# C.3 Fortran 95 Thin BLAS

## C.3.1 Introduction

This paper presents a proposal for a specification of Fortran 95 thin BLAS.

A preliminary version of the F90 Blas proposal has been circulated informally (but not very widely) for about 4 years, and code which implements that version has been available in the Fortran 90 software repository maintained by NAG Ltd (**http://www.nag.co.uk**).

This proposal is designed to cover — as far as seems sensible — roughly the same functionality as the original Level 1, 2 and 3 (Fortran 77) BLAS. It does not address sparse matrices or vectors, nor does it explicitly address issues of parallel computation.

Many of the Fortran 77 Level 1 BLAS can be replaced by simple array expressions and assignments in Fortran 95, without loss of convenience or performance (assuming a reasonable degree of optimisation by the compiler). Fortran 95 also allows groups of related Level 2 and Level 3 BLAS to be merged together, each group being covered by a single interface.

## C.3.2 Design of Fortran 95 Interfaces

Our proposed design utilizes the following features of the Fortran 95 language.

**Generic interfaces:** all routines are accessed through *generic* interfaces. A single generic name covers several specific instances whose arguments may differ in the following properties:

- **data type** (real or complex). However, the relevant arguments must be either all real or all complex. (We do not, for example, cater for multiplying a real matrix by a complex matrix, though this functionality could easily be added to the design if there was a need for it.)
- **precision** (or equivalently, kind-value). However, all real or complex arguments must have the same precision.
- **rank** Some arguments may either have rank 2 (to store a matrix) or rank 1 (to store a vector).
- **different argument list** Some of the arguments may not appear in a specific instance. In this case a pre-defined value or a pre-defined action is assumed. The following table contains the pre-defined value or action for the argument that may not be used. Some of these arguments are key arguments that will be described later.

| argument | value or action if the argument is not used |
|---|---|
| alpha | 1.0 or (1.0,0.0) |
| beta | 0.0 or (0.0,0.0) |
| op_x | operate with x |
| lower | reference upper triangle only |
| right_side | operate on the left-hand side |
| unit_diag | non-unit triangular |

**Assumed-shape arrays:** all array arguments are *assumed-shape* arrays, which must have the exact shape required to store the corresponding matrix or vector. Hence arguments to specify array-dimensions or problem-dimensions are not required. The routines assume that the supplied arrays have valid and consistent shapes (see Section C.3.5). Zero dimensions (implying empty arrays) are allowed.

**Key arguments:** in the Fortran 77 BLAS, we use character arguments to specify different options for the operation to be performed. In this proposal we suggest using key arguments. A key argument is a dummy argument whose actual argument must be a named constant defined by BLAS. The following table lists the key arguments, the related BLAS named constants and the equivalent F77 BLAS values.

| dummy argument | named constant | meaning | F77 argument |
|---|---|---|---|
| op_x | *not used* | operate with x | TRANSx = 'N' |
| | blas_trans | operate with transpose x | TRANSx = 'T' |
| | blas_conj | operate with conjugate x | TRANSx = 'C' |
| | blas_conj_trans | operate with conjugate-transpose x | TRANSx = 'H' |
| lower | *not used* | reference upper triangle only | UPLO = 'U' |
| | blas_lower | reference lower triangle only | UPLO = 'L' |
| right_side | *not used* | operate on the left-hand side | SIDE = 'L' |
| | blas_right | operate on the right-hand side | SIDE = 'R' |
| unit_diag | *not used* | non-unit triangular | DIAG = 'N' |
| | blas_unit_diag | unit triangular | DIAG = 'U' |

## C.3.3 Interfaces for Real Data

The primary aim of this paper is to convey the flavour of the different generic interfaces.

Therefore we first describe the interfaces as they apply to *real* data. The extra complications which arise when they apply to complex data will be considered in Section C.3.4.

