Chapter f06 – Linear Algebra Support Functions

1. Scope of the Chapter

This Chapter is concerned with basic linear algebra functions which perform elementary algebraic operations involving vectors and matrices.

2. Background

All the functions in this chapter meet the specification of the Basic Linear Algebra Subprograms (BLAS) in C as described in Datardina et al (1992). These in turn were derived from the pioneering work of Dongarra *et al* (1988) and Dongarra *et al* (1990) on Fortran 77 BLAS. The functions described are concerned with matrix-vector operations and matrix-matrix operations. These will be referred to here as the Level-2 BLAS and Level-3 BLAS respectively. The terminology reflects the number of operations involved. For example, a Level-2 function involves $O(n^2)$ operations for an n by n matrix. The Level 1 Blas will be included at a future mark of the C Library.

Table 1.1 indicates the NAG coded naming scheme for the functions in this Chapter.

		Level-2	Level-3
'real'	BLAS function	f06p_c	f06y_c
'complex'	BLAS function	f06s_c	f06z_c

The C BLAS names for these functions are the same as the corresponding Fortran names except that they are in lower case.

The functions in this chapter do not have full function documents, but instead are covered by general descriptions in Section 4 sufficient to enable their use. As this chapter is concerned only with basic linear algebra operations, the functions will not normally be required by the general user. The purpose of each function is indicated in Section 3 so that those users requiring these functions to build specialist linear algebra modules can determine which functions are of interest.

3. References

Datardina S P, Du Croz J J, Hammarling S J and Pont M W (1992) A Proposed Specification of BLAS Routines in C *The Journal of C Language Translation* **3** 295–309.

Dongarra J J, Du Croz J J, Hammarling S and Hanson R J (1988) An Extended Set of FORTRAN Basic Linear Algebra Subprograms *ACM Trans. Math. Softw.* **14** 1–32.

Dongarra J J, Du Croz J J, Duff I S and Hammarling S (1990) A Set of Level 3 Basic Linear Algebra Subprograms *ACM Trans. Math. Softw.* **16** 1–28.

4. Recommendations on Choice and Use of Functions

This section lists the functions in the categories Level-2 (matrix-vector) and Level-3 (matrix-matrix). The corresponding BLAS name is indicated in brackets.

Within each section functions are listed in alphabetic order of the fifth character in the short function name, so that corresponding real and complex functions may have adjacent entries.

4.1. The Level-2 Matrix-vector Functions

The Level-2 functions perform matrix-vector operations, such as forming the product between a matrix and a vector.

Compute a matrix-vector product; real general matrix	`dgemv (f06pac)`
Compute a matrix-vector product; complex general matrix	`zgemv (f06sac)`
Compute a matrix-vector product; real general band matrix	`dgbmv (f06pbc)`
Compute a matrix-vector product; complex general band matrix	`zgbmv (f06sbc)`
Compute a matrix-vector product; real symmetric matrix	`dsymv (f06pcc)`
Compute a matrix-vector product; complex Hermitian matrix	`zhemv (f06scc)`
Compute a matrix-vector product; real symmetric band matrix	`dsbmv (f06pdc)`
Compute a matrix-vector product; complex Hermitian band matrix	`zhbmv (f06sdc)`
Compute a matrix-vector product; real symmetric packed matrix	`dspmv (f06pec)`
Compute a matrix-vector product; complex Hermitian packed matrix	`zhpmv (f06sec)`
Compute a matrix-vector product; real triangular matrix	`dtrmv (f06pfc)`
Compute a matrix-vector product; complex triangular matrix	`ztrmv (f06sfc)`
Compute a matrix-vector product; real triangular band matrix	`dtbmv (f06pgc)`
Compute a matrix-vector product; complex triangular band matrix	`ztbmv (f06sgc)`
Compute a matrix-vector product; real triangular packed matrix	`dtpmv (f06phc)`
Compute a matrix-vector product; complex triangular packed matrix	`ztpmv (f06shc)`
Solve a system of equations; real triangular coefficient matrix	`dtrsv (f06pjc)`
Solve a system of equations; complex triangular coefficient matrix	`ztrsv (f06sjc)`
Solve a system of equations; real triangular band coefficient matrix	`dtbsv (f06pkc)`
Solve a system of equations; complex triangular band coefficient matrix	`ztbsv (f06skc)`
Solve a system of equations; real triangular packed coefficient matrix	`dtpsv (f06plc)`
Solve a system of equations; complex triangular packed coefficient matrix	`ztpsv (f06slc)`
Perform a rank-one update; real general matrix	`dger (f06pmc)`
Perform a rank-one update; complex general matrix (unconjugated vector)	`zgeru (f06smc)`
Perform a rank-one update; complex general matrix (conjugated vector)	`zgerc (f06snc)`
Perform a rank-one update; real symmetric matrix	`dsyr (f06ppc)`
Perform a rank-one update; complex Hermitian matrix	`zher (f06spc)`
Perform a rank-one update; real symmetric packed matrix	`dspr (f06pqc)`
Perform a rank-one update; complex Hermitian packed matrix	`zhpr (f06sqc)`
Perform a rank-two update; real symmetric matrix	`dsyr2 (f06prc)`
Perform a rank-two update; complex Hermitian matrix	`zher2 (f06src)`
Perform a rank-two update; real symmetric packed matrix	`dspr2 (f06psc)`
Perform a rank-two update; complex Hermitian packed matrix	`zhpr2 (f06ssc)`

