MPLAPACK version 1.0.0 user manual

NAKATA Maho[1]

[1]RIKEN Cluster for Pioneering Research, 2-1 Hirosawa, Wako-City, Saitama 351-0198, JAPAN

Abstract

The MPLAPACK (formerly MPACK) is a multiple-precision version of LAPACK (`https://www.netlib.org/lapack/`). MPLAPACK version 1.0.0 is based on LAPACK version 3.9.1 and translated from Fortran 90 to C++ using FABLE, a Fortran to C++ source-to-source conversion tool (`https://github.com/cctbx/cctbx_project/tree/master/fable/`). MPLAPACK version 1.0.0 provides the real and complex version of MPBLAS, and the real version of MPLAPACK supports all LAPACK features: solvers for systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems, and related matrix factorizations except for rectangular complete packed matrix form and mixed-precision routines. Besides, MPLAPACK supports a part of the complex routines of LAPACK, such as diagonalization of Hermite matrices, Cholesky factorization, and matrix inversion. The MPLAPACK defines an API for numerical linear algebra, similar to LAPACK. It is easy to port legacy C/C++ numerical codes using MPLAPACK. MPLAPACK supports binary64, binary128, FP80 (extended double), MPFR, GMP, and QD libraries (double-double and quad-double). Users can choose MPFR or GMP for arbitrary accurate calculations, double-double or quad-double for fast 32 or 64 decimal calculations. We can consider the binary64 version as the C++ version of LAPACK. MPLAPACK is available at GitHub (`https://github.com/nakatamaho/mplapack/`) under 2-clause BSD license.

1 Introduction

Numerical linear algebra aims to solve mathematical problems, such as simultaneous linear equations, eigenvalue problems, and least-squares methods, using arithmetic with finite-precision on a computer [1]. We can formulate many problems as numerical linear algebra in various fields, such as natural science, social science, and engineering. Therefore, its importance is magnificent.

The standards for the numerical linear algebra package are BLAS [2] and LAPACK [3]. The BLAS library defines how computers should perform for vector, matrix-vector, and matrix-matrix operations in FORTRAN77. Almost all other vector and matrix arithmetic libraries are compatible with it or have very similar interfaces. Furthermore, LAPACK solves linear problems such as solving linear equations, singular value problems, eigenvalue problems, and least-square fitting problems using BLAS as a building block.

The main focus in numerical linear algebra has been on the speed and the size of solving the problem at approximately 8 or 16 decimal digits (binary32 or 64) using highly optimized BLAS and LAPACK [4, 5, 6, 7, 8, 9]. Using higher precision numbers than binary64 is not common.

However, there are some problems in numerical linear algebra that require higher precision operations. In particular, when we solve ill-conditioned problems, large-scale simulations, and compute

inversions of matrices, we usually need functions of higher precision numbers [10, 11, 12].

Solving positive semidefinite programming (SDP) is yet another example that requires multi-precision computation. When we solve this problem using binary64 usually yields values up to eight decimal digits. The accumulation of numerical error occurs because the matrices' condition number at the optimal solution usually becomes infinite. As a result, Cholesky factorization fails near the optimal solution; the approximate solution diverges toward the optimal solution using the primal and dual interior point method [13]. Therefore, we require multiple precision calculations if we need more than eight decimal digits for optimal solutions. For this reason, we have developed SDPA-GMP [14, 15, 16], the GNU MP version of semidefinite programming solver based on SDPA [16], which is one of the fastest SDP solvers.

MPLAPACK was developed as a drop-in replacement for BLAS and LAPACK in SDPA since SDPA performs Cholesky decomposition and solves symmetric eigenvalue problems via 50 BLAS and LAPACK routines [17].

The features of MPLAPACK are:

- Provides Application Programming Interface (API) numerical algebra, similar to LAPACK and BLAS. Like BLAS and LAPACK, we can implement an optimized version of MPBLAS and MPLAPACK. In addition, we provide a simple OpenMP version of some MPBLAS routines as proof of concept.
- Completely rewrote LAPACK and BLAS in C++ using FABLE and f2c.
- C style programming. We do not introduce new matrix and vector classes.
- Supports seven floating-point formats by precision independent programming; binary64 (16 decimal digits), binary128 (32 decimal digits), FP80 (extended double; 19 decimal digits), double-double (32 decimal digits), quad-double (64 decimal digits), GMP, and MPFR (arbitrary precision, and the default is 153 decimal digits).
- MPLAPACK 1.0.0 is based on LAPACK 3.9.1.
- Version 1.0.0 supports all real and complex versions of BLAS and real version LAPACK functions; simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems and related matrix factorization except for mixed-precision version, and full packed matrix form version and complex version will be supported in version 2.0.
- Reliability: We extended the original test programs to handle multiple-precision numbers, and most of the MPLAPCK routines have passed the test.
- Runs on Linux/Windows/Mac.
- Released at `https://github.com/nakatamaho/mplapack/` under 2-BSD clause license.

The rest of the sections are organized as follows: Section 2 describes supported CPUs, OSes, and compilers. Section 3 describes how to install. Section 4 describes the supported floating-point format. Section 5 illustrates LAPACK and BLAS routine naming conventions and available MPBLAS and MPLAPACK routines; section 6 describes how to use MPBLAS and MPLAPACK. The examples are based on the assumption that users use Docker to avoid over-complicating the notation with differences

in environments. Section 7 describes how we tested MPBLAS and MPLAPACK. Section 8 describes how we rewrote Fortran90 codes to C++ and extended them to multiple precision versions. Next, section 9 describes benchmarking MPBLAS routines, and section 10 describes the history, Section 11 describes related works. Finally, section 12 describes future plans for MPLAPACK.

2 Supported CPUs, OSes, and compilers

Only 64 bit CPU is supported. Following OSes are supported:

- CentOS 7 (amd64, aarch64)
- CentOS 8 (amd64, aarch64)
- Ubuntu 20.04 (amd64, aarch64)
- Ubuntu 18.04 (amd64)
- Windows 10 (amd64)
- MacOS (Intel)

We support the following compilers:

- GCC (GNU Compiler Collection) 9 and 10
- Intel OneAPI.

Note that we use GCC by MacPorts on macOS, which is NOT the default compiler. We use the mingw64 and Wine64 environments on Linux to build and test the Windows version. On different configurations, MPLAPACK may build and work without problems. We welcome patches from the community.

3 Installation

3.1 Using Docker (recommended for Linux environment)

The easiest way to install MPLAPACK is to build inside Docker [18] and use it inside Docker. For example, the following command will build inside docker on Ubuntu amd64 (also known as x86_64) or aarch64 (also known as arm64); the following command will build inside docker.

```
$ git clone -b v1.0.0 https://github.com/nakatamaho/mplapack/
$ cd mplapack
$ /usr/bin/time docker build -t mplapack:ubuntu2004 -f Dockerfile_ubuntu20.04 . \
   2>&1 | tee log.ubuntu2004
```

Table 1: The name of the Dockerfile and the corresponding OS and CPU

Docker filename	CPU	OS	Compiler
Dockerfile_CentOS7	amd64	CentOS 7	GCC
Dockerfile_CentOS7_AArch64	aarch64 only	CentOS 7	GCC
Dockerfile_CentOS8	all CPUs	CentOS 8	GCC
Dockerfile_ubuntu18.04	amd64 only	Ubuntu 18.04	GCC
Dockerfile_ubuntu20.04	all CPUs	Ubuntu 20.04	GCC
Dockerfile_ubuntu20.04_intel	amd64 only	Ubuntu 20.04	Intel oneAPI
Dockerfile_ubuntu20.04_mingw64	amd64 only	Windows10	GCC

In Table 1, we list corresponding Dockerfiles for CPU and OS.

When the build is finished, all the files will be installed under /home/docker/MPLAPACK.

One can also install MPLAPACK in the desired directry. First, make a tar archive of installed directory,

```
$ docker run -it mplapack:ubuntu2004
$ cd /home/docker/MPLAPACK
$ tar cvfz ../mplapack.tar.gz .
$ cd .. ; scp mplapack.tar.gz YourhostIP:
$ exit
```

Then, on the host OS, copy like following:

```
$ sudo tar xvfz mplapack.tar.gz -C /usr/local/mplapack
```

In Table 1, we list Dockerfiles and corresponding CPU, OS, and compiler.

3.2 Compiling from the source

We list prerequisites for compiling from the source in Table 2; users can satisfy these prerequisites using MacPorts on macOS. Homebrew may be used as an alternative to satisfy these prerequisites.

```
$ sudo port install gcc9 coreutils git ccache
$ git clone -b v1.0.0 https://github.com/nakatamaho/mplapack.git --depth 1
$ cd mplapack
$ pushd mplapack/debug ; bash gen.Makefile.am.sh ; popd
$ autoreconf --force --install ; aclocal ; autoconf ; automake
$ autoreconf --force --install
$ CXX="g++-mp-9" ; export CXX
$ CC="gcc-mp-9" ; export CC
$ FC="gfortran-mp-9"; export FC
$ F77="gfortran-mp-9"; export F77
$ ./configure --prefix=/Users/maho/MPLAPACK --enable-gmp=yes --enable-mpfr=yes \
--enable-_Float128=yes --enable-qd=yes --enable-dd=yes --enable-double=yes \
```

Table 2: Prerequisites for compiling from source

Package	Version
gcc	9 or later
gmake	4.3 or later
git	2.33.0 or later
gsed	4.8 or later
autotools	2.71
automake	1.16
GNU sed	4.1 or later

```
--enable-_Float64x=yes --enable-test=yes
...
$ make -j6
...
$ make install
```

4 Supported Floating point formats

We support binary64, FP80 (extended double), binary128, double-double, quad-double (QD library), GMP, and MPFR in MPLAPACK.

Binary64 [4] is the so-called double precision: the number of significant digits in decimal is about 16, and CPUs perform arithmetic processing in hardware. As a result, the CPU can perform binary64 operations very fast.

FP80 is an extended double precision. IEEE 754-1985 species the minimum requirements for an extended double format [19], and 80-bit format meets the minimum requirement. FP80 has approximately 19 digits accuracy, respectively and FP80 has a much wider exponent than binary64. Recently, ISO defined C real floating type [20] for FP80 as `_Float64x` but not yet a standard of C++. Moreover, GCC has already defined type for FP80 as `__float80`. In any case, we always use `_Float64x` MPLAPACK programs to keep the consistency among different environments. When building the MPLAPACK library, the configure script investigates whether `_Float64x` is available or not. If not, we typedef the appropriate type as `_Float64x`. Only Intel and AMD CPUs support FP80 format. These CPUs implement FP80 as hardware. However, they do not support SIMD-type acceleration like SSE nor AVX; thus, FP80 arithmetic is much slower than binary64.

Binary128, and sometimes called quadruple precision, is defined in IEEE754-2008 [4] and this format has approximately 33 decimal significant digits. Usually binary128 arithmetic is done by software, therefore, operations are very slow. As far as we know, only IBM z processors have been the only commercial platform supporting quadruple-precision [21]. Some processors like aarch64, Sparc, RISCV64, MIPS64 define instructions for quadruple-precision arithmetic. These processors emulate binary128 instructions by software. ISO defined C real floating types [20] as `_Float128` but not yet a standard of C++. GCC has already been providing a type for binary128 as `__float128`. We always use `_Float128` as a type for binary128 numbers in MPLAPACK. This type may be the same as `long double` or `__float128` depends on the environment, we typedef appropriate type to `_Float128`.

We use the QD library [22] to support double-double and quad-double precision. The double-double casts two binary64 numbers as a one number and has approximately 32 decimal significant digits, and the quad-double casts four binary64 numbers as a one number and has approximately 64 decimal significant digits. Using Kunth and Dekker's algorithm [23, 24], which can evaluate the addition and multiplication of two binary64 numbers exactly. Then, we can define the addition and multiplication of double-double numbers. Pros for using these format is speed. Since all arithmetic can be done by binary64 and accelerated by hardware, calculation speed is are approximately ten times faster than software implemented binary128. Cons of using these formats are programs written for expecting IEEE754 feature might not work with these precisions. Note that OSes and compilers for IBM Power CPU and Power Macs, "long double" has been equivalent to the double-double.

GMP [25] is a C library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. We can perform calculations to any accuracy, as long as the machine's resources allow. We support complex numbers by preparing `mpc_class`.

MPFR [26] is a C library for multiple-precision floating-point computations with correct rounding based on GMP. We support complex numbers using MPC, which is a library for multiple-precision complex arithmetic with correct rounding [27] via `mpcomplex` class. Both libraries provide almost the same functionalities, but MPFR and MPC further support trigonometric functions necessary for cosine-sine decomposition, elementary and special functions. Therefore, we will drop GMP support in the future.

`long double` is no longer supported in MPLAPACK since v1.0.0 since the situation regarding `long double` is very different and confusing in different environments. E.g., on Intel CPUs, `long double` may be equivalent to `_Float64x`; however, it also depends on OSes. On Windows, `long double` is equivalent to `double`. Similar confusion happens in the AArch64 environment. ARM defines `long double` should be binary64, but on the OS side, Apple overwrites `long double` should be double. On IBM PowerPC or Power Macs, `long double` have been double-double.

5 LAPACK and BLAS Routine Naming Conventions and available routines

BLAS and LAPACK prefix routine names with "s" and "d" for (real) single and double precision real numbers and "c" and "z" for single and double precision complex numbers. FORTRAN77 and Fortran90 are not case-sensitive in function and subroutine names, while C++ is case-sensitive in function names.

The prefix for real and complex routine names in MPLAPACK is an uppercase "R" for real numbers and "C" for complex numbers. All other letters in the routines are lowercase. Also, while we do not distinguish function names by floating-point class (e.g. `mpf_class, _Float128`), we use the same function name for different floating-point classes to take advantage of function overloading (e.g., no matter which floating-point class is adopted, program calls `Raxpy` appropriately). The floating-point class is added after the routine name for routines that take no arguments or only integers. Otherwise, we add the prefix "M" or insert "M" after "i."

For example,

- `daxpy, zaxpy` → `Raxpy, Caxpy`
- `dgemm, zgemm` → `Rgemm, Cgemm`

- dsterf, dsyev → Rsterf, Rsyev
- dzabs1, dzasum → RCabs1, RCasum
- lsame → Mlsame_mpfr, Mlsame_gmp, Mlsame__Float128 ··· etc.
- dlamch → Rlamch_mpfr, Rlamch_gmp, Rlamch__Float128 ··· etc.
- ilaenv → iMlaenv_mpfr, iMlaenv_gmp, iMlaenv__Float128 ··· etc.

In table 3, we show all supported MPBLAS routines. The prototype definitions of these routines can be found in the following headers.

- /home/docker/MPLAPACK/include/mplapack/mpblas__Float128.h
- /home/docker/MPLAPACK/include/mplapack/mpblas__Float64x.h
- /home/docker/MPLAPACK/include/mplapack/mpblas_dd.h
- /home/docker/MPLAPACK/include/mplapack/mpblas_double.h
- /home/docker/MPLAPACK/include/mplapack/mpblas_gmp.h
- /home/docker/MPLAPACK/include/mplapack/mpblas_mpfr.h
- /home/docker/MPLAPACK/include/mplapack/mpblas_qd.h

A simple OpenMP version of MPBLAS is available. In table 4, we show all OpenMP accelerated MPBLAS routines.

In table 5, we show all supported MPLAPACK driver routines. In table 6, we show all supported MPLAPACK Real computational routines. In table 7, we show all supported MPLAPACK complex computational routines.

We do not list here, but there are many MPLPACK auxiliary routines as well.

We usually use MPLAPACK driver routines and MPBLAS routines directly. MPLAPACK computational routines and auxiliary routines may be implicitly used by the driver routines.

The prototype definitions of the MLAPACK routines can be found in the following headers.