We summarize each interface in the form of a `subroutine` statement (or in one case a `function` statement), in which all the arguments might appear. (This is a convenient way to think of the interface, although such a statement using the generic interface name never appears in the code.) Arguments which need not be supplied are enclosed in square brackets, for example:

```
subroutine trmm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

This is followed by a table which lists the different variants of the operation, depending either on the ranks of some of the arguments or on the key arguments.

The following table shows the values used in the tables and the related named constant for the key arguments.

| dummy argument | value in table | named constant |
|---|---|---|
| op_x | 'T' | blas_trans |
| | 'C' | blas_conj |
| | 'C/T' | blas_conj_trans |
| right_side | 'R' | blas_right |

### Routines using conventional storage for matrices

By conventional storage, we mean storing a matrix in a 2-dimensional array,

```
subroutine gemm( [alpha,] a, [op_a,] b, [op_b], [beta,] c )
```

| rank of `a` | rank of `b` | rank of `c` | `op_a` | `op_b` | operation | F77 BLAS |
|---|---|---|---|---|---|---|
| 2 | 2 | 2 | | | $C \leftarrow \alpha AB + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | | 'T' | $C \leftarrow \alpha AB^T + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | | $C \leftarrow \alpha A^T B + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | 'T' | $C \leftarrow \alpha A^T B^T + \beta C$ | `_GEMM` |
| 2 | 1 | 1 | | | $c \leftarrow \alpha Ab + \beta c$ | `_GEMV` |
| 2 | 1 | 1 | 'T' | | $c \leftarrow \alpha A^T b + \beta c$ | `_GEMV` |
| 1 | 1 | 2 | | | $C \leftarrow \alpha ab^T + \beta C$ | `_GER_` |

```
subroutine symm( [alpha,] a, b, [beta,] c, [lower,] [right_side] )
subroutine hemm( [alpha,] a, b, [beta,] c, [lower,] [right_side] )
```

| rank of `b` | rank of `c` | `right_side` | operation | F77 BLAS |
|---|---|---|---|---|
| 2 | 2 | | $C \leftarrow \alpha AB + \beta C$ | `_SYMM` |
| 2 | 2 | 'R' | $C \leftarrow \alpha BA + \beta C$ | `_SYMM` |
| 1 | 1 | | $c \leftarrow \alpha Ab + \beta c$ | `_SYMV` |

where $A$ is a symmetric matrix.

```
subroutine syrk( [alpha,] a, [op_a,] [beta,] c, [lower] )
subroutine herk( [alpha,] a, [op_a,] [beta,] c, [lower] )
```

| rank of `a` | `op_a` | operation | F77 BLAS |
|---|---|---|---|
| 2 | | $C \leftarrow \alpha AA^T + \beta C$ | `_SYRK` |
| 2 | 'T' | $C \leftarrow \alpha A^T A + \beta C$ | `_SYRK` |
| 1 | | $C \leftarrow \alpha aa^T + \beta C$ | `_SYR1` |

where $C$ is a symmetric matrix.

```
subroutine syr2k( [alpha,] a, [op_a,] b, [beta,] c, [lower] )
subroutine he2rk( [alpha,] a, [op_a,] b, [beta,] c, [lower] )
```

| rank of `a` | rank of `b` | `op_a` | operation | F77 BLAS |
|---|---|---|---|---|
| 2 | 2 | | $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$ | `_SYR2K` |
| 2 | 2 | 'T' | $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$ | `_SYR2K` |
| 1 | 1 | | $C \leftarrow \alpha ab^T + \alpha ba^T + \beta C$ | `_SYR2` |

where $C$ is a symmetric matrix.

```
subroutine trmm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

| rank of `b` | `op_a` | `right_side` | operation | F77 BLAS |
|---|---|---|---|---|
| 2 | | | $B \leftarrow \alpha AB$ | `_TRMM` |
| 2 | 'T' | | $B \leftarrow \alpha A^T B$ | `_TRMM` |
| 2 | | 'R' | $B \leftarrow \alpha BA$ | `_TRMM` |
| 2 | 'T' | 'R' | $B \leftarrow \alpha BA^T$ | `_TRMM` |
| 1 | | | $b \leftarrow \alpha Ab$ | `_TRMV` |
| 1 | 'T' | | $b \leftarrow \alpha A^T b$ | `_TRMV` |

```
subroutine trsm( [alpha,] a, [op_a,] b, [lower,] [right_side,] [unit_diag] )
```

| rank of b | op_a | right_side | operation | F77 BLAS |
|---|---|---|---|---|
| 2 | | | $B \leftarrow \alpha A^{-1} B$ | _TRSM |
| 2 | 'T' | | $B \leftarrow \alpha A^{-T} B$ | _TRSM |
| 2 | | 'R' | $B \leftarrow \alpha B A^{-1}$ | _TRSM |
| 2 | 'T' | 'R' | $B \leftarrow \alpha B A^{-T}$ | _TRSM |
| 1 | | | $b \leftarrow \alpha A^{-1} b$ | _TRSV |
| 1 | 'T' | | $b \leftarrow \alpha A^{-T} b$ | _TRSV |

