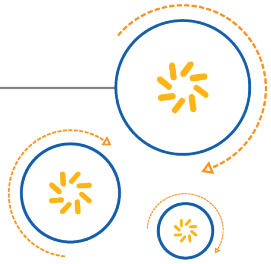
Qualcomm Technologies, Inc.



QML Documentation

Release 1.4.0

Qualcomm Technologies, Inc

January 15, 2020

I Introduction to BLAS **2**

1 History **3**

2 Levels **4**

3 Naming Conventions **5**
3.1 Precision . . . 5
3.2 Matrix storage . . . 5

4 Fortran77 Interface **6**
4.1 Options . . . 6
4.2 Column-Major . . . 6
4.3 One-based Indexing . . . 7
4.4 Example . . . 7

5 C Interface **8**
5.1 Passing arguments . . . 8
5.2 Aliasing . . . 8
5.3 Row- or Column-Major . . . 8
5.4 Zero-based Indexing . . . 9
5.5 Performance Considerations . . . 9
5.6 Example . . . 9

6 Storage Format Examples **10**
6.1 GE . . . 10
6.2 SY . . . 10
6.3 HE . . . 10
6.4 TR . . . 11
6.5 GB . . . 11
6.6 SB . . . 11
6.7 HB . . . 11
6.8 TB . . . 11
6.9 SP . . . 12
6.10 HP . . . 12
6.11 TP . . . 12

7 References **13**

II Introduction to LAPACK 14

8 History 15

9 Conventions 16

10 Interface 17
10.1 Function Pointers . . . 17
10.2 Booleans . . . 17

11 Performance Considerations 18

12 References 19

III Using QML 20

13 QML 21
13.1 OpenMP . . . 21
13.2 QML Directory Structure . . . 21
13.3 ILP64 and LP64 . . . 21
13.4 Using QML on Android . . . 22
13.5 Environment Variables . . . 22
13.6 Example (Linux x86) . . . 22
13.7 Additional Information . . . 22
13.8 Questions and Problems . . . 22

14 Example Programs 23
14.1 Minimal Example . . . 23
14.2 CBLAS Example . . . 24
14.3 BLAS Solve Example . . . 25
14.4 LAPACK Least Squares Example . . . 26
14.5 LAPACK Singular Value Decomposition . . . 29
14.6 Building the Examples . . . 30
14.7 Running the Examples . . . 31

IV QML API Reference 32

15 BLAS 1 Functions 33
15.1 (IS|ID|IC|IZ)AMAX . . . 33
15.2 (IS|ID|IC|IZ)AMIN . . . 34
15.3 (S|D|SC|DZ)ASUM . . . 34
15.4 (S|D|C|Z)AXPBY . . . 35
15.5 (S|D|C|Z)AXPY . . . 36
15.6 (S|D)CABS1 . . . 37
15.7 (S|D|C|Z)COPY . . . 38
15.8 (S|D)DOT . . . 39
15.9 (C|Z)DOTC . . . 40
15.10 (C|Z)DOTU . . . 41
15.11 (S|D|SC|DZ)NRM2 . . . 41
15.12 (S|D|C|CS|Z|ZD)ROT . . . 42
15.13 (S|D|C|Z)ROTG . . . 44
15.14 (S|D)ROTM . . . 45
15.15 (S|D)ROTMG . . . 46

15.16 (S|D|C|Z|CS|ZD)SCAL . . . 47
15.17 (D)SDOT . . . 48
15.18 (SD)SDOT . . . 49
15.19 (S|D|C|Z)SWAP . . . 49

16 BLAS 2 Functions **51**
16.1 (S|D|C|Z)GBMV . . . 51
16.2 (S|D|C|Z)GEMV . . . 53
16.3 (S|D)GER . . . 54
16.4 (C|Z)GERC . . . 55
16.5 (C|Z)GERU . . . 56
16.6 (C|Z)HBMV . . . 57
16.7 (C|Z)HEMV . . . 58
16.8 (C|Z)HER . . . 59
16.9 (C|Z)HER2 . . . 60
16.10 (C|Z)HPMV . . . 61
16.11 (C|Z)HPR . . . 62
16.12 (C|Z)HPR2 . . . 63
16.13 (S|D)SBMV . . . 63
16.14 (S|D)SPMV . . . 64
16.15 (S|D)SPR . . . 65
16.16 (S|D)SPR2 . . . 66
16.17 (S|D)SYMV . . . 67
16.18 (S|D)SYR . . . 68
16.19 (S|D)SYR2 . . . 69
16.20 (S|D|C|Z)TBMV . . . 69
16.21 (S|D|C|Z)TBSV . . . 71
16.22 (S|D|C|Z)TPMV . . . 72
16.23 (S|D|C|Z)TPSV . . . 73
16.24 (S|D|C|Z)TRMV . . . 75
16.25 (S|D|C|Z)TRSV . . . 76

17 BLAS 3 Functions **78**
17.1 (S|D|C|Z)GEMM . . . 78
17.2 (C|Z)HEMM . . . 80
17.3 (C|Z)HERK . . . 81
17.4 (C|Z)HER2K . . . 82
17.5 (S|D|C|Z)SYMM . . . 83
17.6 (S|D|C|Z)SYRK . . . 85
17.7 (S|D|C|Z)SYR2K . . . 86
17.8 (S|D|C|Z)TRMM . . . 88
17.9 (S|D|C|Z)TRSM . . . 90

18 LAPACK Functions **92**
18.1 (S|D|C|Z)BDSQR . . . 92
18.2 (S|D|C|Z)GEBAK . . . 93
18.3 (S|D|C|Z)GEBAL . . . 94
18.4 (S|D|C|Z)GEES . . . 95
18.5 (S|D|C|Z)GEEV . . . 96
18.6 (S|D|C|Z)GEHRD . . . 97
18.7 (S|D|C|Z)GELS . . . 98
18.8 (S|D|C|Z)GELSD . . . 100
18.9 (S|D|C|Z)GEQR2 . . . 101
18.10 (S|D|C|Z)GEQRF . . . 102

18.11 (S|D|C|Z)GESDD . . . 103
18.12 (S|D|C|Z)GESV . . . 104
18.13 (S|D|C|Z)GESVD . . . 105
18.14 (S|D|C|Z)GESVX . . . 106
18.15 (S|D|C|Z)GETF2 . . . 107
18.16 (S|D|C|Z)GETRF . . . 108
18.17 (S|D|C|Z)GETRI . . . 109
18.18 (S|D|C|Z)GETRS . . . 110
18.19 (S|D|C|Z)GGES . . . 111
18.20 (S|D|C|Z)GGEV . . . 112
18.21 (C|Z)HEEV . . . 114
18.22 (C|Z)HEEVD . . . 114
18.23 (C|Z)HETRD . . . 115
18.24 (S|D|C|Z)HSEQR . . . 116
18.25 ILA(S|D|C|Z)LC . . . 118
18.26 ILA(S|D|C|Z)LR . . . 118
18.27 (C|Z)LACGV . . . 119
18.28 (S|D|C|Z)LACPY . . . 119
18.29 (S|D)LADIV . . . 120
18.30 (C|Z)LADIV . . . 121
18.31 (S|D|C|Z)LANGE . . . 121
18.32 (S|D)LAPY2 . . . 122
18.33 (S|D)LAPY3 . . . 122
18.34 (S|D|C|Z)LARF . . . 123
18.35 (S|D|C|Z)LARFB . . . 124
18.36 (S|D|C|Z)LARFG . . . 125
18.37 (S|D|C|Z)LARTG . . . 126
18.38 (S|D|C|Z)LARFT . . . 127
18.39 (S|D)LAS2 . . . 128
18.40 (S|D|C|Z)LASCL . . . 128
18.41 (S|D|C|Z)LASET . . . 129
18.42 (S|D)LASQ1 . . . 130
18.43 (S|D|C|Z)LASR . . . 130
18.44 (S|D)LASRT . . . 132
18.45 (S|D|C|Z)LASSQ . . . 133
18.46 (S|D)LASV2 . . . 133
18.47 (S|D)ORGQR . . . 134
18.48 (S|D)ORGTR . . . 135
18.49 (S|D|C|Z)POSV . . . 136
18.50 (S|D|C|Z)POTF2 . . . 137
18.51 (S|D|C|Z)POTRF . . . 138
18.52 (S|D|C|Z)POTRI . . . 138
18.53 (S|D|C|Z)POTRS . . . 139
18.54 (S|D|C|Z)STEDC . . . 140
18.55 (S|D|C|Z)STEQR . . . 141
18.56 (S|D)SYEV . . . 142
18.57 (S|D)SYEVD . . . 143
18.58 (S|D)SYTRD . . . 143
18.59 (S|D|C|Z)SYTRF . . . 144
18.60 (S|D|C|Z)SYTRI . . . 145
18.61 (S|D|C|Z)TREVC . . . 146
18.62 (S|D|C|Z)TRSYL . . . 147
18.63 (S|D|C|Z)TRTI2 . . . 149
18.64 (S|D|C|Z)TRTRI . . . 150

18.65 (C|Z)UNGHR 150
18.66 (C|Z)UNGQR 151
18.67 (C|Z)UNGTR 152

19 QML Extensions **154**
19.1 (S)CONV_MM 154
19.2 I32I8CONV 155

20 QML Utility Functions **157**
20.1 QMLVersionInfo 157
20.2 QMLVersionString 157
20.3 QMLGetNumThreads 158
20.4 QMLSetNumThreads 158
20.5 QML_IS_SUPPORTED 159

QML is a high performance BLAS and LAPACK implementation for Qualcomm processors. The original BLAS reference and additional documentation can be found at netlib.org.

Part I

Introduction to BLAS

CHAPTER

ONE

HISTORY

The Basic Linear Algebra Subprograms (BLAS) are a set of functions designed to standardize and simplify numerical programming in an efficient and portable way. They are commonly used as building blocks in the design of higher-level linear algebra routines that demand performance and correctness.

The original version of BLAS, now called the *reference implementation*, was developed in Fortran77 starting in 1979 and has been updated with improvements and fixes up to the present. Highly optimized implementations of BLAS tuned for specific machine architectures are available for most development environments. These *optimized implementations* typically follow the same naming scheme and calling conventions as the reference implementation. Many optimized implementations also provide additional functions and options.

QML contains a high performance implementation of BLAS optimized for Qualcomm processors.

CHAPTER

TWO

LEVELS

Functions in BLAS are organized in three levels.

Level	Operations	Examples
BLAS-1	vector	dot product scale a vector
BLAS-2	matrix-vector	matrix-vector product solve a linear system of equations
BLAS-3	matrix-matrix	matrix-matrix multiply solve a linear system of equations with multiple right-hand sides

The BLAS level indicates how many levels of nested loops are required to perform the operation. Scaling a vector requires a single loop iterating over the data. Computing a matrix-matrix product uses three nested loops (in the simplest implementation). The best performance is usually achieved by calling the highest level BLAS function possible to give the library the most oportunity for optimization.

CHAPTER

THREE

NAMING CONVENTIONS

Precision

Many functions can work with several types of data. Function names are prefixed with the precision of the data they support. For example, the function `dgemm` works with double precision floating point values.

Prefix	Precision
S	single
D	double
C	single complex
Z	double complex

Matrix storage

BLAS can operate on matrices that are stored in several different dense formats. After the precision prefix is a secondary prefix that indicates the storage format of the matrix.

Prefix	Matrix type
GE	general
SY	symmetric
HE	hermitian
TR	triangular
GB	general banded
SB	symmetric banded
HB	hermitian banded
TB	triangle banded
SP	symmetric packed
HP	hermitian packed
TP	triangle packed

CHAPTER

FOUR

FORTRAN77 INTERFACE

Because the reference BLAS implementation is written in Fortran77 the interface to BLAS library functions follows Fortran77 conventions. The most important difference from C is that all arguments to functions are passed by reference. In the C function prototypes this means every argument is a pointer, even simple scalar values. Functions that return complex values use a `void` return type and instead put their output into a `result` pointer.

Options

Many functions need to be given information such as whether to use an untransposed or transposed version of a matrix, or whether a triangular matrix is upper or lower triangular. In the Fortran77 interface these types of arguments are passed as character strings. Only the first character in the string is used and is interpreted using the following table.

Option	Character	Interpretation
TRANSx	N	Non-transpose
	T	Transpose
	C	Conjugate transpose
UPLO	U	Upper triangular
	L	Lower triangular
DIAG	N	Non-unit diagonal
	U	Unit diagonal
SIDE	L	Left side
	R	Right side

Column-Major

Following Fortran, all matrices are assumed to be stored in column-major order when using the Fortran77 interface. This means that values linearly stored in increasing addresses in memory are interpreted as values in a column of the matrix going from top to bottom. Every matrix also has a leading dimension. For a matrix A the leading dimension is passed as the integer LDA. The values of a row of matrix A are stored in memory locations spaced apart by LDA values.

For example, a 3x3 matrix with LDA=10 in column-major order has values stored in the following memory locations.

$$\begin{bmatrix} A & A+10 & A+20 \\ A+1 & A+11 & A+21 \\ A+2 & A+12 & A+22 \end{bmatrix}$$

One-based Indexing

The functions AMAX and AMIN return indices into arrays. Fortran uses 1-based indexing where an array with N elements has indices 1 through N. The Fortran BLAS interface follows the Fortran convention and uses 1-based indexing.

Example

For example, suppose we wish to perform a simple matrix multiply:

$$C \leftarrow A * B$$

Here is the Fortran77 prototype for `dgemm`:

```
void dgemm(const char *TRANSA, const char *TRANSB, const qml_long *M,
           const qml_long *N, const qml_long *K, const double *ALPHA,
           const double *A, const qml_long *LDA, const double *B,
           const qml_long *LDB, const double *BETA, double *C,
           const qml_long *LDC);
```

The funtion `dgemm` is a double precision (`d`) general (`ge`) matrix-matrix multiply. We do not wish to transpose `A` or `B`, so we pass `"N"` for `TRANSA` and `TRANSB`. The function actually performs the operation:

$$C \leftarrow \alpha A * B + \beta C$$

We choose $\alpha = 1$ and $\beta = 0$. Even though `ALPHA` and `BETA` are scalar values, we still need to pass the address of the values to use the Fortran interface.

CHAPTER

FIVE

C INTERFACE

CBLAS is a lightweight wrapper around BLAS that provides an interface compatible with the C language. CBLAS functions are prefixed with `cblas_` but otherwise follow the same naming convention as BLAS.

Passing arguments

In the C interface, vectors and matrices are passed by pointer while non-complex scalar values are passed by value. All complex arguments including scalar values are passed by reference. Functions that would return complex values are recast as returning `void`, have a final extra pointer argument `result` that receives the result value, and are renamed to include the suffix `_sub`.

Options such as non-transpose/transpose are passed by value using an enumerated type defined in the `qml_cblas.h` header.

```
enum CBLAS_TRANSPOSE {CblasNoTrans = 111, CblasTrans = 112, CblasConjTrans = 113};
enum CBLAS_UPLO {CblasUpper = 121, CblasLower = 122};
enum CBLAS_DIAG {CblasNonUnit = 131, CblasUnit = 132};
enum CBLAS_SIDE {CblasLeft = 141, CblasRight = 142};
```

Aliasing

Pointers in the C language are generally allowed to alias to the same region of memory. The C interface to BLAS allows aliasing of arguments declared `const` (input arguments) but does not allow aliasing of output arguments (any argument not declared `const`). For example, passing `dgemm` the same memory location for `A` and `B` is allowed because both arguments are input arguments. Passing `dgemm` the same memory location for `A` and `C` is not allowed because `C` is an output argument.

Row- or Column-Major

The CBLAS interface allows either row- or column-major matrix storage order. Functions that accept matrices start with an argument that indicates whether row- or column-major ordering should be used.

```
enum CBLAS_ORDER {CblasRowMajor = 101, CblasColMajor = 102};
```

Row-major ordering means that values linearly stored in increasing addresses in memory are interpreted as values in a row of the matrix going from left to right. Every matrix also has a leading dimension. For a matrix A the leading dimension is passed as the integer LDA. The values of a column of matrix A are stored in memory locations spaced apart by LDA values.

For example, a 3x3 matrix with LDA=10 in row-major order has values stored in the following memory locations.

$$\begin{bmatrix} A & A+1 & A+2 \\ A+10 & A+11 & A+12 \\ A+20 & A+21 & A+22 \end{bmatrix}$$

Zero-based Indexing

The functions AMAX and AMIN return indices into arrays. C uses 0-based indexing where an array with N elements has indices 0 through N-1. The CBLAS interface follows the C convention and uses 0-based indexing.

Performance Considerations

Using row-major ordering has no extra cost for most BLAS functions. Some BLAS-2 functions involving conjugate transpose require a small amount of extra processing and so will see degraded performance compared to column-major ordering. See Legacy BLAS: C Interface to the Legacy BLAS for more information.

Example

For example, suppose we wish to perform a simple matrix multiply:

$$C \leftarrow A * B$$

Here is the CBLAS prototype for `dgemm`:

```
void cblas_dgemm(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_TRANSPOSE TRANSB, const qml_long M,
                 const qml_long N, const qml_long K, const double ALPHA,
                 const double *A, const qml_long LDA, const double *B,
                 const qml_long LDB, const double BETA, double *C,
                 const qml_long LDC);
```

The funtion `cblas_dgemm` is the CBLAS interface (`cblas_`) for a double precision (`d`) general (`ge`) matrix-matrix multiply. We are storing our matrices in row-major order, so we choose `CblasRowMajor` for `ORDER`. We do not wish to transpose `A` or `B`, so we pass `CblasNoTrans` for `TRANSA` and `TRANSB`. The function actually performs the operation:

$$C \leftarrow \alpha A * B + \beta C$$

We choose $\alpha = 1$ and $\beta = 0$, and pass the value 1.0 for `ALPHA` and 0.0 for `BETA`.

CHAPTER

SIX

STORAGE FORMAT EXAMPLES

The following examples show how values are laid out in memory using the different storage formats. The examples are shown for a 4x4 matrix in column-major ordering with LDA=10.

Note that in many formats not all input values are referenced. For example, in SY format with UPLO="L" only the lower triangular part of the matrix is referenced, the upper triangular part is assumed to be symmetric. In some formats elements of the matrix are assumed to be a constant value and are not referenced in memory. For example, in TR format with DIAG="U" the diagonal elements are assumed to be 1 and are never referenced.

GE

General.

$$\begin{bmatrix} A & A+10 & A+20 & A+30 \\ A+1 & A+11 & A+21 & A+31 \\ A+2 & A+12 & A+22 & A+32 \\ A+3 & A+13 & A+23 & A+33 \end{bmatrix}$$

SY

Symmetric, shown with UPLO="L".

$$\begin{bmatrix} A & A+1 & A+2 & A+3 \\ A+1 & A+11 & A+12 & A+13 \\ A+2 & A+12 & A+22 & A+23 \\ A+3 & A+13 & A+23 & A+33 \end{bmatrix}$$

HE

Hermitian, shown with UPLO="L".

$$\begin{bmatrix} A & \overline{A+1} & \overline{A+2} & \overline{A+3} \\ A+1 & A+11 & \overline{A+12} & \overline{A+13} \\ A+2 & A+12 & A+22 & \overline{A+23} \\ A+3 & A+13 & A+23 & A+33 \end{bmatrix}$$

TR

Triangular, shown with UPLO="L" and DIAG="U".

$$\begin{bmatrix} 1 & & & \\ A+1 & 1 & & \\ A+2 & A+12 & 1 & \\ A+3 & A+13 & A+23 & 1 \end{bmatrix}$$

GB

General banded, shown with KL = 1 and KU=1.

$$\begin{bmatrix} A+1 & A+10 & & \\ A+2 & A+11 & A+20 & \\ & A+12 & A+21 & A+30 \\ & & A+22 & A+31 \end{bmatrix}$$

SB

Symmetric banded, shown with UPLO="L" and K=1.

$$\begin{bmatrix} A & A+1 & & \\ A+1 & A+10 & A+11 & \\ & A+11 & A+20 & A+21 \\ & & A+21 & A+30 \end{bmatrix}$$

HB

Hermitian banded, shown with UPLO="L" and K=1.

$$\begin{bmatrix} A & \overline{A+1} & & \\ A+1 & A+10 & \overline{A+11} & \\ & A+11 & A+20 & \overline{A+21} \\ & & A+21 & A+30 \end{bmatrix}$$

TB

Triangular banded, shown with UPLO="L", DIAG="U", and K = 2.

$$\begin{bmatrix} 1 & & & \\ A+1 & 1 & & \\ A+2 & A+11 & 1 & \\ & A+12 & A+21 & 1 \end{bmatrix}$$

SP

Symmetric packed, shown with UPLO="L".

$$\begin{bmatrix} A & A+1 & A+2 & A+3 \\ A+1 & A+4 & A+5 & A+6 \\ A+2 & A+5 & A+7 & A+8 \\ A+3 & A+6 & A+8 & A+9 \end{bmatrix}$$

HP

Hermitian packed, shown with UPLO="L".

$$\begin{bmatrix} A & \overline{A+1} & \overline{A+2} & \overline{A+3} \\ A+1 & A+4 & \overline{A+5} & \overline{A+6} \\ A+2 & A+5 & A+7 & \overline{A+8} \\ A+3 & A+6 & A+8 & A+9 \end{bmatrix}$$

TP

Triangular packed, shown with UPLO="L" and DIAG="U".

$$\begin{bmatrix} 1 & & & \\ A+1 & 1 & & \\ A+2 & A+5 & 1 & \\ A+3 & A+6 & A+8 & 1 \end{bmatrix}$$

CHAPTER

SEVEN

REFERENCES

The official homepage of BLAS is on Netlib.

Netlib also publishes a handy cheat sheet for quick reference to all BLAS functions.

Part II

Introduction to LAPACK

CHAPTER

EIGHT

HISTORY

LAPACK is a library of functions for dealing with dense and banded matrices. It includes functions for solving systems of equations, linear least-squares, eigenvalue problems, singular value problems, and matrix factorizations such as QR, LU, Cholesky, SVD, and Schur forms. Functions in LAPACK are designed to call optimized BLAS functions whenever possible for maximum performance.

The full LAPACK library includes over 1700 functions and remains under active development. For compatibility purposes QML contains implementations of all public LAPACK functions with limitations and caveats discussed below. A subset of higher-level functions is documented in the API Reference. Documentation for other functions can be found on Netlib.

CHAPTER

NINE

CONVENTIONS

Most functions in LAPACK are named following a similar convention to BLAS. The first letter indicates the numerical precision of the computation and is one of *SDCZ*. The next two letters indicate the storage format of the main matrices involved in the routine. Choices include all the formats supported by BLAS along with many more specialized formats. The final three letters indicate the operation to be performed. For example, *DGEQRF* means a double precision computation (*D*) on a general dense matrix (*GE*) performing a QR factorization (*QRF*).

CHAPTER

TEN

INTERFACE

The LAPACK interface follows the same conventions as the *BLAS interface*. As in the BLAS interface, the interface to LAPACK library functions follows Fortran conventions. All arguments to functions are passed by reference. In the C function prototypes this means every argument is a pointer, even simple scalar values. Functions that return complex values use a `void` return type and instead put their output into a `result` pointer.

As in the BLAS interface, matrices are stored in column-major order and indexing is 1-based. No aliasing of pointers is allowed in the LAPACK interface.

Function Pointers

Some LAPACK functions accept functions as arguments. A typical example is *DGEES* which accepts an argument *SELECT* that decides which eigenvalues should be sorted in the output. QML does not currently support function pointers. When interfacing to these functions using C, the type used is *void* *. Any values passed as function pointers are ignored and never called.

Booleans

Many functions accept or return Boolean values or arrays. In the C interface to these functions the type used is *qml_int*. This type is 4 bytes on 32-bit platforms and 8 bytes on 64-bit platforms. This type usually corresponds to the type *LOGICAL* and *INTEGER* in Fortran compilers.

CHAPTER

ELEVEN

PERFORMANCE CONSIDERATIONS

Not all LAPACK functions provided by QML are highly optimized. For best performance limit calls to the functions documented in this guide. Many of the problems solved by LAPACK have significant sequential sections that cannot be easily parallelized so performance will necessarily be limited for these problems.

CHAPTER

TWELVE

REFERENCES

The official homepage of LAPACK is on Netlib. The most convenient way to browse function documentation is through the online explorer.

Part III

Using QML

CHAPTER

THIRTEEN

QML

OpenMP

QML uses the OpenMP runtime for parallelism. More information about OpenMP can be found at http://openmp.org/wp/. In particular, OpenMP environment variables will affect the behavior of QML.

QML Directory Structure

The QML directory structure is organized in the following way:

`/<plat>/<arch>/<interface>/<toolchain>/`

Where possible values for each directory are:

plat: `android, linux, macos, windows`

arch: `arm32, aarch64, x64`

interface: `lp64, ilp64`

toolchain: `ndk-r10e, ndk-r11, gcc-4.x, gcc-5.x, gcc-4.9-ubuntu-12.04, msvc, clang`

Once the platform, architecture, interface, and supported toolchain are selected, the final directory will contain the follow subdirectories:

`/examples`

`/include`

`/lib`

ILP64 and LP64

The BLAS specification uses 32 bit integers for everything from the dimensions of matrices to the stride between elements in a dot product. This is sufficient for most problems and 32 bit architectures, however, certain classes of problem require more bits to indicate things like the length of the input vectors when calling AXPY or DOT. This has lead the community to support both ILP64 and LP64 even on 64 bit architectures. QML also supports both ILP64 and LP64, with the appropriate libraries either in the `ilp64` or `lp64` folder. To aid development on diverse platforms we provide the type `qml_long` in `qml_types.h` which will be 32 bits on 32 bit platforms and 64 bits on 64 bit platforms if the `ilp64` folder header is used. Otherwise if the `lp64` folder header is used, `qml_long` will be 32 bits on all platforms.

Using QML on Android

In order to run QML on Android, the appropriate QML shared library will need to be copied to the device in addition to the program that uses QML. If the device is rooted, the appropriate place for the 32-bit (armv7-a) version of QML is `/system/vendor/lib/`. The 64-bit (aarch64/armv8-a) version of QML belongs in `/system/vendor/lib64/`. Additionally, the 32-bit (armv7-a) version of QML is compiled for Android API level 19 (Android 4.4), whereas the 64-bit (aarch64/armv8-a) version of QML is compiled for Android API level 21 (Android 5.0).

Environment Variables

Three environment variables are available to control the threadpool used by QML:

`OMP_NUM_THREADS=<N>` Controls the number of threads in the OpenMP threadpool at launch time.

`QML_NUM_THREADS=<N>` Controls the degree of parallelism of QML functions, does not affect the size of the OpenMP threadpool. The number of tasks launched by QML will be the minimum of `QML_NUM_THREADS` and `OMP_NUM_THREADS`.

`QML_HET_RATIO=<little:big[:prime]>` Controls the work distribution ratio (as percentage) of parallel operations like GEMM. The first number is for the first CPU cluster (e.g. LITTLE cluster), then for the big cluster and then prime cluster (if applicable). The numbers should be integers, and sum of all numbers must be 100. In case of bad formatting, the default partitioning is used which may not be optimal for all SOCs.

On x86 based systems, over subscription can be a problem for compute bound workloads as it causes cache thrashing which destroys performance. To prevent this it is recommended to set the number of threads using one of these two environment variables to the number of physical cores present rather than the default number of reported cores.

Example (Linux x86)

Let's assume we are using an x86 based machine that supports hyperthreading. This means there are X real cores, however, the system behaves as if it has 2X cores. On modern Linux systems, the first N/2 cores are different cores, the next N/2 are duplicates of the first half. In order to avoid over subscription for compute bound workloads (matrix multiply or GEMM), we don't want to have a threadpool larger than the number of real cores on the machine. We also don't want these threads migrating in the middle of computation as that also causes cache thrashing. To avoid this we will employ both a tool that is provided on Linux as well as the environment variables used by OpenMP. For instance, let's assume the machine has 4 real cores. Then to invoke a program that calls QML GEMM we would do the following:

```
OMP_NUM_THREADS=4 taskset -c 0-3 <program>
```

Additional Information

Some additional information can be found in the release notes.

Questions and Problems

Please send any suggestions, questions, bugs, or integration issues to qml@qti.qualcomm.com.

CHAPTER

FOURTEEN

EXAMPLE PROGRAMS

All the example code can be found in the `examples/` subdirectory at the installation prefix location.

Minimal Example

This example shows the minimal code needed to call a QML function.