- /home/docker/MPLAPACK/include/mplapack/mplapack__Float128.h
- /home/docker/MPLAPACK/include/mplapack/mplapack__Float64x.h
- /home/docker/MPLAPACK/include/mplapack/mplapack_dd.h
- /home/docker/MPLAPACK/include/mplapack/mplapack_double.h
- /home/docker/MPLAPACK/include/mplapack/mplapack_gmp.h
- /home/docker/MPLAPACK/include/mplapack/mplapack_mpfr.h
- /home/docker/MPLAPACK/include/mplapack/mplapack_qd.h

Table 3: Available MPBLAS routines

Crotg	Cscal	Rrotg	Rrot	Rrotm	CRrot	Cswap	Rswap	CRscal	Rscal
Ccopy	Rcopy	Caxpy	Raxpy	Rdot	Cdotc	Cdotu	RCnrm2	Rnrm2	Rasum
iCasum	iRamax	RCabs1	Mlsame	Mxerbla					
Cgemv	Rgemv	Cgbmv	Rgbmv	Chemv	Chbmv	Chpmv	Rsymv	Rsbmv	Ctrmv
Cgemv	Rgemv	Cgbmv	Rgemv	Chemv	Chbmv	Chpmv	Rsymv	Rsbmv	Rspmv
Ctrmv	Rtrmv	Ctbmv	Ctpmv	Rtpmv	Ctrsv	Rtrsv	Ctbsv	Rtbsv	Ctpsv
Rger	Cgeru	Cgerc	Cher	Chpr	Cher2	Chpr2	Rsyr	Rspr	Rsyr2
Rspr2									
Cgemm	Rgemm	Csymm	Rsymm	Chemm	Csyrk	Rsyrk	Cherk	Csyr2k	Rsyr2k
Cher2k	Ctrmm	Rtrmm	Ctrsm	Rtrsm					

Table 4: Available OpenMP version of MPBLAS routines

Raxpy	Rcopy	Rdot	Rgemm

Table 5: Available MPLAPACK Driver routines (1.0.0)

Rgesv	Rgesvx	Rgbsv	Rgbsvx	Rgtsv	Rgtsvx	Rposv	Rposvx	Rppsv	Rppsvx
Rpbsv	Rpvsvx	Rptsv	Rptsvx	Rsysv	Rsysvx	Rpspv	Rspsvx	Rgels	Rgelsy
Rgelss	Rgelsd	Rgglse	Rggglm	Rsyev	Rsyevd	Rsvevd	Rsyevr	Rsepev	Rspevd
Rspevx	Rsbev	Rsbevd	Rsbevx	Rstev	Rstevd	Rstevx	Rstevr	Rgees	Rgeesx
Rgeev	Rgeevx	Rgesvd	Rgesdd	Rsygv	Rsygvd	Rsygvx	Rspgv	Rspgdv	Rspgvx
Rsbgv	Rsbgv	Rsbgvx	Rgges	Rggesx	Rggev	Rggevx	Rggsvd		

Table 6: Available MPLAPACK Computational Real routines (1.0.0)

Rgetrf	Rgetrs	Rgecon	Rgerfs	Rgetri	Rgeequ	Rgbtrf	Rgbtrs	Rgbcon	Rgbrfs
Rgbequ	Rgttrf	Rgttrs	Rgtcon	Rgtrfs	Rpotrf	Rpotrs	Rpocon	Rporfs	Rpotri
Rpoequ	Rpptrf	Rpptrs	Rppcon	Rpprfs	Rpptri	Rppequ	Rpbtrf	Rpbtrs	Rpbcon
Rpbrfs	Rpbequ	Rpttrf	Rpttrs	Rptcon	Rptrfs	Rsytrf	Rsytrs	Rsycon	Rsyrfs
Rsytri	Rsptrf	Rsptrs	Rspcon	Rsprfs	Rsptri	Rtrtrs	Rtrcon	Rtrrfs	Rtrtri
Rtptrs	Rtpcon	Rtprfs	Rtptri	Rtbtrs	Rtbcon	Rtbrfs			
Rgeqp3	Rgeqrf	Rorgqr	Rormqr	Rgelqf	Rorglq	Rormlq	Rgeqlf	Rorgql	Rormql
Rgerqf	Rorgrq	Rormrq	Rtzrzf	Rormrz					
Rsytrd	Rsptrd	Rsbtrd	Rorgtr	Rormtr	Ropgtr	Ropmtr	Rsteqr	Rsterf	Rstedc
Rstegr	Rstebz	Rstein	Rpteqr						
Rgehrd	Rgebal	Rgebak	Rorghr	Rormhr	Rhseqr	Rhsein	Rtrevc	Rtrexc	Rtrsyl
Rtrsna	Rtrsen								
Rgebrd	Rgbbrd	Rorgbr	Rormbr	Rbdsqr	Rbdsdc				
Rsygst	Rspgst	Rpbstf	Rsbgst						
Rgghrd	Rggbal	Rggbak	Rhgeqz	Rtgevc	Rtgexc	Rtgsyl	Rtgsna	Rtgsen	
Rggsvp	Rtgsja								

Table 7: Available MPLAPACK Computational Complex routines (1.0.0)

Cgelq2	Cgeqr2	Cgeqrf	Cgesv	Cgetf2	Cgetrf	Cgetri	Cgetrs	Cheev	Chetd2
Chetrd	Clacgv	Clacrm	Clacrt	Cladiv	Claesy	Claev2	Clahef	Clanhe	Clanht
Clansy	Clarfb	Clarfg	Clarf	Clarft	Clartg	Clascl	Claset	Clasr	Classq
Claswp	Clasyf	Clatrd	Cpotf2	Crot	Cspmv	Cspr	Csteqr	Csymv	Csyr
Ctrti2	Ctrtri	Ctrtrs	Cung2l	Cung2r	Cungql	Cungqr	Cungtr	Cunmqr	

Table 8: The floating-point formats used in MPLAPACK, their type names, and accuracy in decimal digits.

Library or format	type name	accuracy in decimal digits
GMP	`mpf_class, mpc_class`	154 (default) and arbitrary
MPFR	`mpreal, mpcomplex`	154 (default) and arbitrary
double-double	`dd_real, dd_complex`	32
quad-double	`qd_real, qd_complex`	64
binary64	`double, std::complex<double>`	16
extended double	`_Float64x, std::complex<_Float64x>`	19
binary128	`_Float128, std::complex<_Float128>`	33
integer	mplapackint	(32bit on Win, 64bit on Linux and macOS)

6 How to use MPBLAS and MPLPACK

6.1 Types

We use seven kinds of floating-point formats. Multiple precision types are listed in the table 8. For GMP, we use built-in `mpf_class` for real type. For complex type, we developed `mpc_class`. For MPFR, we use a modified version of mpfrc++ (the final LGPL version) to treat like double or float type. For complex type, we developed `mpcomplex.h` using MPFR and MPC. For double-double and quad-double, we use `dd_real` and `qd_real`, respectively. For complex type, we developed `dd_complex` and `qd_complex`, respectively.

For extended double we use `_Float64x` for all Intel and AMD environment. For binary128, we use _Float128 for all OSes and CPUs. For complex types for `double`, `_Float64x` and `_Float128`, we use standard complex implementation of C++. Using `__float80`, `__float128`, or `long double` is strongly discouraged as they are not tested and not portable.

Also, we define `mplapackint` as a 64 bit signed integer to MPLAPACK that can access all the memory spaces regardless of which data type models the environment employs (LLP64 or LP64). On Windows, `mplapackint` is still a 32 bit signed integer.

6.2 MPBLAS

The API of MPBLAS is very similar to the original BLAS and CBLAS. However, we always use a one-dimensional array as a column-major type matrix in MPBLAS, and unlike CBLAS, we cannot specify row- or column-major order.

Following is the prototype definition of the multiple-precision version of matrix-matrix multiplication (`Rgemm`) by MPFR.

```
void Rgemm(const char *transa, const char *transb, mplapackint const m,
mplapackint const n, mplapackint const k, mpreal const alpha, mpreal *a,
mplapackint const lda, mpreal *b, mplapackint const ldb, mpreal const beta,
mpreal*c, mplapackint const ldc);
```

and the following is the definition part of the original DGEMM

```
SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
*     .. Scalar Arguments ..
      DOUBLE PRECISION ALPHA,BETA
      INTEGER K,LDA,LDB,LDC,M,N
      CHARACTER TRANSA,TRANSB
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION A(LDA,*),B(LDB,*),C(LDC,*)
```

There is correspondence between variables in the C++ prototype of Rgemm and variables in the header of `DGEMM`.

- `CHARACTER` → `const char *`
- `INTEGER` → `mplapackint`
- `DOUBLE PRECISION A(LDA, *)` → `mpreal *a`

We can see such correspondences for other MPBLAS and MPLAPACK routines. Let us see how we can multiply matrices using MPFR. The list is the following.

```
#include <mpblas_mpfr.h>

//Matlab/Octave format
void printmat(int N, int M, mpreal * A, int LDA)
{
    mpreal mtmp;
    printf("[ ");
    for (int i = 0; i < N; i++) {
        printf("[ ");
        for (int j = 0; j < M; j++) {
            mtmp = A[i + j * LDA];
            mpfr_printf("%5.2Re", mpfr_ptr(mtmp));
            if (j < M - 1)
                printf(", ");
        }
        if (i < N - 1)
            printf("]; ");
        else
            printf("] ");
    }
    printf("]");
}

int main()
{
    mplapackint n = 3;
//initialization of MPFR
```

```
28      int default_prec = 256;
29      mpfr_set_default_prec(default_prec);
30
31      mpreal *A = new mpreal[n * n];
32      mpreal *B = new mpreal[n * n];
33      mpreal *C = new mpreal[n * n];
34      mpreal alpha, beta;
35
36  //setting A matrix
37      A[0 + 0 * n] = 1;    A[0 + 1 * n] = 8;    A[0 + 2 * n] = 3;
38      A[1 + 0 * n] = 0;    A[1 + 1 * n] = 10;   A[1 + 2 * n] = 8;
39      A[2 + 0 * n] = 9;    A[2 + 1 * n] = -5;   A[2 + 2 * n] = -1;
40
41      B[0 + 0 * n] = 9;    B[0 + 1 * n] = 8;    B[0 + 2 * n] = 3;
42      B[1 + 0 * n] = 3;    B[1 + 1 * n] = -11;  B[1 + 2 * n] = 0;
43      B[2 + 0 * n] = -8;   B[2 + 1 * n] = 6;    B[2 + 2 * n] = 1;
44
45      C[0 + 0 * n] = 3;    C[0 + 1 * n] = 3;    C[0 + 2 * n] = 0;
46      C[1 + 0 * n] = 8;    C[1 + 1 * n] = 4;    C[1 + 2 * n] = 8;
47      C[2 + 0 * n] = 6;    C[2 + 1 * n] = 1;    C[2 + 2 * n] = -2;
48
49      printf("# Rgemm demo...\n");
50
51      printf("A ="); printmat(n, n, A, n); printf("\n");
52      printf("B ="); printmat(n, n, B, n); printf("\n");
53      printf("C ="); printmat(n, n, C, n); printf("\n");
54      alpha = 3.0;
55      beta = -2.0;
56      Rgemm("n", "n", n, n, n, alpha, A, n, B, n, beta, C, n);
57
58      mpfr_printf("alpha = %5.3Re\n", mpfr_ptr(alpha));
59      mpfr_printf("beta  = %5.3Re\n", mpfr_ptr(beta));
60      printf("ans ="); printmat(n, n, C, n); printf("\n");
61      printf("#please check by Matlab or Octave following and ans above\n");
62      printf("alpha * A * B + beta * C =\n");
63      delete[]C;
64      delete[]B;
65      delete[]A;
66  }
```

First, we must include `mpblas_mpfr.h` to use MPFR. `printmat` function prints matrix in Matlab format (lines 4 to 22). We set 256 bit in fraction for mpfrc++ in lines 28 to 29; MPFR Real accuracy setted to 77 decimal digits ($77.06 = \log_{10} 2^{256}$). We allocate the matrix as a one-dimensional array (lines 31 to 33). Then we set the matrix A, B, and C. We always input matrix by row-major format to an array (lines36 to 47). Matrix-matrix multiplication `Rgemm` is called in line 58. One can input the list and save as `Rgemm_mpfr.cpp` or you can find `/home/docker/mplapack/examples/mpblas/` directory. Then, one can compile by one own like following in Docker environment as follows:

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_mpfr.cpp -L/home/docker/MPLAPACK/lib -lmpblas_mpfr -lgmp -lmpfr -lmpc
```

Then, you can run as follows:

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
# Rgemm demo...
A =[ [ 1.00e+00, 8.00e+00, 3.00e+00]; [ 0.00e+00, 1.00e+01, 8.00e+00]; [ 9.00e+00, -5.00e+00, -1.00e+00] ]
B =[ [ 9.00e+00, 8.00e+00, 3.00e+00]; [ 3.00e+00, -1.10e+01, 0.00e+00]; [ -8.00e+00, 6.00e+00, 1.00e+00] ]
C =[ [ 3.00e+00, 3.00e+00, 0.00e+00]; [ 8.00e+00, 4.00e+00, 8.00e+00]; [ 6.00e+00, 1.00e+00, -2.00e+00] ]
alpha = 3.000e+00
beta  = -2.000e+00
ans =[ [ 2.10e+01, -1.92e+02, 1.80e+01]; [ -1.18e+02, -1.94e+02, 8.00e+00]; [ 2.10e+02, 3.61e+02, 8.20e+01] ]
#please check by Matlab or Octave following and ans above
alpha * A * B + beta * C
```

One can check the result by comparing the result of the octave.

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out | octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
A =

    1    8    3
    0   10    8
    9   -5   -1

B =

    9    8    3
    3  -11    0
   -8    6    1

C =

   3   3   0
   8   4   8
   6   1  -2

alpha =  3
beta = -2
ans =

    21  -192    18
  -118  -194     8
   210   361    82

ans =

    21  -192    18
  -118  -194     8
   210   361    82
```

In this case, we see the result below two "`ans =`" are the same. The `Rgemm` result is correct up to 16 decimal digits.

Let us see how we can multiply matrices using `_Float128` (binary128). The list is the following.

```
1 #include <mpblas__Float128.h>
2 #include <stdio.h>
```

```
#define BUFLEN 1024

void printnum(_Float128 rtmp)
{
    int width = 42;
    char buf[BUFLEN];
#if defined ___MPLAPACK_WANT_LIBQUADMATH___
    int n = quadmath_snprintf (buf, sizeof buf, "%+-#*.35Qe", width, rtmp);
#elif defined ___MPLAPACK_LONGDOUBLE_IS_BINARY128___
    snprintf (buf, sizeof buf, "%.35Le", rtmp);
#else
    strfromf128(buf, sizeof(buf), "%.35e", rtmp);
#endif
    printf ("%s", buf);
    return;
}

//Matlab/Octave format
void printmat(int N, int M, _Float128 * A, int LDA)
{
    _Float128 mtmp;

    printf("[ ");
    for (int i = 0; i < N; i++) {
        printf("[ ");
        for (int j = 0; j < M; j++) {
            mtmp = A[i + j * LDA];
            printnum(mtmp);
            if (j < M - 1)
                printf(", ");
        }
        if (i < N - 1)
            printf("]; ");
        else
            printf("] ");
    }
    printf("]");
}

int main()
{
    mplapackint n = 3;

    _Float128 *A = new _Float128[n * n];
    _Float128 *B = new _Float128[n * n];
    _Float128 *C = new _Float128[n * n];
    _Float128 alpha, beta;

//setting A matrix
```

```
    A[0 + 0 * n] = 1;    A[0 + 1 * n] = 8;    A[0 + 2 * n] = 3;
    A[1 + 0 * n] = 2.5;  A[1 + 1 * n] = 10;   A[1 + 2 * n] = 8;
    A[2 + 0 * n] = 9;    A[2 + 1 * n] = -5;   A[2 + 2 * n] = -1;

    B[0 + 0 * n] = 9;    B[0 + 1 * n] = 8;    B[0 + 2 * n] = 3;
    B[1 + 0 * n] = 3;    B[1 + 1 * n] = -11;  B[1 + 2 * n] = 4.8;
    B[2 + 0 * n] = -8;   B[2 + 1 * n] = 6;    B[2 + 2 * n] = 1;