4.2. The Level-3 Matrix-matrix Functions

The Level-3 functions perform matrix-matrix operations, such as forming the product of two matrices.

Compute a matrix-matrix product; two real rectangular matrices	`dgemm (f06yac)`
Compute a matrix-matrix product; two complex rectangular matrices	`zgemm (f06zac)`
Compute a matrix-matrix product; one real symmetric matrix, one real rectangular matrix	`dsymm (f06ycc)`

Compute a matrix-matrix product; one complex Hermitian matrix, one complex rectangular matrix	`zhemm (f06zcc)`
Compute a matrix-matrix product; one real triangular matrix, one real rectangular matrix	`dtrmm (f06yfc)`
Compute a matrix-matrix product; one complex triangular matrix, one complex rectangular matrix	`ztrmm (f06zfc)`
Solve a system of equations with multiple right-hand sides, real triangular coefficient matrix	`dtrsm (f06yjc)`
Solve a system of equations with multiple right-hand sides, complex triangular coefficient matrix	`ztrsm (f06zjc)`
Perform a rank-k update of a real symmetric matrix	`dsyrk (f06ypc)`
Perform a rank-k update of a complex hermitian matrix	`zherk (f06zpc)`
Perform a rank-$2k$ update of a real symmetric matrix	`dsyr2k (f06yrc)`
Perform a rank-$2k$ update of a complex Hermitian matrix	`zher2k (f06zrc)`
Compute a matrix-matrix product: one complex symmetric matrix, one complex rectangular matrix	`zsymm (f06ztc)`
Perform a rank-k update of a complex symmetric matrix	`zsyrk (f06zuc)`
Perform a rank-$2k$ update of a complex symmetric matrix	`zsyr2k (f06zwc)`

5. Description of the f06 Functions

The argument lists use the following data types.

`Integer`:	an integer data type of at least 32 bits.
`double`:	the regular double precision floating-point type.
`Complex`:	a double precision complex type.

plus the enumeration types given by

```
typedef enum { NoTranspose, Transpose, ConjugateTranspose } MatrixTranspose;
typedef enum { UpperTriangle, LowerTriangle } MatrixTriangle;
typedef enum { UnitTriangular, NotUnitTriangular } MatrixUnitTriangular;
typedef enum { LeftSide, RightSide } OperationSide;
```

In this section we describe the purpose of each function and give information on the argument lists, where appropriate indicating their general nature. Usually the association between the function arguments and the mathematical variables is obvious and in such cases a description of the argument is omitted.

Within each section, the argument lists for all functions are presented, followed by the purpose of the functions and information on the argument lists.

Within each section functions are listed in alphabetic order of the fifth character in the function name, so that corresponding real and complex functions may have adjacent entries.