### Routines using packed storage for matrices

By *packed* storage, we mean storing the upper or lower triangle of a symmetric or triangular matrix in a 1-dimensional array (i.e. a vector).

```
subroutine spmv( [alpha,] a, b, [beta,] c, [lower] )
subroutine hpmv( [alpha,] a, b, [beta,] c, [lower] )
```

| operation | F77 BLAS |
|---|---|
| $c \leftarrow \alpha A b + \beta c$ | _SPMV |

where $A$ is a symmetric matrix.

```
subroutine spr1( [alpha,] a, [beta,] c, [lower] )
subroutine hpr1( [alpha,] a, [beta,] c, [lower] )
```

| operation | F77 BLAS |
|---|---|
| $C \leftarrow \alpha a a^T + \beta C$ | _SPR1 |

where $C$ is a symmetric matrix.

```
subroutine syr2( [alpha,] a, b, [beta,] c, [lower] )
subroutine he2r( [alpha,] a, b, [beta,] c, [lower] )
```

| operation | F77 BLAS |
|---|---|
| $C \leftarrow \alpha a b^T + \alpha b a^T + \beta C$ | _SYR2 |

where $C$ is a symmetric matrix.

```
subroutine tpmv( [alpha,] a, [op_a,] b, [lower,] [unit_diag] )
```

| op_a | operation | F77 BLAS |
|---|---|---|
| | $b \leftarrow \alpha A b$ | _TPMV |
| 'T' | $b \leftarrow \alpha A^T b$ | _TPMV |

where $A$ is a triangular matrix.

```
subroutine tpsv( [alpha,] a, [op_a,] b, [lower,] [unit_diag] )
```

| op_a | operation | F77 BLAS |
|---|---|---|
| | $b \leftarrow \alpha A^{-1} b$ | _TPSV |
| 'T' | $b \leftarrow \alpha A^{-T} b$ | _TPSV |

where $A$ is a triangular matrix.

### Routines for band matrices

`subroutine gbmv( [alpha,] a, [op_a,] b, [beta,] c, kd )`

| op_a | operation | F77 BLAS |
|---|---|---|
| | $C \leftarrow \alpha Ab + \beta c$ | _GBMV |
| 'T' | $C \leftarrow \alpha A^T b + \beta c$ | _GBMV |

where $A$ is a general band matrix with `kd` superdiagonals supplied.

```
subroutine sbmv( [alpha,] a, b, [beta,] c, [lower] )
subroutine hbmv( [alpha,] a, b, [beta,] c, [lower] )
```

| operation | F77 BLAS |
|---|---|
| $c \leftarrow \alpha Ab + \beta c$ | _SPMV |

where $A$ is a symmetric band matrix.

`subroutine tbmv( [alpha,] a, [op_a,] , [lower,] [unit_diag] )`

| op_a | operation | F77 BLAS |
|---|---|---|
| | $b \leftarrow \alpha Ab$ | _TBMV |
| 'T' | $b \leftarrow \alpha A^T b$ | _TBMV |

where $A$ is a triangular band matrix.

`subroutine tpsv( [alpha,] a, [op_a,] , [lower,] [unit_diag] )`

| op_a | operation | F77 BLAS |
|---|---|---|
| | $b \leftarrow \alpha A^{-1} b$ | _TBSV |
| 'T' | $b \leftarrow \alpha A^{-T} b$ | _TBSV |

where $A$ is a triangular band matrix.

### Level 1 routines

`function nrm2( x )`

Operation: return $\|x\|_2$.

`subroutine swap( x, y )`

Operation: $x \leftrightarrow y$.

`subroutine rot( x, y, c, s )`

`subroutine rotg( a, b, c, s )`

### C.3.4 Interfaces for Complex Data

In this section we show the subroutine `gemm` for complex arguments. The generic interface is that described for real arguments.

