {
            for (j = 1; j <= i; ++j)
                scanf(" ( %lf , %lf )", &A(i, j).re, &A(i, j).im);
            scanf("%*[\n] ");
        }
    }
    /* Input matrix B */
    for (i = 1; i <= m; ++i)

```

```

    {
        for (j = 1; j <= n; ++j)
            scanf(" ( %lf , %lf )", &B(i, j).re, &B(i, j).im);
        scanf("%*[\n] ");
    }
/* Input matrix C */
for (i = 1; i <= m; ++i)
    {
        for (j = 1; j <= n; ++j)
            scanf(" ( %lf , %lf )", &C(i, j).re, &C(i, j).im);
        scanf("%*[\n] ");
    }

/* nag_zhemm (f16zcc).
 * Hermitian matrix-matrix multiply.
 *
 */
nag_zhemm(order, side, uplo, m, n, alpha, a, pda,
          b, pdb, beta, c, pdc, &fail);
if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_zhemm.\n%s\n", fail.message);
        exit_status = 1;
        goto END;
    }

/* Print result */
/* nag_gen_complx_mat_print (x04dac).
 * Print Complex general matrix (easy-to-use)
 */
fflush(stdout);
nag_gen_complx_mat_print(order, Nag_GeneralMatrix, Nag_NonUnitDiag,
                        m, n, c, pdc, "Matrix Matrix Product",
                        0, &fail);
if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_gen_complx_mat_print (x04dac).\n%s\n",
              fail.message);
        exit_status = 1;
        goto END;
    }

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

return exit_status;
}

```

## 10.2 Program Data

```

nag_zhemm (f16zcc) Example Program Data
3 2 :Values of m, n
Nag_LeftSide : side
Nag_Lower : uplo
( 1.0, 0.0) ( 2.0, 0.0) : alpha, beta
( 1.0, 0.0)
( 1.0,-2.0) ( 2.0, 0.0)
(-2.0,-3.0) ( 1.0,-2.0) ( 3.0, 0.0) : the end of matrix A
( 1.0,-1.0) ( 1.0, 2.0)
(-2.0, 1.0) ( 2.0,-2.0)
( 3.0,-1.0) (-3.0, 1.0) : the end of matrix B
(-3.5,-0.5) ( 1.5, 2.0)
(-4.5, 1.5) (-2.0, 3.5)
(-5.5, 3.5) ( 3.0,-1.5) : the end of matrix C

```

### 10.3 Program Results

nag\_zhemm (f16zcc) Example Program Results

	Matrix	Matrix	Product
		1	2
1	-13.0000	13.0000	
	6.0000	-3.0000	
2	-9.0000	0.0000	
	7.0000	-2.0000	
3	-7.0000	-1.0000	
	8.0000	-13.0000	

---