5.1. The Level-2 Matrix-vector Functions

The matrix-vector functions all have one array argument representing a matrix; usually this is a two-dimensional array but in some cases the matrix is represented by a one-dimensional array.

The size of the matrix is determined by the arguments **m** and **n** for an m by n rectangular matrix; and by the argument **n** for an n by n symmetric, Hermitian, or triangular matrix. Note that it is permissible to call the functions with **m** or **n** $= 0$, in which case the functions exit immediately without referencing their array arguments. For band matrices, the bandwidth is determined by the arguments **kl** and **ku** for a rectangular matrix with **kl** sub-diagonals and **ku** super-diagonals; and by the argument **k** for a symmetric, Hermitian, or triangular matrix with **k** sub-diagonals and/or super-diagonals.

The description of the $m \times n$ matrix consists either of the array name (**a**) followed by the trailing (last) dimension of the array as declared in the calling (sub)program (**tda**), when the matrix is being stored in a two-dimensional array; or the array name (**ap**) alone when the matrix is being stored as a (packed) vector. In the former case the actual array must be allocated at least $((m-1)d+l)$ contiguous elements, where d is the trailing dimension of the array, $d \geq l$, and $l = n$ for arrays representing general, symmetric, Hermitian and triangular matrices, $l = kl + ku + 1$ for arrays representing general band matrices and $l = k + 1$ for symmetric, Hermitian and triangular band matrices. For one-dimensional arrays representing matrices (**packed storage**) the actual array must contain at least $\frac{1}{2}n(n+1)$ elements.

The length of each vector, n, is represented by the argument **n**, and the routines may be called with non-positive values of **n**, in which case the routine returns immediately.

In addition to the argument **n**, each vector argument also has an **increment** argument that immediately follows the vector argument, and whose name consists of the three characters **inc**, followed by the name of the vector. For example, a vector x will be represented by the two arguments **x**, **incx**. The increment argument is the spacing (stride) in the array for which the elements of the vector occur. For instance, if **incx** $= 2$, then the elements of x are in locations $\mathbf{x}[0], \mathbf{x}[2], \ldots, \mathbf{x}[2*\mathbf{n}-2]$ of the array **x** and the intermediate locations $\mathbf{x}[1], \mathbf{x}[3], \ldots, \mathbf{x}[2*\mathbf{n}-3]$ are not referenced.

Zero increments are not permitted. When **incx** > 0, the vector element x_i is in the array element $\mathbf{x}[(i-1)*\mathbf{incx}]$, and when **incx** < 0 the elements are stored in the reverse order so that the vector element x_i is in the array element $\mathbf{x}[-(n-i)*\mathbf{incx}]$ and hence, in particular, the element x_n is in $\mathbf{x}[0]$. The declared length of the array **x** in the calling (sub)program must be at least $(1+(n-1)*|\mathbf{incx}|)$.

The arguments that specify options are enumeration arguments with the names **trans**, **uplo** and **diag**. **trans** is used by the matrix-vector product functions as follows:

Value	**Meaning**
NoTranspose	Operate with the matrix
Transpose	Operate with the transpose of the matrix
ConjugateTranspose	Operate with the conjugate transpose of the matrix

In the real case the values **Transpose** and **ConjugateTranspose** have the same meaning.

uplo is used by the Hermitian, symmetric, and triangular matrix functions to specify whether the upper or lower triangle is being referenced as follows:

Value	**Meaning**
UpperTriangle	Upper triangle
LowerTriangle	Lower triangle

diag is used by the triangular matrix functions to specify whether or not the matrix is unit triangular, as follows:

Value	**Meaning**
UnitTriangular	Unit triangular
NotUnitTriangular	Non-unit triangular

When **diag** is supplied as **UnitTriangular**, the diagonal elements are not referenced.

5.1.1. Matrix storage schemes

Conventional storage

The default scheme for storing matrices is the obvious one: a matrix A is stored in a 2-dimensional array A, with matrix element a_{ij} stored in array element $\mathrm{A}(i,j)$.