This example constructs matrices `A` and `B` that are both 1024x1024 filled with unit values. It then calls `sgemm` to multiply the matrices and store the 1024x1024 result in matrix `C`. The first value in the resulting matrix is displayed, which should be 1024.

```
#include <qml.h>

#include <iostream>

using namespace std;

int main()
{
    const uint32_t matrixSize = 1024*1024;

    float *A = new float[matrixSize];
    float *B = new float[matrixSize];
    float *C = new float[matrixSize];

    for(uint32_t i=0; i < matrixSize; i++)
    {
      A[i] = B[i] = C[i] = 1.0;
    }

    cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, 1024, 1024, 1024,
        1.0, A, 1024, B, 1024, 0.0, C, 1024);

    cout << "Value of C[0] is: " << C[0] << endl;

    delete[] C;
    delete[] B;
    delete[] A;

    return 0;
}
```

CBLAS Example

This example shows how row-major indexing works in CBLAS and how offsets and leading dimensions can be passed to QML functions to operate on subregions of larger matrices.

```
#include <qml.h>

#include <iostream>

/* Example program that uses the CBLAS interface to permute part of a matrix.

   This program constructs a matrix A of size 6x8 with entries:

   [  1,  2,  3,  4,  5,  6,  7,  8; ]
   [  9, 10, 11, 12, 13, 14, 15, 16; ]
   [ 17, 18, 19, 20, 21, 22, 23, 24; ]
   [ 25, 26, 27, 28, 29, 30, 31, 32; ]
   [ 33, 34, 35, 36, 37, 38, 39, 40; ]
   [ 41, 42, 43, 44, 45, 46, 47, 48; ]

   It constructs an explicit permutation matrix of size 4x4 with entries:

   [ 0, 1, 0, 0; ]
   [ 1, 0, 0, 0; ]
   [ 0, 0, 0, 1; ]
   [ 0, 0, 1, 0; ]

   The operation to be performed is to apply the permutation matrix
   to the lower left quadrant of A on the right and store the result in
   a new matrix C. The result matrix C should be:

   [ 17, 18, 19, 20; ]   [ 0, 1, 0, 0; ]   [ 18, 17, 20, 19; ]
   [ 25, 26, 27, 28; ] * [ 1, 0, 0, 0; ] = [ 26, 25, 28, 27; ]
   [ 33, 34, 35, 36; ]   [ 0, 0, 0, 1; ]   [ 34, 33, 36, 35; ]
   [ 41, 42, 43, 44; ]   [ 0, 0, 1, 0; ]   [ 42, 41, 44, 43; ]

   All matrices are stored in row-major order in double precision.
 */

int main()
{
    // A is 6 x 8
    // Create on the heap
    const uint32_t A_rows = 6;
    const uint32_t A_cols = 8;
    double *A = new double[A_rows * A_cols];
    const uint32_t LDA = A_cols;

    // Fill out A
    // Row-major means adjacent memory locations are increasing
    for(uint32_t i=0; i < A_rows * A_cols; i++)
    {
      A[i] = i + 1;
    }

    // P is k x k
    // Create on the stack
    const uint32_t P_size = 4;
    double P[] = { 0, 1, 0, 0,
```

```
                    1, 0, 0, 0,
                    0, 0, 0, 1,
                    0, 0, 1, 0 };
    const uint32_t LDP = P_size;

    // C is k x k
    // Create on the heap
    double *C = new double[P_size * P_size];
    const uint32_t LDC = P_size;

    // CBLAS call to dgemm
    // Operation:
    //     C := 1.0 * P * A[3..6][1..4] + 0.0 * C
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, P_size, P_size,
        P_size, 1.0, A + LDA * 2, LDA, P, LDP, 0.0, C, LDC);

    // Show result matrix C in row-major order
    for (uint32_t row = 0; row < P_size; row++)
    {
        for (uint32_t col = 0; col < P_size; col++)
        {
            std::cout << C[col + LDC * row] << " ";
        }
        std::cout << std::endl;
    }

    // Cleanup
    delete[] A;
    delete[] C;
    return 0;
}
```

BLAS Solve Example

This example shows how to use the Fortran BLAS interface from C++ code to solve a system of linear equations.

```
#include <qml.h>

#include <iostream>

/* Example program that uses the BLAS interface to solve a system of
   equations.

   This program constructs a matrix A of size 7x7 with entries:

   [ 1,  0,   0,     0,  0 ]
   [ 0,  2,  -1,     0,  0 ]
   [ 0,  0, 1.5,    -1,  0 ]
   [ 0,  0,   0, 1.333, -1 ]
   [ 0,  0,   0,     0,  1 ]

   and a vector b with entries:

   [     1 ]
   [     2 ]
   [     2 ]
   [ 2.333 ]
```

```
   [     1 ]

   We would like to solve the system of equations A * x = b for x.

   Matrix A is stored in column-major order using single precision.
 */

int main()
{
    // A is 5x5
    // Create on the heap
    const qml_long A_size = 5;
    float *A = new float[A_size * A_size]{}; // Initialize to 0
    const qml_long LDA = A_size;

    // Fill out non-zero A entries
    A[0 + 0 * LDA] = 1.0f;
    A[1 + 1 * LDA] = 2.0f;
    A[2 + 2 * LDA] = 1.5f;
    A[3 + 3 * LDA] = 1.33333333f;
    A[4 + 4 * LDA] = 1.0f;
    A[1 + 2 * LDA] = -1.0f;
    A[2 + 3 * LDA] = -1.0f;
    A[3 + 4 * LDA] = -1.0f;

    // b is length 7
    // Create on the stack, name V since it will also hold x
    const qml_long V_size = 5;
    float V[] = { 1.0f, 2.0f, 2.0f, 2.33333333f, 1.0f };
    const qml_long INCV = 1;

    // BLAS call to strsv (single precision triangular solver)
    // Operation:
    //     Solve A x = b for x
    //     Upper triangular
    //     Non-transposed solve
    //     Non-unit diagonal
    // Note we pass addresses of scalars following Fortran convention
    strsv("U", "N", "N", &A_size, A, &LDA, V, &INCV);

    // Show result vector
    for (qml_long i = 0; i < V_size; i++)
    {
        std::cout << V[i] << std::endl;
    }

    // Cleanup
    delete[] A;
    return 0;
}
```

LAPACK Least Squares Example

This example shows how to use the LAPACK interface of QML from C++ code to find the best-fit quadratic equation through a set of points. The example calls `dgels` to compute the linear least squares solution.

```
#include <qml.h>

#include <iostream>

/* Example program that uses the LAPACK interface to compute
   the best-fit quadratic equation through a set of data points.

   The problem is to find values for beta_0, beta_1, and beta_2 so that
   the formula

   y = beta_0 + beta_1 * x + beta_2 * x^2

   is the "best" approximation to the following observed data points.

    x   | y
   -----+-----
    1   | 3
    2   | 4
    4   | 13
    5   | 27

   Here "best" means minimizing the sum of the squares of the
   differences.

   We use the LAPACK GELS function to compute a least-squares solution
   to the overdetermined A * x = b.

   A is the matrix:

   [ 1    1    1  ]
   [ 1    2    4  ]
   [ 1    4    16 ]
   [ 1    5    25 ]

   x is the unknown vector:

   [ beta_0 ]
   [ beta_1 ]
   [ beta_2 ]

   b is the matrix:

   [  3 ]
   [  4 ]
   [ 13 ]
   [ 27 ]

   Matrix A is stored in column-major order in double precision.
   Even though x and b are vectors here, GELS can handle multiple right
   hand sides simultaneously so it takes x and b as matrices.
 */

int main()
{
    // A is 4x3
    // Create on the heap
    const qml_long A_rows = 4, A_cols = 3;
    // Column major, each text row below is a column of A
    double *A = new double[A_rows * A_cols]{
```

```
        1.0, 1.0, 1.0, 1.0,
        1.0, 2.0, 4.0, 5.0,
        1.0, 4.0, 16.0, 25.0
    };
    const qml_long LDA = A_rows;
    const qml_long NRHS = 1;

    // b is length 4
    // Will contain x after dgels returns
    const qml_long B_size = 4;
    const qml_long X_size = 3;
    double *B = new double[B_size]{ 3.0, 4.0, 13.0, 27.0 };
    const qml_long LDB = B_size;

    // Query required workspace size
    qml_long lwork = -1; // query into first position of workspace
    double opt_worksize;
    qml_long info;
    dgels("N", &A_rows, &A_cols, &NRHS, A, &LDA, B, &LDB, &opt_worksize, &lwork, &info);

    // Allocate optimal scratch space
    // (Need cast to integer type because optimal size is given as a double)
    lwork = static_cast<qml_long>(opt_worksize);
    double *WORK = new double[lwork]{};

    // LAPACK call to dgels (double precision least squares solver)
    dgels("N", &A_rows, &A_cols, &NRHS, A, &LDA, B, &LDB, WORK, &lwork, &info);

    // Check for success
    if (info != 0)
    {
        std::cout << "ERROR computing least squares solution" << std::endl;
        return 1;
    }

    // Show result (should be y = 8.73333 * x^0  + -7.3 * x^1  + 2.16667 * x^2)
    std::cout << "Best fit equation is y = ";
    for (qml_long i = 0; i < X_size; i++)
    {
        std::cout << B[i] << " * x^" << i << " ";
        if (i < X_size - 1)
        {
            std::cout << " + ";
        }
    }
    std::cout << std::endl;

    // Cleanup
    delete[] A;
    delete[] B;
    delete[] WORK;
    return 0;
}
```

LAPACK Singular Value Decomposition

This example shows how to use the LAPACK interface of QML from C++ code to find the singular value decomposition of a matrix. It calls `dgesvd` to do the decomposition.

```
#include <qml.h>

#include <iostream>

/* Example program that uses the LAPACK interface to compute
   the singular value decomposition (SVD) of a matrix.

   The problem is to find U, SIGMA, and V^T such that

   U * SIGMA * V^T = [ 3  2  2 ]
                     [ 2  3 -2 ]

   where U is a 2x2 orthogonal matrix, V^T is a 3x3 orthogonal matrix,
   and SIGMA is a 2x3 matrix with non-zero elements only on the diagonal.
 */

int main()
{
    // A is 2x3
    // Create on the heap
    const qml_long rows = 2, cols = 3;
    // Column major, each text row below is a column of A
    double *A = new double[rows * cols]{
        3.0, 2.0,
        2.0, 3.0,
        2.0, -2.0
    };

    // S is diagonal entries of SIGMA
    double *S = new double[rows]{};

    // U is 2x2
    double *U = new double[rows * rows]{};

    // VT is 3x3
    double *VT = new double[cols * cols]{};

    // Query required workspace size
    qml_long lwork = -1; // query into first position of workspace
    double opt_worksize;
    qml_long info;
    dgesvd("A", "A", &rows, &cols, A, &rows, S, U, &rows, VT, &cols, &opt_worksize, &lwork, &info);

    // Allocate optimal scratch space
    // (Need cast to integer type because optimal size is given as a double)
    lwork = static_cast<qml_long>(opt_worksize);
    double *WORK = new double[lwork]{};

    // Do SVD
    dgesvd("A", "A", &rows, &cols, A, &rows, S, U, &rows, VT, &cols, WORK, &lwork, &info);

    // Check for success
    if (info != 0)
    {
```

```
        std::cout << "ERROR computing SVD" << std::endl;
        return 1;
    }

    // Show singular values (should be [ 5 3 ])
    std::cout << "Singular values are [ ";
    for (qml_long i = 0; i < rows; i++)
    {
        std::cout << S[i] << " ";
    }
    std::cout << "]\n";

    // Show the first left singular vector (should be [ -0.707107 -0.707107 ])
    std::cout << "First left singular vector is [ ";
    for (qml_long i = 0; i < rows; i++)
    {
        std::cout << U[i] << " ";
    }
    std::cout << "]\n";

    // Cleanup
    delete[] A;
    delete[] S;
    delete[] U;
    delete[] VT;
    delete[] WORK;
    return 0;
}
```

Building the Examples

One way to build your application or library and link against QML is to use the official Android NDK. You declare that QML is a pre-built library using directives in an `Android.mk` file inside the `jni` directory of an application tree.

```
include $(CLEAR_VARS)
LOCAL_MODULE := QML
LOCAL_SRC_FILES := <install-prefix>/lib<name>.so
LOCAL_EXPORT_C_INCLUDES := <install-prefix>/include
include $(PREBUILT_SHARED_LIBRARY)
```

Once this module has been declared, you can let the build system know that your application links against QML by adding the following directive to your application's `Android.mk`:

```
LOCAL_SHARED_LIBRARIES += QML
```

To have access to newer C++11 features you also need to link against a full-featured version of the C++ runtime library and enable C++11 features in the compiler. These settings are set in `Application.mk`.

```
APP_STL := gnustl_shared
APP_CPPFLAGS += -std=c++11
```

To build the provided example programs in this way, run `ndk-build` from the `examples/` directory of the installation within the desired architecture. This will produce executables and library files inside `libs/<arch>/`. Within the context of a complete application, these executables and libraries will be included as part of the final APK and installed in the correct location on application install.

Running the Examples

There are two main ways to run the examples. If you control the Android platform on the device, you can add QML to the system libraries that are available to all applications. If you do not control the platform then you will include the libraries as part of your application so they will be installed in the private application area during installation.

Platform control

If you control the Android platform, you can install the QML into the system libraries location of the device. The typical location would be `/system/vendor/lib/` for 32-bit architectures or `/system/vendor/lib64/` for 64-bit architectures. With root access in ADB this can be done for testing purposes using `adb push`.

Once the libraries are installed in system locations, running the examples requires copying the executables to any location on the device and running them.

Local install

To run the executables without root access, copy the libraries and executables from `libs/<arch>/` to an accessible directory on the device such as `/data/local/test`. Once QML, and the test application are all in the same directory, run the executable with a command such as:

```
LD_LIBRARY_PATH=. ./MinimalExample
```

Complete applications packaged inside an APK will automatically install the libraries into the correct locations during installation if the libraries are declared as previously described in the `Android.mk` file.

Part IV

QML API Reference

CHAPTER

FIFTEEN

BLAS 1 FUNCTIONS

The BLAS 1 consists of vector-vector operations. Below are the BLAS 1 functions supported by QML.

(IS|ID|IC|IZ)AMAX

Single, double, single complex, and double complex AMAX.

Description

Returns the index of the element with the maximum absolute value.

BLAS Interface

```
qml_long isamax(const qml_long *N, const float *X, const qml_long *INCX);

qml_long idamax(const qml_long *N, const double *X, const qml_long *INCX);

qml_long icamax(const qml_long *N, const qml_single_complex *X,
                const qml_long *INCX);

qml_long izamax(const qml_long *N, const qml_double_complex *X,
                const qml_long *INCX);
```

CBLAS Interface

```
qml_long cblas_isamax(const qml_long N, const float *X, const qml_long INCX);

qml_long cblas_idamax(const qml_long N, const double *X, const qml_long INCX);

qml_long cblas_icamax(const qml_long N, const qml_single_complex *X,
                      const qml_long INCX);

qml_long cblas_izamax(const qml_long N, const qml_double_complex *X,
                      const qml_long INCX);
```

Arguments

N	Number of elements in X
X	Input vector, must be at least: $(N-1)*\|INCX\|+1$
INCX	Distance between individual elements in X
Result	Index of element with maximum absolute value

(IS|ID|IC|IZ)AMIN

Single, double, single complex and double complex AMIN.

Description

Returns the index of the element with the minimum absolute value.

BLAS Interface

```
qml_long isamin(const qml_long *N, const float *X, const qml_long *INCX);

qml_long idamin(const qml_long *N, const double *X, const qml_long *INCX);

qml_long icamin(const qml_long *N, const qml_single_complex *X,
                const qml_long *INCX);

qml_long izamin(const qml_long *N, const qml_double_complex *X,
                const qml_long *INCX);
```

CBLAS Interface

```
qml_long cblas_isamin(const qml_long N, const float *X, const qml_long INCX);

qml_long cblas_idamin(const qml_long N, const double *X, const qml_long INCX);

qml_long cblas_icamin(const qml_long N, const qml_single_complex *X,
                      const qml_long INCX);

qml_long cblas_izamin(const qml_long N, const qml_double_complex *X,
                      const qml_long INCX);
```

Arguments

N	Number of elements in X
X	Input vector, must be at least: $(N-1)*\|INCX\|+1$
INCX	Distance between individual elements in X
Result	Index of element with minimum absolute value

(S|D|SC|DZ)ASUM

Single, double, single complex, and double complex ASUM.

Description

Sum of the absolute value of the elements of the vector.

$$result = \sum_{i=1}^{N} |X_i|$$

BLAS Interface

```
float  sasum(const qml_long *N, const float *X, const qml_long *INCX);

double dasum(const qml_long *N, const double *X, const qml_long *INCX);

float  scasum(const qml_long *N, const qml_single_complex *X,
              const qml_long *INCX);

double dzasum(const qml_long *N, const qml_double_complex *X,
              const qml_long *INCX);
```

CBLAS Interface

```
float  cblas_sasum(const qml_long N, const float *X, const qml_long INCX);

double cblas_dasum(const qml_long N, const double *X, const qml_long INCX);

float  cblas_scasum(const qml_long N, const qml_single_complex *X,
                    const qml_long INCX);

double cblas_dzasum(const qml_long N, const qml_double_complex *X,
                    const qml_long INCX);
```

Arguments

N	Number of elements in X
X	Input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Result	Sum of absolute value of elements of vector X

(S|D|C|Z)AXPBY

Single, double, single complex, and double complex AXPBY.

Description

Extension of AXPY that also scales the vector Y.

$$y \leftarrow \alpha x + \beta y$$

BLAS Interface

```
void saxpby(const qml_long *N, const float *ALPHA, const float *X,
            const qml_long *INCX, const float *BETA, float *Y,
            const qml_long *INCY);

void daxpby(const qml_long *N, const double *ALPHA, const double *X,
            const qml_long *INCX, const double *BETA, double *Y,
            const qml_long *INCY);

void caxpby(const qml_long *N, const qml_single_complex *ALPHA,
            const qml_single_complex *X, const qml_long *INCX,
            const qml_single_complex *BETA, qml_single_complex *Y,
            const qml_long *INCY);

void zaxpby(const qml_long *N, const qml_double_complex *ALPHA,
            const qml_double_complex *X, const qml_long *INCX,
            const qml_double_complex *BETA, qml_double_complex *Y,
            const qml_long *INCY);
```

CBLAS Interface

```
void cblas_saxpby(const qml_long N, const float ALPHA, const float *X,
                  const qml_long INCX, const float BETA, float *Y,
                  const qml_long INCY);

void cblas_daxpby(const qml_long N, const double ALPHA, const double *X,
                  const qml_long INCX, const double BETA, double *Y,
                  const qml_long INCY);

void cblas_caxpby(const qml_long N, const qml_single_complex *ALPHA,
                  const qml_single_complex *X, const qml_long INCX,
                  const qml_single_complex *BETA, qml_single_complex *Y,
                  const qml_long INCY);

void cblas_zaxpby(const qml_long N, const qml_double_complex *ALPHA,
                  const qml_double_complex *X, const qml_long INCX,
                  const qml_double_complex *BETA, qml_double_complex *Y,
                  const qml_long INCY);
```

Arguments

N	Number of elements in X and Y
ALPHA	Scalar to scale vector X by
X	First input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
BETA	Scalar to scale vector Y by
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y

(S|D|C|Z)AXPY

Single, double, single complex, and double complex AXPY.

Description

Scales the vector x and adds to vector y.

$$y \leftarrow \alpha x + y$$

BLAS Interface

```
void saxpy(const qml_long *N, const float *ALPHA, const float *X,
           const qml_long *INCX, float *Y, const qml_long *INCY);

void daxpy(const qml_long *N, const double *ALPHA, const double *X,
           const qml_long *INCX, double *Y, const qml_long *INCY);

void caxpy(const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *X, const qml_long *INCX,
           qml_single_complex *Y, const qml_long *INCY);

void zaxpy(const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *X, const qml_long *INCX,
           qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_saxpy(const qml_long N, const float ALPHA, const float *X,
                 const qml_long INCX, float *Y, const qml_long INCY);

void cblas_daxpy(const qml_long N, const double ALPHA, const double *X,
                 const qml_long INCX, double *Y, const qml_long INCY);

void cblas_caxpy(const qml_long N, const qml_single_complex *ALPHA,
                 const qml_single_complex *X, const qml_long INCX,
                 qml_single_complex *Y, const qml_long INCY);

void cblas_zaxpy(const qml_long N, const qml_double_complex *ALPHA,
                 const qml_double_complex *X, const qml_long INCX,
                 qml_double_complex *Y, const qml_long INCY);
```

Arguments

N	Number of elements in X and Y
ALPHA	Scalar to scale vector X by
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y

(S|D)CABS1

Single complex and double complex CABS1.

Description

Returns the absolute value of a complex number.

$$result = |real(Z)| + |imag(Z)|$$

BLAS Interface

```
float  scabs1(const qml_single_complex *Z);

double dcabs1(const qml_double_complex *Z);
```

CBLAS Interface

```
float  cblas_scabs1(const qml_single_complex *Z);

double cblas_dcabs1(const qml_double_complex *Z);
```

Arguments

Z	Complex input
Result	Absolute value of complex input

(S|D|C|Z)COPY

Single, double, single complex, and double complex COPY.

Description

Copies the contents of vector X to vector Y.

BLAS Interface

```
void scopy(const qml_long *N, const float *X, const qml_long *INCX,
           float *Y, const qml_long *INCY);

void dcopy(const qml_long *N, const double *X, const qml_long *INCX,
           double *Y, const qml_long *INCY);

void ccopy(const qml_long *N, const qml_single_complex *X,
           const qml_long *INCX, qml_single_complex *Y,
           const qml_long *INCY);

void zcopy(const qml_long *N, const qml_double_complex *X,
           const qml_long *INCX, qml_double_complex *Y,
           const qml_long *INCY);
```

CBLAS Interface

```
void cblas_scopy(const qml_long N, const float *X, const qml_long INCX,
                 float *Y, const qml_long INCY);

void cblas_dcopy(const qml_long N, const double *X, const qml_long INCX,
                 double *Y, const qml_long INCY);

void cblas_ccopy(const qml_long N, const qml_single_complex *X,
                 const qml_long INCX, qml_single_complex *Y,
                 const qml_long INCY);

void cblas_zcopy(const qml_long N, const qml_double_complex *X,
                 const qml_long INCX, qml_double_complex *Y,
                 const qml_long INCY);
```

Arguments

N	Number of elements in X and Y
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y

(S|D)DOT

Single and double DOT.

Description

Performs a dot product using the input vectors X and Y.

$$result = \sum_{i=1}^{N} x_i * y_i$$

BLAS Interface

```
float  sdot(const qml_long *N, const float *X, const qml_long *INCX,
            const float *Y, const qml_long *INCY);

double ddot(const qml_long *N, const double *X, const qml_long *INCX,
            const double *Y, const qml_long *INCY);
```

CBLAS Interface

```
float  cblas_sdot(const qml_long N, const float *X, const qml_long INCX,
                  const float *Y, const qml_long INCY);

double cblas_ddot(const qml_long N, const double *X, const qml_long INCX,
                  const double *Y, const qml_long INCY);
```

Arguments

N	Number of elements in X and Y
X	First input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
Result	Result of the dot product

(C|Z)DOTC

Single complex and double complex DOTC.

Description

Performs a dot product of the complex conjugate of x with y.

$$result = \sum_{i=1}^{N} conj(x_i) * y_i$$

BLAS Interface

```
void cdotc(qml_single_complex *result, const qml_long *N,
           const qml_single_complex *X, const qml_long *INCX,
           const qml_single_complex *Y, const qml_long *INCY);

void zdotc(qml_double_complex *result, const qml_long *N,
           const qml_double_complex *X, const qml_long *INCX,
           const qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_cdotc_sub(const qml_long N, const qml_single_complex *X,
                     const qml_long INCX, const qml_single_complex *Y,
                     const qml_long INCY, qml_single_complex *result);

void cblas_zdotc_sub(const qml_long N, const qml_double_complex *X,
                     const qml_long INCX, const qml_double_complex *Y,
                     const qml_long INCY, qml_double_complex *result);
```

Arguments

N	Number of elements in X and Y
X	First input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
Result	Result of the dot product

(C|Z)DOTU

Single complex and double complex DOTU.

Description

Performs a dot product of the complex vectors x and y.

$$result = \sum_{i=1}^{N} x_i * y_i$$

BLAS Interface

```
void cdotu(qml_single_complex *result, const qml_long *N,
           const qml_single_complex *X, const qml_long *INCX,
           const qml_single_complex *Y, const qml_long *INCY);

void zdotu(qml_double_complex *result, const qml_long *N,
           const qml_double_complex *X, const qml_long *INCX,
           const qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_cdotu_sub(const qml_long N, const qml_single_complex *X,
                     const qml_long INCX, const qml_single_complex *Y,
                     const qml_long INCY, qml_single_complex *result);

void cblas_zdotu_sub(const qml_long N, const qml_double_complex *X,
                     const qml_long INCX, const qml_double_complex *Y,
                     const qml_long INCY, qml_double_complex *result);
```

Arguments

N	Number of elements in X and Y
X	First input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
Result	Result of the dot product

(S|D|SC|DZ)NRM2

Single, double, single complex, and double complex NRM2.

Description

Computes the Euclidean norm of a vector.

$$result = \sqrt{\sum_{i=1}^{N} |x_i|^2}$$

BLAS Interface

```
float  snrm2(const qml_long *N, const float *X, const qml_long *INCX);

double dnrm2(const qml_long *N, const double *X, const qml_long *INCX);

float  scnrm2(const qml_long *N, const qml_single_complex *X, const qml_long *INCX);

double dznrm2(const qml_long *N, const qml_double_complex *X, const qml_long *INCX);
```

CBLAS Interface

```
float  cblas_snrm2(const qml_long N, const float *X, const qml_long INCX);

double cblas_dnrm2(const qml_long N, const double *X, const qml_long INCX);

float  cblas_scnrm2(const qml_long N, const qml_single_complex *X,
                    const qml_long INCX);

double cblas_dznrm2(const qml_long N, const qml_double_complex *X,
                    const qml_long INCX);
```

Arguments

N	Number of elements in X
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X

(S|D|C|CS|Z|ZD)ROT

Single, double, single complex, and double complex ROT.

Description

Applies a plane rotation.

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

BLAS Interface

```
void srot(const qml_long *N, float *X, const qml_long *INCX, float *Y,
          const qml_long *INCY, const float *C, const float *S);

void drot(const qml_long *N, double *X, const qml_long *INCX, double *Y,
          const qml_long *INCY, const double *C, const double *S);

void crot(const qml_long *N, qml_single_complex *X, const qml_long *INCX,
          qml_single_complex *Y, const qml_long *INCY, const float *C,
          const qml_single_complex *S);

void csrot(const qml_long *N, qml_single_complex *X, const qml_long *INCX,
           qml_single_complex *Y, const qml_long *INCY, const float *C,
           const float *S);

void zrot(const qml_long *N, qml_double_complex *X, const qml_long *INCX,
          qml_double_complex *Y, const qml_long *INCY, const double *C,
          const qml_double_complex *S);

void zdrot(const qml_long *N, qml_double_complex *X, const qml_long *INCX,
           qml_double_complex *Y, const qml_long *INCY, const double *C,
           const double *S);
```

CBLAS Interface

```
void cblas_srot(const qml_long N, float *X, const qml_long INCX, float *Y,
                const qml_long INCY, const float C, const float S);

void cblas_drot(const qml_long N, double *X, const qml_long INCX, double *Y,
                const qml_long INCY, const double C, const double S);

void cblas_crot(const qml_long N, qml_single_complex *X, const qml_long INCX,
                qml_single_complex *Y, const qml_long INCY, const float C,
                const qml_single_complex *S);

void cblas_csrot(const qml_long N, qml_single_complex *X, const qml_long INCX,
                 qml_single_complex *Y, const qml_long INCY, const float C,
                 const float S);

void cblas_zrot(const qml_long N, qml_double_complex *X, const qml_long INCX,
                qml_double_complex *Y, const qml_long INCY, const double C,
                const qml_double_complex *S);

void cblas_zdrot(const qml_long N, qml_double_complex *X, const qml_long INCX,
                 qml_double_complex *Y, const qml_long INCY, const double C,
                 const double S);
```

Arguments

N	The number of elements in X and Y
X	The X coordinates for the series of points, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	The Y coordinates for the series of points, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
C	$\cos\theta$
S	$\sin\theta$

(S|D|C|Z)ROTG

Single, double, single complex, and double complex ROTG.

Description

Generates a single plane rotation.

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} * \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

BLAS Interface

```
void srotg(float *A, float *B, float *C, float *S);

void drotg(double *A, double *B, double *C, double *S);

void crotg(qml_single_complex *A, const qml_single_complex *B,
           float *C, qml_single_complex *S);

void zrotg(qml_double_complex *A, const qml_double_complex *B,
           double *C, qml_double_complex *S);
```

CBLAS Interface

```
void cblas_srotg(float *A, float *B, float *C, float *S);

void cblas_drotg(double *A, double *B, double *C, double *S);

void cblas_crotg(qml_single_complex *A, const qml_single_complex *B,
                 float *C, qml_single_complex *S);

void cblas_zrotg(qml_double_complex *A, const qml_double_complex *B,
                 double *C, qml_double_complex *S);
```

Arguments

A	First component of vector to rotate
	On output contains R
B	Second component of vector to rotate
	On output contains Z, which is:
	$\begin{array}{rll} \mathtt{S} & \text{if} & \lvert\mathtt{A}\rvert > \lvert\mathtt{B}\rvert \\ 1/\mathtt{C} & \text{if} & \lvert\mathtt{A}\rvert \leq \lvert\mathtt{B}\rvert \text{ and } C \neq 0 \text{ and } R \neq 0 \\ 1 & \text{if} & \lvert\mathtt{A}\rvert \leq \lvert\mathtt{B}\rvert \text{ and } C = 0 \text{ and } R \neq 0 \\ 0 & \text{if} & R = 0 \end{array}$
C	On output contains $\cos\theta$
S	On output contains $\sin\theta$

(S|D)ROTM

Single and double ROTM.

Description

Applies a modified plane rotation to a series of points.