| rank of `a` | rank of `b` | rank of `c` | `op_a` | `op_b` | operation | F77 BLAS |
|---|---|---|---|---|---|---|
| 2 | 2 | 2 | | | $C \leftarrow \alpha AB + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | | 'T' | $C \leftarrow \alpha AB^T + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | | 'C/T' | $C \leftarrow \alpha AB^H + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | | $C \leftarrow \alpha A^T B + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | 'T' | $C \leftarrow \alpha A^T B^T + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | 'C/T' | $C \leftarrow \alpha A^T B^H + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'C/T' | | $C \leftarrow \alpha A^H B + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'C/T' | 'T' | $C \leftarrow \alpha A^H B^T + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'C/T' | 'C/T' | $C \leftarrow \alpha A^H B^H + \beta C$ | `_GEMM` |
| 2 | 1 | 1 | | | $c \leftarrow \alpha Ab + \beta c$ | `_GEMV` |
| 2 | 1 | 1 | 'T' | | $c \leftarrow \alpha A^T b + \beta c$ | `_GEMV` |
| 2 | 1 | 1 | 'C/T' | | $c \leftarrow \alpha A^H b + \beta c$ | `_GEMV` |
| 1 | 1 | 2 | | | $C \leftarrow \alpha ab^T + \beta C$ | `_GERU` |
| 1 | 1 | 2 | | 'C' | $C \leftarrow \alpha ab^H + \beta C$ | `_GERC` |

### C.3.5 Error checking

We propose that the Fortran 95 thin BLAS perform no checks on their arguments.

### C.3.6 Comparison with the Fortran 77 BLAS

We consider in more detail each Level of BLAS in turn. In referring to particular operations performed by the BLAS, we use the traditional BLAS names, except that we omit the initial letter (S, D, C, Z) which indicates the data type — for example, `SWAP`. The resulting names are also the generic names which we propose for the Fortran 95 interfaces.

#### Level 1

We include in this proposal only the following:

```
SWAP
ROT
NRM2
ROTG
```

`ROT` and `ROTG` have been extended to cover complex rotations.

#### Level 2

We propose to combine many of the Level 2 BLAS with the corresponding Level 3 BLAS in a single generic interface, the different instances being distinguished by the ranks of some of the arguments. In order to do this, we propose to remove some minor inconsistencies between the specifications of the Level 2 and Level 3 routines:

We propose adding one new routine:

`REFG`

to generate an elementary reflector (that is, a Householder matrix), following the same specification as the LAPACK auxiliary routine xLARTG.

The scope of the proposed BLAS has been extended slightly compared with the Fortran 77 BLAS: for example, we propose Level 1 BLAS for generating an elementary reflector (Householder matrix), and for generating and applying complex plane rotations; we also propose Level 2 BLAS for complex symmetric matrices. On the other hand, many of the Fortran 77 Level 1 BLAS can be replaced in Fortran 95 by simple array constructs, and they have been omitted.

For the thin BLAS we propose that the code does not do any checks on the arguments.

We propose generic interfaces that cover — wherever relevant — both Level 2 and Level 3 BLAS (for example, xTRSV and xTRSM), and have modified the specification of some Level 2 BLAS to make them more consistent with the Level 3 BLAS (for example, xTRSV now has an argument `alpha`).

For each procedure we specify a number of arguments that must be supplied and another set of arguments that need not be supplied. We specify a value or action for each argument which need not be supplied.

We propose that the thin BLAS contain a specific instance for each possible case and no checks or branching is used within the code.

We propose that the early implementations for the thin BLAS will contain simple calls to the reliable and tested F77 BLAS.

For example, the generic `gemm` will consist of the following specific procedures:

- 36 specific procedures each of which calls the F77 BLAS procedure `ZGEMM` (3 settings for each of `op_a` and `op_b`, and 2 settings for each of `alpha` and `beta`).
- 12 specific procedures each of which calls the F77 BLAS procedure `ZGEMV` (3 settings for `op_a`, and 2 settings for each of `alpha` and `beta`).
- 4 specific procedures each of which calls the F77 BLAS procedure `ZGERU` (2 settings for each of `alpha` and `beta`).
- 4 specific procedures each of which calls the F77 BLAS procedure `ZGERC` (2 settings for each of `alpha` and `beta`).
- 36 specific procedures each of which calls the F77 BLAS procedure `DGEMM` (3 settings for each of `op_a` and `op_b`, and 2 settings for each of `alpha` and `beta`). Only 16 procedures are needed, but we allow for `op_a = blas_conj_trans` for similarity with the complex case.
- 12 specific procedures each of which calls the F77 BLAS procedure `DGEMV` (3 settings for `op_a`, and 2 settings for each of `alpha` and `beta`). Only 8 procedures are needed, but we allow for `op_a = blas_conj_trans` for similarity with the complex case.
- 4 specific procedures each of which calls the F77 BLAS procedure `DGER` (2 settings for each of `alpha` and `beta`).
- 4 specific procedures each of which calls the F77 BLAS procedure `DGER` (2 settings for each of `alpha` and `beta`). These are similar to the previous case but have been added to allow `op_b = blas_conj` (as in the complex case for `ZGERC`).

