Go to the first, previous, next, last section, table of contents.

BLAS Support

The Basic Linear Algebra Subprograms (BLAS) define a set of fundamental operations on vectors and matrices which can be used to create optimized higher-level linear algebra functionality.

The library provides a low-level layer which corresponds directly to the C-language BLAS standard, referred to here as "CBLAS", and a higher-level interface for operations on GSL vectors and matrices. Users who are interested in simple operations on GSL vector and matrix objects should use the high-level layer, which is declared in the file `gsl_blas.h`. This should satisfy the needs of most users. Note that GSL matrices are implemented using dense-storage so the interface only includes the corresponding dense-storage BLAS functions. The full BLAS functionality for band-format and packed-format matrices is available through the low-level CBLAS interface.

The interface for the `gsl_cblas` layer is specified in the file `gsl_cblas.h`. This interface corresponds the BLAS Technical Forum's draft standard for the C interface to legacy BLAS implementations. Users who have access to other conforming CBLAS implementations can use these in place of the version provided by the library. Note that users who have only a Fortran BLAS library can use a CBLAS conformant wrapper to convert it into a CBLAS library. A reference CBLAS wrapper for legacy Fortran implementations exists as part of the draft CBLAS standard and can be obtained from Netlib. The complete set of CBLAS functions is listed in an appendix (see section GSL CBLAS Library).

There are three levels of BLAS operations,

Level 1
: Vector operations, e.g. y = \alpha x + y

Level 2
: Matrix-vector operations, e.g. y = \alpha A x + \beta y

Level 3
: Matrix-matrix operations, e.g. C = \alpha A B + C

Each routine has a name which specifies the operation, the type of matrices involved and their precisions. Some of the most common operations and their names are given below,

DOT
: scalar product, x^T y

AXPY
: vector sum, \alpha x + y

MV
: matrix-vector product, A x

SV
: matrix-vector solve, inv(A) x

MM
: matrix-matrix product, A B

SM
: matrix-matrix solve, inv(A) B

The type of matrices are,

GE
: general

GB
: general band

SY
: symmetric

SB
: symmetric band

SP
: symmetric packed

HE
: hermitian

HB
: hermitian band

HP
: hermitian packed

TR
: triangular

TB
: triangular band

TP
: triangular packed

Each operation is defined for four precisions,

S
: single real

D
: double real

C
: single complex

Z
: double complex

Thus, for example, the name SGEMM stands for "single-precision general matrix-matrix multiply" and ZGEMM stands for "double-precision complex matrix-matrix multiply".

GSL BLAS Interface

GSL provides dense vector and matrix objects, based on the relevant built-in types. The library provides an interface to the BLAS operations which apply to these objects. The interface to this functionality is given in the file `gsl_blas.h`.

Level 1

Function: int **gsl_blas_sdsdot** *(float alpha, const gsl_vector_float * x, const gsl_vector_float * y, float * result)*
Function: int **gsl_blas_dsdot** *(const gsl_vector_float * x, const gsl_vector_float * y, double * result)*
These functions compute the sum \alpha + x^T y for the vectors *x* and *y*, returning the result in *result*.

Function: int **gsl_blas_sdot** *(const gsl_vector_float * x, const gsl_vector_float * y, float * result)*
Function: int **gsl_blas_ddot** *(const gsl_vector * x, const gsl_vector * y, double * result)*
These functions compute the scalar product x^T y for the vectors *x* and *y*, returning the result in *result*.