The exact form of the rotation matrix will depend on *FLAG* which is the first element of the *PARAM* input array.

$$\mathtt{PARAM} = \begin{bmatrix} \mathtt{FLAG} \\ \mathtt{H11} \\ \mathtt{H21} \\ \mathtt{H12} \\ \mathtt{H22} \end{bmatrix}$$

If $\mathtt{FLAG} = -1$:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} \mathtt{H11} & \mathtt{H12} \\ \mathtt{H21} & \mathtt{H22} \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

If $\mathtt{FLAG} = 0$:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} 1 & \mathtt{H12} \\ \mathtt{H21} & 1 \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

If $\mathtt{FLAG} = 1$:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} \mathtt{H11} & 1 \\ -1 & \mathtt{H22} \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

If $\mathtt{FLAG} = -2$:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} \leftarrow \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

BLAS Interface

```
void srotm(const qml_long *N, float *X, const qml_long *INCX, float *Y,
           const qml_long *INCY, const float *PARAM);

void drotm(const qml_long *N, double *X, const qml_long *INCX, double *Y,
           const qml_long *INCY, const double *PARAM);
```

CBLAS Interface

```
void cblas_srotm(const qml_long N, float *X, const qml_long INCX, float *Y,
                 const qml_long INCY, const float *PARAM);

void cblas_drotm(const qml_long N, double *X, const qml_long INCX, double *Y,
                 const qml_long INCY, const double *PARAM);
```

Arguments

N	The number of elements in X and Y
X	The X coordinates for the series of points, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	The Y coordinates for the series of points, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
PARAM	Rotation parameter array of length 5

(S|D)ROTMG

Single and double ROTMG.

Description

Generate a single modified plane rotation.

$$H * \begin{bmatrix} \sqrt{D_1} * X_1 \\ \sqrt{D_2} * Y_1 \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

The form of the generated rotation matrix H will depend on *FLAG* which is the first element of the *PARAM* input array.

$$\texttt{PARAM} = \begin{bmatrix} \texttt{FLAG} \\ \texttt{H11} \\ \texttt{H21} \\ \texttt{H12} \\ \texttt{H22} \end{bmatrix}$$

If $\texttt{FLAG} = -1$:

$$H = \begin{bmatrix} \texttt{H11} & \texttt{H12} \\ \texttt{H21} & \texttt{H22} \end{bmatrix}$$

If $\texttt{FLAG} = 0$:

$$H = \begin{bmatrix} 1 & \texttt{H12} \\ \texttt{H21} & 1 \end{bmatrix}$$

If $\texttt{FLAG} = 1$:

$$H = \begin{bmatrix} \texttt{H11} & 1 \\ -1 & \texttt{H22} \end{bmatrix}$$

If $\texttt{FLAG} = -2$:

$$H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

BLAS Interface

```
void srotmg(float *D1, float *D2, float *X1, const float *Y1, float *PARAM);

void drotmg(double *D1, double *D2, double *X1, const double *Y1,
            double *PARAM);
```

CBLAS Interface

```
void cblas_srotmg(float *D1, float *D2, float *X1, const float Y1,
                  float *PARAM);

void cblas_drotmg(double *D1, double *D2, double *X1, const double Y1,
                  double *PARAM);
```

Arguments

D1	First scaling factor
D2	Second scaling factor
X1	First component of vector to rotate
Y1	Second component of vector to rotate
PARAM	Rotation parameter array of length 5

(S|D|C|Z|CS|ZD)SCAL

Single, double, single complex, double complex, single complex by single, and double complex by double SCAL.

Description

Scale the contents of a vector by a constant.

$$X \leftarrow \alpha X$$

BLAS Interface

```
void sscal(const qml_long *N, const float *ALPHA, float *X,
           const qml_long *INCX);

void dscal(const qml_long *N, const double *ALPHA, double *X,
           const qml_long *INCX);

void cscal(const qml_long *N, const qml_single_complex *ALPHA,
           qml_single_complex *X, const qml_long *INCX);

void zscal(const qml_long *N, const qml_double_complex *ALPHA,
           qml_double_complex *X, const qml_long *INCX);

void csscal(const qml_long *N, const float *ALPHA, qml_single_complex *X,
            const qml_long *INCX);

void zdscal(const qml_long *N, const double *ALPHA, qml_double_complex *X,
            const qml_long *INCX) ;
```

CBLAS Interface

```
void cblas_sscal(const qml_long N, const float ALPHA, float *X,
                 const qml_long INCX);

void cblas_dscal(const qml_long N, const double ALPHA, double *X,
                 const qml_long INCX);

void cblas_cscal(const qml_long N, const qml_single_complex *ALPHA,
                 qml_single_complex *X, const qml_long INCX);

void cblas_zscal(const qml_long N, const qml_double_complex *ALPHA,
                 qml_double_complex *X, const qml_long INCX);

void cblas_csscal(const qml_long N, const float ALPHA, qml_single_complex *X,
                  const qml_long INCX);

void cblas_zdscal(const qml_long N, const double ALPHA,
                  qml_double_complex *X, const qml_long INCX);
```

Arguments

N	Number of elements to scale in X
ALPHA	Scale factor
X	Input vector, must be at least size: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X

(D)SDOT

Double with extended internal precision SDOT.

Description

Performs a dot product of the input vectors X and Y using extended internal precision for accumulation.

$$result = \sum_{i=1}^{N} x_i * y_i$$

BLAS Interface

```
double dsdot(const qml_long *N, const float *X, const qml_long *INCX,
             const float *Y, const qml_long *INCY);
```

CBLAS Interface

```
double cblas_dsdot(const qml_long N, const float *X, const qml_long INCX,
                   const float *Y, const qml_long INCY);
```

Arguments

N	Number of elements in X and Y
X	First input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
Result	Result of the dot product

(SD)SDOT

Single with extended internal precision SDOT.

Description

Performs a dot product of the input vectors X and Y using extended internal precision for accumulation.

$$result = B + \sum_{i=1}^{N} x_i * y_i$$

BLAS Interface

```
float sdsdot(const qml_long *N, const float *B, const float *X,
             const qml_long *INCX, const float *Y, const qml_long *INCY);
```

CBLAS Interface

```
float cblas_sdsdot(const qml_long N, const float B, const float *X,
                   const qml_long INCX, const float *Y,
                   const qml_long INCY);
```

Arguments

N	Number of elements in X and Y
B	Initial accumulator value
X	First input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
Result	Result of the dot product

(S|D|C|Z)SWAP

Single, double, single complex, and double complex SWAP.

Description

Swap the contents of vectors X and Y.

BLAS Interface

```
void sswap(const qml_long *N, float *X, const qml_long *INCX, float *Y,
           const qml_long *INCY);

void dswap(const qml_long *N, double *X, const qml_long *INCX, double *Y,
           const qml_long *INCY);

void cswap(const qml_long *N, qml_single_complex *X, const qml_long *INCX,
           qml_single_complex *Y, const qml_long *INCY);

void zswap(const qml_long *N, qml_double_complex *X, const qml_long *INCX,
           qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_sswap(const qml_long N, float *X, const qml_long INCX,
                 float *Y, const qml_long INCY);

void cblas_dswap(const qml_long N, double *X, const qml_long INCX,
                 double *Y, const qml_long INCY);

void cblas_cswap(const qml_long N, qml_single_complex *X, const qml_long INCX,
                 qml_single_complex *Y, const qml_long INCY);

void cblas_zswap(const qml_long N, qml_double_complex *X, const qml_long INCX,
                 qml_double_complex *Y, const qml_long INCY);
```

Arguments

N	Number of elements in X and Y
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y

CHAPTER

SIXTEEN

BLAS 2 FUNCTIONS

The BLAS 2 consists of matrix-vector operations. Below are the BLAS 2 functions supported by QML.

(S|D|C|Z)GBMV

Single, double, single complex, and double complex GBMV.

Description

Computes a general banded matrix-vector product.

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void sgbmv(const char *TRANS, const qml_long *M, const qml_long *N,
           const qml_long *KL, const qml_long *KU, const float *ALPHA,
           const float *A, const qml_long *LDA, const float *X,
           const qml_long *INCX, const float *BETA, float *Y,
           const qml_long *INCY);

void dgbmv(const char *TRANS, const qml_long *M, const qml_long *N,
           const qml_long *KL, const qml_long *KU, const double *ALPHA,
           const double *A, const qml_long *LDA, const double *X,
           const qml_long *INCX, const double *BETA, double *Y,
           const qml_long *INCY);

void cgbmv(const char *TRANS, const qml_long *M, const qml_long *N,
           const qml_long *KL, const qml_long *KU,
           const qml_single_complex *ALPHA, const qml_single_complex *A,
           const qml_long *LDA, const qml_single_complex *X,
           const qml_long *INCX, const qml_single_complex *BETA,
           qml_single_complex *Y, const qml_long *INCY);

void zgbmv(const char *TRANS, const qml_long *M, const qml_long *N,
           const qml_long *KL, const qml_long *KU,
           const qml_double_complex *ALPHA, const qml_double_complex *A,
           const qml_long *LDA, const qml_double_complex *X,
           const qml_long *INCX, const qml_double_complex *BETA,
           qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_sgbmv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N, const qml_long KL,
                 const qml_long KU, const float ALPHA, const float *A,
                 const qml_long LDA, const float *X, const qml_long INCX,
                 const float BETA, float *Y, const qml_long INCY);

void cblas_dgbmv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N, const qml_long KL,
                 const qml_long KU, const double ALPHA, const double *A,
                 const qml_long LDA, const double *X, const qml_long INCX,
                 const double BETA, double *Y, const qml_long INCY);

void cblas_cgbmv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N, const qml_long KL,
                 const qml_long KU, const qml_single_complex *ALPHA,
                 const qml_single_complex *A, const qml_long LDA,
                 const qml_single_complex *X, const qml_long INCX,
                 const qml_single_complex *BETA, qml_single_complex *Y,
                 const qml_long INCY);

void cblas_zgbmv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N, const qml_long KL,
                 const qml_long KU, const qml_double_complex *ALPHA,
                 const qml_double_complex *A, const qml_long LDA,
                 const qml_double_complex *X, const qml_long INCX,
                 const qml_double_complex *BETA, qml_double_complex *Y,
                 const qml_long INCY);
```

Arguments

TRANS	Specifies how to read matrix A
	Possible values: Non-Transpose, Tranpose, Complex Conjugate Transpose
M	Number of rows of matrix A
N	Number of columns of matrix A
KL	Number of sub-diagonals of A, with $0 \leq$ KL
KU	Number of super-diagonals of A, with $0 \leq$ KU
ALPHA	Scalar multiplied with the matrix-vector product
A	Input matrix A
LDA	Leading dimension of matrix A
X	First input vector, must be at least:
	When Non-Tranpose: $(\mathrm{N}-1) * \lvert\mathrm{INCX}\rvert + 1$
	When Tranpose: $(\mathrm{M}-1) * \lvert\mathrm{INCX}\rvert + 1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second input vector, must be at least:
	When Non-Transpose: $(\mathrm{M}-1) * \lvert\mathrm{INCY}\rvert + 1$
	When Transpose: $(\mathrm{N}-1) * \lvert\mathrm{INCY}\rvert + 1$
INCY	Distance between individual elements in Y

(S|D|C|Z)GEMV

Single, double, single complex, and double complex GEMV.

Description

Computes a general matrix-vector product.

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void sgemv(const char *TRANS, const qml_long *M, const qml_long *N,
           const float *ALPHA, const float *A, const qml_long *LDA,
           const float *X, const qml_long *INCX, const float *BETA, float *Y,
           const qml_long *INCY);

void dgemv(const char *TRANS, const qml_long *M, const qml_long *N,
           const double *ALPHA, const double *A, const qml_long *LDA,
           const double *X, const qml_long *INCX, const double *BETA, double *Y,
           const qml_long *INCY);

void cgemv(const char *TRANS, const qml_long *M, const qml_long *N,
           const qml_single_complex *ALPHA, const qml_single_complex *A,
           const qml_long *LDA, const qml_single_complex *X,
           const qml_long *INCX, const qml_single_complex *BETA,
           qml_single_complex *Y, const qml_long *INCY);

void zgemv(const char *TRANS, const qml_long *M, const qml_long *N,
           const qml_double_complex *ALPHA, const qml_double_complex *A,
           const qml_long *LDA, const qml_double_complex *X,
           const qml_long *INCX, const qml_double_complex *BETA,
           qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_sgemv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N, const float ALPHA,
                 const float *A, const qml_long LDA, const float *X,
                 const qml_long INCX, const float BETA, float *Y,
                 const qml_long INCY);

void cblas_dgemv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N, const double ALPHA,
                 const double *A, const qml_long LDA, const double *X,
                 const qml_long INCX, const double BETA, double *Y,
                 const qml_long INCY);

void cblas_cgemv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *A,
                 const qml_long LDA, const qml_single_complex *X,
                 const qml_long INCX, const qml_single_complex *BETA,
                 qml_single_complex *Y, const qml_long INCY);
```

```
void cblas_zgemv(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANS,
                 const qml_long M, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *A,
                 const qml_long LDA, const qml_double_complex *X,
                 const qml_long INCX, const qml_double_complex *BETA,
                 qml_double_complex *Y, const qml_long INCY);
```

Arguments

TRANS	Specifies how to read matrix A
	Possible values: Non-Transpose, Tranpose, Complex Conjugate Transpose
M	Number of rows of matrix A
N	Number of columns of matrix A
ALPHA	Scalar multiplied with the matrix-vector product
A	Input matrix A
LDA	Leading dimension of matrix A
X	First input vector, must be at least:
	When Non-Tranpose: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
	When Tranpose: $(\texttt{M}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second input vector, must be at least:
	When Non-Transpose: $(\texttt{M}-1)*\lvert\texttt{INCY}\rvert+1$
	When Transpose: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y

(S|D)GER

Single and double GER.

Description

Computes the rank-1 operation:

$$A \leftarrow \alpha x * y^T + A$$

BLAS Interface

```
void sger(const qml_long *M, const qml_long *N, const float *ALPHA,
          const float *X, const qml_long *INCX, const float *Y,
          const qml_long *INCY, float *A, const qml_long *LDA);

void dger(const qml_long *M, const qml_long *N, const double *ALPHA,
          const double *X, const qml_long *INCX, const double *Y,
          const qml_long *INCY, double *A, const qml_long *LDA);
```

CBLAS Interface

```
void cblas_sger(const CBLAS_ORDER ORDER, const qml_long M, const qml_long N,
                const float ALPHA, const float *X, const qml_long INCX,
```

```
                const float *Y, const qml_long INCY, float *A,
                const qml_long LDA);

void cblas_dger(const CBLAS_ORDER ORDER, const qml_long M, const qml_long N,
                const double ALPHA, const double *X, const qml_long INCX,
                const double *Y, const qml_long INCY, double *A,
                const qml_long LDA);
```

Arguments

M	Number of rows of matrix A
N	Number of columns of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	First input vector, must be at least: $(\texttt{M}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
A	Input matrix A
LDA	Leading dimension of matrix A

(C|Z)GERC

Single complex and double complex GERC.

Description

Computes the rank-1 operation:

$$A \leftarrow \alpha x * y^H + A$$

BLAS Interface

```
void cgerc(const qml_long *M, const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *X, const qml_long *INCX,
           const qml_single_complex *Y, const qml_long *INCY,
           qml_single_complex *A, const qml_long *LDA);

void zgerc(const qml_long *M, const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *X, const qml_long *INCX,
           const qml_double_complex *Y, const qml_long *INCY,
           qml_double_complex *A, const qml_long *LDA);
```

CBLAS Interface

```
void cblas_cgerc(const CBLAS_ORDER ORDER, const qml_long M, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *X,
                 const qml_long INCX, const qml_single_complex *Y,
                 const qml_long INCY, qml_single_complex *A,
                 const qml_long LDA);

void cblas_zgerc(const CBLAS_ORDER ORDER, const qml_long M, const qml_long N,
```

```
                const qml_double_complex *ALPHA, const qml_double_complex *X,
                const qml_long INCX, const qml_double_complex *Y,
                const qml_long INCY, qml_double_complex *A,
                const qml_long LDA);
```

Arguments

M	Number of rows of matrix A
N	Number of columns of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	First input vector, must be at least: $(\mathtt{M}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
A	Input matrix A
LDA	Leading dimension of matrix A

(C|Z)GERU

Single complex and double complex GERU.

Description

Computes the rank-1 operation:

$$A \leftarrow \alpha x * y^T + A$$

BLAS Interface

```
void cgeru(const qml_long *M, const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *X, const qml_long *INCX,
           const qml_single_complex *Y, const qml_long *INCY,
           qml_single_complex *A, const qml_long *LDA);

void zgeru(const qml_long *M, const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *X, const qml_long *INCX,
           const qml_double_complex *Y, const qml_long *INCY,
           qml_double_complex *A, const qml_long *LDA);
```

CBLAS Interface

```
void cblas_cgeru(const CBLAS_ORDER ORDER, const qml_long M, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *X,
                 const qml_long INCX, const qml_single_complex *Y,
                 const qml_long INCY, qml_single_complex *A,
                 const qml_long LDA);

void cblas_zgeru(const CBLAS_ORDER ORDER, const qml_long M, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *X,
                 const qml_long INCX, const qml_double_complex *Y,
                 const qml_long INCY, qml_double_complex *A,
                 const qml_long LDA);
```

Arguments

M	Number of rows of matrix A
N	Number of columns of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	First input vector, must be at least: $(\texttt{M}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
A	Input matrix A
LDA	Leading dimension of matrix A

(C|Z)HBMV

Single complex and double complex HBMV.

Description

Computes a hermitian banded matrix-vector product.

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void chbmv(const char *UPLO, const qml_long *N, const qml_long *K,
           const qml_single_complex *ALPHA, const qml_single_complex *A,
           const qml_long *LDA, const qml_single_complex *X,
           const qml_long *INCX, const qml_single_complex *BETA,
           qml_single_complex *Y, const qml_long *INCY);

void zhbmv(const char *UPLO, const qml_long *N, const qml_long *K,
           const qml_double_complex *ALPHA, const qml_double_complex *A,
           const qml_long *LDA, const qml_double_complex *X,
           const qml_long *INCX, const qml_double_complex *BETA,
           qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_chbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_long K, const qml_single_complex *ALPHA,
                 const qml_single_complex *A, const qml_long LDA,
                 const qml_single_complex *X, const qml_long INCX,
                 const qml_single_complex *BETA, qml_single_complex *Y,
                 const qml_long INCY);

void cblas_zhbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_long K, const qml_double_complex *ALPHA,
                 const qml_double_complex *A, const qml_long LDA,
                 const qml_double_complex *X, const qml_long INCX,
                 const qml_double_complex *BETA, qml_double_complex *Y,
                 const qml_long INCY);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
K	Number of super-diagonals of A, with $0 \leq \texttt{K}$
ALPHA	Scalar multiplied with the matrix-vector product
A	Input matrix A
LDA	Leading dimension of matrix A, with $\texttt{LDA} \geq \texttt{K} + 1$
X	First input vector, must be at least: $(\texttt{N} - 1) * \lvert\texttt{INCX}\rvert + 1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second vector, must be at least: $(\texttt{N} - 1) * \lvert\texttt{INCY}\rvert + 1$
INCY	Distance between individual elements in Y

(C|Z)HEMV

Single complex and double complex HEMV.

Description

Computes a hermitian matrix-vector product.

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void chemv(const char *UPLO, const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *A, const qml_long *LDA,
           const qml_single_complex *X, const qml_long *INCX,
           const qml_single_complex *BETA, qml_single_complex *Y,
           const qml_long *INCY);

void zhemv(const char *UPLO, const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *A, const qml_long *LDA,
           const qml_double_complex *X, const qml_long *INCX,
           const qml_double_complex *BETA, qml_double_complex *Y,
           const qml_long *INCY);
```

CBLAS Interface

```
void cblas_chemv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *A,
                 const qml_long LDA, const qml_single_complex *X,
                 const qml_long INCX, const qml_single_complex *BETA,
                 qml_single_complex *Y, const qml_long INCY);

void cblas_zhemv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *A,
                 const qml_long LDA, const qml_double_complex *X,
                 const qml_long INCX, const qml_double_complex *BETA,
                 qml_double_complex *Y, const qml_long INCY);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the matrix-vector product
A	Input matrix A
LDA	Leading dimension of matrix A
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y

(C|Z)HER

Single complex and double complex HER.

Description

Computes the hermitian rank-1 operation:

$$A \leftarrow \alpha x * x^H + A$$

BLAS Interface

```
void cher(const char *UPLO, const qml_long *N, const float *ALPHA,
          const qml_single_complex *X, const qml_long *INCX,
          qml_single_complex *A, const qml_long *LDA);

void zher(const char *UPLO, const qml_long *N, const double *ALPHA,
          const qml_double_complex *X, const qml_long *INCX,
          qml_double_complex *A, const qml_long *LDA);
```

CBLAS Interface

```
void cblas_cher(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const float ALPHA, const qml_single_complex *X, const qml_long INCX,
                qml_single_complex *A, const qml_long LDA);

void cblas_zher(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const double ALPHA, const qml_double_complex *X, const qml_long INCX,
                qml_double_complex *A, const qml_long LDA);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	Input vector, must be at least: $(\texttt{N}-1)*\lvert\texttt{INCX}\rvert+1$
INCX	Distance between individual elements in X
A	Input matrix A
LDA	Leading dimension of matrix A

(C|Z)HER2

Single complex and double complex HER2.

Description

Computes the hermitian rank-2 operation:

$$A \leftarrow \alpha x * y^H + conj(\alpha) y * x^H + A$$

BLAS Interface

```
void cher2(const char *UPLO, const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *X, const qml_long *INCX,
           const qml_single_complex *Y, const qml_long *INCY,
           qml_single_complex *A, const qml_long *LDA);

void zher2(const char *UPLO, const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *X, const qml_long *INCX,
           const qml_double_complex *Y, const qml_long *INCY,
           qml_double_complex *A, const qml_long *LDA);
```

CBLAS Interface

```
void cblas_cher2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *X,
                 const qml_long INCX, const qml_single_complex *Y,
                 const qml_long INCY, qml_single_complex *A, const qml_long LDA);

void cblas_zher2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *X,
                 const qml_long INCX, const qml_double_complex *Y,
                 const qml_long INCY, qml_double_complex *A, const qml_long LDA);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
A	Input matrix A
LDA	Leading dimension of matrix A

(C|Z)HPMV

Single complex and double complex HPMV.

Description

Computes the packed hermitian matrix-vector operation:

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void chpmv(const char *UPLO, const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *AP, const qml_single_complex *X,
           const qml_long *INCX, const qml_single_complex *BETA,
           qml_single_complex *Y, const qml_long *INCY);

void zhpmv(const char *UPLO, const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *AP, const qml_double_complex *X,
           const qml_long *INCX, const qml_double_complex *BETA,
           qml_double_complex *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_chpmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *AP,
                 const qml_single_complex *X, const qml_long INCX,
                 const qml_single_complex *BETA, qml_single_complex *Y,
                 const qml_long INCY);

void cblas_zhpmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *AP,
                 const qml_double_complex *X, const qml_long INCX,
                 const qml_double_complex *BETA, qml_double_complex *Y,
                 const qml_long INCY);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the matrix-vector product
AP	Matrix A stored in packed triangular form, must be at least: $\mathtt{N} * (\mathtt{N}+1)/2$
X	First input vector, must be at least: $(\mathtt{N}-1) * \lvert\mathtt{INCX}\rvert + 1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second input vector, must be at least: $(\mathtt{M}-1) * \lvert\mathtt{INCY}\rvert + 1$
INCY	Distance between individual elements in Y

(C|Z)HPR

Single complex and double complex HPR.

Description

Computes the packed hermitian rank-1 operation:

$$A \leftarrow \alpha x * x^H + A$$

BLAS Interface

```
void chpr(const char *UPLO, const qml_long *N, const float *ALPHA,
          const qml_single_complex *X, const qml_long *INCX,
          qml_single_complex *AP);

void zhpr(const char *UPLO, const qml_long *N, const double *ALPHA,
          const qml_double_complex *X, const qml_long *INCX,
          qml_double_complex *AP);
```

CBLAS Interface

```
void cblas_chpr(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const float ALPHA, const qml_single_complex *X,
                const qml_long INCX, qml_single_complex *AP);

void cblas_zhpr(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const double ALPHA, const qml_double_complex *X,
                const qml_long INCX, qml_double_complex *AP);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	Input vector, must be at least: $(\mathtt{N}-1) * \lvert\mathtt{INCX}\rvert + 1$
INCX	Distance between individual elements in X
AP	Matrix A stored in packed triangular form, must be at least: $\mathtt{N} * (\mathtt{N}+1)/2$

(C|Z)HPR2

Single complex and double complex HPR2.

Description

Computes the hermitian rank-2 operation:

$$A \leftarrow \alpha x * y^H + conj(\alpha) y * x^H + A$$

BLAS Interface

```
void chpr2(const char *UPLO, const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *X, const qml_long *INCX,
           const qml_single_complex *Y, const qml_long *INCY,
           qml_single_complex *AP);

void zhpr2(const char *UPLO, const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *X, const qml_long *INCX,
           const qml_double_complex *Y, const qml_long *INCY,
           qml_double_complex *AP);
```

CBLAS Interface

```
void cblas_chpr2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *X,
                 const qml_long INCX, const qml_single_complex *Y,
                 const qml_long INCY, qml_single_complex *AP);

void cblas_zhpr2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *X,
                 const qml_long INCX, const qml_double_complex *Y,
                 const qml_long INCY, qml_double_complex *AP);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	First input vector, must be at least: $(\mathtt{N}-1) * \lvert\mathtt{INCX}\rvert + 1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1) * \lvert\mathtt{INCY}\rvert + 1$
INCY	Distance between individual elements in Y
AP	Matrix A stored in packed triangular form, must be at least: $\mathtt{N} * (\mathtt{N}+1)/2$

(S|D)SBMV

Single and double SBMV.

Description

Computes a symmetric banded matrix-vector product.

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void ssbmv(const char *UPLO, const qml_long *N, const qml_long *K,
           const float *ALPHA, const float *A, const qml_long *LDA,
           const float *X, const qml_long *INCX, const float *BETA,
           float *Y, const qml_long *INCY);

void dsbmv(const char *UPLO, const qml_long *N, const qml_long *K,
           const double *ALPHA, const double *A, const qml_long *LDA,
           const double *X, const qml_long *INCX, const double *BETA,
           double *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_ssbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_long K, const float ALPHA, const float *A,
                 const qml_long LDA, const float *X, const qml_long INCX,
                 const float BETA, float *Y, const qml_long INCY);

void cblas_dsbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const qml_long K, const double ALPHA, const double *A,
                 const qml_long LDA, const double *X, const qml_long INCX,
                 const double BETA, double *Y, const qml_long INCY);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
K	Number of super-diagonals of A, with $0 \leq$ K
ALPHA	Scalar multiplied with the matrix-vector product
A	Input matrix A
LDA	Leading dimension of matrix A, with LDA $\geq$ K $+ 1$
X	First input vector, must be at least: $(\text{N} - 1) * \lvert\text{INCX}\rvert + 1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second vector, must be at least: $(\text{N} - 1) * \lvert\text{INCY}\rvert + 1$
INCY	Distance between individual elements in Y

(S|D)SPMV

Single and double SPMV.

Description

Computes the symmetric packed matrix-vector operation:

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void sspmv(const char *UPLO, const qml_long *N, const float *ALPHA,
           const float *AP, const float *X, const qml_long *INCX,
           const float *BETA, float *Y, const qml_long *INCY);

void dspmv(const char *UPLO, const qml_long *N, const double *ALPHA,
           const double *AP, const double *X, const qml_long *INCX,
           const double *BETA, double *Y, const qml_long *INCY);
```

CBLAS Interface

```
void cblas_sspmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const float ALPHA, const float *AP, const float *X,
                 const qml_long INCX, const float BETA, float *Y,
                 const qml_long INCY);

void cblas_dspmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const double ALPHA, const double *AP, const double *X,
                 const qml_long INCX, const double BETA, double *Y,
                 const qml_long INCY);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the matrix-vector product
AP	Matrix A stored in packed triangular form, must be at least: $\mathtt{N} * (\mathtt{N}+1)/2$
ALPHA	Scalar multiplied with the matrix-vector product
X	First input vector, must be at least: $(\mathtt{N}-1) * \lvert\mathtt{INCX}\rvert + 1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second vector, must be at least: $(\mathtt{M}-1) * \lvert\mathtt{INCY}\rvert + 1$
INCY	Distance between individual elements in Y

(S|D)SPR

Single and double SPR.

Description

Computes the symmetric packed rank-1 operation:

$$A \leftarrow \alpha x * x^T + A$$

BLAS Interface

```
void sspr(const char *UPLO, const qml_long *N, const float *ALPHA,
          const float *X, const qml_long *INCX, float *AP);

void dspr(const char *UPLO, const qml_long *N, const double *ALPHA,
          const double *X, const qml_long *INCX, double *AP);
```

CBLAS Interface

```
void cblas_sspr(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const float ALPHA, const float *X, const qml_long INCX, float *AP);

void cblas_dspr(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const double ALPHA, const double *X, const qml_long INCX, double *AP);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	Input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
AP	Matrix A stored in packed triangular form, must be at least: $\mathtt{N}*(\mathtt{N}+1)/2$

(S|D)SPR2

Single and double SPR2.

Description

Computes the symmetric packed rank-2 operation:

$$A \leftarrow \alpha x * y^T + \alpha y * x^T + A$$

BLAS Interface

```
void sspr2(const char *UPLO, const qml_long *N, const float *ALPHA,
           const float *X, const qml_long *INCX, const float *Y,
           const qml_long *INCY, float *AP);

void dspr2(const char *UPLO, const qml_long *N, const double *ALPHA,
           const double *X, const qml_long *INCX, const double *Y,
           const qml_long *INCY, double *AP);
```

CBLAS Interface

```
void cblas_sspr2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const float ALPHA, const float *X, const qml_long INCX,
                 const float *Y, const qml_long INCY, float *AP);
```

```
void cblas_dspr2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                 const double ALPHA, const double *X, const qml_long INCX,
                 const double *Y, const qml_long INCY, double *AP);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
AP	Matrix A stored in packed triangular form, must be at least: $\mathtt{N}*(\mathtt{N}+1)/2$

(S|D)SYMV

Single and double SYMV.