Appendix C.3.8 contains a list of these specific procedures (only double precision procedures are listed).

A proposed document for this procedure is given in a separate document.

### C.3.7 Conclusion

Our principal purpose in presenting this specification at this meeting is to provide additional input to the discussion about different levels of genericity in the interface to linear algebra routines. The thin BLAS are designed principally as building-blocks for software developers and for the BLAS itself.

### C.3.8 Further Details: Specific procedures for gemm

This appendix contains a list of the specific procedures for the generic procedure gemm.

```
!
! 36 procedures each calls the F77 BLAS subroutine ZGEMM
!  a, b and c are rank-2
!
!           alpha      op_a       a     op_b        b beta c   operation
!
! zgemm_301 (alpha,blas_conj_trans,a,blas_conj_trans,b,beta,c)  C < alpha A(H) B(H) + beta C
! zgemm_302 (alpha,blas_conj_trans,a,blas_conj_trans,b,     c)  C < alpha A(H) B(H) + C
! zgemm_303 (alpha,blas_conj_trans,a,blas_trans     ,b,beta,c)  C < alpha A(H) B(T) + beta C
! zgemm_304 (alpha,blas_conj_trans,a,blas_trans     ,b,     c)  C < alpha A(H) B(T) + C
! zgemm_305 (alpha,blas_conj_trans,a,                b,beta,c)  C < alpha A(H) B + beta C
! zgemm_306 (alpha,blas_conj_trans,a,                b,     c)  C < alpha A(H) B + C
! zgemm_307 (alpha,blas_trans     ,a,blas_conj_trans,b,beta,c)  C < alpha A(T) B(H) + beta C
! zgemm_308 (alpha,blas_trans     ,a,blas_conj_trans,b,     c)  C < alpha A(T) B(H) + C
! zgemm_309 (alpha,blas_trans     ,a,blas_trans     ,b,beta,c)  C < alpha A(T) B(T) + beta C
! zgemm_310 (alpha,blas_trans     ,a,blas_trans     ,b,     c)  C < alpha A(T) B(T) + C
! zgemm_311 (alpha,blas_trans     ,a,                b,beta,c)  C < alpha A(T) B + beta C
! zgemm_312 (alpha,blas_trans     ,a,                b,     c)  C < alpha A(T) B + C
! zgemm_313 (alpha,                a,blas_conj_trans,b,beta,c)  C < alpha A B(H) + beta C
! zgemm_314 (alpha,                a,blas_conj_trans,b,     c)  C < alpha A B(H) + C
! zgemm_315 (alpha,                a,blas_trans     ,b,beta,c)  C < alpha A B(T) + beta C
! zgemm_316 (alpha,                a,blas_trans     ,b,     c)  C < alpha A B(T) + C
! zgemm_317 (alpha,                a,                b,beta,c)  C < alpha A B + beta C
! zgemm_318 (alpha,                a,                b,     c)  C < alpha A B + C
! zgemm_319 (      blas_conj_trans,a,blas_conj_trans,b,beta,c)  C < A(H) B(H) + beta C
! zgemm_320 (      blas_conj_trans,a,blas_conj_trans,b,     c)  C < A(H) B(H) + C
! zgemm_321 (      blas_conj_trans,a,blas_trans     ,b,beta,c)  C < A(H) B(T) + beta C
! zgemm_322 (      blas_conj_trans,a,blas_trans     ,b,     c)  C < A(H) B(T) + C
! zgemm_323 (      blas_conj_trans,a,                b,beta,c)  C < A(H) B + beta C
! zgemm_324 (      blas_conj_trans,a,                b,     c)  C < A(H) B + C
! zgemm_325 (      blas_trans     ,a,blas_conj_trans,b,beta,c)  C < A(T) B(H) + beta C
! zgemm_326 (      blas_trans     ,a,blas_conj_trans,b,     c)  C < A(T) B(H) + C
! zgemm_327 (      blas_trans     ,a,blas_trans     ,b,beta,c)  C < A(T) B(T) + beta C
! zgemm_328 (      blas_trans     ,a,blas_trans     ,b,     c)  C < A(T) B(T) + C
! zgemm_329 (      blas_trans     ,a,                b,beta,c)  C < A(T) B + beta C
! zgemm_330 (      blas_trans     ,a,                b,     c)  C < A(T) B + C
! zgemm_331 (                      a,blas_conj_trans,b,beta,c)  C < A B(H) + beta C
! zgemm_332 (                      a,blas_conj_trans,b,     c)  C < A B(H) + C
! zgemm_333 (                      a,blas_trans     ,b,beta,c)  C < A B(T) + beta C
! zgemm_334 (                      a,blas_trans     ,b,     c)  C < A B(T) + C
! zgemm_335 (                      a,                b,beta,c)  C < A B + beta C
! zgemm_336 (                      a,                b,     c)  C < A B + C
!
!
! 12 procedures each calls the F77 BLAS subroutine ZGEMV
!  a is rank-2, and b and c are rank-1
!
!           alpha      op_a       a     op_b        b beta c   operation