    C[0 + 0 * n] = 3;    C[0 + 1 * n] = 3;    C[0 + 2 * n] = 1.2;
    C[1 + 0 * n] = 8;    C[1 + 1 * n] = 4;    C[1 + 2 * n] = 8;
    C[2 + 0 * n] = 6;    C[2 + 1 * n] = 1;    C[2 + 2 * n] = -2;

    printf("# Rgemm demo...\n");

    printf("A ="); printmat(n, n, A, n); printf("\n");
    printf("B ="); printmat(n, n, B, n); printf("\n");
    printf("C ="); printmat(n, n, C, n); printf("\n");
    alpha = 3.0;
    beta = -2.0;
    Rgemm("n", "n", n, n, n, alpha, A, n, B, n, beta, C, n);

    printf("alpha = "); printnum(alpha); printf("\n");
    printf("beta  = "); printnum(beta);  printf("\n");
    printf("ans ="); printmat(n, n, C, n); printf("\n");
    printf("#please check by Matlab or Octave following and ans above\n");
    printf("alpha * A * B + beta * C \n");
    delete[]C;
    delete[]B;
    delete[]A;
}
```

We do not show the output since the output is almost the same as the MPFR version.

The `_Float128` version of progarm list is almost similar to the MPFR version, too. However, `printnum` part is bit complicated. Because there are at least three kinds of binary128 support depend on CPUs and OSes: (i) GCC only supports `__float128` using `libquadmath` (Windows, macOS), (ii) GCC supports `_Float128` directly and there are libc support as well (Linux amd64), (iii) `long double` is already binary128 and no special support is necessary (AArch64). For (i), we internally define `___MPLAPACK_WANT_LIBQUADMATH___`, for (ii) we define `___MPLAPACK__FLOAT128_ONLY___`, and for (iii), we define `___MPLAPACK_LONGDOUBLE_IS_BINARY128___`. Users can use these internal defines to write a portable program.

As a summary, we show how we compile and run binary64, FP80, binary128, double-double, quad-double, GMP and MPFR versions of `Rgemm` demo programs in `/home/docker/mplapack/examples/mpblas` as follows:

- binary64 (double) version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_double.cpp -L/home/docker/MPLAPACK/lib -lmpblas_double
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- FP80 (extended double) version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm__Float64x.cpp -L/home/docker/MPLAPACK/lib -lmpblas__Float64x
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- binary128 version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm__Float128.cpp -L/home/docker/MPLAPACK/lib -lmpblas__Float128
 LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

  or (on macOS, mingw64 and CentOS7 amd64)

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm__Float128.cpp -L/home/docker/MPLAPACK/lib -lmpblas__Float128 -lquadmath
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- double-double version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_dd.cpp -L/home/docker/MPLAPACK/lib -lmpblas_dd -lqd
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- quad-double version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_qd.cpp -L/home/docker/MPLAPACK/lib -lmpblas_qd -lqd
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- GMP version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_gmp.cpp -L/home/docker/MPLAPACK/lib -lmpblas_gmp -lgmpxx -lgmp
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- MPFR version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_mpfr.cpp -L/home/docker/MPLAPACK/lib -lmpblas_mpfr -lmpfr -lmpc -lgmp
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

6.3 OpenMP accelerated MPBLAS

We have provided reference implementation, and a simple OpenMP accelerated version for MPBLAS, which are shown in Table 4. Even though the number of optimized routines is small; however, acceleration of `Rgemm` is significant as it impacts the performance. To change linking against optimized version, we must add "`_opt`" for `MPBLAS` library as follows:

- `-lmpblas_mpfr` → `-lmpblas_mpfr_opt`
- `-lmpblas_gmp` → `-lmpblas_gmp_opt`
- `-lmpblas_double` → `-lmpblas_double_opt`
- `-lmpblas__Float128` → `-lmpblas__Float128_opt`
- `-lmpblas_dd` → `-lmpblas_dd_opt`
- `-lmpblas_qd` → `-lmpblas_qd_opt`
- `-lmpblas__Float128` → `-lmpblas__Float128_opt`
- `-lmpblas__Float64x` → `-lmpblas__Float64x_opt`.

In the previous examples, we compile the demo program as follows for the `MPFR` version.

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_mpfr.cpp -L/home/docker/MPLAPACK/lib -lmpblas_mpfr -lmpfr -lmpc -lgmp
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

and to link against optimized version, we complie the program as follows:

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgemm_mpfr.cpp -L/home/docker/MPLAPACK/lib -lmpblas_mpfr_opt -lmpfr -lmpc -lgmp
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

Performance comparison is presented in Section 9.

6.4 MPLAPACK

The API of MPLAPACK is very similar to the original LAPACK. We always use a one-dimensional array as a column-major type matrix in MPLAPACK. We do not have an interface like LAPACKE at the moment, and we cannot specify raw- or column-major order in MPLAPACK.

Let us show how we diagonalize a real symmetric matrix A

$$\begin{bmatrix} 5 & 4 & 1 & 1 \\ 4 & 5 & 1 & 1 \\ 1 & 1 & 4 & 2 \\ 1 & 1 & 2 & 4 \end{bmatrix}$$

using GMP. Eigenvalues are $\lambda_1 = 10, \lambda_2 = 5, \lambda_3 = 2, \lambda_4 = 1$ and eigenvectors are

$$x_1 = \begin{bmatrix} 2 \\ 2 \\ 1 \\ 1 \end{bmatrix}, \quad x_2 = \begin{bmatrix} -1 \\ -1 \\ 2 \\ 2 \end{bmatrix}, \quad x_3 = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}, \quad x_4 = \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

and $Ax_i = \lambda x_i$ for $1 \leq i \leq 4$ [28].

Following is the prototype definition of the diagonalization of a symmetric matrix (Rsyev) by GMP.

```
void Rsyev(const char *jobz, const char *uplo, mplapackint const n,
mpf_class *a, mplapackint const lda, mpf_class *w, mpf_class *work,
mplapackint const lwork, mplapackint &info);
```

and the following is the definition of the original dsyev

```
      SUBROUTINE dsyev( JOBZ, UPLO, N, A, LDA, W, WORK, LWORK, INFO )
*
*  -- LAPACK driver routine --
*  -- LAPACK is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*
*     .. Scalar Arguments ..
      CHARACTER          JOBZ, UPLO
      INTEGER            INFO, LDA, LWORK, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), W( * ), WORK( * )
*     ..
*
```

There are clear correspondences between variables in the C++ prototype of Rsyev and variables in the header of DSYEV.

- CHARACTER → const char *
- INTEGER → mplapackint
- DOUBLE PRECISION A(LDA, *) → mpf_class *a
- DOUBLE PRECISION W(*) → mpf_class *w
- DOUBLE PRECISION WORK(*) → mpf_class *work

The list is the following.

```
1 //public domain
2 #include <mpblas_gmp.h>
3 #include <mplapack_gmp.h>
```

```
4
5  #define GMP_FORMAT "%+68.64Fe"
6  #define GMP_SHORT_FORMAT "%+20.16Fe"
7
8  inline void printnum(mpf_class rtmp) { gmp_printf(GMP_SHORT_FORMAT, rtmp.get_mpf_t()); }
9  inline void printnum_short(mpf_class rtmp) { gmp_printf(GMP_SHORT_FORMAT, rtmp.get_mpf_t()
       ); }
10
11 //Matlab/Octave format
12 void printvec(mpf_class *a, int len) {
13     mpf_class tmp;
14     printf("[ ");
15     for (int i = 0; i < len; i++) {
16         tmp = a[i];
17         printnum(tmp);
18         if (i < len - 1)
19             printf(", ");
20     }
21     printf("]");
22 }
23
24 void printmat(int n, int m, mpf_class * a, int lda)
25 {
26     mpf_class mtmp;
27
28     printf("[ ");
29     for (int i = 0; i < n; i++) {
30         printf("[ ");
31         for (int j = 0; j < m; j++) {
32             mtmp = a[i + j * lda];
33             printnum(mtmp);
34             if (j < m - 1)
35                 printf(", ");
36         }
37         if (i < n - 1)
38             printf("]; ");
39         else
40             printf("] ");
41     }
42     printf("]");
43 }
44 int main()
45 {
46     mplapackint n = 4;
47     mplapackint lwork, info;
48
49     mpf_class *A = new mpf_class[n * n];
50     mpf_class *w = new mpf_class[n];
51
```

```
//setting A matrix
    A[0 + 0 * n] = 5;    A[0 + 1 * n] = 4;    A[0 + 2 * n] = 1;    A[0 + 3 * n] = 1;
    A[1 + 0 * n] = 4;    A[1 + 1 * n] = 5;    A[1 + 2 * n] = 1;    A[1 + 3 * n] = 1;
    A[2 + 0 * n] = 1;    A[2 + 1 * n] = 1;    A[2 + 2 * n] = 4;    A[2 + 3 * n] = 2;
    A[3 + 0 * n] = 1;    A[3 + 1 * n] = 1;    A[3 + 2 * n] = 2;    A[3 + 3 * n] = 4;

    printf("A ="); printmat(n, n, A, n); printf("\n");
//work space query
    lwork = -1;
    mpf_class *work = new mpf_class[1];

    Rsyev("V", "U", n, A, n, w, work, lwork, info);
    lwork = (int) cast2double (work[0]);
    delete[]work;
    work = new mpf_class[std::max((mplapackint) 1, lwork)];
//inverse matrix
    Rsyev("V", "U", n, A, n, w, work, lwork, info);
//print out some results.
    printf("#eigenvalues \n");
    printf("w ="); printmat(n, 1, w, 1); printf("\n");

    printf("#eigenvecs \n");
    printf("U ="); printmat(n, n, A, n); printf("\n");
    printf("#you can check eigenvalues using octave/Matlab by:\n");
    printf("eig(A)\n");
    printf("#you can check eigenvectors using octave/Matlab by:\n");
    printf("U'*A*U\n");

    delete[]work;
    delete[]w;
    delete[]A;
}
```

One can input the list and save as `Rsyev_test_gmp.cpp` or you can find `/home/docker/mplapack/examples/mplapack/03_SymmetricEigenproblems` directory. Then, one can compile by one own like following in Docker environment as follows:

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test_gmp.cpp -L/home/docker/MPLAPACK/lib -lmplapack_gmp -lmpblas_gmp -lgmpxx -lgmp
```

Finally, you can run as follows:

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
A =[ [ +5.0000000000000000e+00, +4.0000000000000000e+00, +1.0000000000000000e+00, +1.0000000000000000e+00];
[ +4.0000000000000000e+00, +5.0000000000000000e+00, +1.0000000000000000e+00, +1.0000000000000000e+00];]
#eigenvalues
w =[ [ +1.0000000000000000e+00]; [ +2.0000000000000000e+00]; [ +5.0000000000000000e+00]; [ +1.0000000000000000e+01] ]
#eigenvecs
U =[ [ +7.0710678118654752e-01, -1.3882760712710340e-155, -3.1622776601683793e-01, +6.3245553203367587e-01];
[ -7.0710678118654752e-01, -1.0249769098010974e-155, -3.1622776601683793e-01, +6.3245553203367587e-01]
#you can check eigenvalues using octave/Matlab by:
eig(A)
```

```
#you can check eigenvectors using octave/Matlab by:
U'*A*U
```

One can check the result by comparing the result of the octave.

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out | octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
A =

   5   4   1   1
   4   5   1   1
   1   1   4   2
   1   1   2   4

w =

    1
    2
    5
   10

U =

   0.70711  -0.00000  -0.31623   0.63246
  -0.70711  -0.00000  -0.31623   0.63246
   0.00000  -0.70711   0.63246   0.31623
   0.00000   0.70711   0.63246   0.31623

ans =

    1.00000
    2.00000
    5.00000
   10.00000

ans =

    1.0000e+00  -2.8639e-155    2.6715e-17   -5.3429e-17
  -2.6070e-155    2.0000e+00   -4.1633e-18   -2.0817e-18
   -6.7450e-17    1.7912e-16    5.0000e+00    8.5557e-17
   -3.5824e-16   -1.3490e-16    1.1229e-16    1.0000e+01
```

As a summary, we show how we compile and run binary64, FP80, binary128, double-double, quad-double, GMP and MPFR versions of `Rsyev` demo programs in `/home/docker/mplapack/examples/mplapack/03_SymmetricEigenproblems` as follows:

- binary64 (double) version

  ```
  $ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
  Rsyev_test_double.cpp -L/home/docker/MPLAPACK/lib -lmplapack_double -lmpblas_double
  $ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
  ```

- FP80 (extended double) version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test__Float64x.cpp -L/home/docker/MPLAPACK/lib -lmplapack__Float64x \
-lmpblas__Float64x
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- binary128 version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test__Float128.cpp -L/home/docker/MPLAPACK/lib -lmplapack__Float128 \
-lmpblas__Float128
 LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

or (on macOS, mingw64 and CentOS7 amd64)

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test__Float128.cpp -L/home/docker/MPLAPACK/lib -lmplapack__Float128 \
-lmpblas__Float128 -lquadmath
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- double-double version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test_dd.cpp -L/home/docker/MPLAPACK/lib -lmplapack_dd -lmpblas_dd -lqd
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- quad-double version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test_qd.cpp -L/home/docker/MPLAPACK/lib -lmplapack_qd -lmpblas_qd -lqd
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- GMP version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test_gmp.cpp -L/home/docker/MPLAPACK/lib -lmplapack_gmp -lmpblas_gmp \
-lgmpxx -lgmp
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

- MPFR version

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rsyev_test_mpfr.cpp -L/home/docker/MPLAPACK/lib -lmplapack_mpfr -lmpblas_mpfr \
-lmpfr -lmpc -lgmp
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
```

The following example shows how to solve a real non-symmetric eigenvalue problem. For example, let A is four times four matrices:

$$A = \begin{bmatrix} -2 & 2 & 2 & 2 \\ -3 & 3 & 2 & 2 \\ -2 & 0 & 4 & 2 \\ -1 & 0 & 0 & 5 \end{bmatrix}$$

with eigenvalues $\lambda_1 = 1$, $\lambda_2 = 2$, $\lambda_3 = 3$, $\lambda_4 = 4$ [28]. First, we solve it with `dd_real` class with `Rgees`. This routine computes for an N-by-N real nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization $A = ZTZ^t$. The matrix A is overwritten by T. The prototype definition of `Rgees` is following:

```
void Rgees(const char *jobvs, const char *sort, bool (*select)(dd_real, dd_real),
mplapackint const n, dd_real *a, mplapackint const lda, mplapackint &sdim,
dd_real *wr, dd_real *wi, dd_real *vs, mplapackint const ldvs, dd_real *work,
mplapackint const lwork, bool *bwork, mplapackint &info);
```

The corresponding LAPACK routine is `DGEES`. We show 14 lines of `DGEES` are following.

```
*       SUBROUTINE DGEES( JOBVS, SORT, SELECT, N, A, LDA, SDIM, WR, WI,
*                         VS, LDVS, WORK, LWORK, BWORK, INFO )
*
*       .. Scalar Arguments ..
*       CHARACTER          JOBVS, SORT
*       INTEGER            INFO, LDA, LDVS, LWORK, N, SDIM
*       ..
*       .. Array Arguments ..
*       LOGICAL            BWORK( * )
*       DOUBLE PRECISION   A( LDA, * ), VS( LDVS, * ), WI( * ), WORK( * ),
*      $                   WR( * )
*       ..
*       .. Function Arguments ..
*       LOGICAL            SELECT
*       EXTERNAL           SELECT
```

A sample program is following:

```
1  //public domain
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  #include <cstring>
6  #include <algorithm>
7
8  #include <mpblas_dd.h>
9  #include <mplapack_dd.h>
10
11 #define DD_PRECISION_SHORT 16
12
13 inline void printnum(dd_real rtmp) {
```

```
    std::cout.precision(DD_PRECISION_SHORT);
    if (rtmp >= 0.0) {
        std::cout << "+" << rtmp;
    } else {
        std::cout << rtmp;
    }
    return;
}

//Matlab/Octave format
void printvec(dd_real *a, int len) {
    dd_real tmp;
    printf("[ ");
    for (int i = 0; i < len; i++) {
        tmp = a[i];
        printnum(tmp);
        if (i < len - 1)
            printf(", ");
    }
    printf("]");
}
void printmat(int n, int m, dd_real * a, int lda)
{
    dd_real mtmp;
    printf("[ ");
    for (int i = 0; i < n; i++) {
        printf("[ ");
        for (int j = 0; j < m; j++) {
            mtmp = a[i + j * lda];
            printnum(mtmp);
            if (j < m - 1)
                printf(", ");
        }
        if (i < n - 1)
            printf("]; ");
        else
            printf("] ");
    }
    printf("]");
}
bool rselect(dd_real ar, dd_real ai) {
    // sorting rule for eigenvalues.
    return false;
}
bool rselect(dd_real ar, dd_real ai) {
    // sorting rule for eigenvalues.
    return false;
}

```

```
int main() {
    mplapackint n = 4;

    dd_real *a = new dd_real[n * n];
    dd_real *vs = new dd_real[n * n];
    mplapackint sdim = 0;
    mplapackint lwork = 3 * n;
    dd_real *wr = new dd_real[n];
    dd_real *wi = new dd_real[n];
    dd_real *work = new dd_real[lwork];
    bool bwork[n];
    mplapackint info;
    // setting A matrix
    a[0 + 0 * n] = -2.0; a[0 + 1 * n] = 2.0; a[0 + 2 * n] = 2.0; a[0 + 3 * n] = 2.0;
    a[1 + 0 * n] = -3.0; a[1 + 1 * n] = 3.0; a[1 + 2 * n] = 2.0; a[1 + 3 * n] = 2.0;
    a[2 + 0 * n] = -2.0; a[2 + 1 * n] = 0.0; a[2 + 2 * n] = 4.0; a[2 + 3 * n] = 2.0;
    a[3 + 0 * n] = -1.0; a[3 + 1 * n] = 0.0; a[3 + 2 * n] = 0.0; a[3 + 3 * n] = 5.0;

    printf("# octave check\n");
    printf("a ="); printmat(n, n, a, n); printf("\n");
    Rgees("V", "S", rselect, n, a, n, sdim, wr, wi, vs, n, work, lwork, bwork, info);
    printf("vs ="); printmat(n, n, vs, n); printf("\n");
    printf("t ="); printmat(n, n, a, n); printf("\n");
    printf("vs*t*vs'\n");
    printf("eig(a)\n");
    for (int i = 1; i <= n; i = i + 1) {
        printf("w_%d = ", (int)i); printnum(wr[i - 1]); printf(" "); printnum(wi[i - 1]);
   printf("i\n");
    }
    delete[] work;
    delete[] wr;
    delete[] wi;
    delete[] vs;
    delete[] a;
}
```

One can compile this source code by:

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgees_test_dd.cpp -L/home/docker/MPLAPACK/lib -lmplapack_dd -lmpblas_dd -lqd
```

The output of the executable is following:

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
# octave check
a =[ [ -2.0000000000000000e+00, +2.0000000000000000e+00, +2.0000000000000000e+00, +2.0000000000000000e+00];
[ -3.0000000000000000e+00, +3.0000000000000000e+00, +2.0000000000000000e+00, +2.0000000000000000e+00];
[ -2.0000000000000000e+00, +0.0000000000000000e+00, +4.0000000000000000e+00, +2.0000000000000000e+00];
[ -1.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00, +5.0000000000000000e+00] ]
vs =[ [ -7.3029674334022148e-01, -6.8313005106397323e-01, +0.0000000000000000e+00, +0.0000000000000000e+00];
[ -5.4772255750516611e-01, +5.8554004376911991e-01, +5.9761430466719682e-01, -2.7760873845656105e-33];
```

```
[ -3.6514837167011074e-01, +3.9036002917941327e-01, -7.1713716560063618e-01, -4.4721359549995794e-01];
[ -1.8257418583505537e-01, +1.9518001458970664e-01, -3.5856858280031809e-01, +8.9442719099991588e-01] ]
t =[ [ +1.0000000000000000e+00, -6.9487922897230340e+00, +2.5313275267375116e+00, -1.9595917942265425e+00];
[ +0.0000000000000000e+00, +2.0000000000000000e+00, -1.3063945294843617e+00, +7.8558440484957257e-01];
[ +0.0000000000000000e+00, +0.0000000000000000e+00, +3.0000000000000000e+00, -1.0690449676496975e+00];
[ +0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00, +4.0000000000000000e+00] ]
vs*t*vs'
eig(a)
w_1 = +1.0000000000000000e+00 +0.0000000000000000e+00i
w_2 = +2.0000000000000000e+00 +0.0000000000000000e+00i
w_3 = +3.0000000000000000e+00 +0.0000000000000000e+00i
w_4 = +4.0000000000000000e+00 +0.0000000000000000e+00i
```

We see that we calculated all the eigenvalues correctly. Moreover, you can check the Schur matrix by octave.

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out | octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
a =

  -2   2   2   2
  -3   3   2   2
  -2   0   4   2
  -1   0   0   5

vs =

  -0.73030  -0.68313   0.00000   0.00000
  -0.54772   0.58554   0.59761  -0.00000
  -0.36515   0.39036  -0.71714  -0.44721
  -0.18257   0.19518  -0.35857   0.89443

t =

   1.00000  -6.94879   2.53133  -1.95959
   0.00000   2.00000  -1.30639   0.78558
   0.00000   0.00000   3.00000  -1.06904
   0.00000   0.00000   0.00000   4.00000

ans =

  -2.0000e+00   2.0000e+00   2.0000e+00   2.0000e+00
  -3.0000e+00   3.0000e+00   2.0000e+00   2.0000e+00
  -2.0000e+00   5.1394e-16   4.0000e+00   2.0000e+00
  -1.0000e+00   2.5697e-16  -4.5393e-17   5.0000e+00

ans =

   1.00000
   2.00000
   3.00000
   4.00000

w_1 =  1
w_2 =  2
```

```
w_3 =  3
w_4 =  4
```

We can see that we calculated all the eigenvalues, the Schur form, and Schur vectors correctly.

Another example shows how to solve real non-symmetric eigenvalue problems with eigenvectors. `Rgeev` computes the eigenvalues, and, optionally, the left and/or right eigenvectors for an N by N real nonsymmetric matrix A. For example, let A is four times four matrices:

$$A = \begin{bmatrix} 4 & -5 & 0 & 3 \\ 0 & 4 & -3 & -5 \\ 5 & -3 & 4 & 0 \\ 3 & 0 & 5 & 4 \end{bmatrix}$$

Then, the eigenvalues are $\lambda_1 = 12$, $\lambda_2 = 1 + 5i$, $\lambda_3 = 1 - 5i$ and $\lambda_4 = 2$ and the right eigenvectors are

$$x_1 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}, \quad x_2 = \begin{bmatrix} 1 \\ -i \\ -i \\ -1 \end{bmatrix}, \quad x_3 = \begin{bmatrix} 1 \\ i \\ i \\ -1 \end{bmatrix}, \quad x_4 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix},$$

and the left eigenvectors are

$$y_1 = [1, -1, 1, 1], \quad y_2 = [1, i, i, -1], \quad y_3 = [1, -i, -i, -1], \quad y_4 = [1, 1, -1, 1].$$

Thus, $Ax_1 = 12x_1$, $Ax_2 = (1 + 5i)x_2$, $Ax_3 = (1 - 5i)x_3$ and $Ax_4 = 2x_4$, and $y_1A = 12y_1$, and $y_2A = (1 + 5i)y_2$, $y_3A = (1 - 5i)y_3$, $y_4A = 2y_4$.

Let us solve this eigenvalue problem with left and right eigenvectors for A using MPLAPACK `qd_real`. The prototype definition of `Rgeev` is following:

```
void Rgeev(const char *jobvl, const char *jobvr, mplapackint const n, qd_real *a,
mplapackint const lda, qd_real *wr, qd_real *wi, qd_real *vl, mplapackint const ldvl,
qd_real *vr, mplapackint const ldvr, qd_real *work, mplapackint const lwork,
mplapackint &info);
```

The corresponding LAPACK routine is `DGEEV`. We show the 14 lines of `DGEEV` are following.

```
      SUBROUTINE dgeev( JOBVL, JOBVR, N, A, LDA, WR, WI, VL, LDVL, VR,
      $                  LDVR, WORK, LWORK, INFO )
       implicit none
*
*  -- LAPACK driver routine --
*  -- LAPACK is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*
*     .. Scalar Arguments ..
       CHARACTER          JOBVL, JOBVR
       INTEGER            INFO, LDA, LDVL, LDVR, LWORK, N
*     ..
*     .. Array Arguments ..
       DOUBLE PRECISION   A( LDA, * ), VL( LDVL, * ), VR( LDVR, * ),
      $                   wi( * ), work( * ), wr( * )
```

A sample program is following:

```
//public domain
#include <iostream>
#include <string>
#include <sstream>
#include <cstring>
#include <algorithm>

#include <mpblas_qd.h>
#include <mplapack_qd.h>

#define QD_PRECISION_SHORT 16

inline void printnum(qd_real rtmp) {
    std::cout.precision(QD_PRECISION_SHORT);
    if (rtmp >= 0.0) {
        std::cout << "+" << rtmp;
    } else {
        std::cout << rtmp;
    }
    return;
}

//Matlab/Octave format
void printvec(qd_real *a, int len) {
    qd_real tmp;
    printf("[ ");
    for (int i = 0; i < len; i++) {
        tmp = a[i];
        printnum(tmp);
        if (i < len - 1)
            printf(", ");
    }
    printf("]");
}

void printmat(int n, int m, qd_real * a, int lda)
{
    qd_real mtmp;
    printf("[ ");
    for (int i = 0; i < n; i++) {
        printf("[ ");
        for (int j = 0; j < m; j++) {
            mtmp = a[i + j * lda];
            printnum(mtmp);
            if (j < m - 1)
                printf(", ");
        }
        if (i < n - 1)
```

```
            printf("]; ");
        else
            printf("] ");
    }
    printf("]");
}
bool rselect(qd_real ar, qd_real ai) {
    // sorting rule for eigenvalues.
    return false;
}

int main() {
    mplapackint n = 4;
    qd_real *a = new qd_real[n * n];
    qd_real *vl = new qd_real[n * n];
    qd_real *vr = new qd_real[n * n];
    mplapackint lwork = 4 * n;
    qd_real *wr = new qd_real[n];
    qd_real *wi = new qd_real[n];
    qd_real *work = new qd_real[lwork];
    mplapackint info;
    // setting A matrix
    a[0 + 0 * n] = 4.0; a[0 + 1 * n] = -5.0; a[0 + 2 * n] =  0.0;  a[0 + 3 * n] =  3.0;
    a[1 + 0 * n] = 0.0; a[1 + 1 * n] =  4.0; a[1 + 2 * n] = -3.0;  a[1 + 3 * n] = -5.0;
    a[2 + 0 * n] = 5.0; a[2 + 1 * n] = -3.0; a[2 + 2 * n] =  4.0;  a[2 + 3 * n] =  0.0;
    a[3 + 0 * n] = 3.0; a[3 + 1 * n] =  0.0; a[3 + 2 * n] =  5.0;  a[3 + 3 * n] =  4.0;

    printf("# octave check\n");
    printf("split_long_rows(0)\n");
    printf("a =");
    printmat(n, n, a, n);
    printf("\n");
    Rgeev("V", "V", n, a, n, wr, wi, vl, n, vr, n, work, lwork, info);
    printf("# right vectors\n");
    for (int j = 1; j <= n; j = j + 1) {
        if (abs(wi[j - 1]) < 1e-15) {
            printf("vr_%d =[ ", j);
            for (int i = 1; i <= n - 1; i = i + 1) {
                printnum(vr[(i - 1) + (j - 1) * n]); printf(", ");
            }
            printnum(vr[(n - 1) + (j - 1) * n]); printf("];\n");
        } else {
            printf("vr_%d =[ ", j);
            for (int i = 1; i <= n - 1; i = i + 1) {
                printnum(vr[(i - 1) + (j - 1) * n]); printnum(-vr[(i - 1) + j * n]);
   printf("i, ");
            }
            printnum(vr[(n - 1) + (j - 1) * n]); printnum(-vr[(n - 1) + j * n]); printf("i
    ];\n");
```

```
            printf("vr_%d =[ ", j + 1);
            for (int i = 1; i <= n - 1; i = i + 1) {
                printnum(vr[(i - 1) + (j - 1) * n]); printnum(vr[(i - 1) + j * n]); printf("i, ");
            }
            printnum(vr[(n - 1) + (j - 1) * n]); printnum(vr[(n - 1) + j * n]); printf("i];\n");
            j++;
        }
    }
    printf("# left vectors\n");
    for (int j = 1; j <= n; j = j + 1) {
        if (abs(wi[j - 1]) < 1e-15) {
            printf("vl_%d =[ ", j);
            for (int i = 1; i <= n - 1; i = i + 1) {
                printnum(vl[(i - 1) + (j - 1) * n]); printf(", ");
            }
            printnum(vl[(n - 1) + (j - 1) * n]); printf("];\n");
        } else {
            printf("vl_%d =[ ", j);
            for (int i = 1; i <= n - 1; i = i + 1) {
                printnum(vl[(i - 1) + (j - 1) * n]); printnum(-vl[(i - 1) + j * n]); printf("i, ");
            }
            printnum(vl[(n - 1) + (j - 1) * n]); printnum(-vl[(n - 1) + j * n]); printf("i];\n");
            printf("vl_%d =[ ", j + 1);
            for (int i = 1; i <= n - 1; i = i + 1) {
                printnum(vl[(i - 1) + (j - 1) * n]); printnum(vl[(i - 1) + j * n]); printf("i, ");
            }
            printnum(vl[(n - 1) + (j - 1) * n]); printnum(vl[(n - 1) + j * n]); printf("i];\n");
            j++;
        }
    }
    for (int i = 1; i <= n; i = i + 1) {
        printf("w_%d = ", (int)i); printnum(wr[i - 1]); printf(" "); printnum(wi[i - 1]); printf("i\n");
    }
    for (int i = 1; i <= n; i = i + 1) {
        printf("disp (\"a * vr_%d\")\n", i);
        printf("a * vr_%d'\n", i);
        printf("disp (\"w_%d * vr_%d\")\n", i, i);
        printf("w_%d * vr_%d'\n", i, i);
        printf("disp (\"vr_%d\")\n", i);
        printf("vr_%d'\n", i);
    }

```

```
    for (int i = 1; i <= n; i = i + 1) {
        printf("disp (\"vl_%d * a \")\n", i);
        printf("vl_%d * a\n", i);
        printf("disp (\"w_%d * vl_%d \")\n", i, i);
        printf("w_%d * vl_%d\n", i, i);
        printf("disp (\"vl_%d\")\n", i);
        printf("vl_%d\n", i);
    }
    delete[] work;
    delete[] wr;
    delete[] wi;
    delete[] vr;
    delete[] vl;
    delete[] a;
}
```

You can find this file as /home/docker/mplapack/examples/mplapack/04_NonsymmetricEigenproblems/Rgeev_test_qd.cpp , and compile this source code by:

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgeev_test_qd.cpp -L/home/docker/MPLAPACK/lib -lmplapack_qd -lmpblas_qd -lqd
```

The output of the executable is following:

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
# octave check
split_long_rows(0)
a =[ [ +4.0000000000000000e+00, -5.0000000000000000e+00, +0.0000000000000000e+00, +3.0000000000000000e+00];
[ +0.0000000000000000e+00, +4.0000000000000000e+00, -3.0000000000000000e+00, -5.0000000000000000e+00];
[ +5.0000000000000000e+00, -3.0000000000000000e+00, +4.0000000000000000e+00, +0.0000000000000000e+00];
[ +3.0000000000000000e+00, +0.0000000000000000e+00, +5.0000000000000000e+00, +4.0000000000000000e+00] ]
# right vectors
vr_1 =[ -5.0000000000000000e-01, +5.0000000000000000e-01, -5.0000000000000000e-01, -5.0000000000000000e-01];
vr_2 =[ +2.8486703237279396e-65-5.0000000000000000e-01i, +5.0000000000000000e-01+0.0000000000000000e+00i,
+5.0000000000000000e-01+0.0000000000000000e+00i, -9.0207893584718088e-65+5.0000000000000000e-01i ];
vr_3 =[ +2.8486703237279396e-65+5.0000000000000000e-01i, +5.0000000000000000e-01+0.0000000000000000e+00i,
+5.0000000000000000e-01+0.0000000000000000e+00i, -9.0207893584718088e-65-5.0000000000000000e-01i ];
vr_4 =[ +5.0000000000000000e-01, +5.0000000000000000e-01, -5.0000000000000000e-01, +5.0000000000000000e-01];
# left vectors
vl_1 =[ -5.0000000000000000e-01, +5.0000000000000000e-01, -5.0000000000000000e-01, -5.0000000000000000e-01];
vl_2 =[ -4.7477838728798994e-65-5.0000000000000000e-01i, +5.0000000000000000e-01+0.0000000000000000e+00i,
+5.0000000000000000e-01-1.7092021942367638e-64i, -3.4184043884735275e-64+5.0000000000000000e-01i ];
vl_3 =[ -4.7477838728798994e-65+5.0000000000000000e-01i, +5.0000000000000000e-01+0.0000000000000000e+00i,
+5.0000000000000000e-01+1.7092021942367638e-64i, -3.4184043884735275e-64-5.0000000000000000e-01i ];
vl_4 =[ +5.0000000000000000e-01, +5.0000000000000000e-01, -5.0000000000000000e-01, +5.0000000000000000e-01];
w_1 = +1.2000000000000000e+01 +0.0000000000000000e+00i
w_2 = +1.0000000000000000e+00 +5.0000000000000000e+00i
w_3 = +1.0000000000000000e+00 -5.0000000000000000e+00i
w_4 = +2.0000000000000000e+00 +0.0000000000000000e+00i
...
```

This can be confirmed by passing the output of the terminal to Octave, as shown below.

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out | octave
a =

   4  -5   0   3
   0   4  -3  -5
   5  -3   4   0
   3   0   5   4

w_1 =  12
w_2 =  1 + 5i
w_3 =  1 - 5i
w_4 =  2
a * vr_1
ans =

  -6
   6
  -6
  -6

w_1 * vr_1
ans =

  -6
   6
  -6
  -6

vr_1
ans =

  -0.50000
   0.50000
  -0.50000
  -0.50000

a * vr_2
ans =

  -2.50000 + 0.50000i
   0.50000 + 2.50000i
   0.50000 + 2.50000i
   2.50000 - 0.50000i

w_2 * vr_2
ans =

  -2.50000 + 0.50000i
   0.50000 + 2.50000i
   0.50000 + 2.50000i
   2.50000 - 0.50000i

vr_2
ans =

   0.00000 + 0.50000i
```

```
  0.50000 - 0.00000i
  0.50000 - 0.00000i
 -0.00000 - 0.50000i

...
```

We only show the first part of the output, since the whole output is lengthy. `w_1`, `w_2`, `w_3` and `w_4` are the eigenvalues. `vr_1` and `vr_2` are right eigenvectors. You will see you obtained correct eigenvalues and eigenvectors.

The final example shows how to solve a singular value problem [29]. Let A is 4×5 as follows:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Then, a singular value decomposition of this matrix is given by $A = U\Sigma V^t$ as follows.

$$U = \begin{bmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{bmatrix},$$

$$\Sigma = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{5} & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

and

$$V^t = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ -\sqrt{0.2} & 0 & 0 & 0 & -\sqrt{0.8} \\ 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}.$$

Let us solve the singular value problem for matrix A using MPLAPACK `dd_real`. The prototype definition of `Rgesvd` is following:

```
void Rgesvd(const char *jobu, const char *jobvt, mplapackint const m,
mplapackint const n, dd_real *a, mplapackint const lda, dd_real *s, dd_real *u,
mplapackint const ldu, dd_real *vt, mplapackint const ldvt, dd_real *work,
mplapackint const lwork, mplapackint &info);
```

The corresponding LAPACK routine is `DGESVD`. We show some lines of `DGESVD` are following.