Description

Computes a symmetric matrix-vector product.

$$y \leftarrow \alpha A * x + \beta y$$

BLAS Interface

```
void ssymv(const char *UPLO, const qml_long *N, const float *ALPHA,
           const float *A, const qml_long *LDA, const float *X,
           const qml_long *INCX, const float *BETA, float *Y,
           const qml_long *INCY);

void dsymv(const char *UPLO, const qml_long *N, const double *ALPHA,
           const double *A, const qml_long *LDA, const double *X,
           const qml_long *INCX, const double *BETA, double *Y,
           const qml_long *INCY);
```

CBLAS Interface

```
void cblas_ssymv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const qml_long N, const float ALPHA, const float *A,
                 const qml_long LDA, const float *X, const qml_long INCX,
                 const float BETA, float *Y, const qml_long INCY);

void cblas_dsymv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const qml_long N, const double ALPHA, const double *A,
                 const qml_long LDA, const double *X, const qml_long INCX,
                 const double BETA, double *Y, const qml_long INCY);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the matrix-vector product
A	Input matrix A
LDA	Leading dimension of matrix A
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
BETA	Scalar multiplied with vector Y
Y	Second vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y

(S|D)SYR

Single and double SYR.

Description

Computes the symmetric rank-1 operation:

$$A \leftarrow \alpha x * x^T + A$$

BLAS Interface

```
void ssyr(const char *UPLO, const qml_long *N, const float *ALPHA,
          const float *X, const qml_long *INCX, float *A, const qml_long *LDA);

void dsyr(const char *UPLO, const qml_long *N, const double *ALPHA,
          const double *X, const qml_long *INCX, double *A, const qml_long *LDA);
```

CBLAS Interface

```
void cblas_ssyr(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const float ALPHA, const float *X, const qml_long INCX, float *A,
                const qml_long LDA);

void cblas_dsyr(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO, const qml_long N,
                const double ALPHA, const double *X, const qml_long INCX, double *A,
                const qml_long LDA);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	Input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
A	Input matrix A
LDA	Leading dimension of matrix A

(S|D)SYR2

Single and double SYR2.

Description

Computes the symmetric rank-2 operation:

$$A \leftarrow \alpha x * y^T + \alpha y * x^T + A$$

BLAS Interface

```
void ssyr2(const char *UPLO, const qml_long *N, const float *ALPHA,
           const float *X, const qml_long *INCX, const float *Y,
           const qml_long *INCY, float *A, const qml_long *LDA);

void dsyr2(const char *UPLO, const qml_long *N, const double *ALPHA,
           const double *X, const qml_long *INCX, const double *Y,
           const qml_long *INCY, double *A, const qml_long *LDA);
```

CBLAS Interface

```
void cblas_ssyr2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const qml_long N, const float ALPHA, const float *X,
                 const qml_long INCX, const float *Y, const qml_long INCY,
                 float *A, const qml_long LDA);

void cblas_dsyr2(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const qml_long N, const double ALPHA, const double *X,
                 const qml_long INCX, const double *Y, const qml_long INCY,
                 double *A, const qml_long LDA);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
N	Order of matrix A
ALPHA	Scalar multiplied with the vector-vector product
X	First input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCX}\rvert+1$
INCX	Distance between individual elements in X
Y	Second input vector, must be at least: $(\mathtt{N}-1)*\lvert\mathtt{INCY}\rvert+1$
INCY	Distance between individual elements in Y
A	Input matrix A
LDA	Leading dimension of matrix A

(S|D|C|Z)TBMV

Single, double, single complex, and double complex TBMV.

Description

Computes a triangle banded matrix-vector product.

$$x \leftarrow \alpha A * x$$

or

$$x \leftarrow \alpha A^T * x$$

or

$$x \leftarrow \alpha A^H * x$$

BLAS Interface

```
void stbmv(const char *UPLO, const char *TRANS, const char *DIAG,
           const qml_long *N, const qml_long *K, const float *A,
           const qml_long *LDA, float *X, const qml_long *INCX);

void dtbmv(const char *UPLO, const char *TRANS, const char *DIAG,
           const qml_long *N, const qml_long *K, const double *A,
           const qml_long *LDA, double *X, const qml_long *INCX);

void ctbmv(const char *UPLO, const char *TRANS, const char *DIAG,
           const qml_long *N, const qml_long *K, const qml_single_complex *A,
           const qml_long *LDA, qml_single_complex *X, const qml_long *INCX);

void ztbmv(const char *UPLO, const char *TRANS, const char *DIAG,
           const qml_long *N, const qml_long *K, const qml_double_complex *A,
           const qml_long *LDA, qml_double_complex *X, const qml_long *INCX);
```

CBLAS Interface

```
void cblas_stbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K, const float *A,
                 const qml_long LDA, float *X, const qml_long INCX);

void cblas_dtbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K, const double *A,
                 const qml_long LDA, double *X, const qml_long INCX);

void cblas_ctbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K,
                 const qml_single_complex *A, const qml_long LDA,
                 qml_single_complex *X, const qml_long INCX);

void cblas_ztbmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K,
                 const qml_double_complex *A, const qml_long LDA,
                 qml_double_complex *X, const qml_long INCX);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
TRANS	Specifies which product is computed:
	Non-Transpose: $x \leftarrow \alpha A * x$
	Transpose: $x \leftarrow \alpha A^T * x$
	Complex Conjugate Transpose: $x \leftarrow \alpha A^H * x$
DIAG	Whether the diagonal is unit or not
N	Order of matrix A
K	Number of super- or sub-diagonals of A, with $0 \leq \texttt{K}$
ALPHA	Scalar multiplied with the matrix-vector product
A	Input matrix A
LDA	Leading dimension of matrix A, with $\texttt{LDA} \geq \texttt{K} + 1$
X	Vector, must be at least: $(\texttt{N} - 1) * \lvert\texttt{INCX}\rvert + 1$
INCX	Distance between individual elements in X

(S|D|C|Z)TBSV

Single, double, single complex, and double complex TBSV.

Description

Solves one of the following systems of equations for triangle banded A:

$$A * x = b$$

or

$$A^T * x = b$$

or

$$A^H * x = b$$

BLAS Interface

```
void stbsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_long *K, const float *A, const qml_long *LDA, float *X,
           const qml_long *INCX);

void dtbsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_long *K, const double *A, const qml_long *LDA, double *X,
           const qml_long *INCX);

void ctbsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_long *K, const qml_single_complex *A, const qml_long *LDA,
           qml_single_complex *X, const qml_long *INCX);

void ztbsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_long *K, const qml_double_complex *A, const qml_long *LDA,
           qml_double_complex *X, const qml_long *INCX);
```

CBLAS Interface

```
void cblas_stbsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K, const float *A,
                 const qml_long LDA, float *X, const qml_long INCX);

void cblas_dtbsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K, const double *A,
                 const qml_long LDA, double *X, const qml_long INCX);

void cblas_ctbsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K, const qml_single_complex *A,
                 const qml_long LDA, qml_single_complex *X, const qml_long INCX);

void cblas_ztbsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_long K, const qml_double_complex *A,
                 const qml_long LDA, qml_double_complex *X, const qml_long INCX);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
TRANS	Specifies which equation is solved:
	Non-Transpose: $A * x = b$
	Transpose: $A^T * x = b$
	Complex Conjugate Transpose: $A^H * x = b$
DIAG	Whether the diagonal is unit or not
N	Order of matrix A
K	Number of super- or sub-diagonals of A, with $0 \leq$ K
A	Input matrix A
LDA	Leading dimension of matrix A, with LDA $\geq$ K $+ 1$
X	Input vector, must be at least: $(\text{N} - 1) * \lvert\text{INCX}\rvert + 1$
	On output, overwritten with solution vector
INCX	Distance between individual elements in X

(S|D|C|Z)TPMV

Single, double, single complex, and double complex TPMV.

Description

Computes a packed banded triangle matrix-vector product.

$$x \leftarrow A * x$$

or

$$x \leftarrow A^T * x$$

or

$$x \leftarrow A^H * x$$

BLAS Interface

```
void stpmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const float *AP, float *X, const qml_long *INCX);

void dtpmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const double *AP, double *X, const qml_long *INCX);

void ctpmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_single_complex *AP, qml_single_complex *X, const qml_long *INCX);

void ztpmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_double_complex *AP, qml_double_complex *X, const qml_long *INCX);
```

CBLAS Interface

```
void cblas_stpmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const float *AP, float *X, const qml_long INCX);

void cblas_dtpmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const double *AP, double *X, const qml_long INCX);

void cblas_ctpmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG, const qml_long N,
                 const qml_single_complex *AP, qml_single_complex *X,
                 const qml_long INCX);

void cblas_ztpmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG, const qml_long N,
                 const qml_double_complex *AP, qml_double_complex *X,
                 const qml_long INCX);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
TRANS	Specifies which product is computed:
	Non-Transpose: $x \leftarrow A * x$
	Transpose: $x \leftarrow A^T * x$
	Complex Conjugate Transpose: $x \leftarrow A^H * x$
DIAG	Whether the diagonal is unit or not
N	Order of matrix A
AP	Matrix A stored in packed triangular form, must be at least: $\texttt{N} * (\texttt{N}+1)/2$
X	Vector, must be at least: $(\texttt{N}-1) * \lvert\texttt{INCX}\rvert + 1$
INCX	Distance between individual elements in X

(S|D|C|Z)TPSV

Single, double, single complex, and double complex TPSV.

Description

Solves one of the following systems of equations for packed banded triangle A:

$$A * x = b$$

or

$$A^T * x = b$$

or

$$A^H * x = b$$

BLAS Interface

```
void stpsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const float *AP, float *X, const qml_long *INCX);

void dtpsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const double *AP, double *X, const qml_long *INCX);

void ctpsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_single_complex *AP, qml_single_complex *X,
           const qml_long *INCX);

void ztpsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_double_complex *AP, qml_double_complex *X,
           const qml_long *INCX);
```

CBLAS Interface

```
void cblas_stpsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const float *AP, float *X,
                 const qml_long INCX);

void cblas_dtpsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const double *AP, double *X,
                 const qml_long INCX);

void cblas_ctpsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_single_complex *AP,
                 qml_single_complex *X, const qml_long INCX);

void cblas_ztpsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_double_complex *AP,
                 qml_double_complex *X, const qml_long INCX);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
TRANS	Specifies which equation is solved:
	Non-Transpose: $A * x = b$
	Transpose: $A^T * x = b$
	Complex Conjugate Transpose: $A^H * x = b$
DIAG	Whether the diagonal is unit or not
N	Order of matrix A
AP	Matrix A stored in packed triangular form, must be at least: $\texttt{N} * (\texttt{N} + 1)/2$
X	Input vector, must be at least: $(\texttt{N} - 1) * \lvert\texttt{INCX}\rvert + 1$
	On output, overwritten with solution vector
INCX	Distance between individual elements in X

(S|D|C|Z)TRMV

Single, double, single complex, and double complex TRMV.

Description

Computes a triangle matrix-vector product.

$$x \leftarrow A * x$$

or

$$x \leftarrow A^T * x$$

or

$$x \leftarrow A^H * x$$

BLAS Interface

```
void strmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const float *A, const qml_long *LDA, float *X, const qml_long *INCX);

void dtrmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const double *A, const qml_long *LDA, double *X, const qml_long *INCX);

void ctrmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_single_complex *A, const qml_long *LDA, qml_single_complex *X,
           const qml_long *INCX);

void ztrmv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_double_complex *A, const qml_long *LDA, qml_double_complex *X,
           const qml_long *INCX);
```

CBLAS Interface

```
void cblas_strmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
```

```
                const qml_long N, const float *A, const qml_long LDA,
                float *X, const qml_long INCX);

void cblas_dtrmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                const qml_long N, const double *A, const qml_long LDA,
                double *X, const qml_long INCX);

void cblas_ctrmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                const qml_long N, const qml_single_complex *A,
                const qml_long LDA, qml_single_complex *X, const qml_long INCX);

void cblas_ztrmv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                const qml_long N, const qml_double_complex *A,
                const qml_long LDA, qml_double_complex *X, const qml_long INCX);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
TRANS	Specifies which product is computed:
	Non-Transpose: $x \leftarrow A * x$
	Transpose: $x \leftarrow A^T * x$
	Complex Conjugate Transpose: $x \leftarrow A^H * x$
DIAG	Whether the diagonal is unit or not
N	Order of matrix A
A	Input matrix A
LDA	Leading dimension of matrix A
X	Vector, must be at least: $(\texttt{N} - 1) * \lvert\texttt{INCX}\rvert + 1$
INCX	Distance between individual elements in X

(S|D|C|Z)TRSV

Single, double, single complex, and double complex TRSV.

Description

Solves one of the following systems of equations for triangle A:

$$A * x = b$$

or

$$A^T * x = b$$

or

$$A^H * x = b$$

BLAS Interface

```
void strsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const float *A, const qml_long *LDA, float *X, const qml_long *INCX);

void dtrsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const double *A, const qml_long *LDA, double *X, const qml_long *INCX);

void ctrsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_single_complex *A, const qml_long *LDA,
           qml_single_complex *X, const qml_long *INCX);

void ztrsv(const char *UPLO, const char *TRANS, const char *DIAG, const qml_long *N,
           const qml_double_complex *A, const qml_long *LDA,
           qml_double_complex *X, const qml_long *INCX);
```

CBLAS Interface

```
void cblas_strsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const float *A, const qml_long LDA,
                 float *X, const qml_long INCX);

void cblas_dtrsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const double *A, const qml_long LDA,
                 double *X, const qml_long INCX);

void cblas_ctrsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_single_complex *A,
                 const qml_long LDA, qml_single_complex *X,
                 const qml_long INCX);

void cblas_ztrsv(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const CBLAS_DIAG DIAG,
                 const qml_long N, const qml_double_complex *A,
                 const qml_long LDA, qml_double_complex *X,
                 const qml_long INCX);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix A will be used
TRANS	Specifies which equation is solved:
	Non-Transpose: $A * x = b$
	Transpose: $A^T * x = b$
	Complex Conjugate Transpose: $A^H * x = b$
DIAG	Whether the diagonal is unit or not
N	Order of matrix A
A	Input matrix A
LDA	Leading dimension of matrix A, with $\texttt{LDA} \geq \texttt{K} + 1$
X	Input vector, must be at least: $(\texttt{N} - 1) * \lvert\texttt{INCX}\rvert + 1$
	On output, overwritten with solution vector
INCX	Distance between individual elements in X

CHAPTER

SEVENTEEN

BLAS 3 FUNCTIONS

The BLAS 3 consists of matrix-matrix operations. Below are the BLAS 3 functions supported by QML.

(S|D|C|Z)GEMM

Single, double, single complex, and double complex GEMM.

Description

Computes a general matrix-matrix product.

$$C \leftarrow \alpha A * B + \beta C$$

BLAS Interface

```
void sgemm(const char *TRANSA, const char *TRANSB, const qml_long *M,
           const qml_long *N, const qml_long *K, const float *ALPHA,
           const float *A, const qml_long *LDA, const float *B,
           const qml_long *LDB, const float *BETA, float *C,
           const qml_long *LDC);

void dgemm(const char *TRANSA, const char *TRANSB, const qml_long *M,
           const qml_long *N, const qml_long *K, const double *ALPHA,
           const double *A, const qml_long *LDA, const double *B,
           const qml_long *LDB, const double *BETA, double *C,
           const qml_long *LDC);

void cgemm(const char *TRANSA, const char *TRANSB, const qml_long *M,
           const qml_long *N, const qml_long *K,
           const qml_single_complex *ALPHA, const qml_single_complex *A,
           const qml_long *LDA, const qml_single_complex *B,
           const qml_long *LDB, const qml_single_complex *BETA,
           qml_single_complex *C, const qml_long *LDC);

void zgemm(const char *TRANSA, const char *TRANSB, const qml_long *M,
           const qml_long *N, const qml_long *K,
           const qml_double_complex *ALPHA, const qml_double_complex *A,
           const qml_long *LDA, const qml_double_complex *B,
           const qml_long *LDB, const qml_double_complex *BETA,
           qml_double_complex *C, const qml_long *LDC);
```

CBLAS Interface

```
void cblas_sgemm(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_TRANSPOSE TRANSB, const qml_long M,
                 const qml_long N, const qml_long K, const float ALPHA,
                 const float *A, const qml_long LDA, const float *B,
                 const qml_long LDB, const float BETA, float *C,
                 const qml_long LDC);

void cblas_dgemm(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_TRANSPOSE TRANSB, const qml_long M,
                 const qml_long N, const qml_long K, const double ALPHA,
                 const double *A, const qml_long LDA, const double *B,
                 const qml_long LDB, const double BETA, double *C,
                 const qml_long LDC);

void cblas_cgemm(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_TRANSPOSE TRANSB, const qml_long M,
                 const qml_long N, const qml_long K,
                 const qml_single_complex *ALPHA,
                 const qml_single_complex *A, const qml_long LDA,
                 const qml_single_complex *B, const qml_long LDB,
                 const qml_single_complex *BETA, qml_single_complex *C,
                 const qml_long LDC);

void cblas_zgemm(const CBLAS_ORDER ORDER, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_TRANSPOSE TRANSB, const qml_long M,
                 const qml_long N, const qml_long K,
                 const qml_double_complex *ALPHA,
                 const qml_double_complex *A, const qml_long LDA,
                 const qml_double_complex *B, const qml_long LDB,
                 const qml_double_complex *BETA, qml_double_complex *C,
                 const qml_long LDC);
```

Arguments

TRANSA	Specifies how to read matrix A
	Possible values: Non-Transpose, Tranpose, Complex Conjugate Transpose
TRANSB	Specifies how to read matrix B
	Possible values: Non-Transpose, Tranpose, Complex Conjugate Transpose
M	Number of rows of matrix A
N	Number of columns of matrix B
K	Number of columns of matrix A
ALPHA	Scalar multiplied with the matrix-matrix product
A	Input matrix A
LDA	Leading dimension of matrix A
B	Input matrix B
LDB	Leading dimension of matrix B
BETA	Scalar multiplied with matrix C
C	Result matrix
LDC	Leading dimension of matrix C

(C|Z)HEMM

Single complex and double complex HEMM.

Description

Computes a complex matrix-matrix product with hermitian matrix A

$$C \leftarrow \alpha A * B + \beta C$$

or

$$C \leftarrow \alpha B * A + \beta C$$

BLAS Interface

```
void chemm(const char *SIDE, const char *UPLO, const qml_long *M,
           const qml_long *N, const qml_single_complex *ALPHA,
           const qml_single_complex *A, const qml_long *LDA,
           const qml_single_complex *B, const qml_long *LDB,
           const qml_single_complex *BETA, qml_single_complex *C,
           const qml_long *LDC);

void zhemm(const char *SIDE, const char *UPLO, const qml_long *M,
           const qml_long *N, const qml_double_complex *ALPHA,
           const qml_double_complex *A, const qml_long *LDA,
           const qml_double_complex *B, const qml_long *LDB,
           const qml_double_complex *BETA, qml_double_complex *C,
           const qml_long *LDC);
```

CBLAS Interface

```
void cblas_chemm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const qml_long M,
                 const qml_long N, const qml_single_complex *ALPHA,
                 const qml_single_complex *A, const qml_long LDA,
                 const qml_single_complex *B, const qml_long LDB,
                 const qml_single_complex *BETA, qml_single_complex *C,
                 const qml_long LDC);

void cblas_zhemm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const qml_long M,
                 const qml_long N, const qml_double_complex *ALPHA,
                 const qml_double_complex *A, const qml_long LDA,
                 const qml_double_complex *B, const qml_long LDB,
                 const qml_double_complex *BETA, qml_double_complex *C,
                 const qml_long LDC);
```

Arguments

SIDE	Specify which side of the product matrix A should be
	For left, computes: $C \leftarrow \alpha A * B + \beta C$
	For right, computes: $C \leftarrow \alpha B * A + \beta C$
UPLO	Specify whether the upper or lower triangle of matrix A will be read
M	Number of rows of matrix C
N	Number of columns of matrix C
ALPHA	Scalar multiplied with the matrix-matrix product
A	Hermitian input matrix A
LDA	Leading dimension of matrix A
B	Input matrix B
LDB	Leading dimension of matrix B
BETA	Scalar multiplied with matrix C
C	Result matrix
LDC	Leading dimension of matrix C

(C|Z)HERK

Single complex and double complex HERK.

Description

Performs one of the following rank-k updates on the hermitian matrix C.

$$C = \alpha A * A^H + \beta C$$

or

$$C = \alpha A^H * A + \beta C$$

BLAS Interface

```
void cherk(const char *UPLO, const char *TRANS, const qml_long *N,
           const qml_long *K, const float *ALPHA,
           const qml_single_complex *A, const qml_long *LDA,
           const float *BETA, qml_single_complex *C, const qml_long *LDC);

void zherk(const char *UPLO, const char *TRANS, const qml_long *N,
           const qml_long *K, const double *ALPHA,
           const qml_double_complex *A, const qml_long *LDA,
           const double *BETA, qml_double_complex *C, const qml_long *LDC);
```

CBLAS Interface

```
void cblas_cherk(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const qml_long N,
                 const qml_long K, const float ALPHA,
                 const qml_single_complex *A, const qml_long LDA,
                 const float BETA, qml_single_complex *C,
                 const qml_long LDC);
```

```
void cblas_zherk(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const qml_long N,
                 const qml_long K, const double ALPHA,
                 const qml_double_complex *A, const qml_long LDA,
                 const double BETA, qml_double_complex *C,
                 const qml_long LDC);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix C will be used
TRANS	Specifies which rank-k update is performed:
	Non-Transpose: $C = \alpha A * A^H + \beta C$
	Complex Conjugate Transpose: $C = \alpha A^H * A + \beta C$
N	Order of matrix C
K	For Non-Transpose, K is the number of columns of matrix A
	For Transpose and Complex Conjugate Transpose, K is the number of rows of matrix A
ALPHA	Scalar multiplied with the matrix-matrix product
A	Input matrix A
LDA	Leading dimension of matrix A
BETA	Scalar multiplied with matrix C
C	Hermitian matrix C
LDC	Leading dimension of matrix C

(C|Z)HER2K

Single complex and double complex HER2K.

Description

Performs one of the following rank-2k updates on the hermitian matrix C.

$$C \leftarrow \alpha A * B^H + conj(\alpha) B * A^H + \beta C$$

or

$$C \leftarrow \alpha A^H * B + conj(\alpha) B^H * A + \beta C$$

BLAS Interface

```
void cher2k(const char *UPLO, const char *TRANS, const qml_long *N,
            const qml_long *K, const qml_single_complex *ALPHA,
            const qml_single_complex *A, const qml_long *LDA,
            const qml_single_complex *B, const qml_long *LDB, const float *BETA,
            qml_single_complex *C, const qml_long *LDC);

void zher2k(const char *UPLO, const char *TRANS, const qml_long *N,
            const qml_long *K, const qml_double_complex *ALPHA,
            const qml_double_complex *A, const qml_long *LDA,
            const qml_double_complex *B, const qml_long *LDB, const double *BETA,
            qml_double_complex *C, const qml_long *LDC);
```

CBLAS Interface

```
void cblas_cher2k(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                  const CBLAS_TRANSPOSE TRANS, const qml_long N,
                  const qml_long K, const qml_single_complex *ALPHA,
                  const qml_single_complex *A, const qml_long LDA,
                  const qml_single_complex *B, const qml_long LDB,
                  const float BETA, qml_single_complex *C, const qml_long LDC);

void cblas_zher2k(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                  const CBLAS_TRANSPOSE TRANS, const qml_long N,
                  const qml_long K, const qml_double_complex *ALPHA,
                  const qml_double_complex *A, const qml_long LDA,
                  const qml_double_complex *B, const qml_long LDB,
                  const double BETA, qml_double_complex *C, const qml_long LDC);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix C will be used
TRANS	Specifies which rank-k update is performed:
	Non-Transpose: $C \leftarrow \alpha A * B^H + conj(\alpha) B * A^H + \beta C$
	Complex Conjugate Transpose: $C \leftarrow \alpha A^H * B + conj(\alpha) B^H * A + \beta C$
N	Order of matrix C
K	For Non-Transpose, K is the number of columns of matrix A and B
	For Transpose and Complex Conjugate Transpose, K is the number of rows of matrix A and B
ALPHA	Scalar multiplied with the matrix-matrix product
A	Input matrix A
LDA	Leading dimension of matrix A
B	Input matrix B
LDB	Leading dimension of matrix B
BETA	Scalar multiplied with matrix C
C	Hermitian matrix C
LDC	Leading dimension of matrix C

(S|D|C|Z)SYMM

Single, double, single complex, and double complex SYMM.

Description

Computes a matrix-matrix product with the symmetric matrix A

$$C \leftarrow \alpha A * B + \beta C$$

or

$$C \leftarrow \alpha B * A + \beta C$$

BLAS Interface

```
void ssymm(const char *SIDE, const char *UPLO, const qml_long *M,
           const qml_long *N, const float *ALPHA, const float *A,
```

```
          const qml_long *LDA, const float *B, const qml_long *LDB,
          const float *BETA, float *C, const qml_long *LDC);

void dsymm(const char *SIDE, const char *UPLO, const qml_long *M,
          const qml_long *N, const double *ALPHA, const double *A,
          const qml_long *LDA, const double *B, const qml_long *LDB,
          const double *BETA, double *C, const qml_long *LDC);

void csymm(const char *SIDE, const char *UPLO, const qml_long *M,
          const qml_long *N, const qml_single_complex *ALPHA,
          const qml_single_complex *A, const qml_long *LDA,
          const qml_single_complex *B, const qml_long *LDB,
          const qml_single_complex *BETA, qml_single_complex *C,
          const qml_long *LDC);

void zsymm(const char *SIDE, const char *UPLO, const qml_long *M,
          const qml_long *N, const qml_double_complex *ALPHA,
          const qml_double_complex *A, const qml_long *LDA,
          const qml_double_complex *B, const qml_long *LDB,
          const qml_double_complex *BETA, qml_double_complex *C,
          const qml_long *LDC);
```

CBLAS Interface

```
void cblas_ssymm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const qml_long M, const qml_long N,
                 const float ALPHA, const float *A, const qml_long LDA,
                 const float *B, const qml_long LDB, const float BETA,
                 float *C, const qml_long LDC);

void cblas_dsymm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const qml_long M, const qml_long N,
                 const double ALPHA, const double *A, const qml_long LDA,
                 const double *B, const qml_long LDB, const double BETA,
                 double *C, const qml_long LDC);

void cblas_csymm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const qml_long M, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *A,
                 const qml_long LDA, const qml_single_complex *B,
                 const qml_long LDB, const qml_single_complex *BETA,
                 qml_single_complex *C, const qml_long LDC);

void cblas_zsymm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const qml_long M, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *A,
                 const qml_long LDA, const qml_double_complex *B,
                 const qml_long LDB, const qml_double_complex *BETA,
                 qml_double_complex *C, const qml_long LDC);
```

Arguments

SIDE	Specify which side of the product matrix A should be
	For left, computes: $C \leftarrow \alpha A * B + \beta C$
	For right, computes: $C \leftarrow \alpha B * A + \beta C$
UPLO	Specify whether the upper or lower triangle of matrix A will be read
M	Number of rows of matrix C
N	Number of columns of matrix C
ALPHA	Scalar multiplied with the matrix-matrix product
A	Symmetric input matrix A
LDA	Leading dimension of matrix A
B	Input matrix B
LDB	Leading dimension of matrix B
BETA	Scalar multiplied with matrix C
C	Result matrix
LDC	Leading dimension of matrix C

(S|D|C|Z)SYRK

Single, double, single complex, and double complex SYRK.

Description

Performs one of the following rank-k updates on the symmetric matrix C.

$$C \leftarrow \alpha A * A^T + \beta C$$

or

$$C \leftarrow \alpha A^T * A + \beta C$$

BLAS Interface

```
void ssyrk(const char *UPLO, const char *TRANS, const qml_long *N,
           const qml_long *K, const float *ALPHA, const float *A,
           const qml_long *LDA, const float *BETA, float *C,
           const qml_long *LDC);

void dsyrk(const char *UPLO, const char *TRANS, const qml_long *N,
           const qml_long *K, const double *ALPHA, const double *A,
           const qml_long *LDA, const double *BETA, double *C,
           const qml_long *LDC);

void csyrk(const char *UPLO, const char *TRANS, const qml_long *N,
           const qml_long *K, const qml_single_complex *ALPHA,
           const qml_single_complex *A, const qml_long *LDA,
           const qml_single_complex *BETA, qml_single_complex *C,
           const qml_long *LDC);

void zsyrk(const char *UPLO, const char *TRANS, const qml_long *N,
           const qml_long *K, const qml_double_complex *ALPHA,
           const qml_double_complex *A, const qml_long *LDA,
           const qml_double_complex *BETA, qml_double_complex *C,
           const qml_long *LDC);
```

CBLAS Interface

```
void cblas_ssyrk(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const qml_long N,
                 const qml_long K, const float ALPHA, const float *A,
                 const qml_long LDA, const float BETA, float *C,
                 const qml_long LDC);

void cblas_dsyrk(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const qml_long N,
                 const qml_long K, const double ALPHA, const double *A,
                 const qml_long LDA, const double BETA, double *C,
                 const qml_long LDC);

void cblas_csyrk(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const qml_long N,
                 const qml_long K, const qml_single_complex *ALPHA,
                 const qml_single_complex *A, const qml_long LDA,
                 const qml_single_complex *BETA, qml_single_complex *C,
                 const qml_long LDC);

void cblas_zsyrk(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                 const CBLAS_TRANSPOSE TRANS, const qml_long N,
                 const qml_long K, const qml_double_complex *ALPHA,
                 const qml_double_complex *A, const qml_long LDA,
                 const qml_double_complex *BETA, qml_double_complex *C,
                 const qml_long LDC);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix C will be used
TRANS	Specifies which rank-k update is performed:
	Non-Transpose: $C \leftarrow \alpha A * A^T + \beta C$
	Transpose: $C \leftarrow \alpha A^T * A + \beta C$
	Complex Conjugate Transpose: $C \leftarrow \alpha A^T * A + \beta C$
N	Order of matrix C
K	For Non-Transpose, K is the number of columns of matrix A
	For Transpose and Complex Conjugate Transpose, K is the number of rows of matrix A
ALPHA	Scalar multiplied with the matrix-matrix product
A	Input matrix A
LDA	Leading dimension of matrix A
BETA	Scalar multiplied with matrix C
C	Symmetric matrix C
LDC	Leading dimension of matrix C

(S|D|C|Z)SYR2K

Single, double, single complex, and double complex SYR2K.