!
! zgemv_201 (alpha,blas_conj_trans,a,                b,beta,c)  c < alpha A(H) b + beta c
! zgemv_202 (alpha,blas_conj_trans,a,                b,     c)  c < alpha A(H) b + c
! zgemv_203 (alpha,blas_trans     ,a,                b,beta,c)  c < alpha A(T) b + beta c
! zgemv_204 (alpha,blas_trans     ,a,                b,     c)  c < alpha A(T) b + c
! zgemv_205 (alpha,                a,                b,beta,c)  c < alpha A b + beta c
! zgemv_206 (alpha,                a,                b,     c)  c < alpha A b + c
! zgemv_207 (      blas_conj_trans,a,                b,beta,c)  c < A(H) b + beta c
! zgemv_208 (      blas_conj_trans,a,                b,     c)  c < A(H) b + c
! zgemv_209 (      blas_trans     ,a,                b,beta,c)  c < A(T) b + beta c
! zgemv_200 (      blas_trans     ,a,                b,     c)  c < A(T) b + c
! zgemv_211 (                      a,                b,beta,c)  c < A b + beta c
! zgemv_212 (                      a,                b,     c)  c < A b + c
!
!
! 4 procedures each calls the F77 BLAS subroutine ZGERU
```

```
!  c is rank-2, and a and b are rank-1
!
!           alpha      op_a       a     op_b        b beta c   operation
!
! zgeru_201 (alpha,               a,               b,beta,c)  C < alpha a b(T) + beta C
! zgeru_202 (alpha,               a,               b,     c)  C < alpha a b(T) + C
! zgeru_203 (                     a,               b,beta,c)  C < a b(T) + beta C
! zgeru_204 (                     a,               b,     c)  C < a b(T) + C
!
!
! 4 procedures each calls the F77 BLAS subroutine ZGERC
!  c is rank-2, and a and b are rank-1
!
!           alpha      op_a       a     op_b        b beta c   operation
!
! zgerc_201 (alpha,blas_conj      a,               b,beta,c)  C < alpha a b(H) + beta C
! zgerc_202 (alpha,blas_conj      a,               b,     c)  C < alpha a b(H) + C
! zgerc_203 (      blas_conj      a,               b,beta,c)  C < a b(H) + beta C
! zgerc_204 (      blas_conj      a,               b,     c)  C < a b(H) + C
!
!
!
! 36 procedures each calls the F77 BLAS subroutine DGEMM
!  a, b and c are rank-2
!
!           alpha      op_a       a     op_b        b beta c   operation
!
! dgemm_301 (alpha,blas_conj_trans,a,blas_conj_trans,b,beta,c)  C < alpha A(H) B(H) + beta C
! dgemm_302 (alpha,blas_conj_trans,a,blas_conj_trans,b,     c)  C < alpha A(H) B(H) + C
! dgemm_303 (alpha,blas_conj_trans,a,blas_trans     ,b,beta,c)  C < alpha A(H) B(T) + beta C
! ...
! ...
! dgemm_334 (                     a,blas_trans     ,b,     c)  C < A B(T) + C
! dgemm_335 (                     a,               b,beta,c)  C < A B + beta C
! dgemm_336 (                     a,               b,     c)  C < A B + C
!
!
! 12 procedures each calls the F77 BLAS subroutine DGEMV
!  a is rank-2, and b and c are rank-1
!
!           alpha      op_a       a     op_b        b beta c   operation
!
! dgemv_201 (alpha,blas_conj_trans,a,              b,beta,c)  c < alpha A(H) b + beta c
! dgemv_202 (alpha,blas_conj_trans,a,              b,     c)  c < alpha A(H) b + c
! ...
! dgemv_211 (                     a,               b,beta,c)  c < A b + beta c
! dgemv_212 (                     a,               b,     c)  c < A b + c
!
!
! 8 procedures each calls the F77 BLAS subroutine DGER
!  c is rank-2, and a and b are rank-1
!
!           alpha      op_a       a     op_b        b beta c   operation
!
! dger_201  (alpha,               a,               b,beta,c)  C < alpha a b(T) + beta C
! dger_202  (alpha,               a,               b,     c)  C < alpha a b(T) + C
! dger_203  (                     a,               b,beta,c)  C < a b(T) + beta C
! dger_204  (                     a,               b,     c)  C < a b(T) + C
! dger_205  (alpha,blas_conj      a,               b,beta,c)  C < alpha a b(H) + beta C
! dger_206  (alpha,blas_conj      a,               b,     c)  C < alpha a b(H) + C
! dger_207  (      blas_conj      a,               b,beta,c)  C < a b(H) + beta C
! dger_208  (      blas_conj      a,               b,     c)  C < a b(H) + C
```