If a matrix is **triangular** (upper or lower, as specified by the argument **uplo**), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set. Such elements are indicated by $*$ in the examples below. For example, when $n = 4$:

uplo	**Triangular matrix A**	**Storage in array A**
UpperTriangle	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{matrix}$
LowerTriangle	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$

Routines which handle **symmetric** or **Hermitian** matrices allow for either the upper or lower triangle of the matrix (as specified by **uplo**) to be stored in the corresponding elements of the array; the remaining elements of the array need not be set. For example, when $n = 4$:

uplo	**Hermitian matrix A**	**Storage in array A**
UpperTriangle	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} \\ \bar{a}_{14} & \bar{a}_{24} & \bar{a}_{34} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ * & a_{22} & a_{23} & a_{24} \\ * & * & a_{33} & a_{34} \\ * & * & * & a_{44} \end{matrix}$
LowerTriangle	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & \bar{a}_{41} \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} a_{11} & * & * & * \\ a_{21} & a_{22} & * & * \\ a_{31} & a_{32} & a_{33} & * \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$

Packed storage

Symmetric, Hermitian or triangular matrices may be stored more compactly, if the relevant triangle (again as specified by **uplo**) is packed by rows in a 1-dimensional array.

- if **uplo** $=$ **UpperTriangle**, a_{ij} is stored in **ap**$[j - 1 + (2n - i)(i - 1)/2]$ for $i \le j$;

- if **uplo** $=$ **LowerTriangle**, a_{ij} is stored in **ap**$[j - 1 + i(i - 1)/2]$ for $j \le i$.

For example:

uplo	**Triangular matrix a**	**Packed storage in array ap**
UpperTriangle	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$\underbrace{a_{11}\, a_{12}\, a_{13}\, a_{14}}\ \underbrace{a_{22}\, a_{23}\, a_{24}}\ \underbrace{a_{33}\, a_{34}}\ \underbrace{a_{44}}$
LowerTriangle	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}}\ \underbrace{a_{21}\, a_{22}}\ \underbrace{a_{31}\, a_{32}\, a_{33}}\ \underbrace{a_{41}\, a_{42}\, a_{43}\, a_{44}}$

Note that for real symmetric matrices, packing the upper triangle by rows is equivalent to packing the lower triangle by columns; packing the lower triangle by rows is equivalent to packing the upper triangle by columns. (For complex Hermitian matrices, the only difference is that the off-diagonal elements are conjugated.)

Band storage

A band matrix with kl subdiagonals and ku superdiagonals may be stored compactly in a 2-dimensional array with $kl + ku + 1$ columns and m rows. Rows of the matrix are stored in corresponding rows of the array, and diagonals of the matrix are stored in columns of the array.

For example, when $n = 5$, $kl = 2$ and $ku = 1$:

Band Matrix a	**Band storage in array ab**
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{array}{cccc} * & * & a_{11} & a_{12} \\ * & a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} & * \end{array}$

The elements marked $*$ in the upper left $k_l \times k_l$ triangle and lower right $k_u \times k_u$ of the array **ab** need not be set, and are not referenced by the routines.

The following code fragment will transfer a band matrix $A(m, n)$ from conventional storage to band storage **ab**

```
for(i=0; i<m; ++i){
  k+kl-i;
  for (j=MAX(0,i-kl); j<=MIN(n-1,i+ku); ++j){
    ab[i][k+j]=A[i][j];
  }
}
```

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular.

For symmetric or Hermitian band matrices with k subdiagonals or superdiagonals, only the upper or lower triangle (as specified by **uplo**) need be stored:

The following code fragments will transfer a symmetric or Hermitian matrix $A(n, n)$ from conventional storage to band storage **ab**

if **uplo**=**UpperTriangle**

```
for(i=0; i<n; ++i){
  l=-i;
  for (j=i; j<=MIN(n-1,i+k); ++j){
    ab[i][l+j]=A[i][j];
  }
}
```

if **uplo**=**LowerTriangle**

```
for(i=0; i<n; ++i){
  l=k-i;
  for (j=MAX(0,i-k); j<=i; ++j){
    ab[i][l+j]=A[i][j];
  }
}
```

For example, when $n = 5$ and $k = 2$: