

# BNCmatmul User's Guide

Tomonori Kouya

<https://github.com/tkouya/bncmatmul>

Version 0.21: May 31, 2023

# Contents

Chapter 1	What's BNCmatmul?	1
1.1	Copyright and condition for distribution . . . . .	1
1.2	How to compile BNCmatmul . . . . .	1
1.3	Mathematical notation . . . . .	2
1.4	Why do we require a multiple-precision floating-point arithmetic library? . . . . .	2
1.5	Algorithms and performance of optimized matrix multiplication . . . . .	2
1.6	Software layer of BNCmatmul . . . . .	4
1.7	History of Version and Todo list . . . . .	6
Chapter 2	Basic datatypes and arithmetic	7
2.1	c_dd_qd.h . . . . .	7
2.2	Error-free transformation . . . . .	8
2.3	Double-double precision arithmetic . . . . .	8
2.4	Quadruple-double precision arithmetic . . . . .	9
2.5	Triple-double precision arithmetic . . . . .	10
2.6	rdd.h . . . . .	11
2.7	AVX2 . . . . .	13
Chapter 3	Basic linear computation	17
3.1	Datatypes of D, DD, TD, QD and MPFR vector and matrix . . . . .	17
3.2	Vector arithmetic . . . . .	18
3.3	Matrix arithmetic . . . . .	20
3.4	File I/O with matrix and vector . . . . .	23
3.5	Generating test matrices . . . . .	24
3.6	Simple, block, and Strassen matrix multiplication and related functions . . . . .	25
3.7	Getting relative errors of vector and matrix . . . . .	26
3.8	Ozaki scheme and related functions . . . . .	27
Chapter 4	LU decomposition	29
4.1	List of functions . . . . .	29
Bibliography		31
Index		32

# Chapter 1

## What's BNCmatmul?

BNCmatmul is one of optimized BLAS(Basic Linear Algebra Subprograms) libraries supporting multiple precision arithmetics such as double(binary64), DD(double-double), TD(triple-double), QD(quadruple-double), and arbitrary precision floating point arithmetic of MPFR.

Now that MPLAPACK/MPBLAS, a library for extended multiple precision linear computation, provides the basis for a standard multiple precision numerical computing environment, there is a growing demand for an optimized library that provides faster basic linear computation. Standard optimization methods include the use of SIMD instructions and parallelization with OpenMP, but for multiple precision matrix multiplication, optimization with algorithms such as divide-and-conquer method and Ozaki scheme are also available. We have developed new version of BNCmatmul that can use all of these optimization methods and supports both fixed-precision computation using the multi-component way and arbitrary-precision computation using the multi-digit way. In this paper, we describe its software structure and present the results of performance evaluation of optimized matrix-vector multiplication and matrix multiplication.

We are developing BNCmatmul on the following environment: Xeon and EPYC. On other x86\_64 linux environment with Intel OneAPI or GNU Compiler Collection, our library can probably be available.

EPYC AMD EPYC 7402P 24 cores, Ubuntu 18.04.6 LTS, Intel Compiler version 2021.4.0, MPLAPACK 1.0.1, MPFR 4.1.0

Xeon Intel Xeon W-2295 3.0GHz 18 cores, Ubuntu 20.04.3 LTS, Intel Compiler version 2021.5.0, MPLAPACK 1.0.1, MPFR 4.1.0

### 1.1 Copyright and condition for distribution

BNCmatmul is originated by Tomonori Kouya (<https://na-inet.jp/>), and including this document is distributed under [GPL] version 2 or later.

[GPL]: <https://www.gnu.org/copyleft/gpl.html>

### 1.2 How to compile BNCmatmul

BNCmatmul can be compiled with Intel OneAPI Compiler (ICC) or GNU Compiler Collection (GCC) as follows:

1. Git clone at your home directory from <https://github.com/tkouya/bncmatmul>.
2. Edit bncmatmul.inc to select ICC or GCC for compilation.
3. Run "make all" to build libbncmatmul\*.a and python/libbncmatmul\*.so.
4. Compile some iterative refinement C++ programs in "test" directory with BNCmatmul and MPLAPACK/MPBLAS.

### 1.3 Mathematical notation

Here, we define  $\mathbb{F}_{bS}$  and  $\mathbb{F}_{bL}$  as sets of the  $S$ - and  $L$ -bit mantissas of floating-point numbers, respectively. For instance,  $\mathbb{F}_{b24}$  and  $\mathbb{F}_{b53}$  refer to sets of IEEE754-1985 binary32 and binary64 floating-point numbers, whereas  $\mathbb{F}_{b106}$ ,  $\mathbb{F}_{b159}$ , and  $\mathbb{F}_{b212}$  represent examples of DD, TD, and QD precision floating-point numbers, respectively. Although any mantissa length can be selected in MPFR arithmetic, the set of MPFR numbers is expressed as  $\mathbb{F}_{bM}$ , which is primarily defined as  $M$ -bit using the `mpfr_set_default_prec` function.

Moreover, we use  $(\mathbf{x})_i (= x_i)$  as the  $i$ -th element of the  $n$ -dimensional vector  $\mathbf{x} = [x_i]_{i=1,2,\dots,n} \in \mathbb{R}^n$ , and  $(A)_{ij} (= a_{ij})$  as the  $(i, j)$ -th element of  $A = [a_{ij}]_{i=1,2,\dots,m,j=1,2,\dots,n} \in \mathbb{R}^{m \times n}$ .

### 1.4 Why do we require a multiple-precision floating-point arithmetic library?

The following is a brief explanation regarding the need for multiple-precision calculations with a mantissa exceeding binary64. First, users may seek to use floating-point arithmetic to obtain a value  $F(x)$  from an input value  $x$  represented by a floating-point number. Ultimately,  $F(x)$  must maintain at least  $U$  significant digits. Intuitively, in numerical computations, the initial error due to rounding is assumed to be included in the input value. Suppose the number of mantissa digits for the floating-point arithmetic to be  $L (> U)$ . Then, the number of significant digits of  $F(x)$  is  $U - R$ , where  $R$  denotes the digits lost in the process of computing  $F(x)$ . Furthermore, it is assumed that the algorithm cannot be modified to ensure accuracy with an  $L$ -digit computation.

Unless the calculation algorithm for  $F(x)$  is modified, the error propagation minimally changes irrespective of the number of mantissa digits. Therefore, we can increase the number of mantissa digits, slightly exceeding  $R$  with some slack of  $\alpha$  digits, to  $L + R + \alpha$  digits. If the initial error can be significantly minimized,  $F(x)$  with  $U$  or more significant digits can be subsequently obtained. This process illustrates how multiple-precision calculations are used to ensure accuracy.

In contrast, the multi-fold calculation method proposed by Rump and Ogita [9] [10] prevents an increase in the initial error by suppressing the rounding error generated by arithmetic operations to the lower digits using error-free transformation techniques. The computational manner of the error-free transformation process is almost the same as that of multiple-precision calculations. Moreover, in this process, floating-point arithmetic with  $L$  or more digits is not used. Accordingly, there is no need to perform renormalization procedures. This offers the advantage of reducing computational complexity compared to that of a multi-component approach that uses error-free transformation techniques.

A conceptual diagram of the two aforementioned approaches is presented in Fig. 1.1.

Currently, not all algorithms used in numerical computation can be applied using multi-fold arithmetic, especially with nonlinear calculations. Therefore, it is necessary to employ a combination of multi-fold and multiple-precision calculations, where the basic linear calculation part, such as explicit extrapolation process solving initial value problems of ordinary differential equations, uses multi-fold calculations to reduce computational time.

The Ozaki scheme, a matrix multiplication algorithm that we incorporated into our library, is a technique based on the multi-fold calculation approach. By leveraging the speed of existing binary32 and binary64 xGEMM algorithms, it significantly optimizes multiple-precision matrix multiplication under certain conditions, as revealed in our benchmark and previous studies[7][11]. We are confident that the acceleration of multiple-precision linear computation libraries will remain an important theme in guaranteeing the accuracy of broader numerical computation algorithms, following the development trend of multiple-precision numerical algorithms.

### 1.5 Algorithms and performance of optimized matrix multiplication

We focused on the optimization of multiple-precision matrix multiplication, starting with MPI support for arbitrary-precision numerical calculations[4]. Subsequently, owing to current popularity of multi-core

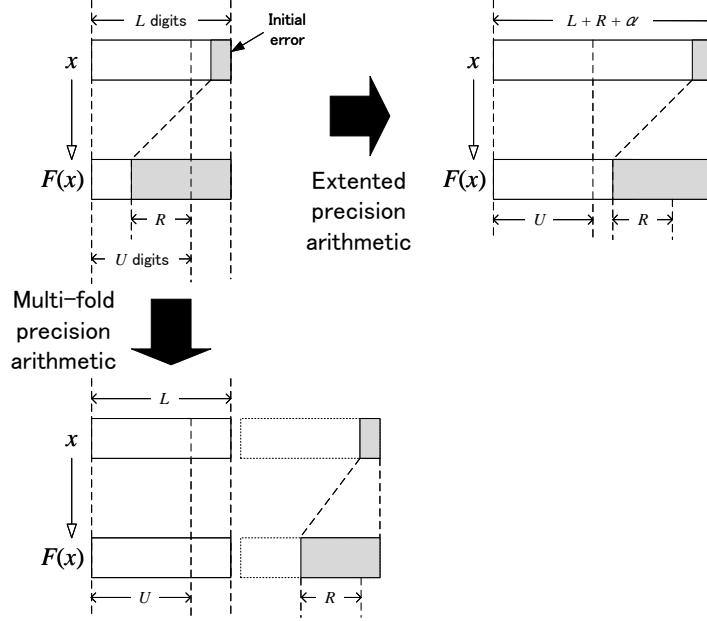


Fig. 1.1 Extended precision and multi-fold precision arithmetics

CPUs, we accelerated multiple-precision matrix multiplication via parallelization support using OpenMP. As far as multiple-precision calculations are concerned, no other such results have been achieved by incorporating divide-and-conquer methods, such as the Strassen and Winograd algorithms. Although limited to multi-component methods, our proposed method is the only approach that supports AVX2 and has been converted to OpenMP for achieving higher speeds. However, the current code does not improve performance beyond eight threads for OpenMP acceleration, and a drastic reformulation is necessary to further improve parallelization performance. Algorithm 1 is the Strassen matrix multiplication.

---

**Algorithm 1** Strassen algorithm for matrix multiplication

---

**Input:**  $A = [A_{ij}]_{i,j=1,2} \in \mathbb{R}^{m \times l}$ ,  $A_{ij} \in \mathbb{R}^{m/2 \times l/2}$ ,  $B = [A_{ij}]_{i,j=1,2} \in \mathbb{R}^{l \times n}$ ,  $B_{ij} \in \mathbb{R}^{l/2 \times n/2}$

**Output:**  $C := \text{Strassen}(A, B) = AB \in \mathbb{R}^{m \times n}$

**if**  $m < m_0$  &&  $n < n_0$  **then**

$C := AB$

**end if**

$P_1 := \text{Strassen}(A_{11} + A_{22}, B_{11} + B_{22})$

$P_2 := \text{Strassen}(A_{21} + A_{22}, B_{11})$

$P_3 := \text{Strassen}(A_{11}, B_{12} - B_{22})$

$P_4 := \text{Strassen}(A_{22}, B_{21} - B_{11})$

$P_5 := \text{Strassen}(A_{11} + A_{12}, B_{22})$

$P_6 := \text{Strassen}(A_{21} - A_{11}, B_{11} + B_{12})$

$P_7 := \text{Strassen}(A_{12} - A_{22}, B_{21} + B_{22})$

$C_{11} := P_1 + P_4 - P_5 + P_7$ ;  $C_{12} := P_3 + P_5$

$C_{21} := P_2 + P_4$ ;  $C_{22} := P_1 + P_3 - P_2 + P_6$

$C := [C_{ij}]_{i,j=1,2}$

---

The usefulness of the Ozaki scheme is also clear at multiple-precision levels owing to the success of Mukunoki et al. in accelerating the float128 precision matrix multiplication[7]. The float128 arithmetic supported by GCC features TD to QD precision performance for addition and multiplication and is likewise expected to be sufficient for this precision range.

The Ozaki scheme is an algorithm that aims to simultaneously accelerate performance and improve accuracy by dividing matrices into more matrices with elements represented by shorter digits. Similarly to the “Split” method used in the error-free transformation technique, this approach leverages on the speed of optimized short-precision matrix multiplication (xGEMM) functions. For a given matrix  $A \in \mathbb{R}^{m \times l}$  and  $B \in \mathbb{R}^{l \times n}$ , to obtain a matrix product  $C := AB \in \mathbb{R}^{m \times n}$  of long  $L$ -bit precision,  $A$  and  $B$  are divided using the Ozaki scheme, where  $D \in \mathbb{N}$  is the maximum number of divisions of short  $S$ -bit precision matrices ( $S \ll L$ ), as depicted in Algorithm 2. The  $S$ -bit arithmetic is used for calculations where no particular description is given, and the  $L$ -bit arithmetic is used only where high-precision operations are required.

---

**Algorithm 2** Ozaki scheme for multiple-precision matrix multiplication

---

**Input:**  $A \in \mathbb{F}_{bL}^{m \times l}, B \in \mathbb{F}_{bL}^{l \times n}$   
**Output:**  $C \in \mathbb{F}_{bL}^{m \times n}$   
 $A^{(S)} := A, B^{(S)} := B : A^{(S)} \in \mathbb{F}_{bS}^{m \times l}, B^{(S)} \in \mathbb{F}_{bS}^{l \times n}$   
 $\mathbf{e} := [1 \ 1 \ \dots \ 1]^T \in \mathbb{F}_{bS}^l$   
 $\alpha := 1$   
**while**  $\alpha < D$  **do**  
     $\mu_A := [\max_{1 \leq p \leq l} |(A^{(S)})_{ip}|]_{i=1,2,\dots,m} \in \mathbb{F}_{bS}^m$   
     $\mu_B := [\max_{1 \leq q \leq l} |(B^{(S)})_{qj}|]_{j=1,2,\dots,n} \in \mathbb{F}_{bS}^n$   
     $\tau_A := [2^{\lceil \log_2((\mu_A)_i) \rceil + \lceil (S + \log_2(l))/2 \rceil}]_{i=1,2,\dots,m} \in \mathbb{F}_{bS}^m$   
     $\tau_B := [2^{\lceil \log_2((\mu_B)_j) \rceil + \lceil (S + \log_2(l))/2 \rceil}]_{j=1,2,\dots,n} \in \mathbb{F}_{bS}^n$   
     $S_A := \tau_A \mathbf{e}^T$   
     $S_B := \mathbf{e} \tau_B^T$   
     $A_\alpha := (A^{(S)} + S_A) - S_A : A_\alpha \in \mathbb{F}_{bS}^{m \times l}$   
     $B_\alpha := (B^{(S)} + S_B) - S_B : B_\alpha \in \mathbb{F}_{bS}^{l \times n}$   
     $A := A - A_\alpha, B := B - B_\alpha : L\text{-bit FP arithmetic}$   
     $A^{(S)} := A, B^{(S)} := B$   
     $\alpha := \alpha + 1$   
**end while**  
 $A_D := A^{(S)}, B_D := B^{(S)}$   
 $C := O$   
**for**  $\alpha = 1, 2, \dots, D$  **do**  
    **for**  $\beta = 1, 2, \dots, D - \alpha + 1$  **do**  
         $C_{\alpha\beta} := A_\alpha B_\beta$   
    **end for**  
     $C := C + \sum_{\beta=1}^{D-\alpha+1} C_{\alpha\beta} : L\text{-bit FP arithmetic}$   
**end for**

---

Fig. 1.2 illustrates an example of the Ozaki scheme when  $A$  and  $B \in \mathbb{R}^{3 \times 3}$  are divided into three short-digit matrices. The most important feature of the Ozaki scheme is that the matrices  $A$  and  $B$  are divided into  $A_1, A_2, A_3, B_1, B_2,$  and  $B_3$  to fit within a short precision, thereby avoiding rounding errors in fast, low-precision matrix multiplication. An error-free matrix product  $C_{ij} := A_i B_j (i, j = 1, 2, 3)$  can be obtained as a result, and a highly accurate matrix product  $C$  can be obtained using a multiple-precision matrix addition operation for  $C := \sum_{i,j} C_{ij}$ . Although the number of divisions of  $A$  and  $B$  is finite, it is difficult to determine the minimum number of divisions required to guarantee a certain accuracy threshold. As a result, benchmark tests must be performed to determine if the algorithm can be executed faster than other multiplication algorithms.

## 1.6 Software layer of BNCmatmul

We have already used QD[1] as multi-component-type fixed precision C++ class library, CAMPARY[3] as multi-component-type arbitrary precision library, and MPFR[8] baased on GNU MP[2]’s MPN kernel. MPLAPACK/MPBLAS[6] is greeding these all reliable MP libraries and providing the coresponding APIs of LAPACK/BLAS[5]. Our BNCmatmul developed aims to get better performance than MPBLAS

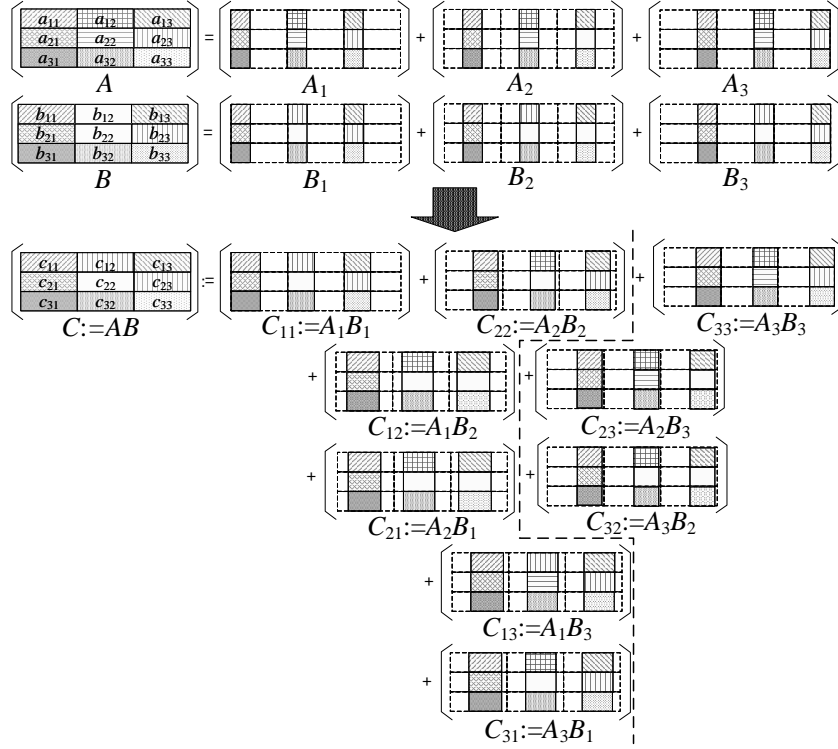


Fig. 1.2 Matrix multiplication based on Ozaki scheme when the matrices are divided into three components

as ATLAS, OpenBLAS and Intel Math Kernel, and it has the following features:

- Core functions and macros are written in the traditional way of ANSI C, not C++.
- Supporting native multi-component-type DD, TD, and QD precision arithmetic, and also accelerating vector and matrix computation like BLAS with AVX2. AVX-512 is partly implemented but not recommended to use.
- Direct use of MPFR functions to avoid overhead due to mpreal class library.
- Supporting partly shared-memory-type parallelization based on OpenMP.
- Implementing three algorithms of matrix multiplication: simple triple-loop, blocking (tiling), and Strassen or Winograd algorithm. Ozaki scheme can be used as trial optimization.

The software layer of BNCmatmul except parallelization is shown as Fig. 1.3.

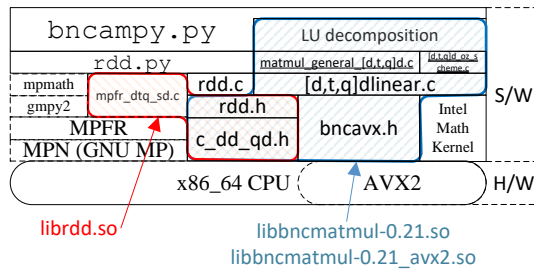


Fig. 1.3 Software layer of BNCmatmul library

Basic DD, TD, and QD precision arithmetic are defined as C inline functions in c\_dd\_qd.h and are

defined as macros in `rdd.h`. To use the Python environment, `rdd.c`, which is based on `rdd.h`, is compiled as `librdd.so`. The DD, TD, and QD classes on Python use the functions defined in `librdd.so`.

BNCmatmul, which is also built on `rdd.h` and `c_dd_qd.h`, includes block and Strassen matrix multiplications that provide more efficient performance than simple triple-loop matrix multiplication. These have been accelerated using SIMDized functions with AVX2 defined in `bncavx2.h`. The LU decomposition has also been accelerated with AVX2.

These functions that are defined in BNCmatmul (Fig. 1.3) can be used from `libbncmatmul-0.21.so` without any SIMDization techniques and `libbncmatmul-0.21_avx2.so` with SIMDization of AVX2. As these two DLLs have the same function names, so users do not need to modify their codes when selecting the BNCmatmul DLLs.

## 1.7 History of Version and Todo list

### ■History of Version

Version 2.0: 2016-06-08 Firstly opening sources at <https://na-inet.jp/na/bnc/bncmatmul-0.2.tar.bz2>, which provides only arbitrary matrix multiplication based on MPFR.

Version 2.1: 2023-05-31 Secondly opening sources including DD, TD, QD and MPFR precision real BLAS functions at <https://github.com/tkouya/bncmatmul/blob/main/bncmatmul-0.21.tar.bz2>.

### ■Todo list

1. Appending complex BLAS functions with DD, TD, QD, and MPFR(MPC)
2. Appending complex LU decomposition and related functions
3. Appending sparse matrix-vector multiplication
4. Showing more sample sources in this manual using BNCmatmul and MPLAPACK/BLAS



## Chapter 2

# Basic datatypes and arithmetic

In this chapter, we present and describe all basic data types and functions defined in BNCmatmul, and show how to use them in order to construct your programs including multiple-precision computing.

As shown in Chapter 1, all basic datatype and functions are written in ANSI C, not C++ way.

### 2.1 c\_dd\_qd.h

(Macro) **DFMA**( $a, b, c$ ) returns  $a \times b + c$  in double precision

(Macro) **SFMA**( $a, b, c$ ) returns  $a \times b + c$  in single precision

(Macro) **QD\_FMA**( $a, b, c$ ) the same as **DFMA**( $a, b, c$ )

(Macro) **QD\_FMS**( $a, b, c$ ) returns  $a \times b - c$  in double precision

(Macro) **DDSIZE** is 2, the number of binary64(double) variables in DD

(Macro) **TDSIZE** is 3, the number of binary64 variables in TD

(Macro) **QDSIZE** is 4, the number of binary64 variables in QD

#### 2.1.1 DD (double-double): 106-bit ( $53\text{bits} \times 2$ ) precision floating-point number

(Macro) **DD\_HI**( $a$ ) is  $a[0]$ .

(Macro) **DD\_LOW**( $a$ ) is  $a[1]$ .

(Macro) **DD\_TRUE** is 1UL.

(Macro) **TD\_TRUE**

(Macro) **QD\_TRUE**

(Macro) **DD\_FALSE** is 0UL.

(Macro) **TD\_FALSE**

(Macro) **QD\_FALSE**

(Macro) **DD\_ISNAN**( $a$ ) checks if  $a$  includes NAN.

(Macro) **TD\_ISNAN**( $a$ )

(Macro) **QD\_ISNAN**( $a$ )

(Macro) **DD\_ISINF**( $a$ ) checks if  $a$  includes INF.

(Macro) **TD\_ISINF**( $a$ )

(Macro) **QD\_ISINF**( $a$ )

(Macro) **DD\_ISZERO**( $a$ ) checks if  $a$  is zero.

(Macro) **TD\_ISZERO**( $a$ )

(Macro) **QD\_ISZERO**( $a$ )

(Macro) **DD\_ISONE**( $a$ ) checks if  $a$  is one.

(Macro) **TD\_ISONE**( $a$ )

(Macro) **QD\_ISONE**( $a$ )

(Macro) **DD\_ISNEGATIVE**( $a$ ) checks if  $a < 0$ .

(Macro) **TD\_ISNEGATIVE**( $a$ )

(Macro) **QD\_ISNEGATIVE**( $a$ )

(Macro) DD\_NAN is FP\\_NAN.

(Macro) TD\_NAN

(Macro) QD\_NAN

## 2.2 Error-free transformation

- `double quick_two_sum(double a, double b, double * err) ...` Computes  $fl(a + b)$  and  $err(a + b)$ . Assumes  $|a| \geq |b|$ .
- `double quick_two_diff(double a, double b, double * err) ...` Computes  $fl(a - b)$  and  $err(a - b)$ . Assumes  $|a| \geq |b|$ .
- `double two_sum(double a, double b, double * err) ...` Computes  $fl(a + b)$  and  $err(a + b)$ .
- `double two_diff(double a, double b, double * err) ...` Computes  $fl(a - b)$  and  $err(a - b)$ .
- `void split(double a, double *hi, double *lo) ...` divide  $a$  to  $hi$  and  $lo$  without error.
- `double two_prod(double a, double b, double * err) ...` Computes  $fl(ab)$  and  $err(ab)$ .
- `double two_sqr(double a, double * err) ...` Computes  $fl(a^2)$  and  $err(a^2)$ . Faster than the above method.

## 2.3 Double-double precision arithmetic

- `dd_bool dd_is_zero(const double * x) ...` Check if  $*x$  is zero.
- `dd_bool dd_is_one(const double * x) ...` Check if  $*x$  is one.
- `void c_dd_set(const double * x, double *ret) ...`  $ret := x$
- `void c_dd_set_dd_d(const double x, double * ret) ...`  $ret := (double)x$
- `void c_dd_set0(double * ret) ...`  $ret := 0$
- `void c_dd_setnan(double * ret) ...`  $ret := NaN$
- `void c_dd_neg(const double * a, double * b) ...`  $b := -a$
- `dd_bool dd_is_positive(const double * x) ...`  $*x > 0$  ?
- `dd_bool dd_is_negative(const double * x) ...`  $*x < 0$  ?
- `void c_dd_comp(const double * a, const double * b, int * result) ...` Compare with  $a$  and  $b$ , and store 0 if  $a == b$ , +1 else if  $a > b$ , -1 else if  $a < b$ .
- `void c_dd_comp_dd_d(const double * a, double b, int * result)`
- `void c_dd_comp_d_dd(double a, const double * b, int * result)`
- `void c_dd_pi(double * a)`
- `void c_d_add(double a, double b, double * c) ...` double-double := double + double
- `void c_dd_add(const double * a, const double * b, double * c) ...`  $c := a + b$
- `void c_dd_add_sloppy(const double * a, const double * b, double * c)`
- `void c_dd_add_dd_d(const double * a, double b, double * c)`
- `void c_dd_add_d_dd(double a, const double * b, double * c)`
- `void c_d_sub(double a, double b, double * c) ...` double-double = double - double
- `void c_dd_sub(const double * a, const double * b, double * c) ...`  $c := a - b$
- `void c_dd_sub_sloppy(const double * a, const double * b, double * c)`
- `void c_dd_sub_dd_d(const double * a, double b, double * c)`
- `void c_dd_sub_d_dd(double a, const double * b, double * c)`
- `void c_d_mul(double a, double b, double * c) ...` double-double := double \* double
- `void c_dd_mul(const double * a, const double * b, double * c)`
- `void c_dd_mul_dd_d(const double * a, double b, double * c)`
- `void c_dd_mul_d_dd(double a, const double * b, double * c)`
- `void c_d_div(double a, double b, double * c) ...`  $c := a/b$
- `void c_dd_div(const double * a, const double * b, double * c) ...`  $c := a/b$
- `void c_dd_sloppy_div(double * a, double * b, double * c)`
- `void c_dd_div_dd_d(const double * a, double b, double * c)`

- void c\_dd\_div\_d\_dd(double a, const double \* b, double \* c)
- void c\_dd\_copy(const double \* a, double \* b)  $b := a$
- void c\_dd\_copy\_d(double a, double \* b)
- void c\_d\_sqr(double a, double \* b)  $\dots b := a^2$
- void c\_dd\_sqr\_d(double a, double \* ret)
- void c\_dd\_sqr(const double \* a, double \* b)
- void c\_dd\_sqrt(const double \* a, double \* b)  $\dots b := \sqrt{a}$
- void c\_dd\_abs(const double \* a, double \* b)  $\dots b := |a|$
- void c\_dd\_floor(const double \* a, double \* b)  $\dots b := \text{floor}(a)$
- void c\_dd\_ceil(const double \* a, double \* b)  $\dots b := \text{ceil}(a)$

## 2.4 Quadruple-double precision arithmetic

- void c\_qd\_copy(const double \* a, double \* b)  $\dots b := a$
- void c\_qd\_copy\_dd(const double \* a, double \* b)
- void c\_qd\_copy\_d(double a, double \* b)
- void quick\_renorm(double \* c0, double \* c1, double \* c2, double \* c3, double \* c4)  $\dots$  Renormalize c0 ...
- void renorm(double \* c0, double \* c1, double \* c2, double \* c3)
- void renorm4(double \* c0, double \* c1, double \* c2, double \* c3, double \* c4)
- void three\_sum(double \* a, double \* b, double \* c)
- void three\_sum2(double \* a, double \* b, double \* c)
- void c\_qd\_add\_qd\_dd(const double \* a, const double \* b, double \* c)  $\dots c := a + b$
- void c\_qd\_add(const double \* a, const double \* b, double \* c)
- void c\_qd\_add\_sloppy(const double \* a, const double \* b, double \* c)  $\dots$  not so accurate but fast addition  $c := a + b$
- void c\_td\_addq(const double \* a, const double \* b, double \* c)  $\dots$  triple double addition based on quad-double way
- void c\_qd\_selfadd(const double \* a, double \* b)  $\dots b := b + a$
- void c\_qd\_selfadd\_dd(const double \* a, double \* b)
- void c\_qd\_selfadd\_d(double a, double \* b)
- void c\_qd\_neg(const double \* a, double \* b)  $\dots b := -a$
- void c\_qd\_neg\_dd(const double \* a, double \* b)
- void c\_qd\_neg\_d(const double a, double \* b)
- void c\_qd\_sub(const double \* a, const double \* b, double \* c)  $\dots c := a - b$
- void c\_qd\_sub\_qd\_dd(const double \* a, const double \* b, double \* c)
- void c\_qd\_sub\_dd\_qd(const double \* a, const double \* b, double \* c)
- void c\_qd\_sub\_qd\_d(const double \* a, double b, double \* c)
- void c\_qd\_sub\_d\_qd(double a, const double \* b, double \* c)
- void c\_qd\_selfsub(const double \* a, double \* b)  $\dots b := b + (-a) = b - a$
- void c\_qd\_selfsub\_dd(const double \* a, double \* b)
- void c\_qd\_selfsub\_d(double a, double \* b)
- void c\_qd\_mul(const double \* a, const double \* b, double \* c)  $\dots c := a \times b$
- void c\_qd\_mul\_qd\_dd(const double \* a, const double \* b, double \* c)
- void c\_qd\_mul\_dd\_qd(const double \* a, const double \* b, double \* c)
- void c\_qd\_mul\_qd\_d(const double \* a, double b, double \* c)
- void c\_qd\_mul\_d\_qd(double a, const double \* b, double \* c)
- void c\_qd\_mul\_sloppy(const double \* a, const double \* b, double \* c)
- void c\_qd\_sqr(const double \* a, double \* c)  $\dots c = a^2$
- void c\_qd\_selfmul(const double \* a, double \* b)  $\dots b := a * b$
- void c\_qd\_selfmul\_dd(const double \* a, double \* b)
- void c\_qd\_selfmul\_d(double a, double \* b)

- void c\_qd\_div\_accurate(const double \* a, const double \* b, double \* c) ...  $c := a/b$
- void c\_qd\_div\_sloppy(const double \* a, const double \* b, double \* c) ... not so accurate but fast division (default)

(Macro) USE\_QD\_DIV\_ACCURATE() ... use accurate qd division if true

- void c\_qd\_div\_qd\_dd(const double \* a, const double \* b, double \* c)
- void c\_qd\_div\_qd\_d(const double \* a, double b, double \* c)
- void c\_qd\_div\_d\_qd(double a, const double \* b, double \* c)
- void c\_qd\_selfdiv(const double \* a, double \* b) ...  $b := b/a$
- void c\_qd\_selfdiv\_dd(const double \* a, double \* b)
- void c\_qd\_selfdiv\_d(double a, double \* b)
- void c\_qd\_mul\_pwr2(const double \* a, double b, double \* c) ...  $c := (a[0]*b, a[1]*b, a[2]*b, a[3]*b)$
- void c\_qd\_set(const double \* x, double \* qdval) ...  $qdval := c$
- void c\_qd\_set0double \* qdval() ...  $qdval := 0$
- void c\_qd\_sqrt(const double \* a, double \* b) ...  $b := \sqrt{a}$
- void c\_qd\_abs(const double \* a, double \* b) ...  $b := |a|$

(Macro) nint(a) ... round(a) as integer

- void c\_qd\_nint(const double \* a, double \* b)
- void c\_qd\_floor(const double \* a, double \* b) ...  $b := \text{floor}(a)$
- void c\_qd\_ceil(const double \* a, double \* b) ...  $b := \text{ceil}(a)$
- void c\_qd\_comp(const double \* a, const double \* b, int \* result) ... return -1 if  $a < b$ , 0 if  $a == b$ , and +1 if  $a > b$ .
- void c\_qd\_comp\_qd\_d(const double \* a, double b, int \* result)
- void c\_qd\_comp\_d\_qd(double a, const double \* b, int \* result)

## 2.5 Triple-double precision arithmetic

- void c\_td\_copy(const double \* a, double \* b) ...  $b := a$
- void c\_qd\_copy\_td(const double \* a, double \* c)
- void c\_td\_copy\_qd(const double \* a, double \* b)
- void c\_td\_copy\_dd(const double \* a, double \* b)
- void c\_td\_copy\_d(double a, double \* b)
- void vec\_sum(double \* e, const double \* x, int n) ...  $e[n] := \text{vec\_sum}(x[n])$
- void vseb(double \* y, int ny, const double \* e, int ne) ...  $y[n] := \text{vec\_sum\_err\_branch}(vseb)(k)(e[n])$
- void c\_to\_td(double \* r, double a, double b, double c) ...  $r[3] := \text{to\_td}(a, b, c)$
- void merge(double \* c, double \* a, int na, double \* b, int nb) ... Merge  $a[na]$  and  $b[nb]$  into  $c[na + nb]$
- void c\_td\_add(double \* a, double \* b, double \* c) ...  $c := a + b$
- void c\_td\_add\_td\_d(double \* a, double b, double \* c)
- void c\_td\_neg(const double \* a, double \* c) ...  $c := -a$
- //
- void c\_td\_sub(double \* c, double \* a, double \* b) ...  $c := a - b$
- void c\_td\_sub(double \* a, double \* b, double \* c)
- void c\_td\_subq(const double \* a, const double \* b, double \* c)
- void c\_td\_sub\_d\_td(double a, double \* b, double \* c) //
- void c\_td\_sub\_td\_d(double \* c, double \* a, double b)
- void c\_td\_sub\_td\_d(double \* a, double b, double \* c)
- void c\_td\_mul\_accurate(double \* a, double \* b, double \* c) ...  $c := a \times b$
- void c\_td\_mul\_sloppy(double \* a, double \* b, double \* c) ... not so accurate but fast  $c := a \times b$

(Macro) USE\_TD\_MUL\_ACCURATE() ... Use accurate td multiplication if true

- void c\_td\_mul\_dd\_td\_sloppy(double \* a, double \* b, double \* c)
- void c\_td\_mul\_dd\_td\_accurate(double \* a, double \* b, double \* c)

(Macro) USE\_TD\_MUL\_DD\_TD\_ACCURATE() ... Use accurate td-dd multiplication if true

- void c\_td\_mul\_d\_td(const double a, const double \* b, double \* c)
- void c\_td\_mul\_td\_d(const double \* a, const double b, double \* c)
- void c\_td\_abs(const double \* a, double \* b) ...  $b := |a|$
- void c\_td\_comp(const double \* a, const double \* b, int \* result) ... return -1 if  $a < b$ , 0 if  $a == b$ , and +1 if  $a > b$ .
- void c\_td\_comp\_td\_d(const double \* a, double b, int \* result)
- void c\_td\_comp\_d\_td(double a, const double \* b, int \* result)
- void c\_td\_2mtw\_dd\_td(double a[DDSIZE], double b[TDSIZE], double c[TDSIZE])
- (Macro) ONE\_P\_2DBL\_EPS() ...  $1 + 2 \times \text{DBL\_EPSILON} = (1.00000000000000044e + 00)$
- (Macro) ONE\_M\_2DBL\_EPS() ...  $1 - 2 \times \text{DBL\_EPSILON} = (9.99999999999999556e - 01)$
- void c\_td\_reciprocal(double \* a, double \* c)
- void c\_td\_divt(double \* a, double \* b, double \* c)
- (Macro) c\_td\_div() ... the same as c\_td\_divtq
- void c\_td\_divq(const double \* a, const double \* b, double \* c) ... td division based on quad-double division.
- void c\_td\_div\_td\_d(double \* a, double b, double \* c)
- void c\_td\_sqrt(double \* a, double \* c) ...  $c := \sqrt{a}$
- void c\_td\_sqrt\_d(double a, double \* c)
- void c\_td\_sqr(double \* a, double \* c) ...  $c := a^2$

## 2.6 rdd.h

```
// ddfloat, tdfloat, qdfloat
typedef struct { double val[DDSIZE]; } ddfloat; // 53 * 2 = 106
typedef struct { double val[TDSIZE]; } tdfloat; // 53 * 3 = 159
typedef struct { double val[QDSIZE]; } qdfloat; // 53 * 4 = 212
```

(Macro) SET0\_DD(val) ...  $val := 0$  val[0] = (double)0.0; val[1] = (double)0.0;

(Macro) SET0\_TD(val)

(Macro) SET0\_QD(val)

- void rdd\_out\_str\_base(FILE \* fp, int base, int length, double val[DDSIZE]) ... DD print(no appending CR)
- int rdd\_cmp(double a[DDSIZE], double b[DDSIZE]) ... return -1 if  $a < b$ , 0 if  $a == b$ , and +1 if  $a > b$ .
- int rdd\_cmp\_d(double a[DDSIZE], double b)
- void rdd\_sqrt\_d(double ret[DDSIZE], double a) ... return  $\sqrt{ret}$
- void rdd\_fma(double ret[DDSIZE], double a[DDSIZE], double b[DDSIZE], double c[DDSIZE]) ...  $ret := a \times b + c$
- void rdd\_pow(double ret[DDSIZE], double base[DDSIZE], double power[DDSIZE]) ...  $ret := \text{base}^{\text{power}}$
- void rqd\_out\_str\_base(FILE \*fp, int base, int length, double val[QDSIZE]) ... print(no appending CR)
- int rqd\_cmp(double a[QDSIZE], double b[QDSIZE]) ... return -1 if  $a < b$ , 0 if  $a == b$ , and +1 if  $a > b$ .
- int rqd\_cmp\_d(double a[QDSIZE], double b)
- void rqd\_sqrt\_d(double ret[QDSIZE], double a) ... return  $\sqrt{ret}$
- void rqd\_fma(double ret[QDSIZE], double a[QDSIZE], double b[QDSIZE], double c[QDSIZE]) ...  $ret := a \times b + c$
- void rqd\_pow(double ret[QDSIZE], double base[QDSIZE], double power[QDSIZE]) ...  $ret := \text{base}^{\text{power}}$
- void rtd\_out\_str\_base(FILE \*fp, int base, int length, double val[TDSIZE])
- int rtd\_cmp(double a[TDSIZE], double b[TDSIZE]) ... return -1 if  $a < b$ , 0 if  $a == b$ , and +1 if  $a > b$ .

- $a < b$ .
- `int rtd_cmp_d(double a[TDSIZE], double b)`
- `void rtd_sqrt_d(double ret[TDSIZE], double a) ... return  $\sqrt{ret}$`
- `void rtd_fma(double ret[TDSIZE], double a[TDSIZE], double b[TDSIZE], double c[TDSIZE]) ...`  
 $ret := a \times b + c$
- `void rtd_pow(double ret[TDSIZE], double base[TDSIZE], double power[TDSIZE]) ...`  
 $ret := base^{power}$

(Macro) `set0_dd(val) val := 0`

(Macro) `rdd_set0(val)`

(Macro) `rdd_add(ret, a, b) ret := a + b`

(Macro) `rdd_sub(ret, a, b) ret := a - b`

(Macro) `rdd_mul(ret, a, b) ret := a  $\times$  b`

(Macro) `rdd_div(ret, a, b) ret := a/b`

(Macro) `rdd_sqrt(ret, a) ret :=  $\sqrt{a}$`

(Macro) `rdd_sqrt_d(ret, a)`

(Macro) `rdd_sqrt_ui(ret, a)`

(Macro) `rdd_get_d(a)`

(Macro) `rdd_set_d(ret, d) ret := d`

(Macro) `rdd_set_ui(ret, d)`

(Macro) `rdd_set(ret, d)`

(Macro) `rdd_neg(ret, a) ret :=  $-a$`

(Macro) `rdd_abs(ret, a) ret :=  $|a|$`

(Macro) `rdd_cmp_ui(a, b) return  $-1$  if  $a > b$ ,  $0$  if  $a == b$ , and  $-1$  if  $a < b$`

(Macro) `rdd_ui_div(ret, a, b) ret :=  $a/b$`

(Macro) `rdd_ui_sub(ret, a, b) ret :=  $a - b$`

(Macro) `rdd_div_d(ret, a, b) ret :=  $a/b$`

(Macro) `rdd_add_d(ret, a, b) ret :=  $a + b$`

(Macro) `rdd_sub_d(ret, a, b) ret :=  $a - b$`

(Macro) `rdd_mul_d(ret, a, b) ret :=  $a \times b$`

(Macro) `rdd_div_ui(ret, a, b) ret :=  $a/b$`

(Macro) `rdd_add_ui(ret, a, b) ret :=  $a + b$`

(Macro) `rdd_sub_ui(ret, a, b) ret :=  $a - b$`

(Macro) `rdd_mul_ui(ret, a, b) ret :=  $a \times b$`

(Macro) `set0_td(val) SET0_TD(val)`

(Macro) `rtd_set0(val) SET0_TD(val)`

(Macro) `rtd_add(ret, a, b) ret :=  $a + b$`

(Macro) `rtd_addt(ret, a, b)`

(Macro) `rtd_addq(ret, a, b)`

(Macro) `rtd_sub(ret, a, b) ret :=  $a - b$`

(Macro) `rtd_subt(ret, a, b) ret :=  $a - b$`

(Macro) `rtd_subq(ret, a, b) ret :=  $a - b$`

(Macro) `rtd_mul(ret, a, b) ret :=  $a \times b$`

(Macro) `rtd_divt(ret, a, b) ret :=  $a/b$`

(Macro) `rtd_divtq(ret, a, b)`

(Macro) `rtd_divq(ret, a, b)`

(Macro) `rtd_div(ret, a, b)`

(Macro) `rtd_sqrt(ret, a) ret :=  $\sqrt{a}$`

(Macro) `rtd_sqrt_d(ret, a)`

(Macro) `rtd_sqrt_ui(ret, a)`

(Macro) `rtd_get_d(a) return (double)a`

(Macro) `rtd_set_d(ret, d) ret := d`

(Macro) `rtd_set_ui(ret, d)`

```

(Macro) rtd_set(ret, d)
(Macro) rtd_neg(ret, a) ret :=  $-a$ 
(Macro) rtd_abs(ret, a) ret :=  $|a|$ 
(Macro) rtd_cmp_ui(a, b) return  $-1$  if  $a > b$ ,  $0$  if  $a == b$ , and  $-1$  if  $a < b$ 
(Macro) rtd_ui_div(ret, a, b) ret :=  $a/b$ 
(Macro) rtd_ui_sub(ret, a, b) ret :=  $a - b$ 
(Macro) rtd_div_d(ret, a, b)  $\dots c := a/b$ 
(Macro) rtd_add_d(ret, a, b)  $\dots c := a + b$ 
(Macro) rtd_sub_d(ret, a, b)  $\dots c := a - b$ 
(Macro) rtd_mul_d(ret, a, b)  $\dots c := a \times b$ 
(Macro) rtd_div_ui(ret, a, b)  $\dots c := a/b$ 
(Macro) rtd_add_ui(ret, a, b)  $\dots c := a + b$ 
(Macro) rtd_sub_ui(ret, a, b)  $\dots c := a - b$ 
(Macro) rtd_mul_ui(ret, a, b)  $\dots c := a \times b$ 
(Macro) set0_qd(val)  $\dots val := 0$ 
(Macro) rqd_set0(val)
(Macro) rqd_add(ret, a, b)  $\dots c := a + b$ 
(Macro) rqd_sub(ret, a, b)  $\dots c := a - b$ 
(Macro) rqd_mul(ret, a, b)  $\dots c := a \times b$ 
(Macro) rqd_div(ret, a, b)  $\dots c := a/b$ 
(Macro) rqd_sqrt(ret, a)  $\dots ret := \sqrt{a}$ 
(Macro) rqd_sqrt_d(ret, a) ret :=  $\sqrt{a}$ 
(Macro) rqd_sqrt_ui(ret, a)
(Macro) rqd_get_d(a) return (double)a
(Macro) rqd_set_d(ret, d) ret :=  $d$ 
(Macro) rqd_set_ui(ret, d)
(Macro) rqd_set(ret, d)
(Macro) rqd_neg(ret, a) ret :=  $-a$ 
(Macro) rqd_abs(ret, a) ret :=  $|a|$ 
(Macro) rqd_cmp_ui(a, b) return  $-1$  if  $a > b$ ,  $0$  if  $a == b$ , and  $-1$  if  $a < b$ 
(Macro) rqd_ui_div(ret, a, b) ret :=  $a/b$ 
(Macro) rqd_ui_sub(ret, a, b) ret :=  $a - b$ 
(Macro) rqd_div_d(ret, a, b)  $\dots c := a/b$ 
(Macro) rqd_add_d(ret, a, b)  $\dots c := a + b$ 
(Macro) rqd_sub_d(ret, a, b)  $\dots c := a - b$ 
(Macro) rqd_mul_d(ret, a, b)  $\dots c := a \times b$ 
(Macro) rqd_div_ui(ret, a, b)  $\dots c := a/b$ 
(Macro) rqd_add_ui(ret, a, b)  $\dots c := a + b$ 
(Macro) rqd_sub_ui(ret, a, b)  $\dots c := a - b$ 
(Macro) rqd_mul_ui(ret, a, b)  $\dots c := a \times b$ 

```

## 2.7 AVX2

### 2.7.1 float

- `__m256 _bncavx2_fabsf(__m256 val8)  $\dots val8 := \max(val8, 0 - val8)$`
- `__m256 _bncavx2_fneg(__m256 a)  $\dots$  return  $-a$`
- `void _bncavx2_ffma(float ret[], float a[], float b[], float c[], int dim)  $\dots ret := a \times b + c$`
- `void _bncavx2_fmud(float ret[], float a[], float b[], int dim)  $\dots ret := a \times b$`

## 2.7.2 double

- `__m256d _bncavx2_fabs(__m256d val4) ... val4 := max(val4, -val4)`
- `__m256d _bncavx2_dneg(__m256d a) ... -a`
- `void _bncavx2_dfma(double ret[], double a[], double b[], double c[], int dim) ... ret :=  $a \times b + c$`
- `void _bncavx2_dmul(double ret[], double a[], double b[], int dim) ... ret :=  $a \times b$`
- `void _bncavx2_ddiv(double ret[], double a[], double b[], int dim) ... ret :=  $a/b$`
- `void _bncavx2_dadd(double ret[], double a[], double b[], int dim) ... ret :=  $a + b$`
- `void _bncavx2_dsub(double ret[], double a[], double b[], int dim) ... ret :=  $a - b$`
- `double _bncavx2_ddotp(double a[], double b[], int dim) ... ret :=  $\sum_{i=0}^{\dim-1} a[i] \times b[i]$`

## 2.7.3 Error-free transformation

- `__m256d _bncavx2_dquick_two_sum(__m256d a, __m256d b, __m256d * err) ...` Computes  $\text{fl}(a + b)$  and  $\text{err}(a + b)$ . Assume  $|a| \geq |b|$ .
- `__m256d _bncavx2_dquick_two_diff(__m256d a, __m256d b, __m256d * err) ...` Computes  $\text{fl}(a - b)$  and  $\text{err}(a - b)$ . Assume  $|a| \geq |b|$ .
- `__m256d _bncavx2_dtwo_sum(__m256d a, __m256d b, __m256d * err) ...` Computes  $\text{fl}(a + b)$  and  $\text{err}(a + b)$ .
- `__m256d _bncavx2_dtwo_diff(__m256d a, __m256d b, __m256d * err) ...` Computes  $\text{fl}(a - b)$  and  $\text{err}(a - b)$ .
- `__m256d _bncavx2_dtwo_prod(__m256d a, __m256d b, __m256d * err) ...` Computes  $\text{fl}(a \times b)$  and  $\text{err}(a \times b)$ .

## 2.7.4 Double-double precision arithmetic

- `void _bncavx2_set0_dd(__m256d) ret[DDSIZE] ... ret := 0`
- (Macro) `_bncavx2_rdd_set0(ret) ...` is the same as `_bncavx2_set0_dd(ret)`
- `void _bncavx2_get_dd_m256d_i(ddfloat *ret, __m256d ret4[DDSIZE], int avx_index) ... ret := ret4[[avx_index]`
- `void _bncavx2_rdd_sum256d(double ret[DDSIZE], __m256d ret4[DDSIZE]) ... ret := ret4[0] + ret4[1] + ret4[2] + ret4[3]`
- `void _bncavx2_rdd_abssum256d(double ret[DDSIZE], __m256d ret4[DDSIZE]) ... ret := |ret4[0]| + |ret4[1]| + |ret4[2]| + |ret4[3]|`
- `void _bncavx2_rdd_absmax256d(double ret[DDSIZE], __m256d ret4[DDSIZE]) ... ret := max(|ret4[0]|, |ret4[1]|, |ret4[2]|, |ret4[3]|)`
- `void _bncavx2_rdd_norm256d(double ret[DDSIZE], __m256d ret4[DDSIZE]) ... ret := ||ret4[0]2 + ret4[1]2 + ret4[2]2 + ret4[3]2||2`
- `void _bncavx2_rdd_add(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE]) ... ret :=  $a + b$`
- `void _bncavx2_rdd_sub(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE]) ... ret :=  $a - b$`
- `void _bncavx2_rdd_mul(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE]) ... ret :=  $a \times b$`
- `void _bncavx2_rdd_div(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE]) ... ret :=  $a/b$`
- `void _bncavx2_rdd_abs(__m256d ret[DDSIZE], __m256d a[DDSIZE]) ... ret :=  $|a|$`



## 2.7.5 Triple-double precision arithmetic

- `void _bncavx2_set0_td(__m256d ret[TDSIZE]) ... ret := 0`
  - `void _bncavx2_get_td_m256d_i(tdfloat * ret, __m256d ret4[TDSIZE], int avx_index) ... ret := ret4[avx_index]`
  - `void _bncavx2_rtd_sum256d(double ret[TDSIZE], __m256d ret4[TDSIZE]) ... ret := ret4[0] + ret4[1] + ret4[2]`
  - `void _bncavx2_rtd_abs(__m256d ret[TDSIZE], __m256d a[TDSIZE]) ... ret := |a|`
  - `void _bncavx2_rtd_abssum256d(double ret[TDSIZE], __m256d ret4[TDSIZE]) // ret := |ret4[0]| + |ret4[1]| + |ret4[2]| + |ret4[3]|`
  - `void _bncavx2_rtd_absmax256d(double ret[TDSIZE], __m256d ret4[TDSIZE]) // ret := max(|ret4[0]|, |ret4[1]|, |ret4[2]|, |ret4[3]|)`
  - `void _bncavx2_rtd_norm256d(double ret[TDSIZE], __m256d ret4[TDSIZE])`
  - `void _bncavx2_vec_sum(__m256d e[], const __m256d x[], int n) ... e := vec_sum(x)`
  - `void _bncavx2_vseb(__m256d y[], int ny, const __m256d e[], int ne) ... y := vec_sum_err_branch(vseb)(k)(e)`
  - `void _bncavx2_merge(__m256d c[], __m256d a[], int na, __m256d b[], int nb) ... Merge a[na] & b[nb] into c[na + nb]`
- (Macro) `_bncavx2_rtd_add() _bncavx2_rtd_addq`
- `void _bncavx2_rtd_addq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ... ret := a + b`
  - `void _bncavx2_rtd_addt(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE])`
  - `void _bncavx2_rtd_mulq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ... ret := a × b`
  - `void _bncavx2_rtd_mul(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE])`
  - `void _bncavx2_rtd_neg(__m256d c[TDSIZE], __m256d a[TDSIZE]) ... c := -a`
  - `void _bncavx2_rtd_sub(__m256d c[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ... c := a - b`
  - `void _bncavx2_rtd_subq(__m256d c[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE])`
  - `void _bncavx2_rtd_divq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ... c := a/b`
  - `void _bncavx2_to_td(__m256d r[TDSIZE], __m256d a, __m256d b, __m256d c) ... r := TD(a, b, c)`
  - `void _bncavx2_rtd_mul_d(__m256d c[TDSIZE], __m256d a, __m256d b[TDSIZE]) ... c := a × b`
  - `void _bncavx2_rtd_mul_dd(__m256d c[TDSIZE], __m256d a[DDSIZE], __m256d b[TDSIZE]) ... c := a × b`
- (Macro) `_bncavx2_rtd_div() ... _bncavx2_rtd_divtq`
- `void _bncavx2_rtd_divt(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ... c := a/sb`
  - `void _bncavx2_rtd_divtq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE])`

## 2.7.6 Quadruple-double precision arithmetic

- `void _bncavx2_set0_qd(__m256d ret[QDSIZE]) ... ret := 0`
- `void _bncavx2_get_qd_m256d_i(qdfloat * ret, __m256d ret4[QDSIZE], int avx_index) ret := ret4[avx_index]`
- `void _bncavx2_rqd_sum256d(double ret[QDSIZE], __m256d ret4[QDSIZE]) ... ret := mmval[0] + ... + mmval[7]`
- `void _bncavx2_rqd_abs__m256d ret[QDSIZE], __m256d a[QDSIZE] ... ret := |a|`
- `void _bncavx2_rqd_abssum256d(double ret[QDSIZE], __m256d ret4[QDSIZE]) ... ret := |ret4[0]| + |ret4[1]| + |ret4[2]| + |ret4[3]|`
- `void _bncavx2_rqd_absmax256d(double ret[QDSIZE], __m256d ret4[QDSIZE]) ... ret :=`

- $\max(|\text{ret4}[0]|, |\text{ret4}[1]|, |\text{ret4}[2]|, |\text{ret4}[3]|)$
- `void _bncavx2_rqd_norm256d(double ret[QDSIZE], __m256d ret4[QDSIZE]) ... ret :=  $\|\text{ret4}[0]^2 + \text{ret4}[1]^2 + \text{ret4}[2]^2 + \text{ret4}[3]^2\|_2$`
- `void _bncavx2_renorm(__m256d * c0, __m256d * c1, __m256d * c2, __m256d * c3) ... renorm(double *c0, double *c1, double *c2, double *c3)`
- `void _bncavx2_renorm4(__m256d * c0, __m256d * c1, __m256d * c2, __m256d * c3, __m256d * c4) ... renorm4(double *c0, double *c1, double *c2, double *c3, double *c4)`
- `void _bncavx2_three_sum(__m256d * a, __m256d * b, __m256d * c) ... three_sum(double *a, double *b, double *c)`
- `void _bncavx2_three_sum2(__m256d * a, __m256d * b, __m256d * c) ... void three_sum2(double *a, double *b, double *c)`
- `void _bncavx2_rqd_add(__m256d ret[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE]) ... ret :=  $a + b$`
- `void _bncavx2_rtd_addq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ... ret :=  $a + b$`
- `void _bncavx2_rtd_mulq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ... ret :=  $a \times b$`
- `void _bncavx2_rqd_mul(__m256d ret[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE]) ... ret :=  $a \times b$`
- `void _bncavx2_rqd_sub(__m256d c[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE]) ... ret :=  $a - b$`
- `void _bncavx2_rqd_mul_d(__m256d c[QDSIZE], const __m256d a[QDSIZE], __m256d b) ... ret :=  $a \times b$`
- `void _bncavx2_rtd_divq(__m256d c[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) ...  $c := a/b$`
- `void _bncavx2_rqd_div(__m256d c[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE])`

## Chapter 3

# Basic linear computation

### 3.1 Datatypes of D, DD, TD, QD and MPFR vector and matrix

Vector datatypes of DD, TD, QD and MPFR precision are shown as follows:

DDVector, TDVector, QDVector

```
typedef struct
{
    long int dim; // dim <= real_dim
    long int real_dim; // multiplier of _BNC_D_WIDTH
    double *element[DDSIZE]; // [TDSIZE] and [QDSIZE] used as TDVector and
    ↪ QDVector
} ddvector;

typedef *DDVector; // DDVector, TDVector, QDVector are pointers.
```

MPFVector

```
typedef struct{
    unsigned long int prec; // default precision of element
    mpf_t *element;
    long int dim;
    long int real_dim;
} mpfvector;

typedef mpfvector *MPFVector; // MPFVector is a pointer.
```

Matrix datatypes of DD, TD, QD and MPFR precision are shown as follows:

DDMatrix, TDMatrix, QDMatrix

```
// DD matrix
typedef struct{
    long int row_dim, col_dim;
    long int real_row_dim, real_col_dim; // multiplier of _BNC_D_WIDTH
    double *element[DDSIZE];
} ddmatrix;

typedef *DDMatrix;
```

MPFMatrix

```
typedef struct{
```

```

    unsigned long int prec;
    mpf_t *element;
    long int row_dim, col_dim;
    long int real_row_dim, real_col_dim;
    void *element_block; // mantissa block
} mpfmatrix;

typedef mpfmatrix *MPFMatrix;

```

## 3.2 Vector arithmetic

- `ddfloat get_ddvector_i_ddfloat(DDVector vec, long int index)` ... Get index-th element of `vec` as `ddfloat` datatype.
- `tdfloat get_tdvector_i_tdfloat(TDVector vec, long int index)`
- `qdfloat get_qdvector_i_qdfloat(QDVector vec, long int index)`
- (Macro) `GET_DDVECTOR_I(vec, index)` ... Get index-th element as pointer to `ddfloat`.
- (Macro) `get_ddvector_i(vec, index)`
- (Macro) `GET_TDVECTOR_I(vec, index)` ... Get index-th element as pointer to `tdfloat`.
- (Macro) `get_tdvector_i(vec, index)`
- (Macro) `GET_QDVECTOR_I(vec, index)` ... Get index-th element as pointer to `qdfloat`.
- (Macro) `get_qdvector_i(vec, index)`
- (Macro) `GET_MPFVECTOR_I(vec, index)` ... Get index-th element as pointer to `mpf_t` (`mpfr_t`).
- (Macro) `get_mpfvector_i(vec, index)`
  - `void set_ddvector_i(DDVector vec, long int index, double * val)` ... Store `val` to index-th element of `vec`.
- (Macro) `SET_DDVECTOR_I(vec, index, val)`
  - `void set_tdvector_i(TDVector vec, long int index, double * val)`
- (Macro) `SET_TDVECTOR_I(vec, index, val)`
  - `void set_qdvector_i(QDVector vec, long int index, double * val)`
- (Macro) `SET_QDVECTOR_I(vec, index, val)`
  - `void set_qdvector_i(QDVector vec, long int index, double * val)`
- (Macro) `SET_QDVECTOR_I_D(vec, index, val)`
  - `void set_ddvector_i_d(DDVector vec, long int index, double val)` ... Store `val` to index-th element of `vec` as `double`.
- (Macro) `SET_DDVECTOR_I_D(vec, index, val)`
  - `void set_tdvector_i_d(TDVector vec, long int index, double val)`
- (Macro) `SET_TDVECTOR_I_D(vec, index, val)`
  - `void set_qdvector_i_d(QDVector vec, long int index, double val)`
- (Macro) `SET_QDVECTOR_I_D(vec, index, val)`
  - `void set0_ddvector_i(DDVector vec, long int index)` ... Store zero to index-th element of `vec`.
- (Macro) `SET0_DDVECTOR_I(vec, index)`
  - `void set0_tdvector_i(TDVector vec, long int index)`
- (Macro) `SET0_TDVECTOR_I(vec, index)`
  - `void set0_qdvector_i(QDVector vec, long int index)`
- (Macro) `SET0_QDVECTOR_I(vec, index)`
  - `DDVector init_ddvector(in dimension)` ... Allocate and initialize a vector and set it as zero vector.
  - `TDVector init_tdvector(in dimension)`
  - `QDVector init_qdvector(in dimension)`
  - `MPFVector init_mpfvector(in dimension)` ... Allocate and initialize a `MPFVector` in default precision in bits with `bnc\_set\_default\_prec` and set it as zero vector.

- `MPFVector init2_mpfvector(int dimension, unsigned long prec) ...` Initialize a MPFVector in `prec` bits and set it as zero vector.
- `void free_ddvector(DDVector vec) ...` Free `vec`.
- `void free_tdvector(TDVector vec)`
- `void free_qdvector(QDVector vec)`
- `void free_mpfvector(MPFVector vec)`
- `void set_ddfloat_ddvec(ddfloat ret[], int ret_dim, DDVector vec) ...` Convert `vec` to `ddfloat` array.
- `void set_tdfloat_tdvec(tdfloat ret[], int ret_dim, TDVector vec) ...` Convert `vec` to `tdfloat` array.
- `void set_qdfloat_qdvec(qdfloat ret[], int ret_dim, QDVector vec) ...` Convert `vec` to `qdfloat` array.
- `void set_ddvector_ddfloat(DDVector ret, ddfloat array[], int array_dim) ...` Convert `ddfloat` array to `DDVector` `ret`.
- `void set_tdvector_tdfloat(TDVector ret, tdfloat array[], int array_dim) ...` Convert `tdfloat` array to `TDVector` `ret`.
- `void set_qdvector_qdfloat(QDVector ret, qdfloat array[], int array_dim) ...` Convert `qdfloat` array to `QDVector` `ret`.
- `void print_ddvector(DDVector vec) ...` Print `vec`.
- `void print_tdvector(TDVector vec)`
- `void print_qdvector(QDVector vec)`
- `void print_mpfvector(MPFVector vec)`
- `void set0_ddvector(DDVector vec) ...` Set `vec` as zero vector.
- `void set0_tdvector(TDVector vec)`
- `void set0_qdvector(QDVector vec)`
- `void set0_mpfvector(MPFVector vec)`
- `void set_ddvector_i_str(DDVector vec, long int index, const char * str) ...` Set `*str` expressed in decimal to `index`-th element of `vec`.
- `void set_tdvector_i_str(TDVector vec, long int index, const char * str)`
- `void set_qdvector_i_str(QDVector vec, long int index, const char * str)`
- `void set_mpfvector_i_str(MPFVector vec, long int index, const char * str)`
- `void add_ddvector(DDVector c, DDVector a, DDVector b) ...`  $c := a + b$
- `void add_tdvector(TDVector c, TDVector a, TDVector b)`
- `void add_qdvector(QDVector c, QDVector a, QDVector b)`
- `void add_mpfvector(MPFVector c, MPFVector a, MPFVector b)`
- `void add2_ddvector(DDVector c, DDVector a) ...`  $c := c + a$
- `void add2_tdvector(TDVector c, TDVector a)`
- `void add2_qdvector(TDVector c, QDVector a)`
- `void add2_mpfvector(MPFVector c, MPFVector a)`
- `void sub_ddvector(DDVector c, DDVector a, DDVector b) ...`  $c := a - b$
- `void sub_tdvector(TDVector c, TDVector a, TDVector b)`
- `void sub_qdvector(QDVector c, QDVector a, QDVector b)`
- `void sub_mpfvector(MPFVector c, MPFVector a, MPFVector b)`
- `void sub2_ddvector(DDVector c, DDVector a) ...`  $c -= a$
- `void sub2_tdvector(TDVector c, TDVector a)`
- `void sub2_qdvector(QDVector c, QDVector a)`
- `void sub2_mpfvector(MPFVector c, MPFVector a)`
- `void cmul_ddvector(DDVector c, double val [DDSIZE], DDVector a) ...`  $c := val \times a$
- `void cmul_tdvector(TDVector c, double val [TDSIZE], TDVector a)`
- `void cmul_qdvector(QDVector c, double val [QDSIZE], QDVector a)`
- `void cmul_mpfvector(MPFVector c, mpf_t val, MPFVector a)`
- `void cmul2_ddvector(DDVector c, double val [DDSIZE]) ...`  $c := val \times c$
- `void cmul2_qdvector(TDVector c, double val [TDSIZE])`

- void cmul2\_ddvector(QDVector c, double val [QDSIZE])
- void cmul2\_ddvector(MPFVector c, mpf\_t val)
- void add\_cmul\_ddvector(DDVector c, DDVector a, double val [DDSIZE], DDVector b)  $\cdots c := a + val * b$
- void add\_cmul\_tdvector(TDVector c, TDVector a, double val [TDSIZE], TDVector b)
- void add\_cmul\_qdvector(QDVector c, QDVector a, double val [QDSIZE], QDVector b)
- void add\_cmul\_mpfvector(MPFVector c, MPFVector a, mpf\_t val, MPFVector b)
- void ip\_ddvector(double ret [DDSIZE], DDVector a, DDVector b)  $\cdots$  Calculate the dot product of  $a$  and  $b$
- void ip\_tdvector(double ret [TDSIZE], TDVector a, TDVector b)
- void ip\_qdvector(double ret [QDSIZE], QDVector a, QDVector b)
- void ip\_mpfvector(mpf\_t ret, MPFVector a, MPFVector b)
- void neg\_ddvector(DDVector c, DDVector a)  $\cdots c := -a$
- void neg\_tdvector(TDVector c, TDVector a)
- void neg\_qdvector(QDVector c, QDVector a)
- void neg\_mpfvector(MPFVector c, MPFVector a)
- void norm1\_ddvector(double ret [DDSIZE], DDVector a)  $\cdots \|a\|_1$
- void norm1\_tdvector(double ret [TDSIZE], TDVector a)
- void norm1\_qdvector(double ret [QDSIZE], QDVector a)
- void norm1\_mpfvector(mpf\_t ret, MPFVector a)
- void normi\_ddvector(double ret [DDSIZE], DDVector a)  $\cdots \|a\|_\infty$
- void normi\_tdvector(double ret [TDSIZE], TDVector a)
- void normi\_qdvector(double ret [QDSIZE], QDVector a)
- void normi\_mpfvector(mpf\_t ret, MPFVector a)
- void norm2\_ddvector(double ret [DDSIZE], DDVector vec)  $\cdots \|a\|_2$
- void norm2\_tdvector(double ret [TDSIZE], TDVector vec)
- void norm2\_qdvector(double ret [QDSIZE], QDVector vec)
- void norm2\_mpfvector(mpf\_t ret, MPFVector vec)

### 3.3 Matrix arithmetic

- ddfloat get\_ddmatrix\_ij\_ddfloat(DDMatrix mat, long int i, long int j)  $\cdots$  Get the  $(i, j)$ -th element of mat.
- (Macro) GET\_DDMATRIX\_IJ(mat, i, j) ((get\_ddmatrix\_ij\_ddfloat((mat), (i), (j)).val))
- (Macro) get\_ddmatrix\_ij(mat, i, j) ((get\_ddmatrix\_ij\_ddfloat((mat), (i), (j)).val))
  - tdfloat get\_tdmatrix\_ij\_tdfloat(TDMatrix mat, long int i, long int j)
- (Macro) GET\_TDMATRIX\_IJ(mat, i, j) ((get\_tdmatrix\_ij\_tdfloat((mat), (i), (j)).val))
- (Macro) get\_tdmatrix\_ij(mat, i, j) ((get\_tdmatrix\_ij\_tdfloat((mat), (i), (j)).val))
  - qdfloat get\_qdmatrix\_ij\_qdfloat(QDMatrix mat, long int i, long int j)
- (Macro) GET\_QDMATRIX\_IJ(mat, i, j) ((get\_qdmatrix\_ij\_qdfloat((mat), (i), (j)).val))
- (Macro) get\_qdmatrix\_ij(mat, i, j) ((get\_qdmatrix\_ij\_qdfloat((mat), (i), (j)).val))
- (Macro) GET\_MPFMATRIX\_IJ(mat, i, j) ((get\_mpfmatrix\_ij((mat), (i), (j))))
  - mpf\_ptr get\_mpfmatrix\_ij(MPFMatrix mat, long int i, long int j)
  - void set\_ddmatrix\_ij(DDMatrix mat, long int i, long int j, double \* val)
- (Macro) SET\_DDMATRIX\_IJ(mat, i, j, val) set\_ddmatrix\_ij((mat), (i), (j), (val))
  - void set\_tdmatrix\_ij(TDMatrix mat, long int i, long int j, double \* val)
- (Macro) SET\_TDMATRIX\_IJ(mat, i, j, val) set\_tdmatrix\_ij((mat), (i), (j), (val))
  - void set\_qdmatrix\_ij(QDMatrix mat, long int i, long int j, double \* val)
- (Macro) SET\_QDMATRIX\_IJ(mat, i, j, val) set\_qdmatrix\_ij((mat), (i), (j), (val))
  - void set\_mpfmatrix\_ij(MPFMatrix mat, long int i, long int j, mpf\_t val)
- (Macro) SET\_MPFMATRIX\_IJ(mat, i, j, val) set\_mpfmatrix\_ij((mat), (i), (j), (val))
  - void set\_ddmatrix\_ij\_d(DDMatrix mat, long int i, long int j, double val)

```

(Macro) SET_DDMATRIX_IJ_D(mat, i, j, val) set_ddmatrix_ij_d((mat), (i), (j), (val))
(Macro) SET_DDMATRIX_IJ_UI(mat, i, j, val) set_ddmatrix_ij_d((mat), (i), (j), (double)(val))
(Macro) set_ddmatrix_ij_ui(mat, i, j, val) set_ddmatrix_ij_d((mat), (i), (j), (double)(val))
    • void set0_ddmatrix_ij(DDMatrix mat, long int i, long int j)
(Macro) SET0_DDMATRIX_IJ(mat, i, j) set0_ddmatrix_ij((mat), (i), (j))
    • void set0_ddmatrix(DDMatrix mat) ... set mat as zero matrix
    • void set0_tdmatrix(TDMatrix mat)
    • void set0_qdmatrix(QDMatrix mat)
    • void set0_mpfmatrix(MPFMatrix mat)
    • DDMatrix init_ddmatrix(long int row_dim, long int col_dim) ... Initialize DDMatrix.
    • TDMatrix init_tdmatrix(long int row_dim, long int col_dim) ... Initialize TDMatrix.
    • QDMatrix init_qdmatrix(long int row_dim, long int col_dim) ... Initialize QDMatrix.
    • MPFMatrix init_mpfmatrix(long int row_dim, long int col_dim) ... Initialize MPFMatrix.
    • MPFMatrix init2_mpfmatrix(long int row_dim, long int col_dim, unsigned long prec) ... Initialize
      MPFMatrix as prec-bit precision.
    • void free_ddmatrix(DDMatrix mat) ... Free mat
    • void free_tdmatrix(TDMatrix mat)
    • void free_qdmatrix(QDMatrix mat)
    • void free_mpfmatrix(MPFMatrix mat)
    • void print_ddmatrix(DDMatrix mat) ... Print mat.
    • void print_tdmatrix(TDMatrix mat)
    • void print_qdmatrix(QDMatrix mat)
    • void print_mpfmatrix(MPFMatrix mat)
    • void set_ddfloat_ddmat(ddfloat ret[], int ret_dim, DDMatrix mat) ... Convert DDMatrix mat
      to ddfloat array.
    • void set_tdfloat_tdmatrix(tdfloat ret[], int ret_dim, DDMatrix mat) ... Convert TDMatrix mat
      to tdfloat array.
    • void set_qdfloat_qdmat(qdfloat ret[], int ret_dim, DDMatrix mat) ... Convert QDMatrix mat
      to qdfloat array.
    • void set_ddmatrix_ddfloat(DDMatrix ret, ddfloat array[], int array_dim) ... Convert ddfloat
      array to DDMatrix.
    • void set_tdmatrix_tdfloat(TDMatrix ret, tdfloat array[], int array_dim) ... Convert tdfloat
      array to TDMatrix.
    • void set_qdmatrix_qdfloat(QDMatrix ret, qdfloat array[], int array_dim) ... Convert qdfloat
      array to QDMatrix
    • void mul_ddmatrix(DDMatrix ret, DDMatrix a, DDMatrix b) ... Simple matrix multiplication.
    • void mul_tdmatrix(TDMatrix ret, TDMatrix a, TDMatrix b)
    • void mul_qdmatrix(QDMatrix ret, QDMatrix a, QDMatrix b)
    • void mul_mpfmatrix(MPFMatrix ret, MPFMatrix a, MPFMatrix b)
    • void normf_ddmatrix(double ret[DDSIZE], DDMatrix mat) ... Get the value of Frobenius norm
      of mat.
    • void normf_tdmatrix(double ret[TDSIZE], TDMatrix mat)
    • void normf_qdmatrix(double ret[QDSIZE], QDMatrix mat)
    • void normf_mpfmatrix(mpf_t ret, MPFMatrix mat)
    • void print_normf_ddmatrix(const char * str, DDMatrix mat) ... print the Frobenius norm of
      mat.
    • void normi_ddmatrix(double ret[DDSIZE], DDMatrix mat) ... Infinity norm of matrix.
    • void normi_tdmatrix(double ret[TDSIZE], TDMatrix mat)
    • void normi_qdmatrix(double ret[QDSIZE], QDMatrix mat)
    • void normi_mpfmatrix(mpf_t ret, TDMatrix mat)
    • void norm1_ddmatrix(double ret[DDSIZE], DDMatrix mat) ... 1-norm of matrix.
    • void norm1_tdmatrix(double ret[TDSIZE], TDMatrix mat)
    • void norm1_qdmatrix(double ret[QDSIZE], QDMatrix mat)

```

- void norm1\_mpfmatrix(mpf\_t ret, MPFMatrix mat)
- void add\_ddmatrix(DDMatrix c, DDMatrix a, DDMatrix b)  $\cdots c := a + b$
- void add\_tdmatrix(TDMatrix c, TDMatrix a, TDMatrix b)
- void add\_qdmatrix(QDMatrix c, QDMatrix a, QDMatrix b)
- void add\_mpfmatrix(MPFMatrix c, MPFMatrix a, MPFMatrix b)
- void sub\_ddmatrix(DDMatrix c, DDMatrix a, DDMatrix b)  $\cdots c := a - b$
- void sub\_tdmatrix(TDMatrix c, TDMatrix a, TDMatrix b)
- void sub\_qdmatrix(QDMatrix c, QDMatrix a, QDMatrix b)
- void sub\_mpfmatrix(MPFMatrix c, MPFMatrix a, MPFMatrix b)
- void cmul\_ddmatrix(DDMatrix c, double sc[DDSIZE], DDMatrix a)  $\cdots c := sc \times a$
- void cmul\_tdmatrix(TDMatrix c, double sc[TDSIZE], TDMatrix a)
- void cmul\_qdmatrix(QDMatrix c, double sc[QDSIZE], QDMatrix a)
- void cmul\_mpfmatrix(MPFMatrix c, mpf\_t sc, MPFMatrix a)
- void transpose\_ddmatrix(DDMatrix c, DDMatrix a)  $\cdots c := a^T$
- void transpose\_tdmatrix(TDMatrix c, TDMatrix a)
- void transpose\_qdmatrix(QDMatrix c, QDMatrix a)
- void transpose\_mpfmatrix(MPFMatrix c, MPFMatrix a)
- void subst\_ddmatrix(DDMatrix c, DDMatrix a)  $\cdots c := a$
- void subst\_tdmatrix(TDMatrix c, TDMatrix a)
- void subst\_qdmatrix(QDMatrix c, QDMatrix a)
- void subst\_mpfmatrix(MPFMatrix c, MPFMatrix a)
- void setI\_ddmatrix(DDMatrix c)  $\cdots c := I$
- void setI\_tdmatrix(TDMatrix c)
- void setI\_qdmatrix(QDMatrix c)
- void setI\_mpfmatrix(MPFMatrix c)
- void mul\_ddmatrix\_ddvec(DDVector v, DDMatrix a, DDVector vb)  $\cdots v := a * vb$
- void mul\_tdmatrix\_tdvec(TDVector v, TDMatrix a, TDVector vb)
- void mul\_qdmatrix\_qdvec(QDVector v, QDMatrix a, QDVector vb)
- void mul\_mpfmatrix\_mpfvec(MPFVector v, MPFMatrix a, MPFVector vb)
- void mul\_ddmatrixt\_ddvec(DDVector v, DDMatrix a, DDVector vb)  $\cdots v := a^T * vb$
- void mul\_tdmatrixt\_tdvec(TDVector v, TDMatrix a, TDVector vb)
- void mul\_qdmatrixt\_qdvec(QDVector v, QDMatrix a, QDVector vb)
- void mul\_mpfmatrixt\_mpfvec(MPFVector v, MPFMatrix a, MPFVector vb)
- void inv\_ddmatrix(DDMatrix a)  $\cdots a := a^{-1}$  (only for square matrix)
- void inv\_tdmatrix(TDMatrix a)
- void inv\_qdmatrix(QDMatrix a)
- void inv\_mpfmatrix(MPFMatrix a)
- void subst\_mpfvector\_ddvec(MPFVector c, DDVector a)  $\cdots c := (mpf)a$
- void subst\_ddvector\_mpfvec(DDVector c, MPFVector MPFVector a)  $\cdots c := (dd)a$
- void subst\_mpfmatrix\_ddmat(MPFMatrix c, DDMatrix a)  $\cdots c := (mpf)a$
- void subst\_ddmatrix\_mpfmat(DDMatrix c, MPFMatrix a)  $\cdots c := (dd)a$
- void relerr\_ddvector\_mpfvec(double relerr[DDSIZE], DDVector approx\_vec, MPFVector true\_vec, int norm\_type)  $\cdots$  Norm relative error of vector.
- void relerr\_element\_ddvector\_mpf(double max\_relerr[DDSIZE], double min\_relerr[DDSIZE], double norm\_relerr[DDSIZE], DDVector approx\_vec, MPFVector true\_vec, int norm\_type)  $\cdots$  Elementwise relative errors of vector.  
/\* c := (dd)a \*/
- void subst\_ddvector\_dvec(DDVector c, DVector sa)  $\cdots c := (dd)a$   
/\* c := (d)a \*/
- void subst\_dvector\_ddvec(DVector c, DDVector a)  $\cdots c := (double)a$   
/\* c := (dd)a \*/
- void subst\_ddmatrix\_dmat(DDMatrix c, DMatrix a)  $\cdots c := (dd)a$



- `/* c := (d)a */`
- `void subst_dmatrix_ddmat(DMatrix c, DDMatrix a) ... c := (double)a`
- `void relerr_ddvector(double relerr[DDSIZE], DDVector approx_vec, DDVector true_vec, int norm_type) ...` Norm relative error of vector.
- `void relerr_element_ddvector(double max_relerr[DDSIZE], double min_relerr[DDSIZE], double norm_relerr[DDSIZE], DDVector approx_vec, DDVector true_vec, int norm_type) ...` Elementwise relative errors of vector.
- `void row_swap_ddmatrix(DDMatrix mat, long int row_index0, long int row_index1, long int col_start, long int col_end) ...` Exchange a(row\_index0, col\_start:col\_end) to a(row\_index1, col\_start:col\_end)

### 3.4 File I/O with matrix and vector

- `void fread_ddmatrix(FILE * fp, DDMatrix mat) ...` Read matrix elements from fp and store theme in mat.
- `void fread_tdmatrix(FILE * fp, TDMatrix mat)`
- `void fread_qdmatrix(FILE * fp, QDMatrix mat)`
- `void fread_mpfmatrix(FILE * fp, MPFMatrix mat)`
- `void fread_ddmatrix_fname(const char * fname, DDMatrix mat) ...` Read matrix elements from fname and store theme in mat.
- `void fread_tdmatrix_fname(const char * fname, TDMatrix mat)`
- `void fread_qdmatrix_fname(const char * fname, QDMatrix mat)`
- `void fread_mpfmatrix_fname(const char * fname, MPFMatrix mat)`
- `void fwrite_ddmatrix(FILE * fp, DDMatrix mat) ...` Write matrix elements of mat to fp.
- `void fwrite_tdmatrix(FILE * fp, TDMatrix mat)`
- `void fwrite_qdmatrix(FILE * fp, QDMatrix mat)`
- `void fwrite_mpfmatrix(FILE * fp, MPFMatrix mat)`
- `void fwrite_ddmatrix_fname(const char * fname, DDMatrix mat) ...` Write matrix elements of mat to fname.
- `void fwrite_tdmatrix_fname(const char * fname, TDMatrix mat)`
- `void fwrite_qdmatrix_fname(const char * fname, QDMatrix mat)`
- `void fwrite_mpfmatrix_fname(const char * fname, MPFMatrix mat)`
- `void fread_ddvector(FILE * fp, DDVector vec) ...` Read vector elements from fp and store theme in vec.
- `void fread_tdvector(FILE * fp, TDVector vec)`
- `void fread_qdvector(FILE * fp, QDVector vec)`
- `void fread_mpfvector(FILE * fp, MPFVector vec)`
- `void fread_ddvector_fname(const char * fname, DDVector vec) ...` Read vector elements from fname and store theme in vec.
- `void fread_tdvector_fname(const char * fname, TDVector vec)`
- `void fread_qdvector_fname(const char * fname, QDVector vec)`
- `void fread_mpfvector_fname(const char * fname, MPFVector vec)`
- `void fwrite_ddvector(FILE * fp, DDVector vec) ...` Write vector elements of vec to fp.
- `void fwrite_tdvector(FILE * fp, TDVector vec)`
- `void fwrite_qdvector(FILE * fp, QDVector vec)`
- `void fwrite_mpfvector(FILE * fp, MPFVector vec)`
- `void fwrite_ddvector_fname(const char * fname, DDVector vec) ...` Write vector elements of vec to fname.
- `void fwrite_tdvector_fname(const char * fname, TDVector vec)`
- `void fwrite_qdvector_fname(const char * fname, QDVector vec)`
- `void fwrite_mpfvector_fname(const char * fname, MPFVector vec)`
- `void read_test_linear_eq_dd(DDMatrix A, DDVector true_x, DDVector b, long int dim, const`

- char \* fname\_A, const char \* fname\_true\_x, const char \* fname\_b) ... Read the coefficient matrix A from fname\_A, the true\_x from fname\_true\_x, and the vector b from fname\_b.
- void read\_test\_linear\_eq\_td(TDMatrix A, TDVector true\_x, TDVector b, long int dim, const char \* fname\_A, const char \* fname\_true\_x, const char \* fname\_b)
- void read\_test\_linear\_eq\_qd(QDMatrix A, QDVector true\_x, QDVector b, long int dim, const char \* fname\_A, const char \* fname\_true\_x, const char \* fname\_b)
- void read\_test\_linear\_eq\_mpf(MPFMatrix A, MPFVector true\_x, MPFVector b, long int dim, const char \* fname\_A, const char \* fname\_true\_x, const char \* fname\_b)

### 3.5 Generating test matrices

The following functions provide various test matrices for benchmark tests.