# Procedure gemm

## Description

`gemm` is a generic procedure which performs one of following operations:

| rank of `a` | rank of `b` | rank of `c` | `op_a` | `op_b` | operation | F77 BLAS |
|---|---|---|---|---|---|---|
| 2 | 2 | 2 | | | $C \leftarrow \alpha AB + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | | 'T' | $C \leftarrow \alpha AB^T + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | | 'C/T' | $C \leftarrow \alpha AB^H + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | | $C \leftarrow \alpha A^T B + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | 'T' | $C \leftarrow \alpha A^T B^T + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'T' | 'C/T' | $C \leftarrow \alpha A^T B^H + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'C/T' | | $C \leftarrow \alpha A^H B + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'C/T' | 'T' | $C \leftarrow \alpha A^H B^T + \beta C$ | `_GEMM` |
| 2 | 2 | 2 | 'C/T' | 'C/T' | $C \leftarrow \alpha A^H B^H + \beta C$ | `_GEMM` |
| 2 | 1 | 1 | | | $c \leftarrow \alpha Ab + \beta c$ | `_GEMV` |
| 2 | 1 | 1 | 'T' | | $c \leftarrow \alpha A^T b + \beta c$ | `_GEMV` |
| 2 | 1 | 1 | 'C/T' | | $c \leftarrow \alpha A^H b + \beta c$ | `_GEMV` |
| 1 | 1 | 2 | | | $C \leftarrow \alpha ab^T + \beta C$ | `_GER_` |
| 1 | 1 | 2 | | 'C' | $C \leftarrow \alpha ab^H + \beta C$ | `_GER_` |

(If $A$ is real, then $A^H = A^T$.)

## Usage

```
CALL gemm([alpha], [op_a], a, [op_b], b, [beta], c)
```

One or more of the arguments in square brakets can be dropped. The order of the supplied arguments must remain unchanged.

## Interfaces

Distinct interfaces are provided for each of the combinations of the following cases:

Real / complex data

**Real data:** `alpha`, `a`, `b`, `beta` and `c` are of type real(kind=*wp*).

**Complex data:** `alpha`, `a`, `b`, `beta` and `c` are of type complex(kind=*wp*).

different ranks

**f77_gemm:** `a`, `b` and `c` are rank-2 arrays.

**f77_gemv:** `a` is a rank-2 array while `b` and `c` are rank-1 arrays.

**f77_ger:** `c` is a rank-2 array while `a` and `b` are rank-1 arrays.

## Arguments

**All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as 'x($n$)' is used in the argument descriptions to specify that the array `x` must have exactly $n$ elements.**

The procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$m$ — the first dimension of `c`, if `c` is rank-2 ($m$ =`SIZE(c,1)`), or the size of `c` if it is rank-1 ($m$ =`SIZE(c)`)

$n$ — the second dimension of `c` if it is rank-2 ($n$ =`SIZE(c,2)`)

$k$ — the intermediate dimension

## Mandatory arguments

One or more of the arguments `alpha`, `op_a`, `op_b` and `beta` can be dropped. The order of the supplied arguments must remain unchanged.

**alpha** — real(kind=*wp*)/ complex(kind=*wp*), intent(in)

*Input:* the value of $\alpha$ if different from one.