Function: int **gsl_blas_cdotu** *(const gsl_vector_complex_float * x, const gsl_vector_complex_float * y, gsl_complex_float * dotu)*
Function: int **gsl_blas_zdotu** *(const gsl_vector_complex * x, const gsl_vector_complex * y, gsl_complex * dotu)*
These functions compute the complex scalar product x^T y for the vectors *x* and *y*, returning the result in *result*

Function: int **gsl_blas_cdotc** *(const gsl_vector_complex_float * x, const gsl_vector_complex_float * y, gsl_complex_float * dotc)*
Function: int **gsl_blas_zdotc** *(const gsl_vector_complex * x, const gsl_vector_complex * y, gsl_complex * dotc)*
These functions compute the complex conjugate scalar product x^H y for the vectors *x* and *y*, returning the result in *result*

Function: float **gsl_blas_snrm2** *(const gsl_vector_float * x)*
Function: double **gsl_blas_dnrm2** *(const gsl_vector * x)*
These functions compute the Euclidean norm ||x||_2 = \sqrt {\sum x_i^2} of the vector *x*.

Function: float **gsl_blas_scnrm2** *(const gsl_vector_complex_float * x)*
Function: double **gsl_blas_dznrm2** *(const gsl_vector_complex * x)*
These functions compute the Euclidean norm of the complex vector *x*,

```
||x||_2 = \sqrt {\sum (\Re(x_i)^2 + \Im(x_i)^2)}.
```

Function: float **gsl_blas_sasum** *(const gsl_vector_float * x)*
Function: double **gsl_blas_dasum** *(const gsl_vector * x)*
These functions compute the absolute sum \sum |x_i| of the elements of the vector *x*.

Function: float **gsl_blas_scasum** *(const gsl_vector_complex_float * x)*
Function: double **gsl_blas_dzasum** *(const gsl_vector_complex * x)*
These functions compute the absolute sum \sum |\Re(x_i)| + |\Im(x_i)| of the elements of the vector *x*.

Function: CBLAS_INDEX_t **gsl_blas_isamax** *(const gsl_vector_float * x)*
Function: CBLAS_INDEX_t **gsl_blas_idamax** *(const gsl_vector * x)*
Function: CBLAS_INDEX_t **gsl_blas_icamax** *(const gsl_vector_complex_float * x)*
Function: CBLAS_INDEX_t **gsl_blas_izamax** *(const gsl_vector_complex * x)*
These functions return the index of the largest element of the vector *x*. The largest element is determined

by its absolute magnitude for real vector and by the sum of the magnitudes of the real and imaginary parts |\Re(x_i)| + |\Im(x_i)| for complex vectors. If the largest value occurs several times then the index of the first occurrence is returned.

Function: int **gsl_blas_sswap** *(gsl_vector_float * x, gsl_vector_float * y)*
Function: int **gsl_blas_dswap** *(gsl_vector * x, gsl_vector * y)*
Function: int **gsl_blas_cswap** *(gsl_vector_complex_float * x, gsl_vector_complex_float * y)*
Function: int **gsl_blas_zswap** *(gsl_vector_complex * x, gsl_vector_complex * y)*
These functions exchange the elements of the vectors *x* and *y*.

Function: int **gsl_blas_scopy** *(const gsl_vector_float * x, gsl_vector_float * y)*
Function: int **gsl_blas_dcopy** *(const gsl_vector * x, gsl_vector * y)*
Function: int **gsl_blas_ccopy** *(const gsl_vector_complex_float * x, gsl_vector_complex_float * y)*
Function: int **gsl_blas_zcopy** *(const gsl_vector_complex * x, gsl_vector_complex * y)*
These functions copy the elements of the vector *x* into the vector *y*.

Function: int **gsl_blas_saxpy** *(float alpha, const gsl_vector_float * x, gsl_vector_float * y)*
Function: int **gsl_blas_daxpy** *(double alpha, const gsl_vector * x, gsl_vector * y)*
Function: int **gsl_blas_caxpy** *(const gsl_complex_float alpha, const gsl_vector_complex_float * x, gsl_vector_complex_float * y)*
Function: int **gsl_blas_zaxpy** *(const gsl_complex alpha, const gsl_vector_complex * x, gsl_vector_complex * y)*
These functions compute the sum y = \alpha x + y for the vectors *x* and *y*.

Function: void **gsl_blas_sscal** *(float alpha, gsl_vector_float * x)*
Function: void **gsl_blas_dscal** *(double alpha, gsl_vector * x)*
Function: void **gsl_blas_cscal** *(const gsl_complex_float alpha, gsl_vector_complex_float * x)*
Function: void **gsl_blas_zscal** *(const gsl_complex alpha, gsl_vector_complex * x)*
Function: void **gsl_blas_csscal** *(float alpha, gsl_vector_complex_float * x)*
Function: void **gsl_blas_zdscal** *(double alpha, gsl_vector_complex * x)*
These functions rescale the vector *x* by the multiplicative factor *alpha*.

Function: int **gsl_blas_srotg** *(float a[], float b[], float c[], float s[])*
Function: int **gsl_blas_drotg** *(double a[], double b[], double c[], double s[])*
These functions compute a Givens rotation (c,s) which zeroes the vector (a,b),

```
[  c  s ] [ a ] = [ r ]
[ -s  c ] [ b ]   [ 0 ]
```

The variables *a* and *b* are overwritten by the routine.

Function: int **gsl_blas_srot** *(gsl_vector_float * x, gsl_vector_float * y, float c, float s)*
Function: int **gsl_blas_drot** *(gsl_vector * x, gsl_vector * y, const double c, const double s)*
These functions apply a Givens rotation (x', y') = (c x + s y, -s x + c y) to the vectors *x*, *y*.

Function: int **gsl_blas_srotmg** *(float d1[], float d2[], float b1[], float b2, float P[])*
Function: int **gsl_blas_drotmg** *(double d1[], double d2[], double b1[], double b2, double P[])*
These functions compute a modified Given's transformation.

Function: int **gsl_blas_srotm** *(gsl_vector_float * x, gsl_vector_float * y, const float P[])*
Function: int **gsl_blas_drotm** *(gsl_vector * x, gsl_vector * y, const double P[])*
These functions apply a modified Given's transformation.

Level 2

Function: int **gsl_blas_sgemv** *(CBLAS_TRANSPOSE_t TransA, float alpha, const gsl_matrix_float * A, const gsl_vector_float * x, float beta, gsl_vector_float * y)*
Function: int **gsl_blas_dgemv** *(CBLAS_TRANSPOSE_t TransA, double alpha, const gsl_matrix * A, const gsl_vector * x, double beta, gsl_vector * y)*
Function: int **gsl_blas_cgemv** *(CBLAS_TRANSPOSE_t TransA, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_vector_complex_float * x, const gsl_complex_float beta, gsl_vector_complex_float * y)*
Function: int **gsl_blas_zgemv** *(CBLAS_TRANSPOSE_t TransA, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_vector_complex * x, const gsl_complex beta, gsl_vector_complex * y)*
These functions compute the matrix-vector product and sum y = \alpha op(A) x + \beta y, where op(A) = A, A^T, A^H for *TransA* = `CblasNoTrans`, `CblasTrans`, `CblasConjTrans`.

Function: int **gsl_blas_strmv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_float * A, gsl_vector_float * x)*
Function: int **gsl_blas_dtrmv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix * A, gsl_vector * x)*
Function: int **gsl_blas_ctrmv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex_float * A, gsl_vector_complex_float * x)*
Function: int **gsl_blas_ztrmv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex * A, gsl_vector_complex * x)*
These functions compute the matrix-vector product and sum y =\alpha op(A) x + \beta y for the triangular matrix *A*, where op(A) = A, A^T, A^H for *TransA* = `CblasNoTrans`, `CblasTrans`, `CblasConjTrans`. When *Uplo* is `CblasUpper` then the upper triangle of *A* is used, and when *Uplo* is `CblasLower` then the lower triangle of *A* is used. If *Diag* is `CblasNonUnit` then the diagonal of the matrix is used, but if *Diag* is `CblasUnit` then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