Description

Performs one of the following rank-2k updates on the symmetric matrix C.

$$C \leftarrow \alpha A * B^T + \alpha B * A^T + \beta C$$

or

$$C \leftarrow \alpha A^T * B + \alpha B^T * A + \beta C$$

BLAS Interface

```
void ssyr2k(const char *UPLO, const char *TRANS, const qml_long *N,
            const qml_long *K, const float *ALPHA, const float *A,
            const qml_long *LDA, const float *B, const qml_long *LDB,
            const float *BETA, float *C, const qml_long *LDC);

void dsyr2k(const char *UPLO, const char *TRANS, const qml_long *N,
            const qml_long *K, const double *ALPHA, const double *A,
            const qml_long *LDA, const double *B, const qml_long *LDB,
            const double *BETA, double *C, const qml_long *LDC);

void csyr2k(const char *UPLO, const char *TRANS, const qml_long *N,
            const qml_long *K, const qml_single_complex *ALPHA,
            const qml_single_complex *A, const qml_long *LDA,
            const qml_single_complex *B, const qml_long *LDB,
            const qml_single_complex *BETA, qml_single_complex *C,
            const qml_long *LDC);

void zsyr2k(const char *UPLO, const char *TRANS, const qml_long *N,
            const qml_long *K, const qml_double_complex *ALPHA,
            const qml_double_complex *A, const qml_long *LDA,
            const qml_double_complex *B, const qml_long *LDB,
            const qml_double_complex *BETA, qml_double_complex *C,
            const qml_long *LDC);
```

CBLAS Interface

```
void cblas_ssyr2k(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                  const CBLAS_TRANSPOSE TRANS, const qml_long N,
                  const qml_long K, const float ALPHA, const float *A,
                  const qml_long LDA, const float *B, const qml_long LDB,
                  const float BETA, float *C, const qml_long LDC);

void cblas_dsyr2k(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                  const CBLAS_TRANSPOSE TRANS, const qml_long N,
                  const qml_long K, const double ALPHA, const double *A,
                  const qml_long LDA, const double *B, const qml_long LDB,
                  const double BETA, double *C, const qml_long LDC);

void cblas_csyr2k(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                  const CBLAS_TRANSPOSE TRANS, const qml_long N,
                  const qml_long K, const qml_single_complex *ALPHA,
                  const qml_single_complex *A, const qml_long LDA,
                  const qml_single_complex *B, const qml_long LDB,
                  const qml_single_complex *BETA, qml_single_complex *C,
                  const qml_long LDC);
```

```
void cblas_zsyr2k(const CBLAS_ORDER ORDER, const CBLAS_UPLO UPLO,
                  const CBLAS_TRANSPOSE TRANS, const qml_long N,
                  const qml_long K, const qml_double_complex *ALPHA,
                  const qml_double_complex *A, const qml_long LDA,
                  const qml_double_complex *B, const qml_long LDB,
                  const qml_double_complex *BETA, qml_double_complex *C,
                  const qml_long LDC);
```

Arguments

UPLO	Specify whether the upper or lower triangle of matrix C will be used
TRANS	Specifies which rank-k update is performed:
	Non-Transpose: $C \leftarrow \alpha A * B^T + \alpha B * A^T + \beta C$
	Transpose: $C \leftarrow \alpha A^T * B + \alpha B^T * A + \beta C$
	Complex Conjugate Transpose: $C \leftarrow \alpha A^T * B + \alpha B^T * A + \beta C$
N	Order of matrix C
K	For Non-Transpose, K is the number of columns of matrix A and B
	For Transpose and Complex Conjugate Transpose, K is the number of rows of matrix A and B
ALPHA	Scalar multiplied with the matrix-matrix product
A	Input matrix A
LDA	Leading dimension of matrix A
B	Input matrix B
LDB	Leading dimension of matrix B
BETA	Scalar multiplied with matrix C
C	Symmetric matrix C
LDC	Leading dimension of matrix C

(S|D|C|Z)TRMM

Single, double, single complex, and double complex TRMM.

Description

Computes a matrix-matrix product with triangular matrix A.

$$B \leftarrow \alpha A * B$$

or

$$B \leftarrow \alpha B * A$$

BLAS Interface

```
void strmm(const char *SIDE, const char *UPLO, const char *TRANSA,
           const char *DIAG, const qml_long *M, const qml_long *N,
           const float *ALPHA, const float *A, const qml_long *LDA,
           float *B, const qml_long *LDB);

void dtrmm(const char *SIDE, const char *UPLO, const char *TRANSA,
           const char *DIAG, const qml_long *M, const qml_long *N,
```

```
            const double *ALPHA, const double *A, const qml_long *LDA,
            double *B, const qml_long *LDB);

void ctrmm(const char *SIDE, const char *UPLO, const char *TRANSA,
            const char *DIAG, const qml_long *M, const qml_long *N,
            const qml_single_complex *ALPHA, const qml_single_complex *A,
            const qml_long *LDA, qml_single_complex *B,
            const qml_long *LDB);

void ztrmm(const char *SIDE, const char *UPLO, const char *TRANSA,
            const char *DIAG, const qml_long *M, const qml_long *N,
            const qml_double_complex *ALPHA, const qml_double_complex *A,
            const qml_long *LDA, qml_double_complex *B,
            const qml_long *LDB);
```

CBLAS Interface

```
void cblas_strmm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const float ALPHA, const float *A, const qml_long LDA,
                 float *B, const qml_long LDB);

void cblas_dtrmm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const double ALPHA, const double *A, const qml_long LDA,
                 double *B, const qml_long LDB);

void cblas_ctrmm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *A,
                 const qml_long LDA, qml_single_complex *B,
                 const qml_long LDB);

void cblas_ztrmm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *A,
                 const qml_long LDA, qml_double_complex *B,
                 const qml_long LDB);
```

Arguments

SIDE	Specify which side of the product matrix A should be
	For left, computes: $B \leftarrow \alpha A * B$
	For right, computes: $B \leftarrow \alpha B * A$
UPLO	Specify whether the upper or lower triangle of matrix A will be read
TRANSA	Specifies how to read matrix A
	Possible values: Non-Transpose, Tranpose, Complex Conjugate Transpose
DIAG	Whether the diagonal is unit or not
M	Number of rows of matrix B
N	Number of columns of matrix B
ALPHA	Scalar multiplied with the matrix-matrix product
A	Input matrix A
LDA	Leading dimension of matrix A
B	Result matrix B
LDB	Leading dimension of matrix B

(S|D|C|Z)TRSM

Single, double, single complex, and double complex TRSM.

Description

Solves one of the following equations:

$$\alpha B = A * X$$

or

$$\alpha B = X * A$$

BLAS Interface

```
void strsm(const char *SIDE, const char *UPLO, const char *TRANSA,
           const char *DIAG, const qml_long *M, const qml_long *N,
           const float *ALPHA, const float *A, const qml_long *LDA,
           float *B, const qml_long *LDB);

void dtrsm(const char *SIDE, const char *UPLO, const char *TRANSA,
           const char *DIAG, const qml_long *M, const qml_long *N,
           const double *ALPHA, const double *A, const qml_long *LDA,
           double *B, const qml_long *LDB);

void ctrsm(const char *SIDE, const char *UPLO, const char *TRANSA,
           const char *DIAG, const qml_long *M, const qml_long *N,
           const qml_single_complex *ALPHA, const qml_single_complex *A,
           const qml_long *LDA, qml_single_complex *B,
           const qml_long *LDB);

void ztrsm(const char *SIDE, const char *UPLO, const char *TRANSA,
           const char *DIAG, const qml_long *M, const qml_long *N,
           const qml_double_complex *ALPHA, const qml_double_complex *A,
           const qml_long *LDA, qml_double_complex *B,
           const qml_long *LDB);
```

CBLAS Interface

```
void cblas_strsm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const float ALPHA, const float *A, const qml_long LDA,
                 float *B, const qml_long LDB);

void cblas_dtrsm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const double ALPHA, const double *A, const qml_long LDA,
                 double *B, const qml_long LDB);

void cblas_ctrsm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const qml_single_complex *ALPHA, const qml_single_complex *A,
                 const qml_long LDA, qml_single_complex *B,
                 const qml_long LDB);

void cblas_ztrsm(const CBLAS_ORDER ORDER, const CBLAS_SIDE SIDE,
                 const CBLAS_UPLO UPLO, const CBLAS_TRANSPOSE TRANSA,
                 const CBLAS_DIAG DIAG, const qml_long M, const qml_long N,
                 const qml_double_complex *ALPHA, const qml_double_complex *A,
                 const qml_long LDA, qml_double_complex *B,
                 const qml_long LDB);
```

Arguments

SIDE	Specify which side of the product matrix A should be
	For left, computes: $\alpha B = A * X$
	For right, computes: $\alpha B = X * A$
UPLO	Specify whether the upper or lower triangle of matrix A will be read
TRANSA	Specifies how to read matrix A
	Possible values: Non-Transpose, Tranpose, Complex Conjugate Transpose
DIAG	Whether the diagonal is unit or not
M	Number of rows of matrix B
N	Number of columns of matrix B
ALPHA	Scalar multiplied with the result matrix B
A	Input matrix A
LDA	Leading dimension of matrix A
B	Result matrix B
LDB	Leading dimension of matrix B

CHAPTER

EIGHTEEN

LAPACK FUNCTIONS

LAPACK is a collection of numerical algorithms implemented using the BLAS. Below are the LAPACK functions supported by QML.

(S|D|C|Z)BDSQR

Single, double, single complex, and double complex BDSQR.

Description

Computes the singular values and (optionally) the right/left singular vectors from the singular value decomposition of a square bidiagonal matrix.

A general dense matrix A can be bidiagonalized to matrix B before calling BDSQR.

$$A = U * B * VT$$

BDSQR then computes the singular value decomposition of B as:

$$B = Q * S * P^H$$

The result can be interpreted as the singular value decomposition of the original matrix A:

$$A = (U * Q) * S * (P^H * VT)$$

The matrices B, U, and VT are provided as input. BDSQR computes S, $U * Q$, and $P^H * VT$.

LAPACK Interface

```
void sbdsqr(const char *UPLO, const qml_long *N, const qml_long *NCVT,
    const qml_long *NRU, const qml_long *NCC, float *D, float *E,
    float *VT, const qml_long *LDVT, float *U, const qml_long *LDU,
    float *C, const qml_long *LDC, float *RWORK, qml_long *INFO);

void dbdsqr(const char *UPLO, const qml_long *N, const qml_long *NCVT,
    const qml_long *NRU, const qml_long *NCC, double *D, double *E,
    double *VT, const qml_long *LDVT, double *U, const qml_long *LDU,
    double *C, const qml_long *LDC, double *RWORK, qml_long *INFO);

void cbdsqr(const char *UPLO, const qml_long *N, const qml_long *NCVT,
    const qml_long *NRU, const qml_long *NCC, float *D, float *E,
    qml_single_complex *VT, const qml_long *LDVT, qml_single_complex *U,
```

```
    const qml_long *LDU, qml_single_complex *C, const qml_long *LDC,
    float *RWORK, qml_long *INFO);

void zbdsqr(const char *UPLO, const qml_long *N, const qml_long *NCVT,
    const qml_long *NRU, const qml_long *NCC, double *D, double *E,
    qml_double_complex *VT, const qml_long *LDVT, qml_double_complex *U,
    const qml_long *LDU, qml_double_complex *C, const qml_long *LDC,
    double *RWORK, qml_long *INFO);
```

Arguments

UPLO	Specify whether B is upper ('U') or lower ('L') triangular
N	Order of matrix B
NCVT	Number of columns of VT
NRU	Number of rows of U
NCC	Number of columns of C
D	Vector of diagonal elements of B of length N, overwritten by sorted singular values on exit
E	Vector of off-diagonal elements of B of length N - 1, destroyed on exit
VT	Matrix of size N x NCVT, overwritten by $P^H * VT$ on exit
LDVT	Leading dimension of VT
U	Matrix of size NRU x N, overwritten by $U * Q$ on exit
LDU	Leading dimension of U
C	Matrix of size N x NCC, overwritten by $Q^H * C$ on exit
LDC	Leading dimension of C
RWORK	Work space of size 4N
INFO	0 on success, <0 for bad arguments, >0 if no convergence

(S|D|C|Z)GEBAK

Single, double, single complex, and double complex GEBAK.

Description

Computes the transformed right and left eigenvectors for a balanced matrix using back transformation.

This routine takes eigenvectors computed using a routine such as TREVC together with the SCALE output of GEBAL and performs back substitution to adjust the eigenvectors to account for the permutation and scaling operations done during balancing.

LAPACK Interface

```
void sgebak(const char *JOB, const char *SIDE, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, float *SCALE,
    const qml_long *M, float *V, const qml_long *LDV, qml_long *INFO);

void dgebak(const char *JOB, const char *SIDE, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, double *SCALE,
    const qml_long *M, double *V, const qml_long *LDV, qml_long *INFO);

void cgebak(const char *JOB, const char *SIDE, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, float *SCALE,
```

```
    const qml_long *M, qml_single_complex *V, const qml_long *LDV,
    qml_long *INFO);

void zgebak(const char *JOB, const char *SIDE, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, double *SCALE,
    const qml_long *M, qml_double_complex *V, const qml_long *LDV,
    qml_long *INFO);
```

Arguments

JOB	Operations to perform, 'N' none, 'P' permute only, 'S' scale only, 'B' both
SIDE	Choose 'R' right or 'L' left eigenvectors in V
N	Number of rows of V
ILO	Lower index
IHI	Upper index, 1 <= ILO <= IHI <= N
SCALE	Array of size N, permutation and scale factors of columns
M	Number of columns of V
V	Left or right eigenvectors, overwritten with transformed eigenvectors
LDV	Leading dimension of V
INFO	0 on success

(S|D|C|Z)GEBAL

Single, double, single complex, and double complex GEBAL.

Description

Balances a general matrix using permutations and scaling to isolate eigenvalues and make rows and columns more similar in norm.

The permutation stage puts the matrix A into the form:

$$P * A * P^H = \begin{bmatrix} T_1 & X & Y \\ 0 & B & Z \\ 0 & 0 & T_2 \end{bmatrix}$$

where T_1 and T_2 are upper triangular. The indices ILO and IHI define the boundaries of B. Permutation details are recorded in SCALE in components 1 through ILO-1, and IHI+1 through N.

The balance stage consists of diagonal similarity transforms:

$$D^{-1} * B * D$$

The scale factors for D are chosen to make the 1-norm of each row of B and its corresding column as equal as possible. The scale factors for D are recorded in SCALE in components ILO through IHI.

The final balanced matrix is:

$$D^{-1} * P * A * P^H * D = \begin{bmatrix} T_1 & X * D & Y \\ 0 & D^{-1} * B * D & D^{-1} * Z \\ 0 & 0 & T_2 \end{bmatrix}$$

LAPACK Interface

```
void sgebal(const char *JOB, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *ILO, qml_long *IHI, float *SCALE, qml_long *INFO);

void dgebal(const char *JOB, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *ILO, qml_long *IHI, double *SCALE, qml_long *INFO);

void cgebal(const char *JOB, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *ILO, qml_long *IHI, float *SCALE,
    qml_long *INFO);

void zgebal(const char *JOB, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *ILO, qml_long *IHI, double *SCALE,
    qml_long *INFO);
```

Arguments

JOB	Operations to perform, 'N' none, 'P' permute only, 'S' scale only, 'B' both
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten with balanced matrix
LDA	Leading dimension of A
ILO	Lower index
IHI	Upper index
SCALE	Array of size N, scale factors of columns
INFO	0 on success

(S|D|C|Z)GEES

Single, double, single complex, and double complex GEES.

Description

Computes the eigenvalues, Schur form, and (optionally) Schur vectors of a general matrix.

Given an N by N matrix A, computes the Schur factorization:

$$\mathrm{A} = Z * T * Z^H$$

The matrix T is in Schur form, meaning it is upper quasi-triangular with blocks of size 1x1 or 2x2. Blocks of size 2x2 are of the form:

$$\begin{bmatrix} a & b \\ c & a \end{bmatrix}$$

Each 2x2 block corresponds to a pair of eigenvalues $a \pm \sqrt{bc}$.

LAPACK Interface

```
void sgees(const char *JOBVS, const char *SORT, void *SELECT, const qml_long *N,
    float *A, const qml_long *LDA, qml_long *SDIM, float *WR, float *WI,
    float *VS, const qml_long *LDVS, float *WORK, const qml_long *LWORK,
    qml_int *BWORK, qml_long *INFO);
```

```
void dgees(const char *JOBVS, const char *SORT, void *SELECT, const qml_long *N,
    double *A, const qml_long *LDA, qml_long *SDIM, double *WR, double *WI,
    double *VS, const qml_long *LDVS, double *WORK, const qml_long *LWORK,
    qml_int *BWORK, qml_long *INFO);

void cgees(const char *JOBVS, const char *SORT, void *SELECT, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, qml_long *SDIM,
    qml_single_complex *W, qml_single_complex *VS, const qml_long *LDVS,
    qml_single_complex *WORK, const qml_long *LWORK, float *RWORK,
    qml_int *BWORK, qml_long *INFO);

void zgees(const char *JOBVS, const char *SORT, void *SELECT, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, qml_long *SDIM,
    qml_double_complex *W, qml_double_complex *VS, const qml_long *LDVS,
    qml_double_complex *WORK, const qml_long *LWORK, double *RWORK,
    qml_int *BWORK, qml_long *INFO);
```

Arguments

JOBVS	Compute VS if 'V', otherwise do not compute
SORT	Sort eigenvalues, only 'N' supported
SELECT	Function pointers, not supported (pass NULL)
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten with T on exit
LDA	Leading dimension of A
SDIM	Not used
W	Array of size N containing eigenvalues on exit
WR	Real part of W for real versions
WI	Complex part of W for real version
VS	On exit contains Schur vectors matrix Z if JOBVS is 'V'
LDVS	Leading dimension of VS
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least 3N but optimally a larger multiple of N (-1 to query)
RWORK	Work space of size N
BWORK	Not used
INFO	0 on success

(S|D|C|Z)GEEV

Single, double, single complex, and double complex GEEV.

Description

Computes the eigenvalues and (optionally) eigenvectors of a general matrix.

Right eigenvectors satisfy:

$$\mathrm{A} * v_i = \lambda_i v_i$$

Left eigenvectors satisfy:

$${u_i}^H * \mathrm{A} = \lambda {u_i}^H$$

Computed eigenvectors are normalized to have Euclidean unit length and largest real component.

LAPACK Interface

```
void sgeev(const char *JOBVL, const char *JOBVR, const qml_long *N, float *A,
    const qml_long *LDA, float *WR, float *WI, float *VL,
    const qml_long *LDVL, float *VR, const qml_long *LDVR, float *WORK,
    const qml_long *LWORK, qml_long *INFO);

void dgeev(const char *JOBVL, const char *JOBVR, const qml_long *N, double *A,
    const qml_long *LDA, double *WR, double *WI, double *VL,
    const qml_long *LDVL, double *VR, const qml_long *LDVR, double *WORK,
    const qml_long *LWORK, qml_long *INFO);

void cgeev(const char *JOBVL, const char *JOBVR, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *W,
    qml_single_complex *VL, const qml_long *LDVL, qml_single_complex *VR,
    const qml_long *LDVR, qml_single_complex *WORK, const qml_long *LWORK,
    float *RWORK, qml_long *INFO);

void zgeev(const char *JOBVL, const char *JOBVR, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *W,
    qml_double_complex *VL, const qml_long *LDVL, qml_double_complex *VR,
    const qml_long *LDVR, qml_double_complex *WORK, const qml_long *LWORK,
    double *RWORK, qml_long *INFO);
```

Arguments

JOBVL	Compute VL if ‘V’, otherwise do not compute
JOBVR	Compute VR if ‘V’, otherwise do not compute
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten on exit
LDA	Leading dimension of A
W	Array of size N containing eigenvalues on exit
WR	Real part of W for real versions
WI	Complex part of W for real version
VL	On exit contains left eigenvectors if JOBVL is ‘V’
LDVL	Leading dimension of VL
VR	On exit contains right eigenvectors if JOBVR is ‘V’
LDVR	Leading dimension of VR
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least 4N but optimally a larger multiple of N (-1 to query)
RWORK	Work space of size 2N
INFO	0 on success

(S|D|C|Z)GEHRD

Single, double, single complex, and double complex GEHRD.

Description

Reduces a general matrix to upper Hessenberg form using a unitary similarity transform.

$$H = Q^H * A * Q$$

The matrix H is upper Hessenberg and the matrix Q is unitary.

The matrix Q is not stored explicitly. Instead, the elements below the diagonal of A together with TAU store Q as the product of scaled elementary reflectors.

$$Q = H_{ILO} * H_{ILO+1} * \ldots * H_{IHI-1}$$

$$H_i = I - \tau_i * v * v^H$$

The vector v has components 1 through i of zero, component i+1 is 1, components IHI+1 through N are zero, and components i+2 through IHI are stored in A below the subdiagonal in column i.

LAPACK Interface

```
void sgehrd(const qml_long *N, const qml_long *ILO, const qml_long *IHI, float *A,
    const qml_long *LDA, float *TAU, float *WORK, const qml_long *LWORK,
    qml_long *INFO);

void dgehrd(const qml_long *N, const qml_long *ILO, const qml_long *IHI,
    double *A, const qml_long *LDA, double *TAU, double *WORK,
    const qml_long *LWORK, qml_long *INFO);

void cgehrd(const qml_long *N, const qml_long *ILO, const qml_long *IHI,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *TAU,
    qml_single_complex *WORK, const qml_long *LWORK, qml_long *INFO);

void zgehrd(const qml_long *N, const qml_long *ILO, const qml_long *IHI,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *TAU,
    qml_double_complex *WORK, const qml_long *LWORK, qml_long *INFO);
```

Arguments

N	Number of rows and columns of A
ILO	Lower index
IHI	Upper index, 1 <= ILO <= IHI <= N
A	Matrix, overwritten by H and reflector vectors on exit
LDA	Leading dimension of A
TAU	On exit contains vector of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(S|D|C|Z)GELS

Single, double, single complex, and double complex GELS.

Description

Solves an over or underdetermined system of linear equations.

For overdetermined systems, finds the least squares solution:

$$\min ||B - A * X||$$

For underdetermined systems, finds the minimum norm solution:

$$A * X = B$$

For SGELS and DGELS, if TRANS is T then uses the transpose of A. For CGELS and ZGELS, if TRANS is C then uses the Hermitian of A.

The matrix A is assumed to be full rank. Either QR or LQ factorization is used to compute results.

The matrix B encodes multiple right-hand sides and produces multiple solution vectors in the matrix X. Each column of B and column of X can be considered a single solution to

$$A * x = b$$

LAPACK Interface

```
void sgels(const char *TRANS, const qml_long *M, const qml_long *N,
    const qml_long *NRHS, float *A, const qml_long *LDA, float *B,
    const qml_long *LDB, float *WORK, const qml_long *LWORK,
    qml_long *INFO);

void dgels(const char *TRANS, const qml_long *M, const qml_long *N,
    const qml_long *NRHS, double *A, const qml_long *LDA, double *B,
    const qml_long *LDB, double *WORK, const qml_long *LWORK,
    qml_long *INFO);

void cgels(const char *TRANS, const qml_long *M, const qml_long *N,
    const qml_long *NRHS, qml_single_complex *A, const qml_long *LDA,
    qml_single_complex *B, const qml_long *LDB, qml_single_complex *WORK,
    const qml_long *LWORK, qml_long *INFO);

void zgels(const char *TRANS, const qml_long *M, const qml_long *N,
    const qml_long *NRHS, qml_double_complex *A, const qml_long *LDA,
    qml_double_complex *B, const qml_long *LDB, qml_double_complex *WORK,
    const qml_long *LWORK, qml_long *INFO);
```

Arguments

TRANS	One of 'N' (no transpose), 'T' (transpose), or 'C' (Hermitian)
M	Number of rows of A
N	Number of columns of A
NRHS	Number of right hand sides, number of columns of B and X
A	Matrix of size M x N, overwritten by factorization on exit
LDA	Leading dimension of A
B	Matrix of right hand side vectors, overwritten with X on exit
LDB	Leading dimension of B
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least *min(M,N)+max(min(M, N), NRHS)* (-1 to query)
INFO	0 on success, <0 for illegal arguments, >0 if not full rank

(S|D|C|Z)GELSD

Single, double, single complex, and double complex GELSD.

Description

Finds the minimum norm solution to a linear least-squares problem:

$$\min ||b - A * x||_2$$

The matrix A is allowed to be rank deficient.

This routine works by computing the SVD of A. Any singular values less than RCOND times the largest singular value are assumed to be zero. Passing RCOND<0 means machine precision will be used for the cutoff.

The matrix B encodes multiple right-hand sides and produces multiple solution vectors encoded in the matrix X. Each column of B and column of X can be considered a single solution to

$$A * x = b$$

LAPACK Interface

```
void sgelsd(const qml_long *M, const qml_long *N, const qml_long *NRHS, float *A,
    const qml_long *LDA, float *B, const qml_long *LDB, float *S,
    const float *RCOND, qml_long *RANK, float *WORK, const qml_long *LWORK,
    qml_long *IWORK, qml_long *INFO);

void dgelsd(const qml_long *M, const qml_long *N, const qml_long *NRHS, double *A,
    const qml_long *LDA, double *B, const qml_long *LDB, double *S,
    const double *RCOND, qml_long *RANK, double *WORK,
    const qml_long *LWORK, qml_long *IWORK, qml_long *INFO);

void cgelsd(const qml_long *M, const qml_long *N, const qml_long *NRHS,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *B,
    const qml_long *LDB, float *S, const float *RCOND, qml_long *RANK,
    qml_single_complex *WORK, const qml_long *LWORK, float *RWORK,
    qml_long *IWORK, qml_long *INFO);

void zgelsd(const qml_long *M, const qml_long *N, const qml_long *NRHS,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *B,
    const qml_long *LDB, double *S, const double *RCOND, qml_long *RANK,
    qml_double_complex *WORK, const qml_long *LWORK, double *RWORK,
    qml_long *IWORK, qml_long *INFO);
```

Arguments

M	Number of rows of A
N	Number of columns of A
NRHS	Number of right hand sides, number of columns of B and X
A	Matrix of size M x N, overwritten on exit
LDA	Leading dimension of A
B	Matrix of right hand side vectors, overwritten with X on exit
LDB	Leading dimension of B
S	On exit contains singular values in decreasing order, length *min(M, N)*
RCOND	Determines which singular values to consider zero
RANK	On exit contains effective rank of A
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query WORK and IWORK)
RWORK	Work space (query size required with LWORK=-1)
IWORK	Integer work space (query size required with LWORK=-1)
INFO	0 on success, <0 for illegal arguments, >0 if SVD failed

(S|D|C|Z)GEQR2

Single, double, single complex, and double complex GEQR2.

Description

Computes the QR factorization of a matrix using an unblocked algorithm.

$$A = Q * R$$

The upper triangular matrix R is stored in A on the diagonal and above. The matrix Q is not stored explicitly. Instead, the elements below the diagonal of A together with TAU store Q as the product of scaled elementary reflectors.

$$Q = H_1 * H_2 * \ldots * H_k$$

$$H_i = I - \tau_i * v_i * {v_i}^H$$

LAPACK Interface

```
void sgeqr2(const qml_long *M, const qml_long *N, float *A, const qml_long *LDA,
    float *TAU, float *WORK, qml_long *INFO);

void dgeqr2(const qml_long *M, const qml_long *N, double *A, const qml_long *LDA,
    double *TAU, double *WORK, qml_long *INFO);

void cgeqr2(const qml_long *M, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_single_complex *TAU, qml_single_complex *WORK,
    qml_long *INFO);

void zgeqr2(const qml_long *M, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_double_complex *TAU, qml_double_complex *WORK,
    qml_long *INFO);
```

Arguments

M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N
LDA	Leading dimension of A
TAU	On exit contains vector of scale factors for reflectors
WORK	Work space of size N
INFO	0 on success

(S|D|C|Z)GEQRF

Single, double, single complex, and double complex GEQRF.

Description

Computes the QR factorization of a matrix using a blocked algorithm.

$$A = Q * R$$

The upper triangular matrix R is stored in A on the diagonal and above. The matrix Q is not stored explicitly. Instead, the elements below the diagonal of A together with TAU store Q as the product of scaled elementary reflectors.

$$Q = H_1 * H_2 * \ldots * H_k$$

$$H_i = I - \tau_i * v_i * {v_i}^H$$

LAPACK Interface

```
void sgeqrf(const qml_long *M, const qml_long *N, float *A, const qml_long *LDA,
    float *TAU, float *WORK, const qml_long *LWORK, qml_long *INFO);

void dgeqrf(const qml_long *M, const qml_long *N, double *A, const qml_long *LDA,
    double *TAU, double *WORK, const qml_long *LWORK, qml_long *INFO);

void cgeqrf(const qml_long *M, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_single_complex *TAU, qml_single_complex *WORK,
    const qml_long *LWORK, qml_long *INFO);

void zgeqrf(const qml_long *M, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_double_complex *TAU, qml_double_complex *WORK,
    const qml_long *LWORK, qml_long *INFO);
```

Arguments

M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N
LDA	Leading dimension of A
TAU	On exit contains vector of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least N but optimally a larger multiple of N (-1 to query)
INFO	0 on success

(S|D|C|Z)GESDD

Single, double, single complex, and double complex GESDD.

Description

Computes the singular value decomposition (SVD) of a general matrix, and optionally the right/left singular vectors.

The matrix A is decomposed:

$$A = U * \Sigma * V^H$$

The matrix Σ is an MxN matrix that is zero except on the diagonal. U is an MxM unitary matrix, and V is an NxN unitary matrix. The diagonal elements of Σ are the singular values, ordered from largest to smallest in magnitude.

Uses the divide-and-conquer algorithm.

LAPACK Interface

```
void sgesdd(const char *JOBZ, const qml_long *M, const qml_long *N, float *A,
    const qml_long *LDA, float *S, float *U, const qml_long *LDU,
    float *VT, const qml_long *LDVT, float *WORK, const qml_long *LWORK,
    qml_long *IWORK, qml_long *INFO);

void dgesdd(const char *JOBZ, const qml_long *M, const qml_long *N, double *A,
    const qml_long *LDA, double *S, double *U, const qml_long *LDU,
    double *VT, const qml_long *LDVT, double *WORK, const qml_long *LWORK,
    qml_long *IWORK, qml_long *INFO);

void cgesdd(const char *JOBZ, const qml_long *M, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, float *S,
    qml_single_complex *U, const qml_long *LDU, qml_single_complex *VT,
    const qml_long *LDVT, qml_single_complex *WORK, const qml_long *LWORK,
    float *RWORK, qml_long *IWORK, qml_long *INFO);

void zgesdd(const char *JOBZ, const qml_long *M, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, double *S,
    qml_double_complex *U, const qml_long *LDU, qml_double_complex *VT,
    const qml_long *LDVT, qml_double_complex *WORK, const qml_long *LWORK,
    double *RWORK, qml_long *IWORK, qml_long *INFO);
```

Arguments

JOBU	What to compute for U and VT, 'A' (all), 'S' (singular vectors), 'O' (overwrite A), 'N' (none)
M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N, overwritten on exit
LDA	Leading dimension of A
S	Array of size min(M, N), singular values in decreasing order
U	Matrix containing left singular vectors as columns
LDU	Leading dimension of U
VT	Matrix containing right singular vectors as rows
LDVT	Leading dimension of VT
WORK	Workspace of size at least LWORK
LWORK	Size of workspace (-1 to query)
IWORK	Integer workspace of size 8*min(M, N)
INFO	0 on success, <0 for bad arguments, >0 if no convergence

(S|D|C|Z)GESV

Single, double, single complex, and double complex GESV.

Description

Uses LU decomposition with pivoting to solve the equation:

$$A * X = B$$

The LU decomposition with partial pivoting is used to factor A as:

$$A = P * L * U$$

where P is a permutation matrix, L is unit lower triangular and U is upper triangular. On exit, L and U overwrite A (the unit diagonal is not stored), the locations of pivots are stored in IPIV, and the solution matrix X overwrites B.

LAPACK Interface

```
void sgesv(const qml_long *N, const qml_long *NRHS, float *A, const qml_long *LDA,
    qml_long *IPIV, float *B, const qml_long *LDB, qml_long *INFO);

void dgesv(const qml_long *N, const qml_long *NRHS, double *A, const qml_long *LDA,
    qml_long *IPIV, double *B, const qml_long *LDB, qml_long *INFO);

void cgesv(const qml_long *N, const qml_long *NRHS, qml_single_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_single_complex *B,
    const qml_long *LDB, qml_long *INFO);

void zgesv(const qml_long *N, const qml_long *NRHS, qml_double_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_double_complex *B,
    const qml_long *LDB, qml_long *INFO);
```

Arguments

N	Number of linear equations, order of A
NRHS	Number of right hand sides, number of columns of B
A	Matrix of size N x N
LDA	Leading dimension of A
IPIV	Pivot indices indicating rows to interchange
B	Matrix of size N x NRHS, overwritten with solutions
LDB	Leading dimension of B
INFO	0 on success, <0 for illegal arguments, >0 for singular decomposition

(S|D|C|Z)GESVD

Single, double, single complex, and double complex GESVD.

Description

Computes the singular value decomposition (SVD) of a general matrix, and optionally the right/left singular vectors.

The matrix A is decomposed:

$$A = U * \Sigma * V^H$$

The matrix Σ is an MxN matrix that is zero except on the diagonal. U is an MxM unitary matrix, and V is an NxN unitary matrix. The diagonal elements of Σ are the singular values, ordered from largest to smallest in magnitude.

LAPACK Interface

```
void sgesvd(const char *JOBU, const char *JOBVT, const qml_long *M,
    const qml_long *N, float *A, const qml_long *LDA, float *S, float *U,
    const qml_long *LDU, float *VT, const qml_long *LDVT, float *WORK,
    const qml_long *LWORK, qml_long *INFO);

void dgesvd(const char *JOBU, const char *JOBVT, const qml_long *M,
    const qml_long *N, double *A, const qml_long *LDA, double *S,
    double *U, const qml_long *LDU, double *VT, const qml_long *LDVT,
    double *WORK, const qml_long *LWORK, qml_long *INFO);

void cgesvd(const char *JOBU, const char *JOBVT, const qml_long *M,
    const qml_long *N, qml_single_complex *A, const qml_long *LDA,
    float *S, qml_single_complex *U, const qml_long *LDU,
    qml_single_complex *VT, const qml_long *LDVT,
    qml_single_complex *WORK, const qml_long *LWORK, float *RWORK,
    qml_long *INFO);

void zgesvd(const char *JOBU, const char *JOBVT, const qml_long *M,
    const qml_long *N, qml_double_complex *A, const qml_long *LDA,
    double *S, qml_double_complex *U, const qml_long *LDU,
    qml_double_complex *VT, const qml_long *LDVT,
    qml_double_complex *WORK, const qml_long *LWORK, double *RWORK,
    qml_long *INFO);
```

Arguments

JOBU	What to compute for U, 'A' (all), 'S' (left singular vectors), 'O' (overwrite A), 'N' (none)
JOBVT	What to compute for VT, 'A' (all), 'S' (right singular vectors), 'O' (overwrite A), 'N' (none)
M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N, overwritten on exit
LDA	Leading dimension of A
S	Array of size min(M, N), singular values in decreasing order
U	Matrix containing left singular vectors as columns
LDU	Leading dimension of U
VT	Matrix containing right singular vectors as rows
LDVT	Leading dimension of VT
WORK	Workspace of size at least LWORK
LWORK	Size of workspace (-1 to query)
RWORK	Workspace of size 5*min(M, N)
INFO	0 on success, <0 for bad arguments, >0 if no convergence

(S|D|C|Z)GESVX

Single, double, single complex, and double complex GESVX.

Description

Uses LU decomposition with pivoting to solve one of the equations:

$$A * X = B$$

$$A^T * X = B$$

$$A^H * X = B$$

This interface provides more options than the simplified GESV interface. The matrix A may be provided factored or non-factored on input. Equilibration is configurable. Error estimates for the solution are also provided.

If FACT is 'F' then A is supplied both in equilibrated form and in LU factored form with pivots. In this case arguments A, AF, IPIV, EQUED, R, and C are input arguments provided by the caller. Otherwise AF, IPIV, EQUED, R, and C are output arguments.

LAPACK Interface

```
void sgesvx(const char *FACT, const char *TRANS, const qml_long *N,
    const qml_long *NRHS, float *A, const qml_long *LDA, float *AF,
    const qml_long *LDAF, qml_long *IPIV, char *EQUED, float *R, float *C,
    float *B, const qml_long *LDB, float *X, const qml_long *LDX,
    float *RCOND, float *FERR, float *BERR, float *WORK, qml_long *IWORK,
    qml_long *INFO);

void dgesvx(const char *FACT, const char *TRANS, const qml_long *N,
    const qml_long *NRHS, double *A, const qml_long *LDA, double *AF,
    const qml_long *LDAF, qml_long *IPIV, char *EQUED, double *R,
```

```
    double *C, double *B, const qml_long *LDB, double *X,
    const qml_long *LDX, double *RCOND, double *FERR, double *BERR,
    double *WORK, qml_long *IWORK, qml_long *INFO);

void cgesvx(const char *FACT, const char *TRANS, const qml_long *N,
    const qml_long *NRHS, qml_single_complex *A, const qml_long *LDA,
    qml_single_complex *AF, const qml_long *LDAF, qml_long *IPIV,
    char *EQUED, float *R, float *C, qml_single_complex *B,
    const qml_long *LDB, qml_single_complex *X, const qml_long *LDX,
    float *RCOND, float *FERR, float *BERR, qml_single_complex *WORK,
    float *RWORK, qml_long *INFO);

void zgesvx(const char *FACT, const char *TRANS, const qml_long *N,
    const qml_long *NRHS, qml_double_complex *A, const qml_long *LDA,
    qml_double_complex *AF, const qml_long *LDAF, qml_long *IPIV,
    char *EQUED, double *R, double *C, qml_double_complex *B,
    const qml_long *LDB, qml_double_complex *X, const qml_long *LDX,
    double *RCOND, double *FERR, double *BERR, qml_double_complex *WORK,
    double *RWORK, qml_long *INFO);
```

Arguments

FACT	One of 'F' (factorization provided), 'N' (factorization requested), or 'E' (equilibration and factorization requested)
TRANS	One of 'N' (no tranpose), 'T' (transpose), or 'C' (Hermitian)
N	Number of linear equations, order of A
NRHS	Number of right hand sides, number of columns of B and X
A	Matrix of size N x N, must be equilibrated if FACT is 'F' and EQUED is not 'N'
LDA	Leading dimension of A
AF	Factored form of A
LDAF	Leading dimension of AF
IPIV	Pivot indices
EQUED	Type of equilibration, 'N' (none), 'R' (row), 'C' (column), 'B' (both)
R	Array of size N, row scaling factors of A
C	Array of size N, column scaling factors of A
B	Matrix of size N x NRHS
LDB	Leading dimension of B
X	Matrix of size N x NRHS containing solutions
LDX	Leading dimension of X
RCOND	Estimate of reciprocal of condition number of A after equilibration
FERR	Array of size NRHS, estimate of forward errors for each solution
BERR	Array of size NRHS, relative backward error for each solution
WORK	Workspace of size 4N
RWORK	Workspace of size 2N
IWORK	Integer workspace of size N
INFO	0 on success, <0 for illegal arguments, >0 for factorization problems

(S|D|C|Z)GETF2

Single, double, single complex, and double complex GETF2.

Description

Computes the LU decomposition of a matrix using partial pivoting and an unblocked algorithm.

The matrix A is factored as:

$$A = P * L * U$$

where P is a permutation matrix, L is unit lower triangular and U is upper triangular. On exit, L and U overwrite A (the unit diagonal is not stored), the locations of pivots are stored in IPIV, and the solution matrix X overwrites B.

LAPACK Interface

```
void sgetf2(const qml_long *M, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *IPIV, qml_long *INFO);

void dgetf2(const qml_long *M, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *IPIV, qml_long *INFO);

void cgetf2(const qml_long *M, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_long *INFO);

void zgetf2(const qml_long *M, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_long *INFO);
```

Arguments

M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N
LDA	Leading dimension of A
IPIV	Pivot indices indicating rows to interchange
INFO	0 on success, <0 for illegal arguments, >0 for singular decomposition

(S|D|C|Z)GETRF

Single, double, single complex, and double complex GETRF.

Description

Computes the LU decomposition of a matrix using partial pivoting and a blocked algorithm.

The matrix A is factored as:

$$A = P * L * U$$

where P is a permutation matrix, L is unit lower triangular and U is upper triangular. On exit, L and U overwrite A (the unit diagonal is not stored), the locations of pivots are stored in IPIV, and the solution matrix X overwrites B.

LAPACK Interface

```
void sgetrf(const qml_long *M, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *IPIV, qml_long *INFO);

void dgetrf(const qml_long *M, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *IPIV, qml_long *INFO);

void cgetrf(const qml_long *M, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_long *INFO);

void zgetrf(const qml_long *M, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_long *INFO);
```

Arguments

M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N
LDA	Leading dimension of A
IPIV	Pivot indices indicating rows to interchange
INFO	0 on success, <0 for illegal arguments, >0 for singular decomposition

(S|D|C|Z)GETRI

Single, double, single complex, and double complex GETRI.

Description

Computes the inverse of a matrix by using LU decomposition.

Computes the inverse of U, then solves for the inverse of A:

$$A^{-1} * L = U^{-1}$$

LAPACK Interface

```
void sgetri(const qml_long *N, float *A, const qml_long *LDA, const qml_long *IPIV,
    float *WORK, const qml_long *LWORK, qml_long *INFO);

void dgetri(const qml_long *N, double *A, const qml_long *LDA, const qml_long *IPIV,
    double *WORK, const qml_long *LWORK, qml_long *INFO);

void cgetri(const qml_long *N, qml_single_complex *A, const qml_long *LDA,
    const qml_long *IPIV, qml_single_complex *WORK, const qml_long *LWORK,
    qml_long *INFO);

void zgetri(const qml_long *N, qml_double_complex *A, const qml_long *LDA,
    const qml_long *IPIV, qml_double_complex *WORK, const qml_long *LWORK,
    qml_long *INFO);
```

Arguments

N	Order of A
A	Matrix of size N x N
LDA	Leading dimension of A
IPIV	Pivot indices indicating rows to interchange
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least N, optimally a multiple of N (-1 to query)
INFO	0 on success, <0 for illegal arguments, >0 for singular decomposition

(S|D|C|Z)GETRS

Single, double, single complex, and double complex GETRS.

Description

Uses LU decomposition to solve one of the equations:

$$A * X = B$$

$$A^T * X = B$$

$$A^H * X = B$$

LAPACK Interface

```
void sgetrs(const char *TRANS, const qml_long *N, const qml_long *NRHS, float *A,
    const qml_long *LDA, qml_long *IPIV, float *B, const qml_long *LDB,
    qml_long *INFO);

void dgetrs(const char *TRANS, const qml_long *N, const qml_long *NRHS, double *A,
    const qml_long *LDA, qml_long *IPIV, double *B, const qml_long *LDB,
    qml_long *INFO);

void cgetrs(const char *TRANS, const qml_long *N, const qml_long *NRHS,
    qml_single_complex *A, const qml_long *LDA, qml_long *IPIV,
    qml_single_complex *B, const qml_long *LDB, qml_long *INFO);

void zgetrs(const char *TRANS, const qml_long *N, const qml_long *NRHS,
    qml_double_complex *A, const qml_long *LDA, qml_long *IPIV,
    qml_double_complex *B, const qml_long *LDB, qml_long *INFO);
```

Arguments

TRANS	Specifies transform option for A, 'N' for none, 'T' for transpose, 'C' for conjugate transpose
N	Number of linear equations, order of A
NRHS	Number of right hand sides, number of columns of B
A	Matrix of size N x N
LDA	Leading dimension of A
IPIV	Pivot indices indicating rows to interchange
B	Matrix of size N x NRHS, overwritten with solutions
LDB	Leading dimension of B
INFO	0 on success

(S|D|C|Z)GGES

Single, double, single complex, and double complex GGES.

Description

Computes the generalized eigenvalues, Schur form, and (optionally) Schur vectors of a general matrix.

Given two N by N matrices A and B, computes the generalized Schur factorization:

$$(\mathrm{A}, \mathrm{B}) = \left(\mathrm{VSL} * \mathrm{S} * \mathrm{VSR}^{H}, \mathrm{VSL} * \mathrm{T} * \mathrm{VSR}^{H}\right)$$

The output matrix S will be upper block triangular with blocks of size 1x1 or 2x2. Blocks of size 2x2 will only occur in SGGES and DGGES and represent a complex conjugate pair of generalized eigenvalues. The output matrix T will be upper triangular with non-negative diagonal.

A generalized eigenvalue is a solution to the equation:

$$Av = \lambda Bv$$

Generalized eigenvalues are represented as a ratio ALPHA / BETA with separate storage for ALPHA and BETA for increased numerical flexibility.

LAPACK Interface

```
void sgges(const char *JOBVSL, const char *JOBVSR, const char *SORT,
    void *SELCTG, const qml_long *N, float *A, const qml_long *LDA,
    float *B, const qml_long *LDB, qml_long *SDIM, float *ALPHAR,
    float *ALPHAI, float *BETA, float *VSL, const qml_long *LDVSL,
    float *VSR, const qml_long *LDVSR, float *WORK, const qml_long *LWORK,
    qml_int *BWORK, qml_long *INFO);

void dgges(const char *JOBVSL, const char *JOBVSR, const char *SORT,
    void *SELCTG, const qml_long *N, double *A, const qml_long *LDA,
    double *B, const qml_long *LDB, qml_long *SDIM, double *ALPHAR,
    double *ALPHAI, double *BETA, double *VSL, const qml_long *LDVSL,
    double *VSR, const qml_long *LDVSR, double *WORK, const qml_long *LWORK,
    qml_int *BWORK, qml_long *INFO);

void cgges(const char *JOBVSL, const char *JOBVSR, const char *SORT,
    void *SELCTG, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_single_complex *B, const qml_long *LDB,
```

```
    qml_long *SDIM, qml_single_complex *ALPHA, qml_single_complex *BETA,
    qml_single_complex *VSL, const qml_long *LDVSL, qml_single_complex *VSR,
    const qml_long *LDVSR, qml_single_complex *WORK, const qml_long *LWORK,
    float *RWORK, qml_int *BWORK, qml_long *INFO);

void zgges(const char *JOBVSL, const char *JOBVSR, const char *SORT,
    void *SELCTG, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_double_complex *B, const qml_long *LDB,
    qml_long *SDIM, qml_double_complex *ALPHA, qml_double_complex *BETA,
    qml_double_complex *VSL, const qml_long *LDVSL, qml_double_complex *VSR,
    const qml_long *LDVSR, qml_double_complex *WORK, const qml_long *LWORK,
    double *RWORK, qml_int *BWORK, qml_long *INFO);
```

Arguments

JOBVSL	Compute VSL if 'V', otherwise do not compute
JOBVSR	Compute VSR if 'V', otherwise do not compute
SORT	Sort eigenvalues, only 'N' supported
SELCTG	Function pointers, not supported (pass NULL)
N	Number of rows and columns of A, B, VSL, and VSR
A	Matrix of size N x N, overwritten with S on output
LDA	Leading dimension of A
B	Matrix of size N x N, overwritten with T on output
LDB	Leading dimension of B
SDIM	Not used
ALPHA	On exit contains generalized eigenvalues
ALPHAR	Real part of ALPHA for real versions
ALPHAI	Complex part of ALPHA for real version
VSL	On exit contains left Schur vectors if JOBVSL is 'V'
LDVSL	Leading dimension of VSL
VSR	On exit contains right Schur vectors if JOBVSR is 'V'
LDVSR	Leading dimension of VSR
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least 2N but optimally a larger multiple of N (-1 to query)
RWORK	Work space of size 8N
BWORK	Not used
INFO	0 on success

(S|D|C|Z)GGEV

Single, double, single complex, and double complex GGEV.

Description

Computes the generalized eigenvalues and (optionally) generalized eigenvectors of a general matrix.

Given two N by N matrices A and B, computes the generalized eigenvalues and generalized left and right eigenvectors.

A generalized eigenvalue is a solution to the equation:

$$Av = \lambda Bv$$

The vector v is a generalized right eigenvector. Generalized left eigenvectors u satisfy:

$$u^H A = \lambda u^H B$$

Generalized eigenvalues are represented as a ratio ALPHA / BETA with separate storage for ALPHA and BETA for increased numerical flexibility.

LAPACK Interface

```
void sggev(const char *JOBVL, const char *JOBVR, const qml_long *N, float *A,
    const qml_long *LDA, float *B, const qml_long *LDB, float *ALPHAR,
    float *ALPHAI, float *BETA, float *VL, const qml_long *LDVL, float *VR,
    const qml_long *LDVR, float *WORK, const qml_long *LWORK, qml_long *INFO);

void dggev(const char *JOBVL, const char *JOBVR, const qml_long *N, double *A,
    const qml_long *LDA, double *B, const qml_long *LDB, double *ALPHAR,
    double *ALPHAI, double *BETA, double *VL, const qml_long *LDVL, double *VR,
    const qml_long *LDVR, double *WORK, const qml_long *LWORK, qml_long *INFO);

void cggev(const char *JOBVL, const char *JOBVR, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *B,
    const qml_long *LDB, qml_single_complex *ALPHA, qml_single_complex *BETA,
    qml_single_complex *VL, const qml_long *LDVL, qml_single_complex *VR,
    const qml_long *LDVR, qml_single_complex *WORK, const qml_long *LWORK,
    float *RWORK, qml_long *INFO);

void zggev(const char *JOBVL, const char *JOBVR, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *B,
    const qml_long *LDB, qml_double_complex *ALPHA, qml_double_complex *BETA,
    qml_double_complex *VL, const qml_long *LDVL, qml_double_complex *VR,
    const qml_long *LDVR, qml_double_complex *WORK, const qml_long *LWORK,
    double *RWORK, qml_long *INFO);
```

Arguments

JOBVL	Compute VL if ‘V’, otherwise do not compute
JOBVR	Compute VR if ‘V’, otherwise do not compute
N	Number of rows and columns of A, B, VL, and VR
A	Matrix of size N x N, overwritten on output
LDA	Leading dimension of A
B	Matrix of size N x N, overwritten on output
LDB	Leading dimension of B
ALPHA	On exit contains generalized eigenvalues
ALPHAR	Real part of ALPHA for real versions
ALPHAI	Complex part of ALPHA for real versions
VL	On exit contains left eigenvectors if JOBVL is ‘V’
LDVL	Leading dimension of VL
VR	On exit contains right eigenvectors if JOBVR is ‘V’
LDVR	Leading dimension of VR
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least 8N but optimally a larger multiple of N (-1 to query)
INFO	0 on success

(C|Z)HEEV

Single complex and double complex HEEV.

Description

Computes the eigenvalues and (optionally) eigenvectors of a Hermitian matrix.

Right eigenvectors satisfy:

$$\mathtt{A} * v_i = \lambda_i v_i$$

Left eigenvectors satisfy:

$${u_i}^H * \mathtt{A} = \lambda {u_i}^H$$

Computed eigenvectors are normalized to have Euclidean unit length and largest real component.

LAPACK Interface

```
void cheev(const char *JOBZ, const char *UPLO, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, float *W,
    qml_single_complex *WORK, const qml_long *LWORK, float *RWORK,
    qml_long *INFO);

void zheev(const char *JOBZ, const char *UPLO, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, double *W,
    qml_double_complex *WORK, const qml_long *LWORK, double *RWORK,
    qml_long *INFO);
```

Arguments

JOBZ	Compute eigenvectors and store in A if 'V', otherwise do not compute
UPLO	Specify whether to use 'U' upper triangular or 'L' lower triangular part of A
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten on exit
LDA	Leading dimension of A
W	Array of size N containing eigenvalues on exit
WORK	Work space of size at least LWORK
LWORK	Size of work space, at least 2N-1 but optimally a larger multiple of N (-1 to query)
RWORK	Work space of size 3N-2
INFO	0 on success

(C|Z)HEEVD

Single complex and double complex HEEVD.

Description

Computes the eigenvalues and (optionally) eigenvectors of a Hermitian matrix using a divide-and-conquer algorithm.

Right eigenvectors satisfy:

$$\mathrm{A} * v_i = \lambda_i v_i$$

Left eigenvectors satisfy:

$${u_i}^H * \mathrm{A} = \lambda {u_i}^H$$

Computed eigenvectors are normalized to have Euclidean unit length and largest real component.

LAPACK Interface

```
void cheevd(const char *JOBZ, const char *UPLO, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, float *W,
    qml_single_complex *WORK, const qml_long *LWORK, float *RWORK,
    const qml_long *LRWORK, qml_long *IWORK, const qml_long *LIWORK,
    qml_long *INFO);

void zheevd(const char *JOBZ, const char *UPLO, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, double *W,
    qml_double_complex *WORK, const qml_long *LWORK, double *RWORK,
    const qml_long *LRWORK, qml_long *IWORK, const qml_long *LIWORK,
    qml_long *INFO);
```

Arguments

JOBZ	Compute eigenvectors and store in A if 'V', otherwise do not compute
UPLO	Specify whether to use 'U' upper triangular or 'L' lower triangular part of A
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten on exit
LDA	Leading dimension of A
W	Array of size N containing eigenvalues on exit
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
RWORK	Work space of size at least LRWORK
LRWORK	Size of secondary work space (-1 to query)
IWORK	Integer work space of size at least LIWORK
LIWORK	Size of integer work space (-1 to query)
INFO	0 on success, <0 for illegal arguments, >0 if convergence failed

(C|Z)HETRD

Single complex and double complex HETRD.

Description

Reduces a Hermitian matrix to symmetric tridiagonal form using a unitary similarity transform.

$$T = Q^H * A * Q$$

The matrix T is symmetric tridiagonal and the matrix Q is unitary.

The matrix Q is not stored explicitly. Instead, the elements above or below the diagonal of A together with TAU store Q as the product of scaled elementary reflectors.

When UPLO is U:

$$Q = H_{N-1} * \ldots * H_2 * H_1$$

$$H_i = I - \tau_i * v * v^H$$

The vector v has components i+1 through n of zero, component i is 1, and components 1 through i-1 are stored in the columns of A above the superdiagonal.

When UPLO is L:

$$Q = H_1 * H_2 * \ldots * H_{N-1}$$

$$H_i = I - \tau_i * v * v^H$$

The vector v has components 1 through i of zero, component i+1 is 1, and components i+2 through N are stored in the columns of A below the subdiagonal.

LAPACK Interface

```
void chetrd(const char *UPLO, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, float *D, float *E, qml_single_complex *TAU,
    qml_single_complex *WORK, const qml_long *LWORK, qml_long *INFO);

void zhetrd(const char *UPLO, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, double *D, double *E, qml_double_complex *TAU,
    qml_double_complex *WORK, const qml_long *LWORK, qml_long *INFO);
```

Arguments

UPLO	Store upper 'U' or lower 'L' triangular part of A
N	Number of rows and columns of A
A	Hermitian matrix, overwritten by T and reflector vectors on exit
LDA	Leading dimension of A
D	On exit contains diagonal elements of T
E	On exit contains the off diagonal elements of T
TAU	On exit contains vector of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(S|D|C|Z)HSEQR

Single, double, single complex, and double complex HSEQR.

Description

Computes the eigenvalues of a Hessenberg matrix and optionally the Schur decomposition.

The Schur decomposition of the Hessenberg matrix H is:

$$H = Z * T * Z^H$$

where T is upper triangular and Z is the unitary matrix of Schur vectors.

LAPACK Interface

```
void shseqr(const char *JOB, const char *COMPZ, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, float *H,
    const qml_long *LDH, float *WR, float *WI, float *Z,
    const qml_long *LDZ, float *WORK, const qml_long *LWORK,
    qml_long *INFO);

void dhseqr(const char *JOB, const char *COMPZ, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, double *H,
    const qml_long *LDH, double *WR, double *WI, double *Z,
    const qml_long *LDZ, double *WORK, const qml_long *LWORK,
    qml_long *INFO);

void chseqr(const char *JOB, const char *COMPZ, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, qml_single_complex *H,
    const qml_long *LDH, qml_single_complex *W, qml_single_complex *Z,
    const qml_long *LDZ, qml_single_complex *WORK, const qml_long *LWORK,
    qml_long *INFO);

void zhseqr(const char *JOB, const char *COMPZ, const qml_long *N,
    const qml_long *ILO, const qml_long *IHI, qml_double_complex *H,
    const qml_long *LDH, qml_double_complex *W, qml_double_complex *Z,
    const qml_long *LDZ, qml_double_complex *WORK, const qml_long *LWORK,
    qml_long *INFO);
```

Arguments

JOB	Compute Schur matrix T (‘S’) or eigenvalues only (‘E’)
COMPZ	Compute Schur matrix Z (‘I’), Q*Z (‘V’), or do not compute (‘N’)
N	Number of rows and columns of H
ILO	Lower bound of already reduced triangular part
IHI	Upper bound of already reduced triangular part
H	Matrix of size N x N, overwritten on exit
LDH	Leading dimension of H
W	Array of size N containing eigenvalues on exit
WR	Real part of W for real versions
WI	Complex part of W for real version
Z	Matrix of size N x N
LDZ	Leading dimension of Z
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
RWORK	Work space of size 2N
INFO	0 on success

ILA(S|D|C|Z)LC

Single, double, single complex, and double complex ILAxLC.

Description

Scans a matrix to find the last non-zero column.