*Note:* if $\alpha$ is exactly one, you need not supply this argument.

**op_a** — a "key" argument, intent(in)

*Input:* if `op_a` is supplied, it specifies whether the operation involves the transpose $A^T$ or its conjugate-transpose $A^H$ ($= A^T$ if $A$ is real). In this case `op_a` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose $A^T$ rather than the matrix $A$;

`blas_conj_trans`: if the operation involves the conjugate-transpose $A^H$ rather than the matrix $A$.

For further explanation of "key" arguments, see the ????.

*Note:* for real matrices, `blas_conj_trans` is equivalent to `blas_trans`.

*Constraints:* `op_a` *must not* be supplied if `a` is rank-1 or if the operation does not involve the transpose or the conjugate-transpose of $A$.

**a**$(m)$ **/ a**$(p,q)$ — real(kind=$wp$)/ complex(kind=$wp$), intent(in)

*Input:* the matrix $A$ or vector $a$.
If `a` is rank-2 then:

if `op_a` is not supplied, the shape of `a` must be $(m,k)$;

if `op_a` is supplied, the shape of `a` must be $(k,m)$.

**op_b** — a "key" argument, intent(in)

*Input:* if `op_b` is supplied and `b` is rank-2, it specifies whether the operation involves the transpose $B^T$ or its conjugate-transpose $B^H$ ($= B^T$ if $B$ is real). In this case `op_b` must have one of the following values (which are named constants, each of a different derived type, defined by the BLAS, and accessible from the module `blas`):

`blas_trans`: if the operation involves the transpose $B^T$ rather than the matrix $B$;

`blas_conj_trans`: if the operation involves the conjugate-transpose $B^H$ rather than the matrix $B$.

If `op_b` is supplied and `b` is rank-1, it specifies that the operation involves the conjugate of $b^T$ $(b^H)$ rather than $b^T$. In this case `op_b` must have be `blas_conj` (which is a named constant of a derived type, defined by the BLAS, and accessible from the module `blas`).

For further explanation of "key" arguments, see the ????.

*Note:* for real matrices, `blas_conj_trans` is equivalent to `blas_trans`. For real arrays `blas_conj` does not have any effect.

*Constraints:* `op_b` *must not* be supplied if the operation does not involve the conjugate of $b$, the transpose of $B$ or the conjugate-transpose of $B$.

**b**$(r)$ **/ b**$(r,s)$ — real(kind=$wp$)/ complex(kind=$wp$), intent(in)

*Input:* the matrix $B$ or vector $b$.
If `b` is rank-1 then:

if `a` is rank-1, the shape of `b` must be $(m)$;

if `a` is rank-2, the shape of `b` must be `SIZE(op_a(a),2)`.

If `b` is rank-2 then:

if `op_b` is not supplied, the shape of `b` must be $(k,n)$;

if `op_b` is supplied, the shape of `b` must be $(n,k)$.

**beta** — real(kind=$wp$)/ complex(kind=$wp$), intent(in)

*Input:* the value of $\beta$ if different from zero.

*Note:* if $\beta$ is exactly zero, you need not supply this argument.

**c**$(m)$ **/ c**$(m,n)$ — real(kind=$wp$)/ complex(kind=$wp$), intent(inout)

*Input:* the matrix $C$ or vector $c$. If `beta` is not supplied `c` need not be initialized.

*Output:* the matrix $C$ or vector $c$ after applying the operation.

## Examples of usage

One or more of the arguments `alpha`, `op_a`, `op_b` and `beta` can be dropped. The order of the supplied arguments must remain unchanged.

To perform $C \leftarrow \alpha AB^H$ use the call:

```
CALL gemm (alpha,a,blas_conj_trans,b,c)
```

To perform $C \leftarrow AB + \beta C$ use the call:

```
CALL gemm (a,b,beta,c)
```

To perform $C \leftarrow \alpha AB + \beta C$ use the call:

```
CALL gemm (alpha,a,b,beta,c)
```

To perform $C \leftarrow A^T B^H + \beta C$ use the call:

```
CALL gemm (blas_trans,a,blas_her,b,beta,c)
```

To perform $c \leftarrow \alpha A^T b + \beta c$ use the call:

```
CALL gemm (alpha,blas_trans,a,b,beta,c)
```

To perform $C \leftarrow ab^H + \beta C$ use the call:

```
CALL gemm (a,b,blas_conj,beta,c)
```