Function: int **gsl_blas_strsv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_float * A, gsl_vector_float * x)*
Function: int **gsl_blas_dtrsv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix * A, gsl_vector * x)*
Function: int **gsl_blas_ctrsv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex_float * A, gsl_vector_complex_float * x)*
Function: int **gsl_blas_ztrsv** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_matrix_complex * A, gsl_vector_complex *x)*
These functions compute inv(op(A)) x for *x*, where op(A) = A, A^T, A^H for *TransA* = `CblasNoTrans`, `CblasTrans`, `CblasConjTrans`. When *Uplo* is `CblasUpper` then the upper triangle of *A* is used, and when *Uplo* is `CblasLower` then the lower triangle of *A* is used. If *Diag* is `CblasNonUnit` then the diagonal of the matrix is used, but if *Diag* is `CblasUnit` then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

Function: int **gsl_blas_ssymv** *(CBLAS_UPLO_t Uplo, float alpha, const gsl_matrix_float * A, const*

*gsl_vector_float * x, float beta, gsl_vector_float * y)*
Function: int **gsl_blas_dsymv** *(CBLAS_UPLO_t Uplo, double alpha, const gsl_matrix * A, const gsl_vector * x, double beta, gsl_vector * y)*

These functions compute the matrix-vector product and sum y = \alpha A x + \beta y for the symmetric matrix *A*. Since the matrix *A* is symmetric only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *A* are used.

Function: int **gsl_blas_chemv** *(CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_vector_complex_float * x, const gsl_complex_float beta, gsl_vector_complex_float * y)*
Function: int **gsl_blas_zhemv** *(CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_vector_complex * x, const gsl_complex beta, gsl_vector_complex * y)*

These functions compute the matrix-vector product and sum y = \alpha A x + \beta y for the hermitian matrix *A*. Since the matrix *A* is hermitian only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *A* are used. The imaginary elements of the diagonal are automatically assumed to be zero and are not referenced.

Function: int **gsl_blas_sger** *(float alpha, const gsl_vector_float * x, const gsl_vector_float * y, gsl_matrix_float * A)*
Function: int **gsl_blas_dger** *(double alpha, const gsl_vector * x, const gsl_vector * y, gsl_matrix * A)*
Function: int **gsl_blas_cgeru** *(const gsl_complex_float alpha, const gsl_vector_complex_float * x, const gsl_vector_complex_float * y, gsl_matrix_complex_float * A)*
Function: int **gsl_blas_zgeru** *(const gsl_complex alpha, const gsl_vector_complex * x, const gsl_vector_complex * y, gsl_matrix_complex * A)*

These functions compute the rank-1 update A = \alpha x y^T + A of the matrix *A*.

Function: int **gsl_blas_cgerc** *(const gsl_complex_float alpha, const gsl_vector_complex_float * x, const gsl_vector_complex_float * y, gsl_matrix_complex_float * A)*
Function: int **gsl_blas_zgerc** *(const gsl_complex alpha, const gsl_vector_complex * x, const gsl_vector_complex * y, gsl_matrix_complex * A)*

These functions compute the conjugate rank-1 update A = \alpha x y^H + A of the matrix *A*.

Function: int **gsl_blas_ssyr** *(CBLAS_UPLO_t Uplo, float alpha, const gsl_vector_float * x, gsl_matrix_float * A)*
Function: int **gsl_blas_dsyr** *(CBLAS_UPLO_t Uplo, double alpha, const gsl_vector * x, gsl_matrix * A)*

These functions compute the symmetric rank-1 update A = \alpha x x^T + A of the symmetric matrix *A*. Since the matrix *A* is symmetric only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *A* are used.

Function: int **gsl_blas_cher** *(CBLAS_UPLO_t Uplo, float alpha, const gsl_vector_complex_float * x, gsl_matrix_complex_float * A)*
Function: int **gsl_blas_zher** *(CBLAS_UPLO_t Uplo, double alpha, const gsl_vector_complex * x, gsl_matrix_complex * A)*

These functions compute the hermitian rank-1 update A = \alpha x x^H + A of the hermitian matrix *A*. Since the matrix *A* is hermitian only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then

the lower triangle and diagonal of *A* are used. The imaginary elements of the diagonal are automatically set to zero.