```
       SUBROUTINE dgesvd( JOBU, JOBVT, M, N, A, LDA, S, U, LDU,
     $                   VT, LDVT, WORK, LWORK, INFO )
*
*  -- LAPACK driver routine --
*  -- LAPACK is a software package provided by Univ. of Tennessee,    --
```

```
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*
*     .. Scalar Arguments ..
      CHARACTER          JOBU, JOBVT
      INTEGER            INFO, LDA, LDU, LDVT, LWORK, M, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), S( * ), U( LDU, * ),
     $                   vt( ldvt, * ), work( * )
*     ..
```

A sample program for `dd_real` is following:

```
//public domain
#include <iostream>
#include <string>
#include <sstream>
#include <cstring>
#include <algorithm>

#include <mpblas_dd.h>
#include <mplapack_dd.h>

#define DD_PRECISION_SHORT 16

inline void printnum(dd_real rtmp) {
    std::cout.precision(DD_PRECISION_SHORT);
    if (rtmp >= 0.0) {
        std::cout << "+" << rtmp;
    } else {
        std::cout << rtmp;
    }
    return;
}

//Matlab/Octave format
void printvec(dd_real *a, int len) {
    dd_real tmp;
    printf("[ ");
    for (int i = 0; i < len; i++) {
        tmp = a[i];
        printnum(tmp);
        if (i < len - 1)
            printf(", ");
    }
    printf("]");
}
void printmat(int n, int m, dd_real * a, int lda)
{
    dd_real mtmp;
```

```
38      printf("[ ");
39      for (int i = 0; i < n; i++) {
40          printf("[ ");
41          for (int j = 0; j < m; j++) {
42              mtmp = a[i + j * lda];
43              printnum(mtmp);
44              if (j < m - 1)
45                  printf(", ");
46          }
47          if (i < n - 1)
48              printf("]; ");
49          else
50              printf("] ");
51      }
52      printf("]");
53  }
54  int main() {
55      mplapackint n = 5;
56      mplapackint m = 4;
57
58      dd_real *a = new dd_real[m * n];
59      dd_real *s = new dd_real[std::min(m, n)];
60      dd_real *u = new dd_real[m * m];
61      dd_real *vt = new dd_real[n * n];
62      mplapackint lwork = std::max({(mplapackint)1, 3 * std::min(m, n) + std::max(m, n), 5 *
        std::min(m, n)});
63      dd_real *work = new dd_real[lwork];
64      mplapackint info;
65
66      // setting A matrix
67      a[0 + 0 * m] = 1.0; a[0 + 1 * m] = 0.0; a[0 + 2 * m] = 0.0;  a[0 + 3 * m] = 0.0;  a[0
       + 4 * m] = 2.0;
68      a[1 + 0 * m] = 0.0; a[1 + 1 * m] = 0.0; a[1 + 2 * m] = 3.0;  a[1 + 3 * m] = 0.0;  a[1
       + 4 * m] = 0.0;
69      a[2 + 0 * m] = 0.0; a[2 + 1 * m] = 0.0; a[2 + 2 * m] = 0.0;  a[2 + 3 * m] = 0.0;  a[2
       + 4 * m] = 0.0;
70      a[3 + 0 * m] = 0.0; a[3 + 1 * m] = 2.0; a[3 + 2 * m] = 0.0;  a[3 + 3 * m] = 0.0;  a[3
       + 4 * m] = 0.0;
71
72      printf("# octave check\n");
73      printf("a ="); printmat(m, n, a, m); printf("\n");
74      Rgesvd("A", "A", m, n, a, m, s, u, m, vt, n, work, lwork, info);
75      printf("s="); printvec(s, std::min(m, n)); printf("\n");
76      if (m < n)
77          printf("padding=zeros(%d, %d-%d)\n", (int)m, (int)n, (int)m);
78      if (n < m)
79          printf("padding=zeros(%d-%d,%d)\n", (int)m, (int)n, (int)n);
80      printf("u ="); printmat(m, m, u, m); printf("\n");
81      printf("vt ="); printmat(n, n, vt, n); printf("\n");
```

```
    printf("svd(a)\n");
    if (m < n)
        printf("sigma=[diag(s) padding] \n");
    if (n < m)
        printf("sigma=[diag(s); padding] \n");
    if (n == m)
        printf("sigma=[diag(s)] \n");
    printf("sigma \n");
    printf("u * sigma  * vt\n");
    delete[] work;
    delete[] vt;
    delete[] u;
    delete[] s;
    delete[] a;
}
```

You can find corresponding example files for various mutiple-precision verions as `Rgesvd_test__Float128.cpp`, `Rgesvd_test_dd.cpp`, `Rgesvd_test_gmp.cpp`, `Rgesvd_test_qd.cpp`, `Rgesvd_test__Float64x.cpp`, `Rgesvd_test_double.cpp`, and `Rgesvd_test_mpfr.cpp` at `/home/docker/mplapack/examples/mplapack/05_SingularValueDecomposition/`. In `dd_real` case, you can compile `dd_real` version by:

```
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgesvd_test_dd.cpp -L/home/docker/MPLAPACK/lib -lmplapack_dd -lmpblas_dd -lqd
```

The output of the executable is following:

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out
# octave check
a =[ [ +1.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00,
+2.0000000000000000e+00]; [ +0.0000000000000000e+00, +0.0000000000000000e+00, +3.0000000000000000e+00,
+0.0000000000000000e+00, +0.0000000000000000e+00]; [ +0.0000000000000000e+00, +0.0000000000000000e+00,
+0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00]; [ +0.0000000000000000e+00,
+2.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00] ]
s=[ +3.0000000000000000e+00, +2.2360679774997897e+00, +2.0000000000000000e+00, +0.0000000000000000e+00]
padding=zeros(4, 5-4)
u =[ [ +0.0000000000000000e+00, +1.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00];
[ +1.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00];
[ +0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00, -1.0000000000000000e+00];
[ +0.0000000000000000e+00, +0.0000000000000000e+00, +1.0000000000000000e+00, +0.0000000000000000e+00] ]
vt =[ [ +0.0000000000000000e+00, +0.0000000000000000e+00, +1.0000000000000000e+00, +0.0000000000000000e+00,
+0.0000000000000000e+00]; [ +4.4721359549995794e-01, +0.0000000000000000e+00, +0.0000000000000000e+00,
+0.0000000000000000e+00, +8.9442719099991588e-01]; [ +0.0000000000000000e+00, +1.0000000000000000e+00,
+0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00]; [ +0.0000000000000000e+00,
+0.0000000000000000e+00, +0.0000000000000000e+00, +1.0000000000000000e+00, +0.0000000000000000e+00];
[ -8.9442719099991588e-01,
+0.0000000000000000e+00, +0.0000000000000000e+00, +0.0000000000000000e+00, +4.4721359549995794e-01] ]
svd(a)
sigma=[diag(s) padding]
sigma
u * sigma  * vt
```

To better readability, you can pass the output to the octave. The output is following:

```
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out | octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
a =

   1   0   0   0   2
   0   0   3   0   0
   0   0   0   0   0
   0   2   0   0   0

s =

   3.00000   2.23607   2.00000   0.00000

padding =

   0
   0
   0
   0

u =

   0   1   0   0
   1   0   0   0
   0   0   0  -1
   0   0   1   0

vt =

   0.00000   0.00000   1.00000   0.00000   0.00000
   0.44721   0.00000   0.00000   0.00000   0.89443
   0.00000   1.00000   0.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   1.00000   0.00000
  -0.89443   0.00000   0.00000   0.00000   0.44721

ans =

   3.00000
   2.23607
   2.00000
   0.00000

sigma =

   3.00000   0.00000   0.00000   0.00000   0.00000
   0.00000   2.23607   0.00000   0.00000   0.00000
   0.00000   0.00000   2.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000   0.00000

sigma =

   3.00000   0.00000   0.00000   0.00000   0.00000
   0.00000   2.23607   0.00000   0.00000   0.00000
   0.00000   0.00000   2.00000   0.00000   0.00000
   0.00000   0.00000   0.00000   0.00000   0.00000
```

```
ans =

   1   0   0   0   2
   0   0   3   0   0
   0   0   0   0   0
   0   2   0   0   0
```

You will see that `Rgesvd`, the singular value decomposition solver solved the problem correctly.

There are more examples in `/home/docker/mplapack/examples/mplapack/` as follows:

- `00_LinearEquations/Cgetri_test_*.cpp` ... samples of inversion of complex matrices
- `00_LinearEquations/Rgesv_test_*.cpp` ... samples of solving linear equation of real matrices
- `00_LinearEquations/Rgetri_Hilbert_*.cpp` ... samples of inversion of Hilbert matrices of various orders.
- `00_LinearEquations/Rgetri_test*.cpp` ... samples of inversion of real matrices.
- `03_SymmetricEigenproblems/Cheev_test_*.cpp` ... samples of diagonalization of Hermitian matrices.
- `03_SymmetricEigenproblems/Rsyev_Frank_*.cpp` ... samples of diagonalization of real Frank matrices.
- `03_SymmetricEigenproblems/Rsyev_test_*.cpp` ... samples of diagonalization of real matrices.
- `03_SymmetricEigenproblems/Rsyevd_DingDong_*.cpp` ... samples of diagonalization of Ding Dong matrices.
- `03_SymmetricEigenproblems/Rsyevd_Frank_*.cpp` ... samples of diagonalization of real Frank matrices using divide-and-conquer driver.
- `03_SymmetricEigenproblems/Rsyevr_Frank_*.cpp` ... samples of diagonalization of real Frank matrices using relatively robust representation driver.
- `04_NonsymmetricEigenproblems/Rgees_Grcar_*.cpp` ... samples of solving non-symmetic eigenvalues problem using Grcar matrix (Schur form)
- `04_NonsymmetricEigenproblems/Rgees_readfromfile_*.cpp` ... samples of solving non-symmetic eigenvalues problem using matrix from a file (Schur form)
- `04_NonsymmetricEigenproblems/Rgees_test_*.cpp` ... samples of solving simple non-symmetic eigenvalues problem (Schur form)
- `04_NonsymmetricEigenproblems/Rgeev_Frank_*.cpp` ... samples of solving non-symmetic eigenvalues problem for Frank matrix
- `04_NonsymmetricEigenproblems/Rgeev_random_highcond_*.cpp` ... samples of solving non-symmetic eigenvalues problem for high conditioned random matrix

- 04_NonsymmetricEigenproblems/Rgeev_readfromfile_*.cpp ... samples of solving non-symmetic eigenvalues problem using matrix from a file
- 04_NonsymmetricEigenproblems/Rgeev_test_*.cpp .. samples of solving simple non-symmetic eigenvalues problem
- 05_SingularValueDecomposition/Rgesvd_random_highcond_*.cpp ...samples of solving singular value decomposition problem for matrices with high condition numbers
- 05_SingularValueDecomposition/Rgesvd_readfromfile_*.cpp ... samples of singular value decomposition problems using matrix from a file
- 05_SingularValueDecomposition/Rgesvd_test_*.cpp ... samples of simple singular value decomposition problems

6.5 Rlamch values and specifying precision at runtime

For MPLAPACK, Rlamch routines are vital since they return precision-specific constants. Exceedingly, these values control convergence behaviors for such as eigenvalue problems and singular value problems. Thus, for example, we cannot say binary128 version of MPLAPACK routines run output if we use the same Rlamch values for binary64 and vice versa. In table 9, we show Rlamch values for MPFR. The default accuracy for MPFR is approximately 154 decimal digits ($154.127 = 512 \log_{10} 2$). In table 10, we show Rlamch values for MPFR with MPLAPACK_MPFR_PRECISION=65536. This accuracy is approximately 19728 decimal digits ($19728 = 65536 \log_{10} 2$).

In table 11, we show Rlamch values for GMP. The default accuracy for MPFR is approximately 154 decimal digits ($154.127 = 512 \log_{10} 2$). In table 12, we show Rlamch values with for GMP with MPLAPACK_GMP_PRECISION=1024. This accuracy is approximately 308 decimal digits ($308.25 = 1024 \log_{10} 2$). In table 13, we show Rlamch values for _Float128. In table 14, we show Rlamch values for _Float64x. In table 15, we show Rlamch values for double. In table 16, we show Rlamch values for dd_real. We intentionally reduce the overflow values so that sqrt does not fail. In table 17, we show Rlamch values for qd_real, respectively.

We can change the default precision at runtime for GMP and MPFR by setting environment variable, MPLAPACK_GMP_PRECISION and MPLAPACK_MPFR_PRECISION, respectively.

Let us show an example by inverting Hilerbt matrix. Rgetri_Hilbert_mpfr inverts Hilbert matrix of order n, where $n = 1, 2, \cdots$ via LU decomposition. a is the Hilbert matrix and ainv is inverted matrix, and InfnormL is the residuals.

The output is following:

```
$ cd /home/docker/mplapack/examples/mplapack/00_LinearEquations
$ g++ -O2 -I/home/docker/MPLAPACK/include -I/home/docker/MPLAPACK/include/mplapack \
Rgetri_Hilbert_mpfr.cpp -L/home/docker/MPLAPACK/lib -lmplapack_mpfr -lmpblas_mpfr \
-lmpfr -lmpc -lgmp
$ LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out | grep InfnromL
InfnormL:(ainv * a - I)=+0.0000000000000000e+00
InfnormL:(ainv * a - I)=+0.0000000000000000e+00
InfnormL:(ainv * a - I)=+4.7733380679681323e-153
InfnormL:(ainv * a - I)=+7.6373409087490117e-152
```

Table 9: Rlamch values for MPFR (default setting)

```
Rlamch E: Epsilon                       +7.4583407312002067e-155
Rlamch S: Safe minimum                  +9.5302596195518043e-323228497
Rlamch B: Base                          +2.0000000000000000e+00
Rlamch P: Precision                     +1.4916681462400413e-154
Rlamch N: Number of digits in mantissa +5.1200000000000000e+02
Rlamch R: Rounding mode                 +1.0000000000000000e+00
Rlamch M: Minimum exponent:             -1.0737418230000000e+09
Rlamch U: Underflow threshold           +9.5302596195518043e-323228497
Rlamch L: Largest exponent              +1.0737418220000000e+09
Rlamch O: Overflow threshold            +2.0985787164673877e+323228496
Rlamch -: Reciprocal of safe minimum    +1.0492893582336938e+323228496


InfnormL:(ainv * a - I)=+2.4439490907996837e-150
InfnormL:(ainv * a - I)=+1.1730955635838482e-148
InfnormL:(ainv * a - I)=+2.1897783853565166e-147
InfnormL:(ainv * a - I)=+6.0062492855493028e-146
InfnormL:(ainv * a - I)=+2.7228330094490173e-144
InfnormL:(ainv * a - I)=+8.2005323578699814e-143
...
$ MPLAPACK_MPFR_PRECISION=65536 LD_LIBRARY_PATH=/home/docker/MPLAPACK/lib ./a.out | \
grep InfnromL
InfnormL:(ainv * a - I)=+0.0000000000000000e+00
InfnormL:(ainv * a - I)=+0.0000000000000000e+00
InfnormL:(ainv * a - I)=+3.9929525776415436e-19728
InfnormL:(ainv * a - I)=+5.1109792993811758e-19726
InfnormL:(ainv * a - I)=+3.2710267516039525e-19724
InfnormL:(ainv * a - I)=+5.2336428025663240e-19723
InfnormL:(ainv * a - I)=+8.3738284841061184e-19722
InfnormL:(ainv * a - I)=+5.0242970904636710e-19720
InfnormL:(ainv * a - I)=+1.0718500459655832e-19718
...
```

In the first case, we calculate in 512 bit accuracy or 154 decimal digit accuracy ($154.127 = 512 \log_{10} 2$), and the next case we calculate in 65536 bit accuracy or 19728 decimal digits accuracy ($19728 = 65536 \log_{10} 2$). The results show we can specify how many bits are for multiple-precision calculations at the runtime.

6.6 Summary: a basic strategy to use MPBLAS and MPLAPACK

Unfortunately, We do not provide concise manuals for MPBLAS and MPLAPACK since LAPACK is very well documented. If you want to use some specific routines, first consult LAPACK users' guide [3], or alternatively, `https://www.netlib.org/lapack/lug/`.

Following is the basic strategy to use MPLAPACK routines.

Table 10: Rlamch values for MPFR (MPLAPACK_MPFR_PRECISION=65536)

```
Rlamch E: Epsilon                       +4.9911907220519295e-19729
Rlamch S: Safe minimum                  +9.5302596195518043e-323228497
Rlamch B: Base                          +2.0000000000000000e+00
Rlamch P: Precision                     +9.9823814441038589e-19729
Rlamch N: Number of digits in mantissa +6.5536000000000000e+04
Rlamch R: Rounding mode                 +1.0000000000000000e+00
Rlamch M: Minimum exponent:             -1.0737418230000000e+09
Rlamch U: Underflow threshold           +9.5302596195518043e-323228497
Rlamch L: Largest exponent              +1.0737418220000000e+09
Rlamch O: Overflow threshold            +2.0985787164673877e+323228496
Rlamch -: Reciprocal of safe minimum    +1.0492893582336938e+323228496
```

Table 11: Rlamch values for GMP (default setting)

```
Rlamch E: Epsilon                       +7.4583407312002067e-155
Rlamch S: Safe minimum                  +6.4447927660133239e-1292913987
Rlamch B: Base                          +2.0000000000000000e+00
Rlamch P: Precision                     +1.4916681462400413e-154
Rlamch N: Number of digits in mantissa +5.1200000000000000e+02
Rlamch R: Rounding mode                 +1.0000000000000000e+00
Rlamch M: Minimum exponent:             -4.2949672950000000e+09
Rlamch U: Underflow threshold           +6.4447927660133239e-1292913987
Rlamch L: Largest exponent              +4.2949672950000000e+09
Rlamch O: Overflow threshold            +1.5516402719316431e+1292913986
Rlamch -: Reciprocal of safe minimum    +1.5516402719316431e+1292913986
```

Table 12: Rlamch values for GMP with MPLAPACK_GMP_PRECISION=1024

```
Rlamch E: Epsilon                       +5.5626846462680035e-309
Rlamch S: Safe minimum                  +6.4447927660133239e-1292913987
Rlamch B: Base                          +2.0000000000000000e+00
Rlamch P: Precision                     +1.1125369292536007e-308
Rlamch N: Number of digits in mantissa +1.0240000000000000e+03
Rlamch R: Rounding mode                 +1.0000000000000000e+00
Rlamch M: Minimum exponent:             -4.2949672950000000e+09
Rlamch U: Underflow threshold           +6.4447927660133239e-1292913987
Rlamch L: Largest exponent              +4.2949672950000000e+09
Rlamch O: Overflow threshold            +1.5516402719316431e+1292913986
Rlamch -: Reciprocal of safe minimum    +1.5516402719316431e+1292913986
```

Table 13: Rlamch values for _Float128

Rlamch E: Epsilon	+1.9259299443872359e-34
Rlamch S: Safe minimum	+3.3621031431120935e-4932
Rlamch B: Base	+2.0000000000000000e+00
Rlamch P: Precision	+3.8518598887744717e-34
Rlamch N: Number of digits in mantissa	+1.1300000000000000e+02
Rlamch R: Rounding mode	+1.0000000000000000e+00
Rlamch M: Minimum exponent:	-1.6381000000000000e+04
Rlamch U: Underflow threshold	+3.3621031431120935e-4932
Rlamch L: Largest exponent	+1.6384000000000000e+04
Rlamch O: Overflow threshold	+1.1897314953572318e+4932
Rlamch -: Reciprocal of safe minimum	+2.9743287383930794e+4931

Table 14: Rlamch values for _Float64x

Rlamch E: Epsilon	+5.4210108624275222e-20
Rlamch S: Safe minimum	+3.3621031431120935e-4932
Rlamch B: Base	+2.0000000000000000e+00
Rlamch P: Precision	+1.0842021724855044e-19
Rlamch N: Number of digits in mantissa	+6.4000000000000000e+01
Rlamch R: Rounding mode	+1.0000000000000000e+00
Rlamch M: Minimum exponent:	-1.6381000000000000e+04
Rlamch U: Underflow threshold	+3.3621031431120935e-4932
Rlamch L: Largest exponent	+1.6384000000000000e+04
Rlamch O: Overflow threshold	+1.1897314953572318e+4932
Rlamch -: Reciprocal of safe minimum	+2.9743287383930794e+4931

Table 15: Rlamch values for double

Rlamch E: Epsilon	+1.1102230246251565e-16
Rlamch S: Safe minimum	+2.2250738585072014e-308
Rlamch B: Base	+2.0000000000000000e+00
Rlamch P: Precision	+2.2204460492503131e-16
Rlamch N: Number of digits in mantissa	+5.3000000000000000e+01
Rlamch R: Rounding mode	+1.0000000000000000e+00
Rlamch M: Minimum exponent:	-1.0210000000000000e+03
Rlamch U: Underflow threshold	+2.2250738585072014e-308
Rlamch L: Largest exponent	+1.0240000000000000e+03
Rlamch O: Overflow threshold	+1.7976931348623157e+308
Rlamch -: Reciprocal of safe minimum	+4.4942328371557898e+307

Table 16: Rlamch values for `dd_real`

```
Rlamch E: Epsilon                      +4.9303806576313200e-32
Rlamch S: Safe minimum                 +2.0041683600089728e-292
Rlamch B: Base                         +2.0000000000000000e+00
Rlamch P: Precision                    +9.8607613152626399e-32
Rlamch N: Number of digits in mantissa +1.0400000000000000e+02
Rlamch R: Rounding mode                +1.0000000000000000e+00
Rlamch M: Minimum exponent:            -9.6900000000000000e+02
Rlamch U: Underflow threshold          +2.0041683600089728e-292
Rlamch L: Largest exponent             +1.0240000000000000e+03
Rlamch O: Overflow threshold           +1.9958403095347197e+292
Rlamch -: Reciprocal of safe minimum   +4.9896007738367995e+291
```

Table 17: Rlamch values for `qd_real`

```
Rlamch E: Epsilon                      +1.2154326714572501e-63
Rlamch S: Safe minimum                 +1.6259745436952323e-260
Rlamch B: Base                         +2.0000000000000000e+00
Rlamch P: Precision                    +2.4308653429145001e-63
Rlamch N: Number of digits in mantissa +2.0800000000000000e+02
Rlamch R: Rounding mode                +1.0000000000000000e+00
Rlamch M: Minimum exponent:            -8.6300000000000000e+02
Rlamch U: Underflow threshold          +1.6259745436952323e-260
Rlamch L: Largest exponent             +1.0240000000000000e+03
Rlamch O: Overflow threshold           +1.9958403095347197e+292
Rlamch -: Reciprocal of safe minimum   +6.1501577861568104e+259
```

1. See `https://www.netlib.org/lapack/lug/node25.html` to find the name of driver routine.
2. For example, If you want to compute the generalized singular values and vectors of a matrix A, you will find the routine name as `DGGSVD` in `https://www.netlib.org/lapack/lug/node36.html`.
3. Find the definition part of `DGGSVD` by download LAPACK and extract to find `dggsvd.f` or you can find on the internet `https://www.netlib.org/lapack/explore-html/dd/db4/dggsvd_8f.html`.
4. Look for `Rggsvd` prototype definition of MPLAPACK and change the variables appropriately to fit to MPLAPACK.

7 Quality assurance of MPBLAS and MPLAPACK

7.1 Testing MPBLAS routines

We performed quality assurance of MPBLAS as follows.

1. We believe no bugs in the reference BLAS [2].
2. We input random values of binary64 for matrices, vectors, and scalars to BLAS and MPFR version of MPBLAS for various sizes of matrices, vector, and leading dimensions (It means MPBLAS and BLAS may fail with error codes) to a routine.
3. If the difference between the two results is within tolerance, we regard the quality as assured for the MPFR version of MPBLAS the routine.
4. We also check the error codes by `xerbla` and `Mxerbla` and always these error codes are the same for the routine.
5. Do the same checks for all routines in Table 3.
6. Then, we input random values of a multiple precision type (e.g., `_Float128`) for matrices, vectors, and scalars to this version of MPBLAS and MPFR version MPBLAS for various sizes of the matrix, vector, and leading dimensions.
7. If the difference between two results is within tolerance, we regard the quality as assured for the multiple-precision type version of MPBLAS.

Since MPBLAS routines only consist of algebraic operations, this method ensures that all routines are correctly implemented.

7.2 Testing MPLAPACK routines

Until version 0.9.4, we did the same method for quality assurance of MPLAPACK. From version 1.0.0, we ported `lapack-3.9.1/TESTING/LIN` and `lapack-3.9.1/TESTING/EIG` for all precisions using FABLE. The original test programs use the `FORMAT` statement extensively. Interestingly, FABLE and FABLE's Fortran EMulator library translate these `FORMAT` statements almost perfectly except for printing out multiple-precision floating-point numbers. Thus our modifications to the testing routines were minimal. Furthermore, it took several months to pass all the tests for the `MPFR` version.

First, we review how to test LAPACK linear routines:

```
$ cd /home/docker/mplapack/external/lapack/work/internal/lapack-3.9.1/TESTING/LIN/
$ ./xlintstd < ../dtest.in
 Tests of the DOUBLE PRECISION LAPACK routines
 LAPACK VERSION 3.9.1

 The following parameter values will be used:
    M   :       0     1     2     3     5    10    50
    N   :       0     1     2     3     5    10    50
    NRHS:       1     2    15
    NB  :       1     3     3     3    20
    NX  :       1     0     5     9     1
    RANK:      30    50    90

 Routines pass computational tests if test ratio is less than   30.00

 Relative machine underflow is taken to be    0.222507-307
 Relative machine overflow  is taken to be    0.179769+309
 Relative machine precision is taken to be    0.111022D-15


 DGE routines passed the tests of the error exits

 All tests for DGE routines passed the threshold (   3653 tests run)

 DGE drivers passed the tests of the error exits

 All tests for DGE drivers  passed the threshold (   5748 tests run)

 DGB routines passed the tests of the error exits
...
```

We omit the details of how we test the linear routines, but `./xlintstd` tests all the linear routines used in LAPACK. Details can be found in the literature [30].

Corresponding ported testing programs for linear routines `/home/docker/mplapack/mplapack/test/lin` are following:

- `mpfr/xlintstC_mpfr`
- `mpfr/xlintstR_mpfr`
- `double/xlintstC_double`
- `double/xlintstR_double`
- `_Float64x/xlintstC__Float64x`
- `_Float64x/xlintstR__Float64x`
- `_Float128/xlintstC__Float128`

- _Float128/xlintstR__Float128
- dd/xlintstC_dd
- dd/xlintstR_dd
- gmp/xlintstC_gmp
- gmp/xlintstR_gmp
- qd/xlintstC_qd
- qd/xlintstR_qd

where we add suffix "R" + "mplib" or "C" + "mplib" for `xlintst`, respectively.

We test MPLAPACK routines for MPFR as follows:

```
$ cd /home/docker/mplapack/mplapack/test/lin/mpfr
$ ./xlintstR_mpfr < ../Rtest.in
 Tests of the Multiple precision version of LAPACK MPLAPACK VERSION 1.0.0
 Based on the original LAPACK VERSION 3.9.1

The following parameter values will be used:
        M  :     0     1     2     3     5    10    50
        N  :     0     1     2     3     5    10    50
     NRHS  :     1     2    15
       NB  :     1     3     3     3    20
       NX  :     1     0     5     9     1
     RANK  :    30    50    90

 Routines pass computational tests if test ratio is less than   30.00

 Relative machine underflow is taken to be : +9.5302596195518043e-323228497
 Relative machine overflow  is taken to be : +2.0985787164673877e+323228496
 Relative machine precision is taken to be : +7.4583407312002067e-155
 RGE routines passed the tests of the error exits

 All tests for RGE routines passed the threshold (   3653 tests run)
 RGE drivers passed the tests of the error exits

 All tests for RGE drivers  passed the threshold (   5748 tests run)
 RGB routines passed the tests of the error exits

 All tests for RGB routines passed the threshold (  28938 tests run)
 RGB drivers passed the tests of the error exits

 All tests for RGB drivers  passed the threshold (  36567 tests run)
 RGT routines passed the tests of the error exits
```

```
 All tests for RGT routines passed the threshold (   2694 tests run)
 RGT drivers passed the tests of the error exits

 All tests for RGT drivers  passed the threshold (   2033 tests run)
 RPO routines passed the tests of the error exits

 All tests for RPO routines passed the threshold (   1628 tests run)
 RPO drivers passed the tests of the error exits

 All tests for RPO drivers  passed the threshold (   1910 tests run)
 RPS routines passed the tests of the error exits

 All tests for RPS routines passed the threshold (    150 tests run)
 RPP routines passed the tests of the error exits
...
```

The file `Rtest.in` is same as `dtest.in` of LAPACK, except for the first comment line. For other precisions, we similarly performed tests. We verified that all tests passed for all precisions. Currently, complex versions do not pass for driver routines. Nevertheless, users can test MPLAPACK complex linear routine using `Ctest.in` by modifying the 15-th line from "T" to "F" so that not to test driver routines, many tests pass. Thus, users can use complex routines with caution.

Next, we review how to test LAPACK EIG as follows:

```
$ cd /home/docker/mplapack/external/lapack/work/internal/lapack-3.9.1/TESTING/EIG/
$  ./xeigtstd < ../nep.in
 Tests of the Nonsymmetric Eigenvalue Problem routines

 LAPACK VERSION 3.9.1

 The following parameter values will be used:
    M:         0     1     2     3     5    10    16
    N:         0     1     2     3     5    10    16
    NB:        1     3     3     3    20
    NBMIN:     2     2     2     2     2
    NX:        1     0     5     9     1
    INMIN:    11    12    11    15    11
    INWIN:     2     3     5     3     2
    INIBL:     0     5     7     3   200
    ISHFTS:    1     2     4     2     1
    IACC22:    0     1     2     0     1

 Relative machine underflow is taken to be    0.222507-307
 Relative machine overflow  is taken to be    0.179769+309
 Relative machine precision is taken to be    0.111022D-15

 Routines pass computational tests if test ratio is less than   20.00
```

```
 DHS routines passed the tests of the error exits ( 66 tests done)
...
```

There are 20 input files for testing eigenvalue problem, singular value problems, least square fitting problems, other related routines. For MPLAPCK, we use following input files:

```
Rbak.in Rbal.in Rbb.in Rec.in Red.in Rgbak.in
Rgbal.in Rgd.in Rgg.in Rsb.in Rsg.in csd.in glm.in
gqr.in gsv.in lse.in nep.in se2.in sep.in svd.in
```

We modified original `Rbak.in`, `Rbak.in`, `Rbal.in`, `Rbb.in`, `Rec.in Red.in`, `Rgbak.in`, `Rgbal.in`, `Rgd.in`, `Rgg.in`, `Rsb.in` and `Rsg.in` so that testing program can process the data correctly.

Corresponding ported testing programs for eigenvalues and sigular value problme solver routines are in `/home/docker/mplapack/mplapack/test/eig`, and we list them as follows:

- `mpfr/xeigtstC_mpfr`
- `mpfr/xeigtstR_mpfr`
- `_Float128/xeigtstC__Float128`
- `_Float128/xeigtstR__Float128`
- `_Float64x/xeigtstC__Float64x`
- `_Float64x/xeigtstR__Float64x`
- `dd/xeigtstC_dd`
- `dd/xeigtstR_dd`
- `double/xeigtstC_double`
- `double/xeigtstR_double`
- `gmp/xeigtstC_gmp`
- `gmp/xeigtstR_gmp`
- `qd/xeigtstC_qd`
- `qd/xeigtstR_qd`

where we add suffix "R" + "mplib" or "C" + "mplib" for `xeigtst`, respectively.

We test MPLAPACK routines for MPFR as follows:

```
$ cd /home/docker/mplapack/mplapack/test/eig/mpfr
$ ./xeigtstR_mpfr < ../svd.in
 Tests of the Singular Value Decomposition routines
 Tests of the Multiple precision version of LAPACK MPLAPACK VERSION 1.0.0
```

```
 Based on original LAPACK VERSION 3.9.1

The following parameter values will be used:
    M:         0     0     0     1     1     1     2     2     3     3
               3    10    10    16    16    30    30    40    40
    N:         0     1     3     0     1     2     0     1     0     1
               3    10    16    10    16    30    40    30    40
    NB:        1     3     3     3    20
    NBMIN:     2     2     2     2     2
    NX:        1     0     5     9     1
    NS:        2     0     2     2     2

 Relative machine underflow is taken to be+9.5302596195518043e-323228497
 Relative machine overflow  is taken to be+2.0985787164673877e+323228496
 Relative machine precision is taken to be+7.4583407312002067e-155

 Routines pass computational tests if test ratio is less than+5.0000000000000000e+01

 DBD routines passed the tests of the error exits ( 55 tests done)
 Rgesvd passed the tests of the error exits (  8 tests done)
 Rgesdd passed the tests of the error exits (  6 tests done)
 Rgejsv passed the tests of the error exits ( 11 tests done)
 Rgesvdx passed the tests of the error exits ( 12 tests done)
 Rgesvdq passed the tests of the error exits ( 11 tests done)


 SVD:  NB =   1, NBMIN =   2, NX =   1, NRHS =   2

 All tests for DBD routines passed the threshold (  10260 tests run)
....
```

We performed tests for all the 20 testing files for all real precision versions. All the tests passed for `mpfr/xeigtstR_mpfr`, `dd/xeigtstR_dd`, `double/xeigtstR_double`, `_Float128/xeigtstR__Float128` and `_Float64x/xeigtstR__Float64x`. Some failures are using `gmp/xeigtstR_gmp`, and the errors are related to precision loss in arithmetic; we should raise error threshold by a factor of 10 or 100, and we do not consider such precision loesses are serious, as we can raise precision at runtime. We do not perform test for `csd.in`, because `GMP` does not have trigonometric functions. MPLAPACK stops when users try to use cosine sine decomposition. We do not fix these failures and planning to drop the GMP version and integrate it into the `MPFR` version in version 3.0.0. Besides, some failures are using `qd/xeigtstR_qd`, and the errors are related to underflow. This will be fixed in the next version. Currently, complex versions are not ready for use. The LAPACK routines passes the tests, but driver routines are broken. We will fix in version 2.0.

In summary, MPLAPACK routines for all real precisions are reliable.

8 Fortran90 to C++ Conversion

Until MPLAPACK 0.9.3, we used f2c to convert FORTRAN77 routines to C (until version 3.1.1, LAPACK only required the FORTRAN77 compiler). Unfortunately, the resultant source code was tough to read by a human; we replaced many tokens heavily using `sed`, and finally converted them to user-friendly, easy-to-read C++ programs by hand.

We simply use REAL, COMPLEX, INTEGER, LOGICAL, and BOOL to support all floating-point types. These types are abstract, and using typedef to specify the actual floating-point type (e.g., `typedef mpf_class REAL` for GMP. See `mplapack/include/mplapack.h` for details).

The main differences between Fortran and C++ are: (i) array index starts from one, (ii) loop index may be incremented or decremented depending on its context, (iii) treatment of two-dimension arrays, (iv) when calling subroutines, the values are always referenced by FORTRAN, but we can choose call by value or call by reference in C++. For (i), we use the same array index and decrements the index when accessing the array. (ii) we judge the direction of loop by hand, (iii) we expand two-dimension array using the leading dimension; we translate `a(i,j)` to `a[(i-1) + (j-1) * lda]`, and (iv) we changed call by value when possible.

In MPLAPACK 1.0.0, we used FABLE [31] to convert all BLAS and LAPACK routines. In the original paper, they could translate `dsyev.f` without modifications. However, the resultant source code also depends on Fortran Emulation Module (FEM), which is unnecessary for the MPBLAS and MPLAPACK parts, and the treatment of the array is a bit unnatural. To make the resultant C++ code more natural and readable, we further patched `cout.py` and passed through sed. Then, we modified the source codes by hand. For MPBLAS, conversion is almost automatic. For MPLAPACK, there are many hand-corrected minor syntax errors, interpretation of numbers, casts, and other corrections, but we can compile many of them without modifications.

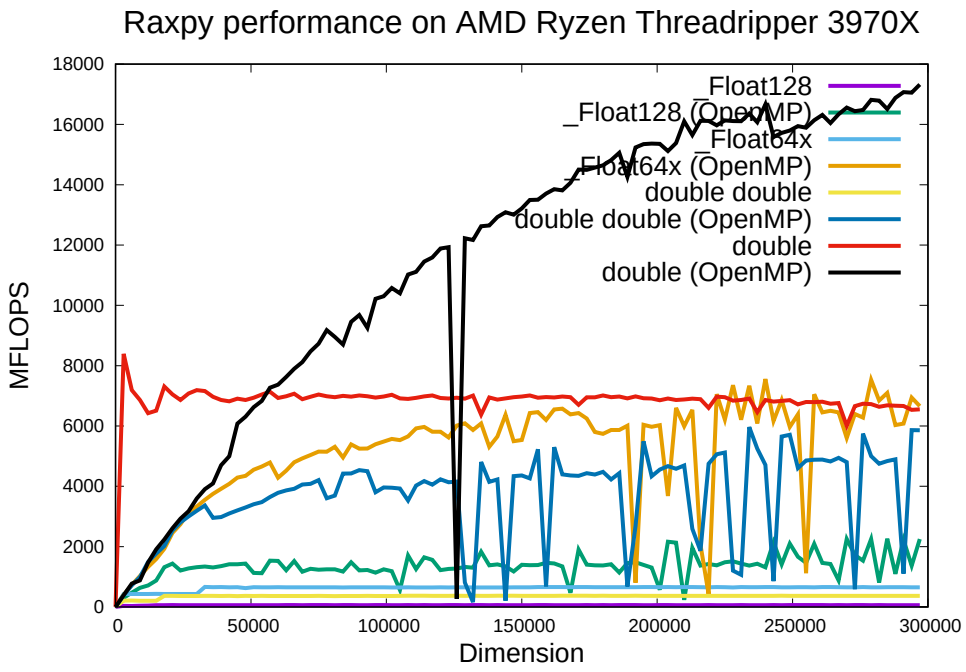
More important part is translation of `TESTING/LIN` and `TESTING/EIG` from Fortran90 to C++. Thanks to FEM, the written statement of Fortran can be used directly without modifications. According to ChangeLog, it took approximately three weeks to just compile programs in `TESTING/LIN/xlint{C,R}_mpfr` and `TESTING/EIG/xeigtst{C,R}_mpfr` for MPFR, although through check the results are required, and it took four months for real version and still ongoing for complex version.

9 Benchmarks

We have provided reference implementation as well as a simple OpenMP accelerated version for MPBLAS. It is shown in Table 4. In Figure 1, we show the result of `Raxpy` performance for `_Float128`, `_Float64x`, `double` and `double-double`, and Figure 2 shows the `Raxpy` performance for `GMP`, `MPFR`, and `quad-double`, with and without optimization. We used CPU for AMD Ryzen 3970X, 256GB of DDR4 3200MHz memory (ECC Unbuffered), and took benchmark inside Docker environment. GCC version was 9.3.0.

For these benchmarks, We performed `Raxpy` calculations for each precision ten times and took the average for OpenMP implementations, and three times for the reference implementations. The peak performance of the reference `_Float128` version was approximately 60 MFlops, and 2.2 GFlops for the OpenMP version. The peak performance of the reference `_Float64x` version was approximately 660 MFlops, and 7.0 GFlops for the OpenMP version. The peak performance of the reference

Figure 1: Raxpy performance on AMD Ryzen 3970X for _Float128, _Float64x, double-double, double and (simple) OpenMP accelrated results.



double-double version was approximately 365 MFlops, and 6.0 GFlops for the OpenMP version. The double version is the fastest, and the peak was 8 GFlops for reference , and 17 GFlops for the OpenMP version. These results are not competitive to OpenBLAS or Intel MKL. The peak performance of the reference GMP version was 20 MFlops (512bit accuracy; default), and 750 MFlops OpenMP enabled. The peak performance of the reference quad-double version was 32 MFlops, and 1.3 GFlops OpenMP enabled. The peak performance of reference MPFR version was 10 MFlops (512bit accuracy; default), and 400 MFlops OpenMP enabled. Thus, the MPFR version is twice as slower than the GMP, this is an unexpected result. One possible reason might be the C++ binding for MPFR do unnecessary copies whereas GMP does not, because GMP C++ binding tries to reduce copies multiple-precision numbers using template programming.

In any case, enabling OpenMP will significantly increase the performance of Raxpy. However, the performances are not stable; as we can see from the figures, there is some considerable performance degradation especially using quad-double, GMP and MPFR.

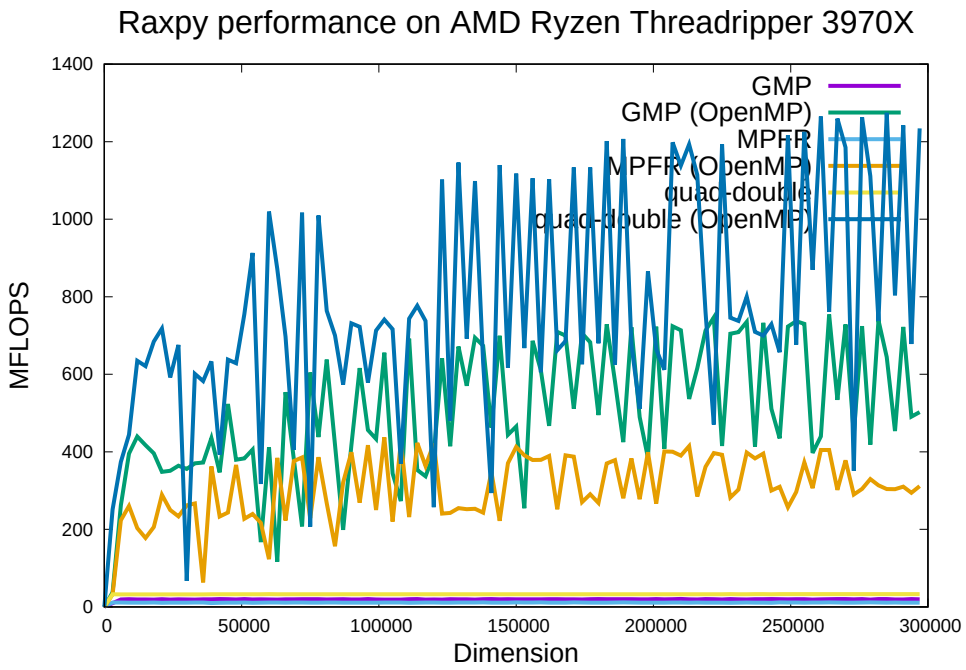
Users can take performance benchmark on your machine:

```
$ cd /home/docker/mplapack/benchmark/mpblas
$ bash -x go.Raxpy.sh
```

and resultant files are Raxpy1.eps and Raxpy2.eps. They correspond to Figure 1 and Figure 2.

In Figure 3, we show the result of Rgemm performance for _Float128, _Float64x, double and double-double, and Figure 4 shows the Rgemm performance for GMP, MPFR, and quad-double, with and without optimization. We used CPU for AMD Ryzen 3970X, 256GB of DDR4 3200MHz memory (ECC is not enabled), and took benchmark inside Docker environment. GCC version was 9.3.0.

Figure 2: Raxpy performance on AMD Ryzen 3970X for `quad-double`, `GMP` and `MPFR`, and (simple) OpenMP accelrated results.



For these benchmarks, We performed `Rgemm` calculations for each precision five times and took the average for OpenMP implementations and three times for the reference implementations. We used square matrices for the benchmarks. The peak performance of the reference `_Float128` version was approximately 64 MFlops, and 2.4 GFlops for the OpenMP version.

The peak performance of the reference `_Float64x` version was approximately 665 MFlops, and 18.8 GFlops for the OpenMP version. The peak performance of the reference `double-double` version was approximately 365 MFlops, and 11.2 GFlops for the OpenMP version. The double version is the fastest, and the peak was 6.8 GFlops for reference and 15 GFlops for the OpenMP version. These results are not competitive to OpenBLAS or Intel MKL. `DGEMM` performance by OpenBLAS was 1.46 TBFlops with Ryzen Threadripper 3970X (This performance is almost 77% of theoretical peak performance at rated core clock; 1.89 TFlops= 3.7[GHz] $\times$ 16 [AVX2] $\times$ 32 (cores)).

The peak performance of the reference `GMP` version was 19 MFlops (512bit accuracy; default), and 693 MFlops OpenMP enabled. The peak performance of the reference `quad-double` version was 33 MFlops, and 1.3 GFlops OpenMP enabled. The peak performance of reference `MPFR` version was 11 MFlops (512bit accuracy; default), and 390 MFlops OpenMP enabled. Thus, the `MPFR` version is twice as slower than the `GMP`.

In any case, enabling OpenMP will significantly increase the performance of `Rgemm`. Moreover, the performances are stabler than `Raxpy` results.

Users can take performance benchmark on your machine:

```
$ cd /home/docker/mplapack/benchmark/mpblas
$ bash -x go.Rgemm.sh
```

Figure 3: Rgemm performance on AMD Ryzen 3970X for `_Float128`, `_Float64x`, `double-double`, `double` and (simple) OpenMP accelrated results.

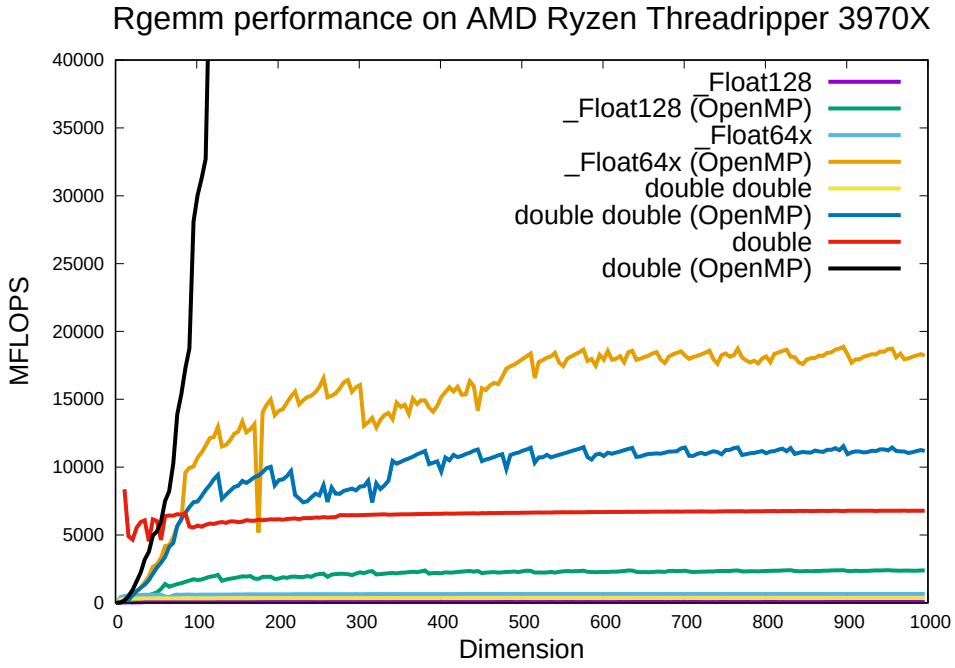
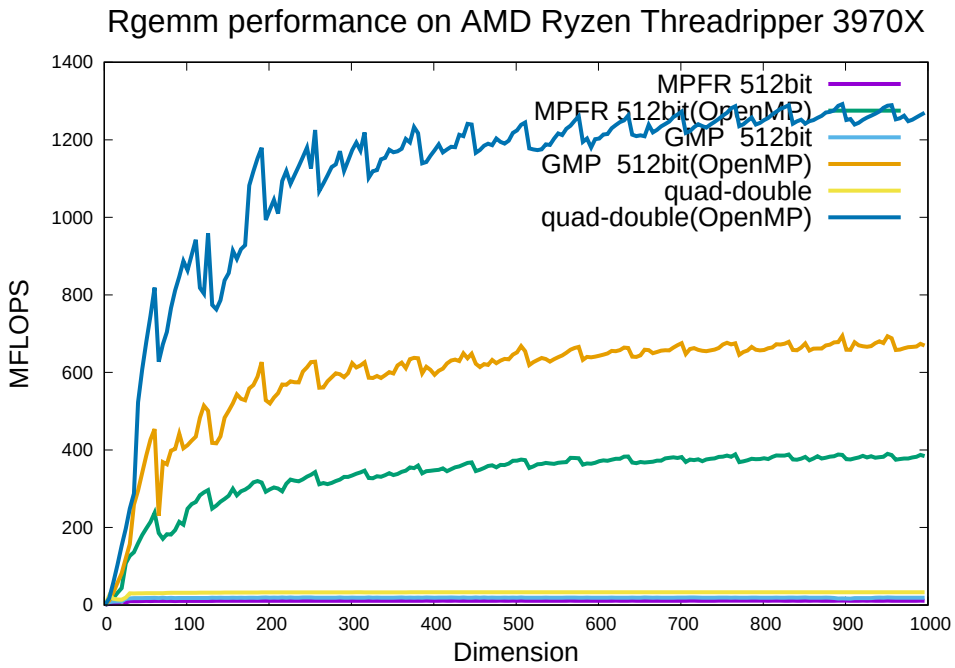


Figure 4: Rgemm performance on AMD Ryzen 3970X for `quad-double`, `GMP` and `MPFR`, and (simple) OpenMP accelrated results.



and resultant files are `Rgemm1.eps` and `Rgemm2.eps`. They correspond to Figure 3 and Figure 4.

10 History

M.N started developing MPLAPACK as a by-product of an arbitrary accurate semidefinite programming solver, SDPA-GMP, around 2006 [14]. The first MPLAPACK (formerly MPACK) version, 0.0.1, was released in 2008-7-15. In 2009-2-5, we released SDPA-GMP 7.1.2, and MPLAPACK supports SDPA-GMP, SDPA-QD, and SDPA-DD [15, 17, 32, 33]. In 2010-01-13, we supported Windows via mingw32. In 2010-5-21, we supported MPFR. In 2012-10-13, we released a fast implementation of a double-double version of Rgemm for NVIDIA C2050 [34, 35]. In 2017-3-29, we moved the web site from `http://mplapack.sourceforge.net/` to `https://github.com/nakatamaho/mplapack/`. In 2021-4-1, we renamed our project to MPLAPACK. In 2021-4-11, we supported AArch64 and released 0.9.4. In 2021-10-1, we released 1.0.0.

11 Related works

We can relate our work in two main directions. (i) acceleration of multiple-precision extension to BLAS (especially on GPU) and (ii) development of a multiple-precision version of linear algebra packages.

For (i), In 2009, Mukunoki*et al.* [36] implemented double-double version of matrix-matrix multiplication kernels for NVIDIA Tesla C1060 and evaluated the performance. In 2010, Mukunoki *et al.* [37] the quadruple precision Basic Linear Algebra Subprograms (BLAS) functions, AXPY, GEMV, and GEMM, on graphics processing units (GPUs), and evaluated their performance. On an NVIDIA Tesla C1060, their BLAS functions are approximately 30 times faster than the existing quadruple precision BLAS on an Intel Core i7 920.

In 2011, Nakasato [38] implemented optimized dense matrix multiplication kernels for AMD Cypress GPU for various precisions. Their result includes double-double precision, and attained peak performance of 30 GFlops.

In 2012, Nakata *et al.* [35] implemented double-double version of `Rgemm` on NVIDIA C2050. In their implementation is complete `Rgemm` implementation; thus, their implementation can be used for real applications; it can handle all the sizes in matrices, and supports transpose of each matrix. They attained 16.1GFlops with CPU-GPU transfer included, and 26.4GFlops (25.7GFlops with CPU-GPU transfer included. Moreover, they applied to semidefinite programming solver and attained ten times acceleration compared to CPU-only implementation.

In 2012, Yamada *et al.* developed QPBLAS packages for CPUs [39], as well as in the QPBLAS-GPU package for GPUs in 2013 [40]. These correspond to the complete set of double-double version parts of MPBLAS, accelerating CPU and GPUs. Note that memory alignment is different from MPBLAS and written in Fortran 90.

In 2014 and 2015, Kouya [41, 42] implemented optimized matrix-matrix multiplications using Strassen and Winograd algorithms using MPFR, double-double, and quad-double precisions and applied them to LU factorizations. Strassen and Winograd versions are twice as fast as double-double and quad-double precisions running one core using a simple blocking algorithm. Furthermore, there is no significant performance difference between the Strassen version and Winograd. However, when we used MPFR, the Winograd version was always faster. He also found a significant loss of accuracy when performing LU factorization using Strassen and Winograd versions.

In 2016, Joldes *et al.* implemented CAMPARY: Cuda Multiple Precision Arithmetic Library and applied it to semi-definite programming [43]. However, it was a prototype implementation and somewhat slower than our CPU implementation of ours [15]. In 2017, Joldes *et al.* implemented an improved version of CAMPARY [44]. The significant result of this paper is that they improved `Rgemm` performance on GPU for various multiple-precision versions; 1.6GFlops for triple-double precision, 976MFlops for quadruple-precision, 660MFlops for quintuple-double, 453MFlops for sextuple-double, 200MFlops for octuple-double.

In 2019, Hishinuma *et al.* implimented `Rgemm` kernels for double-double precision on MIMD type acclearlator PEZY-SC2 [45]. The performance of their implementation of `Rgemm` the PEZY-SC2 attained 75% of the peak performance. This value is 20 times faster than an Intel Xeon E5-2618L v3, even including the communication time between the host CPU and the PEZY-SC2.

In 2020, Isopov *et al.* [46] took a benchmark for several level-1 multi-precision routines for MPFR, ARPREC, MPDECIMAL, MPACK, GARPREC, CUMP,, and MPRES-BLAS. MPRES-BLAS is an ongoing effort to develop a multiple-precision version of BLAS like MPBLAS on GPUS The[47]. MPRES-BLAS defines a new arbitrary precision type suited for GPUs and fastest among CAMPARY [43], CUMP [48] and GARPREC [49] when the fractions range from 424bits to 848 bits.

In 2021, Kouya [50] acceralated `dd_real`, `qd_real` and triple-double precision version of `Rgemm` with AVX2 using Strassen algorithm [51].

For (ii), Kouya has long developed C libraries for multiple-precision calculation using GMP and MPFR, including matrix operations [52]. He calls them as BNCpack and MPIBNCPack. At least, we can back to 2003 to find there was MPIBNCpack version 0.1 [53]. In 2011, version 0.7 had released, and it can solve linear equations and eigenvalue problems, and many multiple-precision functions not directly related to linear algebra. From version 0.8 (2013-03-11), BNCpack and MPIBNCpack are integrated [52, 54].

Saito developed ZKCM [55], a C++ library for multi-precision matrix computation for quantum computer simulation. He implemented matrix inversion, singular-value decomposition of a general matrix, the diagonalization of a Hermitian matrix, and other operations.

Arb is a C library for arbitrary-precision ball arithmetic developed by Johansson [56]. It uses ball arithmetic to track numerical errors automatically. It can perform and solve a wide range of matrix operations, e.g., LU factorization, inversion, eigenvalue problems, but it is not an extension of LAPACK to higher precision since employed arithmetic is different.

Multiprecision Computing Toolbox by Advanpix for MATLAB is a toolbox of MATLAB, can solve many arbitrary precision matrix operations like diagonalization of real and complex non-symmetric matrices, singular value decomposition with outstanding performance. They seem to employ a similar approach to ours to convert LAPACK to multiple precision versions, but details are not open and do not replace LAPACK [57]. Symbolic Math Toolbox (SMT) of Matlab also provides such functionalities [58].

RalphAS has been developing GenericSchur.jl [59], and it calculates Schur decomposition of matrices with generic floating-point element types in Julia. However, it is not a replacement for the LAPACK library.

Johansson *et al.* also has been developing mpmath [60] can handle linear algebra (linear system solving, LU factorization, matrix inverse, matrix norms, matrix exponentials/logarithms/square roots, eigenvalues, singular values, QR factorization), written in python, and it is not a replacement of LAPACK.

Mathematica [61] has multiple-precision versions of the eigenvalue problem of non-symmetric matrices, singular value decomposition problems, and other solvers.

12 Future plans

In version 2.0.0, we finish RFP, complex version, and mixed-precision version, and in version 3.0.0, we plan to implement optimized MPFR C++ wrapper and drop GMP version.

Acknowledgement

First, We are grateful to the LAPACK team for making an excellent library. This work was supported by the Special Postdoctoral Researchers' Program of RIKEN (2008, 2009), Grant-in-Aid for Scientific Research (B) 21300017 from the Japan Society for the Promotion of Science (2009, 2010, 2011), Microsoft Research CORE6 (2010) and the Japan Society for the Promotion of Science (JSPS KAKENHI Grant no. 18H03206). The author would like to thank Dr. Imamura Toshiyuki. Dr. Nakasato Naohito, Dr. Fujisawa Katsuki, Dr. Kouya Tomonori, Dr. Takahashi Daisuke, Dr. Goto Kazushige, Dr. Himeno Ryutaro, Dr. Hishimuna Toshiaki, Dr. Katagiri Takahiro, Dr. Ogita Takeshi, Dr. Kashiwagi Masahide, Dr. Yuasa Fukuko, Dr. Ishikawa Tadashi, Dr. Geshi Masaaki and Mr. Minato Yuichiro for warm encouragement.

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[2] An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135151, June 2002.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[4] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[5] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), May 2008.

[6] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691, 2012.

[7] Jack Dongarra, Vladimir Getov, and Kevin Walsh. The 30th anniversary of the supercomputing conference: Bringing the future closer–supercomputing history and the immortality of now. *Computer*, 51(10):74–85, 2018.

[8] NVIDIA. Basic linear algebra on nvidia gpus, 2021. `https://developer.nvidia.com/cublas`.

[9] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[10] D.H. Bailey, R. Barrio, and J.M. Borwein. High-precision computation: Mathematical physics and dynamics. *Applied Mathematics and Computation*, 218(20):10106–10121, 2012.

[11] David H. Bailey and Jonathan M. Borwein. High-precision arithmetic in mathematical physics. *Mathematics*, 3(2):337–367, 2015.

[12] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[13] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Rev.*, 38(1):4995, March 1996.

[14] Maho Nakata, Bastiaan J. Braams, Katsuki Fujisawa, Mituhiro Fukuda, Jerome K. Percus, Makoto Yamashita, and Zhengji Zhao. Variational calculation of second-order reduced density matrices by strong n-representability conditions and an accurate semidefinite programming solver. *The Journal of Chemical Physics*, 128(16):164113, 2008.

[15] Maho Nakata. A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: Sdpa-gmp, -qd and -dd. In *2010 IEEE International Symposium on Computer-Aided Control System Design*, pages 29–34, 2010.

[16] Makoto Yamashita, Katsuki Fujisawa, Mituhiro Fukuda, Kazuhiro Kobayashi, Kazuhide Nakata, and Maho Nakata. *Latest Developments in the Family for Solving Large-Scale SDPs*, pages 687–713. Springer US, Boston, MA, 2012.

[17] Nakata, Maho. SDPA-GMP, retrieved September 23, 2021. `https://github.com/nakatamaho/sdpa-gmp/`.

[18] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[19] Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985.

[20] *ISO/IEC TS 18661-3:2015 Information Technology - Programming languages, their environments, and system software interfaces - Floating-point extensions for C - Part 3: Interchange and extended types*. the International Organization for Standardization, Chemin de Blandonnet 8 CP 401 1214 Vernier, Geneva Switzerland, 2015.

[21] Cedric Lichtenau, Steven Carlough, and Silvia Melitta Mueller. Quad precision floating point on the ibm z13. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 87–94, 2016.

[22] Xiaoye S. Li Yozo Hida and David H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. In *Technical Report LBNL-46996*. Lawrence Berkley National Laboratory, 2000.

[23] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[24] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971/72.

[25] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. `http://gmplib.org/`.

[26] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13es, June 2007.

[27] Andreas Enge, Mickaël Gastineau, Philippe Théveny, and Paul Zimmermann. *mpc — A library for multiprecision complex arithmetic with exact rounding*. INRIA, 1.1.0 edition, January 2018. `http://mpc.multiprecision.org/`.

[28] R. T. Gregory and D. Karney. A collection of matrices for testing computational algorithms. 1969.

[29] Singular value decomposition, retrieved September 24, 2021. `https://en.wikipedia.org/wiki/Singular_value_decomposition`.

[30] Susan Blackford and Jack Dongarra. Installation guide for LAPACK. LAPACK Working Note 41, March 1992. UT-CS-92-151, March, 1992.

[31] Ralf W. Grosse-Kunstleve, Thomas C. Terwilliger, Nicholas K. Sauter, and Paul D. Adams. Automatic fortran to++ conversion with. *Source Code for Biology and Medicine*, 7:5, 2012. `https://doi.org/10.1186/1751-0473-7-5`.

[32] Nakata, Maho. SDPA-QD, retrieved September 23, 2021. `https://github.com/nakatamaho/sdpa-qd/`.

[33] Nakata, Maho. SDPA-DD, retrieved September 23, 2021. `https://github.com/nakatamaho/sdpa-dd/`.

[34] Maho Nakata, Yasuyoshi Takao, Shigeho Noda, and Ryutaro Himeno. A fast implementation of matrix-matrix product in double-double precision on nvidia c2050 and application to semidefinite programming. In *2012 Third International Conference on Networking and Computing*, pages 68–75, 2012.

[35] Maho Nakata. Poster: Mpack 0.7.0: Multiple precision version of blas and lapack. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1353–1353, 2012.

[36] Mukunoki Daichi and Takahashi Daisuke. Implementation and evaluation of quadruple precision blas on gpu (*in japanese*). In *IPSJ SIG Technical Report*, volume 137, pages 1–6, 2009.

[37] Daichi Mukunoki and Daisuke Takahashi. Implementation and evaluation of quadruple precision blas functions on gpus. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, pages 249–259, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[38] N. Nakasato. A fast gemm implementation on the cypress gpu. *SIGMETRICS Perform. Evaluation Rev.*, 38:50–55, 2011.

[39] Susumu Yamada, Takuya Ina, Narimasa Sasa, Yasuhiro Idomura, Masahiko Machida, and Toshiyuki Imamura. Quadruple-precision blas using bailey's arithmetic with fma instruction: its performance and applications. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1418–1425, 2017.

[40] Japan Atomic Energy Agency. *Quadruple Precision BLAS Routines for GPU QPBLAS-GPU Ver.1.0 User's Manual*, July 2013.

[41] Tomonori Kouya. Accelerated multiple precision matrix multiplication using strassen's algorithm and winograd's variant. *JSIAM Letters*, 6:81–84, 2014.

[42] Tomonori Kouya. Performance evaluation of multiple precision matrix multiplications using parallelized strassen and winograd algorithms. *JSIAM Letters*, 8:21–24, 2016.

[43] Mioara Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. Campary: Cuda multiple precision arithmetic library and applications. In Gert-Martin Greuel, Thorsten Koch, Peter Paule, and Andrew Sommese, editors, *Mathematical Software – ICMS 2016*, pages 232–240, Cham, 2016. Springer International Publishing.

[44] Mioara Joldes, Jean-Michel Muller, and Valentina Popescu. Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 27–34, 2017.

[45] Toshiaki Hishinuma and Maho Nakata. pzqd: Pezy-sc2 acceleration of double-double precision arithmetic library for high-precision blas. In *International Conference on Computational & Experimental Engineering and Sciences*, pages 717–736. Springer, 2019.

[46] Konstantin Isupov. Performance data of multiple-precision scalar and vector blas operations on cpu and gpu. *Data in Brief*, 30:105506, 2020.

[47] Konstantin Isupov and Vladimir Knyazkov. Multiple-precision blas library for graphics processing units. In Vladimir Voevodin and Sergey Sobolev, editors, *Supercomputing*, pages 37–49, Cham, 2020. Springer International Publishing.

[48] Takato Nakayama and D. Takahashi. Implementation of multiple-precision floating-point arithmetic library for gpu computing. In *Proc. 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pages 343–349, 2011.

[49] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[50] Tomonori Kouya. Acceleration of multiple precision matrix multiplication based on multicomponent floating-point arithmetic using avx2. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Chiara Garau, Ivan Blečić, David Taniar, Bernady O. Apduhan, Ana Maria A. C. Rocha, Eufemia Tarantino, and Carmelo Maria Torre, editors, *Computational Science and Its Applications – ICCSA 2021*, pages 202–217, Cham, 2021. Springer International Publishing.

[51] V. STRASSEN. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[52] Tomonori Kouya. *BNCpack 0.7*, September 2011. http://na-inet.jp/na/bnc/.

[53] Tomonori Kouya. *MPIBNCpack 0.1*, September 2003. https://na-inet.jp/na/bnc/mpibncpack.pdf.

[54] Tomonori Kouya. Tuning technique for multiple precision dense matrix multiplication using prediction of computational time, 2017.

[55] Akira SaiToh. Zkcm: A c++ library for multiprecision matrix computation with applications in quantum information. *Computer Physics Communications*, 184(8):2005–2020, 2013.

[56] Fredrik Johansson. Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic, 2016. https://arblib.org/.

[57] Advanpix. Multiprecision Computing Toolbox for MATLAB, retrieved September 23, 2021. https://www.advanpix.com/.

[58] Inc. The MathWorks. *Symbolic Math Toolbox*. Natick, Massachusetts, United State, 2019.

[59] RalphAS. Schur decomposition of matrices with generic floating-point element types in Julia, retrieved September 23, 2021. https://github.com/RalphAS/GenericSchur.jl.

[60] Fredrik Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*, December 2013. http://mpmath.org/.

[61] Wolfram Research, Inc. Mathematica, Version 12.3.1. Champaign, IL, 2021.