LAPACK Interface

```
qml_long ilaslc(const qml_long *M, const qml_long *N, float *A,
    const qml_long *LDA);

qml_long iladlc(const qml_long *M, const qml_long *N, double *A,
    const qml_long *LDA);

qml_long ilaclc(const qml_long *M, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA);

qml_long ilazlc(const qml_long *M, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA);
```

Arguments

M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N
LDA	Leading dimension of A
Result	Last non-zero column

ILA(S|D|C|Z)LR

Single, double, single complex, and double complex ILAxLR.

Description

Scans a matrix to find the last non-zero row.

LAPACK Interface

```
qml_long ilaslr(const qml_long *M, const qml_long *N, float *A,
    const qml_long *LDA);

qml_long iladlr(const qml_long *M, const qml_long *N, double *A,
    const qml_long *LDA);

qml_long ilaclr(const qml_long *M, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA);
```

```
qml_long ilazlr(const qml_long *M, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA);
```

Arguments

M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N
LDA	Leading dimension of A
Result	Last non-zero row

(C|Z)LACGV

Single complex and double complex LACGV.

Description

Conjugates a complex vector in place.

$$X \leftarrow conj(X)$$

LAPACK Interface

```
void clacgv(const qml_long *N, qml_single_complex *X, const qml_long *INCX);

void zlacgv(const qml_long *N, qml_double_complex *X, const qml_long *INCX);
```

Arguments

N	Length of vector X
X	Vector of length N
INCX	Stride between elements of X

(S|D|C|Z)LACPY

Single, double, single complex, and double complex LACPY.

Description

Copies all or part of a matrix from one location to another.

$$B \leftarrow A$$

LAPACK Interface

```
void slacpy(const char *UPLO, const qml_long *M, const qml_long *N,
    const float *A, const qml_long *LDA, float *B, const qml_long *LDB);

void dlacpy(const char *UPLO, const qml_long *M, const qml_long *N,
    const double *A, const qml_long *LDA, double *B, const qml_long *LDB);

void clacpy(const char *UPLO, const qml_long *M, const qml_long *N,
    const qml_single_complex *A, const qml_long *LDA, qml_single_complex *B,
    const qml_long *LDB);

void zlacpy(const char *UPLO, const qml_long *M, const qml_long *N,
    const qml_double_complex *A, const qml_long *LDA, qml_double_complex *B,
    const qml_long *LDB);
```

Arguments

UPLO	Specify whether A is upper 'U' or lower 'L' triangular, or rectangular otherwise
M	Number of rows of A
N	Number of columns of A
A	Source matrix
LDA	Leading dimension of A
B	Destination matrix
LDB	Leading dimension of B

(S|D)LADIV

Single and double LADIV.

Description

Performs complex division while avoiding overflow.

LAPACK Interface

```
void sladiv(const float *A, const float *B, const float *C, const float *D,
    float *P, float *Q);

void dladiv(const double *A, const double *B, const double *C, const double *D,
    double *P, double *Q);
```

Arguments

A	Real part of numerator
B	Imaginary part of numerator
C	Read part of denominator
D	Imaginary part of denominator
P	Real part of result
Q	Imaginary part of result

(C|Z)LADIV

Single complex and double complex LADIV.

Description

Performs complex division while avoiding overflow.

LAPACK Interface

```
void cladiv(qml_single_complex *result, const qml_single_complex *X,
    const qml_single_complex *Y);

void zladiv(qml_double_complex *result, const qml_double_complex *X,
    const qml_double_complex *Y);
```

Arguments

result	Complex result
X	Complex numerator
Y	Complex denominator

(S|D|C|Z)LANGE

Single, double, single complex, and double complex LANGE.

Description

Computes the 1-norm, Frobenius norm, infinity-norm, or largest absolute value of a general rectangular matrix.

NORM	Computes
M	$\max \lvert A_{ij} \rvert$
1 or O	$\lvert\lvert A \rvert\rvert_1$
I	$\lvert\lvert A \rvert\rvert_\infty$
F or E	$\lvert\lvert A \rvert\rvert_F$

LAPACK Interface

```
float slange(const char *NORM, const qml_long *M, const qml_long *N,
    const float *A, const qml_long *LDA, float *WORK);

double dlange(const char *NORM, const qml_long *M, const qml_long *N,
    const double *A, const qml_long *LDA, double *WORK);

float clange(const char *NORM, const qml_long *M, const qml_long *N,
    const qml_single_complex *A, const qml_long *LDA, float *WORK);

double zlange(const char *NORM, const qml_long *M, const qml_long *N,
    const qml_double_complex *A, const qml_long *LDA, double *WORK);
```

Arguments

NORM	Type of norm to compute
M	Number of rows of A
N	Number of columns of A
A	Matrix of size M x N
LDA	Leading dimension of A
WORK	Temporary workspace of size M used for infinity-norm, unused otherwise

(S|D)LAPY2

Single and double LAPY2.

Description

While avoiding overflow, computes:

$$result = \sqrt{X^2 + Y^2}$$

LAPACK Interface

```
float slapy2(const float *X, const float *Y);

double dlapy2(const double *X, const double *Y);
```

Arguments

X	X component of vector
Y	Y component of vector
Result	Magnitude of vector

(S|D)LAPY3

Single and double LAPY3.

Description

While avoiding overflow, computes:

$$result = \sqrt{X^2 + Y^2 + Z^2}$$

LAPACK Interface

```
float slapy3(const float *X, const float *Y, const float *Z);

double dlapy3(const double *X, const double *Y, const double *Z);
```

Arguments

X	X component of vector
Y	Y component of vector
Z	Z component of vector
Result	Magnitude of vector

(S|D|C|Z)LARF

Single, double, single complex, and double complex LARF.

Description

Applies an elementary reflector to a rectangular matrix from the left or right.

If SIDE is 'L',

$$C \leftarrow H * C$$

If SIDE is 'R',

$$C \leftarrow C * H$$

The reflector is:

$$H = I - \tau * v * v^T$$

LAPACK Interface

```
void slarf(const char *SIDE, const qml_long *M, const qml_long *N, const float *V,
    const qml_long *INCV, const float *TAU, float *C, const qml_long *LDC,
    float *WORK);

void dlarf(const char *SIDE, const qml_long *M, const qml_long *N, const double *V,
    const qml_long *INCV, const double *TAU, double *C, const qml_long *LDC,
    double *WORK);

void clarf(const char *SIDE, const qml_long *M, const qml_long *N,
    const qml_single_complex *V, const qml_long *INCV, const qml_single_complex *TAU,
    qml_single_complex *C, const qml_long *LDC, qml_single_complex *WORK);

void zlarf(const char *SIDE, const qml_long *M, const qml_long *N,
    const qml_double_complex *V, const qml_long *INCV, const qml_double_complex *TAU,
    qml_double_complex *C, const qml_long *LDC, qml_double_complex *WORK);
```

Arguments

SIDE	Which side to mulplify reflector from, 'L' or 'R'
M	Number of rows of C
N	Number of columns of C
V	Vector representing reflector
INCV	Stride between elements in V
TAU	Scale factor for elementary reflector
C	Matrix of size M x N
LDC	Leading dimension of C
WORK	Work space of size at least N if SIDE is 'L', or M if SIDE is 'R'

(S|D|C|Z)LARFB

Single, double, single complex, and double complex LARFB.

Description

Applies a block reflector to a rectangular matrix from the left or right.

If SIDE is 'L',

$$C \leftarrow H * C$$

If SIDE is 'R',

$$C \leftarrow C * H$$

If TRANS is 'T' then H^T is used. If TRANS is 'C' then H^H is used.

The block reflector matrix is formed as a product of elementary reflectors. When DIRECT is 'F':

$$H = H_1 * H_2 * \ldots * H_k$$

When DIRECT is 'B':

$$H = H_k * H_{k-1} * \ldots * H_1$$

The vectors that define each elementary reflector are stored either as rows or columns of a matrix V depending on STOREV.

LAPACK Interface

```
void slarfb(const char *SIDE, const char *TRANS, const char *DIRECT, const char *STOREV,
    const qml_long *M, const qml_long *N, const qml_long *K, const float *V,
    const qml_long *LDV, const float *T, const qml_long *LDT, float *C,
    const qml_long *LDC, float *WORK, const qml_long *LDWORK);

void dlarfb(const char *SIDE, const char *TRANS, const char *DIRECT, const char *STOREV,
    const qml_long *M, const qml_long *N, const qml_long *K, const double *V,
    const qml_long *LDV, const double *T, const qml_long *LDT, double *C,
    const qml_long *LDC, double *WORK, const qml_long *LDWORK);

void clarfb(const char *SIDE, const char *TRANS, const char *DIRECT, const char *STOREV,
```

```
    const qml_long *M, const qml_long *N, const qml_long *K,
    const qml_single_complex *V, const qml_long *LDV, const qml_single_complex *T,
    const qml_long *LDT, qml_single_complex *C, const qml_long *LDC,
    qml_single_complex *WORK, const qml_long *LDWORK);

void zlarfb(const char *SIDE, const char *TRANS, const char *DIRECT, const char *STOREV,
    const qml_long *M, const qml_long *N, const qml_long *K,
    const qml_double_complex *V, const qml_long *LDV, const qml_double_complex *T,
    const qml_long *LDT, qml_double_complex *C, const qml_long *LDC,
    qml_double_complex *WORK, const qml_long *LDWORK);
```

Arguments

SIDE	Which side to mulplify reflector from, 'L' or 'R'
TRANS	Whether to transpose reflector matrix, 'N', 'T', or 'C'
DIRECT	Direction to apply elementary reflectors
STOREV	Storage format of elementary reflector vectors
M	Number of rows of C
N	Number of columns of C
K	The number of elementary reflectors in the block
V	Matrix storing reflector vectors
LDV	Leading dimension of V
T	Triangular K x K matrix storing reflectors
LDT	Leading dimension of T
C	Matrix of size M x N
LDC	Leading dimension of C
WORK	Work space matrix of size at least N x K if SIDE is 'L', or M x K if SIDE is 'R'
LDWORK	Leading dimension of WORK matrix

(S|D|C|Z)LARFG

Single, double, single complex, and double complex LARFG.

Description

Generates a single elementary reflector matrix.

$$H^H * \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}$$

$$H^H * H = I$$

The reflector matrix is represented in the form:

$$H = I - \tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^H \end{bmatrix}$$

LAPACK Interface

```
void slarfg(const qml_long *N, float *ALPHA, float *X, const qml_long *INCX,
    float *TAU);
```

```
void dlarfg(const qml_long *N, double *ALPHA, double *X, const qml_long *INCX,
    double *TAU);

void clarfg(const qml_long *N, qml_single_complex *ALPHA, qml_single_complex *X,
    const qml_long *INCX, qml_single_complex *TAU);

void zlarfg(const qml_long *N, qml_double_complex *ALPHA, qml_double_complex *X,
    const qml_long *INCX, qml_double_complex *TAU);
```

Arguments

N	Order of the elementary reflector matrix
ALPHA	On entry α, on exit holds β
X	On entry the vector x, on exit the vector v
INCX	The stride between elements of x
TAU	On exit the value τ

(S|D|C|Z)LARTG

Single, double, single complex, and double complex LARTG.

Description

Generates a plane rotation such that:

$$\begin{bmatrix} \mathrm{CS} & \mathrm{SN} \\ -\overline{\mathrm{SN}} & \mathrm{CS} \end{bmatrix} * \begin{bmatrix} \mathrm{F} \\ \mathrm{G} \end{bmatrix} = \begin{bmatrix} \mathrm{R} \\ 0 \end{bmatrix}$$

where $\mathrm{CS}^2 + |\mathrm{SN}|^2 = 1$.

LAPACK Interface

```
void slartg(const float *F, const float *G, float *CS, float *SN, float *R);

void dlartg(const double *F, const double *G, double *CS, double *SN, double *R);

void clartg(const qml_single_complex *F, const qml_single_complex *G,
    float *CS, qml_single_complex *SN, qml_single_complex *R);

void zlartg(const qml_double_complex *F, const qml_double_complex *G,
    double *CS, qml_double_complex *SN, qml_double_complex *R);
```

Arguments

F	First part of vector to rotate
G	Second part of vector to rotate
CS	Cosine part of rotation matrix
SN	Sine part of rotatione matrix
R	Residual after rotation

(S|D|C|Z)LARFT

Single, double, single complex, and double complex LARFT.

Description

Generates the triangular factor T of a block reflector.

The block reflector matrix H is formed as a product of elementary reflectors.

When DIRECT is 'F':

$$H = H_1 * H_2 * \ldots * H_k$$

When DIRECT is 'B':

$$H = H_k * H_{k-1} * \ldots * H_1$$

If STOREV is 'C', the vectors that define each elementary reflector are stored as columns of V and:

$$H = I - V * T * V^H$$

If STOREV is 'R', the vectors that define each elementary reflector are stored as rows of V and:

$$H = I - V^H * T * V$$

LAPACK Interface

```
void slarft(const char *DIRECT, const char *STOREV, const qml_long *N,
    const qml_long *K, const float *V, const qml_long *LDV, const float *TAU,
    float *T, const qml_long *LDT);

void dlarft(const char *DIRECT, const char *STOREV, const qml_long *N,
    const qml_long *K, const double *V, const qml_long *LDV, const double *TAU,
    double *T, const qml_long *LDT);

void clarft(const char *DIRECT, const char *STOREV, const qml_long *N,
    const qml_long *K, const qml_single_complex *V, const qml_long *LDV,
    const qml_single_complex *TAU, qml_single_complex *T, const qml_long *LDT);

void zlarft(const char *DIRECT, const char *STOREV, const qml_long *N,
    const qml_long *K, const qml_double_complex *V, const qml_long *LDV,
    const qml_double_complex *TAU, qml_double_complex *T, const qml_long *LDT);
```

Arguments

DIRECT	Direction to apply elementary reflectors
STOREV	Storage format of elementary reflector vectors
N	Order of the block reflector H
K	The number of elementary reflectors in the block
V	Matrix storing reflector vectors
LDV	Leading dimension of V
TAU	Vector of scale factors τ_i
T	Triangular K x K matrix storing reflectors
LDT	Leading dimension of T

(S|D)LAS2

Single and double LAS2.

Description

Compute singular values for the 2x2 triangular matrix:

$$\begin{bmatrix} \mathtt{F} & \mathtt{G} \\ 0 & \mathtt{H} \end{bmatrix}$$

LAPACK Interface

```
void slas2(const float *F, const float *G, const float *H, float *SSMIN, float *SSMAX);

void dlas2(const double *F, const double *G, const double *H, double *SSMIN, double *SSMAX);
```

Arguments

F	Element of input matrix
G	Element of input matrix
H	Element of input matrix
SSMIN	Smallest singular value
SSMAX	Largest singular value

(S|D|C|Z)LASCL

Single, double, single complex, and double complex LASCL.

Description

Scales part of a matrix taking care to avoid over/underflow.

$$A \leftarrow \frac{\mathtt{CTO}}{\mathtt{CFROM}} A$$

Supports matrices stored in the following formats:

TYPE	Storage format
G	Full matrix
L	Lower triangular
U	Upper triangular
H	Upper Hessenberg
B	Symmetric band with lower band stored
Q	Symmetric band with upper band stored
Z	Band matrix

LAPACK Interface

```
void slascl(const char *TYPE, const qml_long *KL, const qml_long *KU,
    const float *CFROM, const float *CTO, const qml_long *M, const qml_long *N,
    float *A, const qml_long *LDA, qml_long *INFO);

void dlascl(const char *TYPE, const qml_long *KL, const qml_long *KU,
    const double *CFROM, const double *CTO, const qml_long *M, const qml_long *N,
    double *A, const qml_long *LDA, qml_long *INFO);

void clascl(const char *TYPE, const qml_long *KL, const qml_long *KU,
    const float *CFROM, const float *CTO, const qml_long *M, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, qml_long *INFO);

void zlascl(const char *TYPE, const qml_long *KL, const qml_long *KU,
    const double *CFROM, const double *CTO, const qml_long *M, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, qml_long *INFO);
```

Arguments

TYPE	Specifies storage format of A
KL	Lower bandwidth for banded storage formats
KU	Upper bandwidth for banded storage formats
CFROM	Scaling factor
CTO	Scaling factor
M	Number of rows of A
N	Number of columns of A
A	Source matrix
LDA	Leading dimension of A
INFO	0 on success

(S|D|C|Z)LASET

Single, double, single complex, and double complex LASET.

Description

Initializes diagonal and off-diagonal elements of a matrix.

LASET sets diagonal elements to β. If UPLO is L, lower triangular elements off the diagonal of A are set to α and upper triangular elements are not accessed. If UPLO is U, upper triangular elements off the diagonal of A are set to α and lower triangular elements are not accessed. Otherwise, all off-diagonal elements are set to α.

LAPACK Interface

```
void slaset(const char *uplo, const qml_long *M, const qml_long *N,
    const float *ALPHA, const float *BETA, float *A, const qml_long *LDA);

void dlaset(const char *uplo, const qml_long *M, const qml_long *N,
    const double *ALPHA, const double *BETA, double *A, const qml_long *LDA);

void claset(const char *uplo, const qml_long *M, const qml_long *N,
```

```
    const qml_single_complex *ALPHA, const qml_single_complex *BETA,
    qml_single_complex *A, const qml_long *LDA);

void zlaset(const char *uplo, const qml_long *M, const qml_long *N,
    const qml_double_complex *ALPHA, const qml_double_complex *BETA,
    qml_double_complex *A, const qml_long *LDA);
```

Arguments

UPLO	Specify which part of A to set
M	Number of rows of A
N	Number of columns of A
ALPHA	Value for off-diagonal elements of A
BETA	Value for diagonal elements of A
A	Destination matrix
LDA	Leading dimension of A

(S|D)LASQ1

Single and double LASQ1.

Description

Computes the singular values of a square bidiagonal matrix to high relative accuracy.

LAPACK Interface

```
void slasq1(const qml_long *N, float *D, float *E, float *WORK, qml_long *INFO);

void dlasq1(const qml_long *N, double *D, double *E, double *WORK, qml_long *INFO);
```

Arguments

N	Number of rows and columns of matrix A
D	Vector of diagonal elements of A of length N, overwritten by sorted singular values on exit
E	Vector of off-diagonal elements of A of length N - 1, destroyed on exit
WORK	Work space of size 4N
INFO	0 on success, <0 for bad arguments, >0 if no convergence

(S|D|C|Z)LASR

Single, double, single complex, and double complex LASR.

Description

Applies a sequence of plane rotations to a rectangular matrix.

When SIDE is L, LASR performs:

$$A \leftarrow P * A$$

When SIDE is R, LASR performs:

$$A \leftarrow A * P^T$$

The matrix P is an orthogonal matrix formed as the product of plane rotations.

When DIRECT is F:

$$P = P_K * \ldots * P_2 * P_1$$

When DIRECT is B:

$$P = P_1 * P_2 * \ldots * P_K$$

Each plane rotation is formed around a pivot.

When PIVOT is T:

$$P_k = \begin{bmatrix} c_k & & & & s_k & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ -s_k & & & & c_k & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix}$$

When PIVOT is B:

$$P_k = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_k & & & & s_k \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & 1 & \\ & & & -s_k & & & & c_k \end{bmatrix}$$

When PIVOT is V:

$$P_k = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_k & s_k & & & \\ & & & -s_k & c_k & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}$$

LAPACK Interface

```
void slasr(const char *SIDE, const char *PIVOT, const char *DIRECT,
    const qml_long *M,const qml_long *N, const float *C, const float *S,
    float *A, const qml_long *LDA);

void dlasr(const char *SIDE, const char *PIVOT, const char *DIRECT,
    const qml_long *M, const qml_long *N, const double *C, const double *S,
    double *A, const qml_long *LDA);

void clasr(const char *SIDE, const char *PIVOT, const char *DIRECT,
    const qml_long *M, const qml_long *N, const float *C, const float *S,
    qml_single_complex *A, const qml_long *LDA);

void zlasr(const char *SIDE, const char *PIVOT, const char *DIRECT,
    const qml_long *M, const qml_long *N, const double *C, const double *S,
    qml_double_complex *A, const qml_long *LDA);
```

Arguments

SIDE	Which side to apply plane rotations to
PIVOT	Positions of pivots
DIRECT	Direction to apply rotations
M	Number of rows of A
N	Number of columns of A
C	Array of cosine values for plane rotations
S	Array of sine values for plane rotations
A	Matrix of size M x N
LDA	Leading dimension of A

(S|D)LASRT

Single and double LASRT.

Description

Sorts an array of numbers.

LAPACK Interface

```
void slasrt(const char *ID, const qml_long *N, float *D, qml_long *INFO);

void dlasrt(const char *ID, const qml_long *N, double *D, qml_long *INFO);
```

Arguments

ID	Order to short, 'I' for increasing, 'D' for decreasing
N	Length of the array
D	Array to be sorted, modified in place
INFO	0 on success

(S|D|C|Z)LASSQ

Single, double, single complex, and double complex LASSQ.

Description

Updates a scaled sum of squares value.

The scaled sum of squares storage format consists of two values SCALE and SUMSQ which together represent the value:

$$\texttt{SCALE}^2 * \texttt{SUMSQ}$$

LASSQ updates an existing scaled sum of squares value to include the sum of squares of vector X.

LAPACK Interface

```
void slassq(const qml_long *N, const float *X, const qml_long *INCX,
    float *SCALE, float *SUMSQ);

void dlassq(const qml_long *N, const double *X, const qml_long *INCX,
    double *SCALE, double *SUMSQ);

void classq(const qml_long *N, const qml_single_complex *X, const qml_long *INCX,
    float *SCALE, float *SUMSQ);

void zlassq(const qml_long *N, const qml_double_complex *X, const qml_long *INCX,
    double *SCALE, double *SUMSQ);
```

Arguments

N	Size of the array
X	Input array
INCX	Stride between elements of X
SCALE	Initial scale value, overwritten with updated value
SUMSQ	Initial sum of squares value, overwritten with updated value

(S|D)LASV2

Single and double LASV2.

Description

Computes singular value decomposition for the 2x2 triangular matrix:

$$\begin{bmatrix} \texttt{F} & \texttt{G} \\ 0 & \texttt{H} \end{bmatrix}$$

The larger singular value is returned in SSMAX and the smaller in SSMIN. The left singular vector for SSMAX is (CSL, SNL) and the right singular vector for SSMAX is (CSR, SNR). On exit:

$$\begin{bmatrix} \texttt{CSL} & \texttt{SNL} \\ -\texttt{SNL} & \texttt{CSL} \end{bmatrix} * \begin{bmatrix} \texttt{F} & \texttt{G} \\ 0 & \texttt{H} \end{bmatrix} * \begin{bmatrix} \texttt{CSR} & -\texttt{SNR} \\ \texttt{SNR} & \texttt{CSR} \end{bmatrix} = \begin{bmatrix} \texttt{SSMAX} & 0 \\ 0 & \texttt{SSMIN} \end{bmatrix}$$

LAPACK Interface

```
void slasv2(const float *F, const float *G, const float *H, float *SSMIN,
    float *SSMAX, float *SNR, float *CSR, float *SNL, float *CSL);

void dlasv2(const double *F, const double *G, const double *H, double *SSMIN,
    double *SSMAX, double *SNR, double *CSR, double *SNL, double *CSL);
```

Arguments

F	Element of input matrix
G	Element of input matrix
H	Element of input matrix
SSMIN	Smallest singular value
SSMAX	Largest singular value
SNR	Second part of right singular vector
CSR	First part of right singular vector
SNL	Second part of left singular vector
CSL	First part of left singular vector

(S|D)ORGQR

Single and double ORGQR.

Description

Generates an orthonormal matrix given a set of elementary reflectors and scale factors.

$$Q = H_1 * H_2 * \ldots * H_K$$

The elementary reflectors and scale factors are expected in the same form as generated by GEQRF.

$$H_i = I - \tau_i * v_i * {v_i}^H$$

LAPACK Interface

```
void sorgqr(const qml_long *M, const qml_long *N, const qml_long *K, float *A,
    const qml_long *LDA, float *TAU, float *WORK, const qml_long *LWORK,
    qml_long *INFO);

void dorgqr(const qml_long *M, const qml_long *N, const qml_long *K, double *A,
    const qml_long *LDA, double *TAU, double *WORK, const qml_long *LWORK,
    qml_long *INFO);
```

Arguments

M	Number of rows of Q
N	Number of columns of Q (M >= N)
K	Number of elementary reflectors (N >= K)
A	Matrix of size M x N containing reflectors in columns, overwritten on exit with Q
LDA	Leading dimension of A
TAU	Vector of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(S|D)ORGTR

Single and double ORGTR.

Description

Generates an orthonormal matrix given a set of elementary reflectors and scale factors.

$$Q = H_1 * H_2 * \ldots * H_K$$

The elementary reflectors and scale factors are expected in the same form as generated by SYTRD.

When UPLO is U:

$$Q = H_{N-1} * \ldots * H_2 * H_1$$

$$H_i = I - \tau_i * v * v^T$$

The vector v has components i+1 through n of zero, component i is 1, and components 1 through i-1 are stored in the columns of A above the superdiagonal.

When UPLO is L:

$$Q = H_1 * H_2 * \ldots * H_{N-1}$$

$$H_i = I - \tau_i * v * v^T$$

The vector v has components 1 through i of zero, component i+1 is 1, and components i+2 through N are stored in the columns of A below the subdiagonal.

LAPACK Interface

```
void sorgtr(const char *UPLO, const qml_long *N, float *A, const qml_long *LDA,
    float *TAU, float *WORK, const qml_long *LWORK, qml_long *INFO);

void dorgtr(const char *UPLO, const qml_long *N, double *A, const qml_long *LDA,
    double *TAU, double *WORK, const qml_long *LWORK, qml_long *INFO);
```

Arguments

UPLO	Store upper 'U' or lower 'L' triangular part of A
N	Number of rows and columns of A
A	Elementary reflector vectors, overwritten with Q on exit
LDA	Leading dimension of A
TAU	Array of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(S|D|C|Z)POSV

Single, double, single complex, and double complex POSV.

Description

Solves a system of linear equations for multiple right-hand sides for positive definite matrices.

$$A * X = B$$

Uses Cholesky decomposition to factor A.

If UPLO is 'U' then A is factored using an upper triangular matrix U:

$$A = U^H * U$$

If UPLO is 'L' then A is factored using a lower triangular matrix L:

$$A = L * L^H$$

The factored form is used to solve the system of equations.

LAPACK Interface

```
void sposv(const char *UPLO, const qml_long *N, const qml_long *NRHS, float *A,
    const qml_long *LDA, float *B, const qml_long *LDB, qml_long *INFO);

void dposv(const char *UPLO, const qml_long *N, const qml_long *NRHS, double *A,
    const qml_long *LDA, double *B, const qml_long *LDB, qml_long *INFO);

void cposv(const char *UPLO, const qml_long *N, const qml_long *NRHS,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *B,
    const qml_long *LDB, qml_long *INFO);

void zposv(const char *UPLO, const qml_long *N, const qml_long *NRHS,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *B,
    const qml_long *LDB, qml_long *INFO);
```

Arguments

UPLO	Specify whether to factor using 'U' upper triangular or 'L' lower triangular
N	Number of linear equations, order of the matrix A
NRHS	Number of right hand sides, number of columns of B
A	Matrix of size N x N, overwritten with factorization on exit
LDA	Leading dimension of A
B	On entry, the N x NRHS matrix B, on exit the solution matrix
LDB	Leading dimension of B
INFO	0 on success, <0 on illegal arguments, >0 if not positive definite

(S|D|C|Z)POTF2

Single, double, single complex, and double complex POTF2.

Description

Computes the Cholesky factorization of a positive definite matrix using an unblocked algorithm.

If UPLO is 'U' then A is factored using an upper triangular matrix U:

$$A = U^H * U$$

If UPLO is 'L' then A is factored using a lower triangular matrix L:

$$A = L * L^H$$

The upper or lower triangular part of A is overwritten on exit with U or L. The opposite side is not modified.

LAPACK Interface

```
void spotf2(const char *UPLO, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *INFO);

void dpotf2(const char *UPLO, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *INFO);

void cpotf2(const char *UPLO, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *INFO);

void zpotf2(const char *UPLO, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *INFO);
```

Arguments

UPLO	Specify whether to factor using 'U' upper triangular or 'L' lower triangular
N	Order of the matrix A
A	Matrix of size N x N, overwritten with factorization on exit
LDA	Leading dimension of A
INFO	0 on success, <0 on illegal arguments, >0 if not positive definite

(S|D|C|Z)POTRF

Single, double, single complex, and double complex POTRF.

Description

Computes the Cholesky factorization of a positive definite matrix using a blocked algorithm.

If UPLO is 'U' then A is factored using an upper triangular matrix U:

$$A = U^H * U$$

If UPLO is 'L' then A is factored using a lower triangular matrix L:

$$A = L * L^H$$

The upper or lower triangular part of A is overwritten on exit with U or L. The opposite side is not modified.

LAPACK Interface

```
void spotrf(const char *UPLO, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *INFO);

void dpotrf(const char *UPLO, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *INFO);

void cpotrf(const char *UPLO, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *INFO);

void zpotrf(const char *UPLO, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *INFO);
```

Arguments

UPLO	Specify whether to factor using 'U' upper triangular or 'L' lower triangular
N	Order of the matrix A
A	Matrix of size N x N, overwritten with factorization on exit
LDA	Leading dimension of A
INFO	0 on success, <0 on illegal arguments, >0 if not positive definite

(S|D|C|Z)POTRI

Single, double, single complex, and double complex POTRI.

Description

Computes the inverse of a Hermitian positive definite matrix using Cholesky factorization.

$$\mathrm{A} = \mathrm{U}^H * \mathrm{U}$$

$$\mathrm{A} = \mathrm{L} * \mathrm{L}^H$$

LAPACK Interface

```
void spotri(const char *UPLO, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *INFO);

void dpotri(const char *UPLO, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *INFO);

void cpotri(const char *UPLO, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *INFO);

void zpotri(const char *UPLO, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *INFO);
```

Arguments

UPLO	Set to 'U' for upper triangular factorization, 'L' for lower
N	Number of rows and columns of A
A	Matrix of size N x N containing Cholesky factorization as output by POTRF
LDA	Leading dimension of A
INFO	0 on success, <0 for bad arguments, >0 if no inverse

(S|D|C|Z)POTRS

Single, double, single complex, and double complex POTRS.

Description

Solves a system of linear equations for multiple right-hand sides given the Cholesky factorization of the matrix.

Solves:

$$A * X = B$$

If UPLO is 'U' then A has been factored using an upper triangular matrix U:

$$A = U^H * U$$

If UPLO is 'L' then A has been factored using a lower triangular matrix L:

$$A = L * L^H$$

The factored form is used to solve the system of equations. Only the upper or lower triangular part of A is referenced.

To get the Cholesky factorization of a matrix use POTRF.

LAPACK Interface

```
void spotrs(const char *UPLO, const qml_long *N, const qml_long *NRHS, float *A,
    const qml_long *LDA, float *B, const qml_long *LDB, qml_long *INFO);

void dpotrs(const char *UPLO, const qml_long *N, const qml_long *NRHS, double *A,
    const qml_long *LDA, double *B, const qml_long *LDB, qml_long *INFO);
```

```
void cpotrs(const char *UPLO, const qml_long *N, const qml_long *NRHS,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *B,
    const qml_long *LDB, qml_long *INFO);

void zpotrs(const char *UPLO, const qml_long *N, const qml_long *NRHS,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *B,
    const qml_long *LDB, qml_long *INFO);
```

Arguments

UPLO	Specify whether A has been factored in 'U' upper triangular or 'L' lower triangular form
N	Number of linear equations, order of the matrix A
NRHS	Number of right hand sides, number of columns of B
A	Matrix of size N x N in factored form
LDA	Leading dimension of A
B	On entry, the N x NRHS matrix B, on exit the solution matrix
LDB	Leading dimension of B
INFO	0 on success, <0 on illegal arguments

(S|D|C|Z)STEDC

Single, double, single complex, and double complex STEDC.

Description

Computes the eigenvalues of a symmetric tridiagonal matrix and optionally the eigenvectors.

Uses a divide-and-conquer method. If COMPZ is V then Z must be set to the orthogonal matrix that reduces the original symmetric matrix to tridiagonal form.

LAPACK Interface

```
void sstedc(const char *COMPZ, const qml_long *N, float *D, float *E, float *Z,
    const qml_long *LDZ, float *WORK, const qml_long *LWORK,
    qml_long *IWORK, const qml_long *LIWORK, qml_long *INFO);

void dstedc(const char *COMPZ, const qml_long *N, double *D, double *E, double *Z,
    const qml_long *LDZ, double *WORK, const qml_long *LWORK,
    qml_long *IWORK, const qml_long *LIWORK, qml_long *INFO);

void cstedc(const char *COMPZ, const qml_long *N, float *D, float *E,
    qml_single_complex *Z, const qml_long *LDZ, qml_single_complex *WORK,
    const qml_long *LWORK, float *RWORK, const qml_long *LRWORK,
    qml_long *IWORK, const qml_long *LIWORK, qml_long *INFO);

void zstedc(const char *COMPZ, const qml_long *N, double *D, double *E,
    qml_double_complex *Z, const qml_long *LDZ, qml_double_complex *WORK,
    const qml_long *LWORK, double *RWORK, const qml_long *LRWORK,
    qml_long *IWORK, const qml_long *LIWORK, qml_long *INFO);
```

Arguments

COMPZ	Computes eigenvalues only ('N'), eigenvalues and eigenvectors ('V'), or eigenvectors and eigenvalues of tridiagonal input ('I')
N	Order of the matrix
D	Array of diagonal entries, overwritten with eigenvalues on exit
E	Array of subdiagonal entries, overwritten on exit
Z	Orthogonal matrix to reduce to tridiagonal form, overwritten with eigenvectors
LDZ	Leading dimension of Z
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
RWORK	Work space of size LRWORK
LRWORK	Size of work space (-1 to query)
IWORK	Integer work space of size LIWORK
LIWORK	Size of integer work space (-1 to query)
INFO	0 on success, <0 for illegal arguments, >0 on failure to converge

(S|D|C|Z)STEQR

Single, double, single complex, and double complex STEQR.

Description

Computes the eigenvalues of a symmetric tridiagonal matrix and optionally the eigenvectors.

Uses the QL or QR method. If COMPZ is V then Z must be set to the orthogonal matrix that reduces the original symmetric matrix to tridiagonal form.

LAPACK Interface

```
void ssteqr(const char *COMPZ, const qml_long *N, float *D, float *E, float *Z,
    const qml_long *LDZ, float *WORK, qml_long *INFO);

void dsteqr(const char *COMPZ, const qml_long *N, double *D, double *E, double *Z,
    const qml_long *LDZ, double *WORK, qml_long *INFO);

void csteqr(const char *COMPZ, const qml_long *N, float *D, float *E,
    qml_single_complex *Z, const qml_long *LDZ, float *WORK,
    qml_long *INFO);

void zsteqr(const char *COMPZ, const qml_long *N, double *D, double *E,
    qml_double_complex *Z, const qml_long *LDZ, double *WORK,
    qml_long *INFO);
```

Arguments

COMPZ	Computes eigenvalues only ('N'), eigenvalues and eigenvectors ('V'), or eigenvectors and eigenvalues of tridiagonal input ('I')
N	Order of the matrix
D	Array of diagonal entries, overwritten with eigenvalues on exit
E	Array of subdiagonal entries, overwritten on exit
Z	Orthogonal matrix to reduce to tridiagonal form, overwritten with eigenvectors
LDZ	Leading dimension of Z
WORK	Work space of size at least 2N-2
INFO	0 on success, <0 for illegal arguments, >0 on failure to converge

(S|D)SYEV

Single and double SYEV.

Description

Computes the eigenvalues and (optionally) eigenvectors of a symmetric matrix.

Right eigenvectors satisfy:

$$\mathtt{A} * v_i = \lambda_i v_i$$

Left eigenvectors satisfy:

$${u_i}^T * \mathtt{A} = \lambda {u_i}^T$$

Computed eigenvectors are normalized.

LAPACK Interface

```
void ssyev(const char *JOBZ, const char *UPLO, const qml_long *N, float *A,
    const qml_long *LDA, float *W, float *WORK, const qml_long *LWORK,
    qml_long *INFO);

void dsyev(const char *JOBZ, const char *UPLO, const qml_long *N, double *A,
    const qml_long *LDA, double *W, double *WORK, const qml_long *LWORK,
    qml_long *INFO);
```

Arguments

JOBZ	Compute eigenvectors and store in A if 'V', otherwise do not compute
UPLO	Specify whether to use 'U' upper triangular or 'L' lower triangular part of A
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten on exit
LDA	Leading dimension of A
W	Array of size N containing eigenvalues on exit
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(S|D)SYEVD

Single and double SYEVD.

Description

Computes the eigenvalues and (optionally) eigenvectors of a symmetric matrix using a divide-and-conquer algorithm.

Right eigenvectors satisfy:

$$\mathtt{A} * v_i = \lambda_i v_i$$

Left eigenvectors satisfy:

$${u_i}^T * \mathtt{A} = \lambda {u_i}^T$$

Computed eigenvectors are normalized.

LAPACK Interface

```
void ssyevd(const char *JOBZ, const char *UPLO, const qml_long *N, float *A,
    const qml_long *LDA, float *W, float *WORK, const qml_long *LWORK,
    qml_long *IWORK, const qml_long *LIWORK, qml_long *INFO);

void dsyevd(const char *JOBZ, const char *UPLO, const qml_long *N, double *A,
    const qml_long *LDA, double *W, double *WORK, const qml_long *LWORK,
    qml_long *IWORK, const qml_long *LIWORK, qml_long *INFO);
```

Arguments

JOBZ	Compute eigenvectors and store in A if 'V', otherwise do not compute
UPLO	Specify whether to use 'U' upper triangular or 'L' lower triangular part of A
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten on exit
LDA	Leading dimension of A
W	Array of size N containing eigenvalues on exit
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
IWORK	Integer work space of size at least LIWORK
LIWORK	Size of integer work space (-1 to query)
INFO	0 on success, <0 for illegal arguments, >0 if convergence failed

(S|D)SYTRD

Single and double SYTRD.

Description

Reduces a symmetric matrix to symmetric tridiagonal form using an orthogonal similarity transform.

$$T = Q^T * A * Q$$

The matrix T is symmetric tridiagonal and the matrix Q is orthogonal.

The matrix Q is not stored explicitly. Instead, the elements above or below the diagonal of A together with TAU store Q as the product of scaled elementary reflectors.

When UPLO is U:

$$Q = H_{N-1} * \ldots * H_2 * H_1$$

$$H_i = I - \tau_i * v * v^T$$

The vector v has components i+1 through n of zero, component i is 1, and components 1 through i-1 are stored in the columns of A above the superdiagonal.

When UPLO is L:

$$Q = H_1 * H_2 * \ldots * H_{N-1}$$

$$H_i = I - \tau_i * v * v^T$$

The vector v has components 1 through i of zero, component i+1 is 1, and components i+2 through N are stored in the columns of A below the subdiagonal.

LAPACK Interface

```
void ssytrd(const char *UPLO, const qml_long *N, float *A, const qml_long *LDA,
    float *D, float *E, float *TAU, float *WORK, const qml_long *LWORK,
    qml_long *INFO);

void dsytrd(const char *UPLO, const qml_long *N, double *A, const qml_long *LDA,
    double *D, double *E, double *TAU, double *WORK, const qml_long *LWORK,
    qml_long *INFO);
```

Arguments

UPLO	Store upper 'U' or lower 'L' triangular part of A
N	Number of rows and columns of A
A	Symmetric matrix, overwritten by T and reflector vectors on exit
LDA	Leading dimension of A
D	On exit contains diagonal elements of T
E	On exit contains the off diagonal elements of T
TAU	On exit contains vector of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(S|D|C|Z)SYTRF

Single, double, single complex, and double complex SYTRF.

Description

Computes the factorization of a symmetric matrix using one of the following factorizations:

$$A = U * D * U^T$$

$$A = L * D * L^T$$

Uses the Bunch-Kaufman diagonal pivoting method. U is a product of permutation and unit upper triangular matrices. L is a product of permutation and unit lower triangular matrices. D is symmetric with block diagonal structure, with blocks of size 1x1 or 2x2.

LAPACK Interface

```
void ssytrf(const char *UPLO, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *IPIV, float *WORK, const qml_long *LWORK, qml_long *INFO);

void dsytrf(const char *UPLO, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *IPIV, double *WORK, const qml_long *LWORK, qml_long *INFO);

void csytrf(const char *UPLO, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_single_complex *WORK,
    const qml_long *LWORK, qml_long *INFO);

void zsytrf(const char *UPLO, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_double_complex *WORK,
    const qml_long *LWORK, qml_long *INFO);
```

Arguments

UPLO	Set to 'U' for upper triangular factorization, 'L' for lower
N	Number of rows and columns of A
A	Matrix of size N x N, overwritten with D on exit
LDA	Leading dimension of A
IPIV	Array of pivots of size N, negative indicates 2x2 block pivots
WORK	Work space
LWORK	Size of WORK, -1 to query
INFO	0 on success, <0 for bad arguments, >0 if singular

(S|D|C|Z)SYTRI

Single, double, single complex, and double complex SYTRI.

Description

Computes the inverse of a symmetric indefinite matrix using one of the following factorizations:

$$A = U * D * U^T$$

$$A = L * D * L^T$$

LAPACK Interface

```
void ssytri(const char *UPLO, const qml_long *N, float *A, const qml_long *LDA,
    qml_long *IPIV, float *WORK, qml_long *INFO);

void dsytri(const char *UPLO, const qml_long *N, double *A, const qml_long *LDA,
    qml_long *IPIV, double *WORK, qml_long *INFO);

void csytri(const char *UPLO, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_single_complex *WORK,
    qml_long *INFO);

void zsytri(const char *UPLO, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_long *IPIV, qml_double_complex *WORK,
    qml_long *INFO);
```

Arguments

UPLO	Set to 'U' for upper triangular factorization, 'L' for lower
N	Number of rows and columns of A
A	Matrix of size N x N containing block diagonal matrix D with multipliers, as output by SYTRF
LDA	Leading dimension of A
IPIV	Array of pivots of size N, as provided by SYTRF
WORK	Work space of size at least 2N
INFO	0 on success, <0 for bad arguments, >0 if singular

(S|D|C|Z)TREVC

Single, double, single complex, and double complex TREVC.

Description

Computes the left/right eigenvectors of an upper (quasi) triangular matrix T.

Upper (quasi) triangular matrices are produced during Schur factorization, for example by HSEQR.

Right eigenvectors x satisfy:

$$T * x = \lambda x$$

Left eigenvectors y satisfy:

$$y^H * T = \lambda y^H$$

Eigenvalues λ are not passed explicitly, they are taken from the diagonal of T. Optionally a matrix Q can be given that reduces an original matrix A to upper triangular form, which allows the calculation of the eigenvectors of A (set HOWMNY to B).

The real versions of this function accept quasi triangular form which allows 1x1 and 2x2 blocks along the diagonal of T corresponding to real and pairs of complex conjugate eigenvalues.

LAPACK Interface

```
void strevc(const char *SIDE, const char *HOWMNY, qml_int *SELECT,
    const qml_long *N, float *T, const qml_long *LDT, float *VL,
    const qml_long *LDVL, float *VR, const qml_long *LDVR,
    const qml_long *MM, qml_long *M, float *WORK, qml_long *INFO);

void dtrevc(const char *SIDE, const char *HOWMNY, qml_int *SELECT,
    const qml_long *N, double *T, const qml_long *LDT, double *VL,
    const qml_long *LDVL, double *VR, const qml_long *LDVR,
    const qml_long *MM, qml_long *M, double *WORK, qml_long *INFO);

void ctrevc(const char *SIDE, const char *HOWMNY, qml_int *SELECT,
    const qml_long *N, qml_single_complex *T, const qml_long *LDT,
    qml_single_complex *VL, const qml_long *LDVL, qml_single_complex *VR,
    const qml_long *LDVR, const qml_long *MM, qml_long *M,
    qml_single_complex *WORK, float *RWORK, qml_long *INFO);

void ztrevc(const char *SIDE, const char *HOWMNY, qml_int *SELECT,
    const qml_long *N, qml_double_complex *T, const qml_long *LDT,
    qml_double_complex *VL, const qml_long *LDVL, qml_double_complex *VR,
    const qml_long *LDVR, const qml_long *MM, qml_long *M,
    qml_double_complex *WORK, double *RWORK, qml_long *INFO);
```

Arguments

SIDE	Compute right 'R', left 'L', or both 'B' eigenvectors
HOWMNY	Compute all eigenvectors 'A', backtransformed eigenvectors 'B', or selected eigenvectors 'S'
SELECT	Boolean array choosing eigenvectors used when HOWMNY is 'S'
N	Number of rows and columns of T
T	Upper (quasi) triangular matrix, modified but restored on exit
LDT	Leading dimension of T
VL	Matrix of size N x N containing Q, overwritten with left eigenvectors
LDVL	Leading dimension of VL
VR	Matrix of size N x N containing Q, overwritten with right eigenvectors
LDVR	Leading dimension of VR
MM	Number of columns in VL and VR
M	On exit set to number of columns in VL and VR used to store eigenvectors
WORK	Work space of size 3N
RWORK	Work space of size N
INFO	0 on success

(S|D|C|Z)TRSYL

Single, double, single complex, and double complex TRSYL.

Description

Solves one of the Sylvester equations:

$$A * X + X * B = \alpha C$$

$$A^H * X + X * B = \alpha C$$

$$A * X + X * B^H = \alpha C$$

$$A^H * X + X * B^H = \alpha C$$

$$A * X - X * B = \alpha C$$

$$A^H * X - X * B = \alpha C$$

$$A * X - X * B^H = \alpha C$$

$$A^H * X - X * B^H = \alpha C$$

Matrices A and B are upper triangular, A is M x M and B is N x N. Matrices C and X are M x N. The scale factor α is set less than one to avoid overflow in X.

LAPACK Interface

```
void strsyl(const char *TRANA, const char *TRANB, const qml_long *ISGN,
    const qml_long *M, const qml_long *N, float *A, const qml_long *LDA,
    float *B, const qml_long *LDB, float *C, const qml_long *LDC,
    float *SCALE, qml_long *INFO);

void dtrsyl(const char *TRANA, const char *TRANB, const qml_long *ISGN,
    const qml_long *M, const qml_long *N, double *A, const qml_long *LDA,
    double *B, const qml_long *LDB, double *C, const qml_long *LDC,
    double *SCALE, qml_long *INFO);

void ctrsyl(const char *TRANA, const char *TRANB, const qml_long *ISGN,
    const qml_long *M, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_single_complex *B, const qml_long *LDB,
    qml_single_complex *C, const qml_long *LDC, float *SCALE,
    qml_long *INFO);

void ztrsyl(const char *TRANA, const char *TRANB, const qml_long *ISGN,
    const qml_long *M, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_double_complex *B, const qml_long *LDB,
    qml_double_complex *C, const qml_long *LDC, double *SCALE,
    qml_long *INFO);
```

Arguments

TRANA	Specifies transform option for A, 'N' for none, 'T' for transpose, 'C' for conjugate transpose
TRANB	Specifies transform option for B, 'N' for none, 'T' for transpose, 'C' for conjugate transpose
ISGN	Solve $A * X + X * B = \alpha C$ (+1) or $A * X - X * B = \alpha C$ (-1)
M	Number of rows and columns of A
N	Number of rows and columns of B
A	Upper triangular matrix
LDA	Leading dimension of A
B	Upper triangular matrix
LDB	Leading dimension of B
C	Right hand side matrix, on exit overwritten by solution matrix X
LDC	Leading dimension of C
SCALE	Scale factor α to avoid overflow in X
INFO	0 on success, <0 on bad arguments, 1 if A and B have very close eigenvalues

(S|D|C|Z)TRTI2

Single, double, single complex, and double complex TRTI2.

Description

Computes the inverse of a triangular matrix using an unblocked algorithm.

If UPLO is 'U' then A is assumed to be upper triangular and the lower triangular part is not accessed.

If UPLO is 'L' then A is assumed to be lower triangular and the upper triangular part is not accessed.

If DIAG is 'U' then the diagonal of A is not accessed and is assumed to be 1. If Diag is 'N' then the diagonal of A is accessed.

LAPACK Interface

```
void strti2(const char *UPLO, const char *DIAG, const qml_long *N, float *A,
    const qml_long *LDA, qml_long *INFO);

void dtrti2(const char *UPLO, const char *DIAG, const qml_long *N, double *A,
    const qml_long *LDA, qml_long *INFO);

void ctrti2(const char *UPLO, const char *DIAG, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, qml_long *INFO);

void ztrti2(const char *UPLO, const char *DIAG, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, qml_long *INFO);
```

Arguments

UPLO	Specify whether to use 'U' upper triangular or 'L' lower triangular part of A
DIAG	Specify whether to assume 'N' non-unit triangular or 'U' unit triangular
N	Order of the matrix A
A	Matrix of size N x N, overwritten with inverse on exit
LDA	Leading dimension of A
INFO	0 on success, <0 on illegal arguments

(S|D|C|Z)TRTRI

Single, double, single complex, and double complex TRTRI.

Description

Computes the inverse of a triangular matrix using a blocked algorithm.

If UPLO is 'U' then A is assumed to be upper triangular and the lower triangular part is not accessed.

If UPLO is 'L' then A is assumed to be lower triangular and the upper triangular part is not accessed.

If DIAG is 'U' then the diagonal of A is not accessed and is assumed to be 1. If Diag is 'N' then the diagonal of A is accessed.

LAPACK Interface

```
void strtri(const char *UPLO, const char *DIAG, const qml_long *N, float *A,
    const qml_long *LDA, qml_long *INFO);

void dtrtri(const char *UPLO, const char *DIAG, const qml_long *N, double *A,
    const qml_long *LDA, qml_long *INFO);

void ctrtri(const char *UPLO, const char *DIAG, const qml_long *N,
    qml_single_complex *A, const qml_long *LDA, qml_long *INFO);

void ztrtri(const char *UPLO, const char *DIAG, const qml_long *N,
    qml_double_complex *A, const qml_long *LDA, qml_long *INFO);
```

Arguments

UPLO	Specify whether to use 'U' upper triangular or 'L' lower triangular part of A
DIAG	Specify whether to assume 'N' non-unit triangular or 'U' unit triangular
N	Order of the matrix A
A	Matrix of size N x N, overwritten with inverse on exit
LDA	Leading dimension of A
INFO	0 on success, <0 on illegal arguments, >0 if singular

(C|Z)UNGHR

Single complex and double complex UNGHR.

Description

Generates a unitary matrix given a set of elementary reflectors and scale factors.

$$Q = H_{ILO} * H_{ILO+1} * \ldots * H_{IHI-1}$$

The elementary reflectors and scale factors are expected in the same form as generated by GEHRD.

LAPACK Interface

```
void cunghr(const qml_long *N, const qml_long *ILO, const qml_long *IHI,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *TAU,
    qml_single_complex *WORK, const qml_long *LWORK, qml_long *INFO);

void zunghr(const qml_long *N, const qml_long *ILO, const qml_long *IHI,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *TAU,
    qml_double_complex *WORK, const qml_long *LWORK, qml_long *INFO);
```

Arguments

N	Number of rows and columns of A
ILO	Lower index
IHI	Upper index, 1 <= ILO <= IHI <= N
A	Elementary reflector vectors, overwritten with Q on exit
LDA	Leading dimension of A
TAU	Array of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(C|Z)UNGQR

Single complex and double complex UNGQR.

Description

Generates an orthonormal matrix given a set of elementary reflectors and scale factors.

$$Q = H_1 * H_2 * \ldots * H_K$$

The elementary reflectors and scale factors are expected in the same form as generated by GEQRF.

$$H_i = I - \tau_i * v_i * {v_i}^H$$

LAPACK Interface

```
void cungqr(const qml_long *M, const qml_long *N, const qml_long *K,
    qml_single_complex *A, const qml_long *LDA, qml_single_complex *TAU,
    qml_single_complex *WORK, const qml_long *LWORK, qml_long *INFO);

void zungqr(const qml_long *M, const qml_long *N, const qml_long *K,
    qml_double_complex *A, const qml_long *LDA, qml_double_complex *TAU,
    qml_double_complex *WORK, const qml_long *LWORK, qml_long *INFO);
```

Arguments

M	Number of rows of Q
N	Number of columns of Q (M >= N)
K	Number of elementary reflectors (N >= K)
A	Matrix of size M x N containing reflectors in columns, overwritten on exit with Q
LDA	Leading dimension of A
TAU	Vector of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

(C|Z)UNGTR

Single complex and double complex UNGTR.

Description

Generates an orthonormal matrix given a set of elementary reflectors and scale factors.

$$Q = H_1 * H_2 * \ldots * H_K$$

The elementary reflectors and scale factors are expected in the same form as generated by HETRD.

When UPLO is U:

$$Q = H_{N-1} * \ldots * H_2 * H_1$$

$$H_i = I - \tau_i * v * v^H$$

The vector v has components i+1 through n of zero, component i is 1, and components 1 through i-1 are stored in the columns of A above the superdiagonal.

When UPLO is L:

$$Q = H_1 * H_2 * \ldots * H_{N-1}$$

$$H_i = I - \tau_i * v * v^H$$

The vector v has components 1 through i of zero, component i+1 is 1, and components i+2 through N are stored in the columns of A below the subdiagonal.

LAPACK Interface

```
void cungtr(const char *UPLO, const qml_long *N, qml_single_complex *A,
    const qml_long *LDA, qml_single_complex *TAU,
    qml_single_complex *WORK, const qml_long *LWORK, qml_long *INFO);

void zungtr(const char *UPLO, const qml_long *N, qml_double_complex *A,
    const qml_long *LDA, qml_double_complex *TAU,
    qml_double_complex *WORK, const qml_long *LWORK, qml_long *INFO);
```

Arguments

UPLO	Store upper 'U' or lower 'L' triangular part of A
N	Number of rows and columns of A
A	Elementary reflector vectors, overwritten with Q on exit
LDA	Leading dimension of A
TAU	Array of scale factors for reflectors
WORK	Work space of size at least LWORK
LWORK	Size of work space (-1 to query)
INFO	0 on success

CHAPTER

NINETEEN

QML EXTENSIONS

QML provides a limited number of extensions to the standard primitives such as those offered in LAPACK or BLAS. These primitives are less general and more tightly coupled to specific domains such as machine learning or augmented reality. Below are the extensions supported by QML.

(S)CONV_MM

Single precision CONV_MM.

Description

Takes an image and a set of filters and performs a convolution using matrix multiplication. The image must be stored in channel major format.

Extension Interface

```
void sconv_mm(const bool biased, const float* src, const qml_int srcWidth,
              const qml_int srcHeight, const qml_int srcNumChannels,
              const float* filters, const qml_int numFilters,
              const qml_int filterX, const qml_int filterY, const qml_int padLeft,
              const qml_int padTop, const qml_int strideX, const qml_int strideY,
              float* output, const qml_int outputWidth, const qml_int outputHeight);
```

Arguments

biased	Whether or not a bias has already been applied to the output
src	Input image, assumed to be stored in channel major format
srcX	Width of the image
srcY	Height of the image
channels	Number of channels in the image
filters	Filters, assumed to be stored in channel major format
numFilters	Number of filters
filterX	Width of the filters
filterY	Height of the filters
padX	Amount of padding added to the left and right of the image
padY	Amount of padding added to the top and bottom of the image
strideX	Filter stride in the X dimension
strideY	Filter stride in the Y dimension
output	Output, assumed to be stored in channel major format
outputX	Width of an individual output
outputy	Height of an individual output

I32I8CONV

Convolution with int8 input and int32 output.

Description

Takes an image and a set of filters and performs a convolution using dot-product instruction.

Extension Interface

```
void i32i8conv(const QML_IMAGE_FORMAT inputFormat, const QML_IMAGE_FORMAT outputFormat,
               const bool biased, const qml_int8* src,
               const qml_int srcX, const qml_int srcY,
               const qml_int channels, const qml_int8* filters,
               const qml_int numFilters, const qml_int filterX, const qml_int filterY,
               const qml_int padLeft, const qml_int padTop,
               const qml_int strideX, const qml_int strideY,
               qml_int* output, const qml_int outputX, const qml_int outputY);
```

Arguments

inputFormat	Input image format (channel-major or image-major).
outputFormat	Output image format (channel-major or image-major).
biased	Whether or not a bias has already been applied to the output
src	Input image stored in the specified format
srcX	Width of the image
srcY	Height of the image
channels	Number of channels in the image
filters	Filters, assumed to be stored in channel major format
numFilters	Number of filters
filterX	Width of the filters
filterY	Height of the filters
padX	Amount of padding added to the left and right of the image
padY	Amount of padding added to the top and bottom of the image
strideX	Filter stride in the X dimension
strideY	Filter stride in the Y dimension
output	Output image stored in the specified format
outputX	Width of an individual output
outputy	Height of an individual output

CHAPTER

TWENTY

QML UTILITY FUNCTIONS

QML provides a few utility functions that can be useful in diagnosing problems on a given machine. These utility functions provide some insight into the runtime decisions the library is making as well as general information about the library.

QMLVersionInfo

QMLVersionInfo.

Description

Provides general information about QML.

Extension Interface

```
void QMLVersionInfo(qml_info *info);

struct qml_info
{
    const char *productName;
    const char *vendor;
    qml_int major;
    qml_int minor;
    qml_int mark;
    QML_INFO_TYPE type;
    const char *cpuOptimization;
    const char *cpuArch;
    const char *buildDate;
    qml_int lp64;
};
```

Arguments

info	Pointer to the `qml_info` struct which will be populated with information about QML

QMLVersionString

QMLVersionString.

Description

Provides the version string for QML.

Extension Interface

```
void QMLVersionString(const char **string);
```

Arguments

string	This argument will be modified to point to the QML version string

QMLGetNumThreads

QMLGetNumThreads.

Description

Get the number of threads that are being used internally to parallelize work in QML.

The number of threads used will normally be set by QML to the number of processor cores reported present on the device. The number of threads used can be affected by OpenMP environment variables and by QML environment variables set when the library is loaded (see *Environment Variables*). The number of threads may also be changed by calls to QMLSetNumThreads.

Extension Interface

```
qml_int QMLGetNumThreads(void);
```

Arguments

Result	The number of threads being used

QMLSetNumThreads

QMLSetNumThreads.

Description

Set the number of threads that will be used internally to parallelize work in QML.

This function does not normally need to be called. QML automatically detects the number of cores available in the device when loaded and adjusts the thread pool to an optimal size. A typical use for this function is to pass *1* as the argument to get sequential versions of algorithms that will run in a single thread. This can be useful for programs that already use multiple threads to divide up work.

Extension Interface

```
void QMLSetNumThreads(qml_int numThreads);
```

Arguments

numThreads	The number of threads requested

QML_IS_SUPPORTED

QML_IS_SUPPORTED.

Description

Returns one if QML is supported on this platform, zero otherwise.

Extension Interface

```
qml_int QML_IS_SUPPORTED();
```

Arguments

Result	Returns one if QML is supported on this platform, zero otherwise.