Function: int **gsl_blas_ssyr2** *(CBLAS_UPLO_t Uplo, float alpha, const gsl_vector_float * x, const gsl_vector_float * y, gsl_matrix_float * A)*
Function: int **gsl_blas_dsyr2** *(CBLAS_UPLO_t Uplo, double alpha, const gsl_vector * x, const gsl_vector * y, gsl_matrix * A)*

These functions compute the symmetric rank-2 update A = \alpha x y^T + \alpha y x^T + A of the symmetric matrix *A*. Since the matrix *A* is symmetric only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *A* are used.

Function: int **gsl_blas_cher2** *(CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_vector_complex_float * x, const gsl_vector_complex_float * y, gsl_matrix_complex_float * A)*
Function: int **gsl_blas_zher2** *(CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_vector_complex * x, const gsl_vector_complex * y, gsl_matrix_complex * A)*

These functions compute the hermitian rank-2 update A = \alpha x y^H + \alpha^* y x^H A of the hermitian matrix *A*. Since the matrix *A* is hermitian only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *A* are used. The imaginary elements of the diagonal are automatically set to zero.

Level 3

Function: int **gsl_blas_sgemm** *(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, float alpha, const gsl_matrix_float * A, const gsl_matrix_float * B, float beta, gsl_matrix_float * C)*
Function: int **gsl_blas_dgemm** *(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, double alpha, const gsl_matrix * A, const gsl_matrix * B, double beta, gsl_matrix * C)*
Function: int **gsl_blas_cgemm** *(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_matrix_complex_float * B, const gsl_complex_float beta, gsl_matrix_complex_float * C)*
Function: int **gsl_blas_zgemm** *(CBLAS_TRANSPOSE_t TransA, CBLAS_TRANSPOSE_t TransB, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_matrix_complex * B, const gsl_complex beta, gsl_matrix_complex * C)*

These functions compute the matrix-matrix product and sum C = \alpha op(A) op(B) + \beta C where op(A) = A, A^T, A^H for *TransA* = `CblasNoTrans`, `CblasTrans`, `CblasConjTrans` and similarly for the parameter *TransB*.

Function: int **gsl_blas_ssymm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, float alpha, const gsl_matrix_float * A, const gsl_matrix_float * B, float beta, gsl_matrix_float * C)*
Function: int **gsl_blas_dsymm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, double alpha, const gsl_matrix * A, const gsl_matrix * B, double beta, gsl_matrix * C)*
Function: int **gsl_blas_csymm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_matrix_complex_float * B, const gsl_complex_float beta, gsl_matrix_complex_float * C)*
Function: int **gsl_blas_zsymm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_matrix_complex * B, const gsl_complex beta, gsl_matrix_complex * C)*

These functions compute the matrix-matrix product and sum C = \alpha A B + \beta C for *Side* is `CblasLeft` and C = \alpha B A + \beta C for *Side* is `CblasRight`, where the matrix *A* is symmetric.

When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *A* are used.

Function: int **gsl_blas_chemm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_matrix_complex_float * B, const gsl_complex_float beta, gsl_matrix_complex_float * C)*
Function: int **gsl_blas_zhemm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_matrix_complex * B, const gsl_complex beta, gsl_matrix_complex * C)*

These functions compute the matrix-matrix product and sum C = \alpha A B + \beta C for *Side* is `CblasLeft` and C = \alpha B A + \beta C for *Side* is `CblasRight`, where the matrix *A* is hermitian. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *A* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *A* are used. The imaginary elements of the diagonal are automatically set to zero.

Function: int **gsl_blas_strmm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, float alpha, const gsl_matrix_float * A, gsl_matrix_float * B)*
Function: int **gsl_blas_dtrmm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, double alpha, const gsl_matrix * A, gsl_matrix * B)*
Function: int **gsl_blas_ctrmm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, gsl_matrix_complex_float * B)*
Function: int **gsl_blas_ztrmm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex alpha, const gsl_matrix_complex * A, gsl_matrix_complex * B)*

These functions compute the matrix-matrix product B = \alpha op(A) B for *Side* is `CblasLeft` and B = \alpha B op(A) for *Side* is `CblasRight`. The matrix *A* is triangular and op(A) = A, A^T, A^H for *TransA* = `CblasNoTrans`, `CblasTrans`, `CblasConjTrans` When *Uplo* is `CblasUpper` then the upper triangle of *A* is used, and when *Uplo* is `CblasLower` then the lower triangle of *A* is used. If *Diag* is `CblasNonUnit` then the diagonal of *A* is used, but if *Diag* is `CblasUnit` then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

Function: int **gsl_blas_strsm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, float alpha, const gsl_matrix_float * A, gsl_matrix_float * B)*
Function: int **gsl_blas_dtrsm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, double alpha, const gsl_matrix * A, gsl_matrix * B)*
Function: int **gsl_blas_ctrsm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, gsl_matrix_complex_float * B)*
Function: int **gsl_blas_ztrsm** *(CBLAS_SIDE_t Side, CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t TransA, CBLAS_DIAG_t Diag, const gsl_complex alpha, const gsl_matrix_complex * A, gsl_matrix_complex * B)*

These functions compute the matrix-matrix product B = \alpha op(inv(A)) B for *Side* is `CblasLeft` and B = \alpha B op(inv(A)) for *Side* is `CblasRight`. The matrix *A* is triangular and op(A) = A, A^T, A^H for *TransA* = `CblasNoTrans`, `CblasTrans`, `CblasConjTrans` When *Uplo* is `CblasUpper` then the upper triangle of *A* is used, and when *Uplo* is `CblasLower` then the lower triangle of *A* is used. If *Diag* is `CblasNonUnit` then the diagonal of *A* is used, but if *Diag* is `CblasUnit` then the diagonal elements of the matrix *A* are taken as unity and are not referenced.

Function: int **gsl_blas_ssyrk** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_float * A, float beta, gsl_matrix_float * C)*
Function: int **gsl_blas_dsyrk** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix * A, double beta, gsl_matrix * C)*
Function: int **gsl_blas_csyrk** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_complex_float beta, gsl_matrix_complex_float * C)*
Function: int **gsl_blas_zsyrk** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_complex beta, gsl_matrix_complex * C)*

These functions compute a rank-k update of the symmetric matrix *C*, C = \alpha A A^T + \beta C when *Trans* is `CblasNoTrans` and C = \alpha A^T A + \beta C when *Trans* is `CblasTrans`. Since the matrix *C* is symmetric only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *C* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *C* are used.

Function: int **gsl_blas_cherk** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_complex_float * A, float beta, gsl_matrix_complex_float * C)*
Function: int **gsl_blas_zherk** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix_complex * A, double beta, gsl_matrix_complex * C)*

These functions compute a rank-k update of the hermitian matrix *C*, C = \alpha A A^H + \beta C when *Trans* is `CblasNoTrans` and C = \alpha A^H A + \beta C when *Trans* is `CblasTrans`. Since the matrix *C* is hermitian only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *C* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *C* are used. The imaginary elements of the diagonal are automatically set to zero.

Function: int **gsl_blas_ssyr2k** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, float alpha, const gsl_matrix_float * A, const gsl_matrix_float * B, float beta, gsl_matrix_float * C)*
Function: int **gsl_blas_dsyr2k** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, double alpha, const gsl_matrix * A, const gsl_matrix * B, double beta, gsl_matrix * C)*
Function: int **gsl_blas_csyr2k** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_matrix_complex_float * B, const gsl_complex_float beta, gsl_matrix_complex_float * C)*
Function: int **gsl_blas_zsyr2k** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_matrix_complex * B, const gsl_complex beta, gsl_matrix_complex *C)*

These functions compute a rank-2k update of the symmetric matrix *C*, C = \alpha A B^T + \alpha B A^T + \beta C when *Trans* is `CblasNoTrans` and C = \alpha A^T B + \alpha B^T A + \beta C when *Trans* is `CblasTrans`. Since the matrix *C* is symmetric only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *C* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *C* are used.

Function: int **gsl_blas_cher2k** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex_float alpha, const gsl_matrix_complex_float * A, const gsl_matrix_complex_float * B, float beta, gsl_matrix_complex_float * C)*
Function: int **gsl_blas_zher2k** *(CBLAS_UPLO_t Uplo, CBLAS_TRANSPOSE_t Trans, const gsl_complex alpha, const gsl_matrix_complex * A, const gsl_matrix_complex * B, double beta, gsl_matrix_complex * C)*

These functions compute a rank-2k update of the hermitian matrix *C*, C = \alpha A B^H + \alpha^* B A^H + \beta C when *Trans* is `CblasNoTrans` and C = \alpha A^H B + \alpha^* B^H A + \beta C when

Trans is `CblasTrans`. Since the matrix *C* is hermitian only its upper half or lower half need to be stored. When *Uplo* is `CblasUpper` then the upper triangle and diagonal of *C* are used, and when *Uplo* is `CblasLower` then the lower triangle and diagonal of *C* are used. The imaginary elements of the diagonal are automatically set to zero.

Examples

The following program computes the product of two matrices using the Level-3 BLAS function DGEMM,

```
[ 0.11 0.12 0.13 ]  [ 1011 1012 ]     [ 367.76 368.12 ]
[ 0.21 0.22 0.23 ]  [ 1021 1022 ]  =  [ 674.06 674.72 ]
                    [ 1031 1032 ]
```

The matrices are stored in row major order, according to the C convention for arrays.

```
#include <stdio.h>
#include <gsl/gsl_blas.h>

int
main (void)
{
  double a[] = { 0.11, 0.12, 0.13,
                 0.21, 0.22, 0.23 };

  double b[] = { 1011, 1012,
                 1021, 1022,
                 1031, 1032 };

  double c[] = { 0.00, 0.00,
                 0.00, 0.00 };

  gsl_matrix_view A = gsl_matrix_view_array(a, 2, 3);
  gsl_matrix_view B = gsl_matrix_view_array(b, 3, 2);
  gsl_matrix_view C = gsl_matrix_view_array(c, 2, 2);

  /* Compute C = A B */

  gsl_blas_dgemm (CblasNoTrans, CblasNoTrans,
                  1.0, &A.matrix, &B.matrix,
                  0.0, &C.matrix);

  printf("[ %g, %g\n", c[0], c[1]);
  printf("  %g, %g ]\n", c[2], c[3]);

  return 0;
}
```

Here is the output from the program,

```
$ ./a.out
[ 367.76, 368.12
  674.06, 674.72 ]
```

References and Further Reading

Information on the BLAS standards, including both the legacy and draft interface standards, is available online from the BLAS Homepage and BLAS Technical Forum web-site.

- *BLAS Homepage* http://www.netlib.org/blas/
- *BLAS Technical Forum* http://www.netlib.org/cgi-bin/checkout/blast/blast.pl

The following papers contain the specifications for Level 1, Level 2 and Level 3 BLAS.

- C. Lawson, R. Hanson, D. Kincaid, F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage", *ACM Transactions on Mathematical Software*, Vol. 5 (1979), Pages 308-325.
- J.J. Dongarra, J. DuCroz, S. Hammarling, R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, Vol. 14, No. 1 (1988), Pages 1-32.
- J.J. Dongarra, I. Duff, J. DuCroz, S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, Vol. 16 (1990), Pages 1-28.

Postscript versions of the latter two papers are available from http://www.netlib.org/blas/. A CBLAS wrapper for Fortran BLAS libraries is available from the same location.

Go to the first, previous, next, last section, table of contents.