- void hilbert\_ddmatrix(DDMatrix a, long int dim) ... Hilbert matrix.
- void hilbert\_tdmatrix(TDMatrix a, long int dim)
- void hilbert\_qdmatrix(QDMatrix a, long int dim)
- void hilbert\_mpfmatrix(MPFMatrix a, long int dim)
- void lotkin\_ddmatrix(DDMatrix a, long int dim) ... Lotkin matrix.
- void lotkin\_tdmatrix(TDMatrix a, long int dim)
- void lotkin\_qdmatrix(QDMatrix a, long int dim)
- void lotkin\_mpfmatrix(MPFMatrix a, long int dim)
- void frank\_ddmatrix(DDMatrix a, long int dim) ... Symmetric Frank matrix.
- void frank\_tdmatrix(TDMatrix a, long int dim)
- void frank\_qdmatrix(QDMatrix a, long int dim)
- void frank\_mpfmatrix(MPFMatrix a, long int dim)
- void tridiag\_ddmatrix(DDMatrix a, DDVector low\_subdiag, DDVector diag, DDVector up\_subdiag, long int dim) ... Triagonal matrix.
- void tridiag\_tdmatrix(DDMatrix a, TDVector low\_subdiag, TDVector diag, TDVector up\_subdiag, long int dim)
- void tridiag\_qdmatrix(DDMatrix a, QDVector low\_subdiag, QDVector diag, QDVector up\_subdiag, long int dim)
- void tridiag\_mpfmatrix(DDMatrix a, MPFVector low\_subdiag, MPFVector diag, MPFVector up\_subdiag, long int dim)
- void int\_sym\_rand\_ddmatrix(DDMatrix mat, long int max, long int seed, long int dim) ... Integer Symmetrix Random Matrix.
- void int\_sym\_rand\_tdmatrix(TDMatrix mat, long int max, long int seed, long int dim)
- void int\_sym\_rand\_qdmatrix(QDMatrix mat, long int max, long int seed, long int dim)
- void int\_sym\_rand\_mpfmatrix(MPFMatrix mat, long int max, long int seed, long int dim)
- void int\_unsym\_rand\_ddmatrix(DDMatrix mat, long int max, long int seed, long int dim) ... Integer Unsymmetrix Random Matrix.
- void int\_unsym\_rand\_tdmatrix(TDMatrix mat, long int max, long int seed, long int dim) ... Integer Unsymmetrix Random Matrix.
- void int\_unsym\_rand\_qdmatrix(QDMatrix mat, long int max, long int seed, long int dim) ... Integer Unsymmetrix Random Matrix.
- void int\_unsym\_rand\_mpfmatrix(MPFMatrix mat, long int max, long int seed, long int dim) ... Integer Unsymmetrix Random Matrix.
- void diag\_ddmatrix(DDMatrix mat, DDVector diag, long int dim) ... Real Diagonal Matrix.
- void diag\_tdmatrix(TDMatrix mat, TDVector diag, long int dim)
- void diag\_qdmatrix(QDMatrix mat, QDVector diag, long int dim)
- void diag\_mpfmatrix(MPFMatrix mat, MPFVector diag, long int dim)
- void toeplitz\_ddmatrix(DDMatrix mat, double gamma\_param[DDSIZE], long int dim) ... Toeplitz matrix.

- void toeplitz\_tdmatrix(TDMatrix mat, double gamma\_param[TDSIZE], long int dim)
- void toeplitz\_qdmatrix(QDMatrix mat, double gamma\_param[QDSIZE], long int dim)
- void toeplitz\_mpfmatrix(MPFMatrix mat, mpf\_t gamma\_param, long int dim)
- void pascal\_ddmatrix(DDMatrix ret, long int dim) ... Pascal matrix.
- void pascal\_tdmatrix(TDMatrix ret, long int dim)
- void pascal\_qdmatrix(QDMatrix ret, long int dim)
- void pascal\_mpfmatrix(MPFMatrix ret, long int dim)
- void im\_rand\_ddmatrix(DDMatrix ret, unsigned long seed) ...  $I - random$
- void im\_rand\_tdmatrix(TDMatrix ret, unsigned long seed)
- void im\_rand\_qdmatrix(QDMatrix ret, unsigned long seed)
- void im\_rand\_mpfmatrix(MPFMatrix ret, unsigned long seed)

### 3.6 Simple, block, and Strassen matrix multiplication and related functions

- (Macro) mul\_ddmatrix\_simple(c, a, b) ... mul\_ddmatrixc, a, b ... Matrix multiplication based on simple triple-loop way
- (Macro) mul\_tdmatrix\_simple(c, a, b) ... mul\_tdmatrixc, a, b
- (Macro) mul\_qdmatrix\_simple(c, a, b) ... mul\_qdmatrixc, a, b
- void mul\_mpfmatrix\_simple(MPFMatrix c, MPFMatrix a, MPFMatrix b)
  - void mul\_dmatrix\_strassen(DMatrix ret, DMatrix mat\_a, DMatrix mat\_b, long int min\_dim) ... Matrix multiplication based on Strassen or Winograd algorithms
  - void mul\_ddmatrix\_strassen(DDMatrix ret, DDMatrix mat\_a, DDMatrix mat\_b, long int min\_dim)
  - void mul\_tdmatrix\_strassen(TDMatrix ret, TDMatrix mat\_a, TDMatrix mat\_b, long int min\_dim)
  - void mul\_qdmatrix\_strassen(QDMatrix ret, QDMatrix mat\_a, QDMatrix mat\_b, long int min\_dim)
  - void mul\_mpfmatrix\_strassen(MPFMatrix ret, MPFMatrix mat\_a, MPFMatrix mat\_b, long int min\_dim)
  - void mul\_dmatrix\_block(DMatrix ret, DMatrix mat\_a, DMatrix mat\_b, long int min\_dim) ... Block matrix multiplication
  - void mul\_ddmatrix\_block(DDMatrix ret, DDMatrix mat\_a, DDMatrix mat\_b, long int min\_dim)
  - void mul\_tdmatrix\_block(TDMatrix ret, TDMatrix mat\_a, TDMatrix mat\_b, long int min\_dim)
  - void mul\_qdmatrix\_block(QDMatrix ret, QDMatrix mat\_a, QDMatrix mat\_b, long int min\_dim)
  - void mul\_mpfmatrix\_block(MPFMatrix ret, MPFMatrix mat\_a, MPFMatrix mat\_b, long int min\_dim)
  - void mul\_dmatrix\_strassen\_even(DMatrix ret, DMatrix mat\_a, DMatrix mat\_b, long int min\_dim) ... Strassen's Algorithm (even dimension)
  - void mul\_ddmatrix\_strassen\_even(DDMatrix ret, DDMatrix mat\_a, DDMatrix mat\_b, long int min\_dim)
  - void mul\_tdmatrix\_strassen\_even(TDMatrix ret, TDMatrix mat\_a, TDMatrix mat\_b, long int min\_dim)
  - void mul\_qdmatrix\_strassen\_even(QDMatrix ret, QDMatrix mat\_a, QDMatrix mat\_b, long int min\_dim)
  - void mul\_mpfmatrix\_strassen\_even(MPFMatrix ret, MPFMatrix mat\_a, MPFMatrix mat\_b, long int min\_dim)
  - void mul\_dmatrix\_winograd\_even(DMatrix ret, DMatrix mat\_a, DMatrix mat\_b, long int min\_dim) ... Winograd Variant of Strassen's Algorithm
  - void mul\_ddmatrix\_winograd\_even(DDMatrix ret, DDMatrix mat\_a, DDMatrix mat\_b, long int min\_dim)
  - void mul\_tdmatrix\_winograd\_even(TDMatrix ret, TDMatrix mat\_a, TDMatrix mat\_b, long int min\_dim)

- void mul\_qdmatrix\_winograd\_even(QDMatrix ret, QDMatrix mat\_a, QDMatrix mat\_b, long int min\_dim)
- void mul\_mpfmatrix\_winograd\_even(MPFMatrix ret, MPFMatrix mat\_a, MPFMatrix mat\_b, long int min\_dim)
- void inv\_ddmatrix\_strassen\_even(DDMatrix ret, DDMatrix mat\_a, long int min\_dim) ... Computation of Inverse Matrix by using Strassen's Algorithm
- void inv\_tdmatrix\_strassen\_even(TDMatrix ret, TDMatrix mat\_a, long int min\_dim)
- void inv\_qdmatrix\_strassen\_even(QDMatrix ret, QDMatrix mat\_a, long int min\_dim)

### 3.6.1 OpenMP

- (Macro) `_bncomp_mul_ddmatrix_simple(c, a, b) _bncomp_mul_ddmatrix(c, a, b)` ... Parallelized simple matrix multiplication,
- (Macro) `_bncomp_mul_tdmatrix_simple(c, a, b) _bncomp_mul_tdmatrix(c, a, b)`
- (Macro) `_bncomp_mul_qdmatrix_simple(c, a, b) _bncomp_mul_qdmatrix(c, a, b)`
- (Macro) `_bncomp_mul_mpfmatrix_simple(c, a, b) _bncomp_mul_mpfmatrix(c, a, b)`
- void `_bncomp_mul_ddmatrix_block`(DDMatrix ret, DDMatrix mat\_a, DDMatrix mat\_b, long int min\_dim) ... Parallelized block matrix multiplication.
- void `_bncomp_mul_tdmatrix_block`(TDMatrix ret, TDMatrix mat\_a, TDMatrix mat\_b, long int min\_dim)
- void `_bncomp_mul_qdmatrix_block`(QDMatrix ret, QDMatrix mat\_a, QDMatrix mat\_b, long int min\_dim)
- void `_bncomp_mul_mpfmatrix_block`(MPFMatrix ret, MPFMatrix mat\_a, MPFMatrix mat\_b, long int min\_dim)
- void `_bncomp_mul_ddmatrix_strassen`(DDMatrix ret, DDMatrix mat\_a, DDMatrix mat\_b, long int min\_dim) ... Parallelized Strassen and Winograd matrix multiplication (limited performance due to poor coding)
- void `_bncomp_mul_tdmatrix_strassen`(TDMatrix ret, TDMatrix mat\_a, TDMatrix mat\_b, long int min\_dim)
- void `_bncomp_mul_qdmatrix_strassen`(QDMatrix ret, QDMatrix mat\_a, QDMatrix mat\_b, long int min\_dim)
- void `_bncomp_mul_mpfmatrix_strassen`(MPFMatrix ret, MPFMatrix mat\_a, MPFMatrix mat\_b, long int min\_dim)

## 3.7 Getting relative errors of vector and matrix

- void `relerr3_dvector`(double \* max\_relerr, double \* min\_relerr, double \* norm\_relerr, DVector vec, DVector vec\_true, int kind\_of\_norm) ... Relative errors of vec.
- void `relerr3_ddvector`(double max\_relerr[DDSIZE], double min\_relerr[DDSIZE], double norm\_relerr[DDSIZE], DDVector vec, DDVector vec\_true, int kind\_of\_norm)
- void `relerr3_tdvector`(double max\_relerr[TDSIZE], double min\_relerr[TDSIZE], double norm\_relerr[TDSIZE], TDVector vec, TDVector vec\_true, int kind\_of\_norm)
- void `relerr3_qdvector`(double max\_relerr[QDSIZE], double min\_relerr[QDSIZE], double norm\_relerr[QDSIZE], QDVector vec, QDVector vec\_true, int kind\_of\_norm)
- void `relerr3_mpfvector`(mpf\_t max\_relerr, mpf\_t min\_relerr, mpf\_t norm\_relerr, MPFVector vec, MPFVector vec\_true, int kind\_of\_norm)
- void `relerr3_dmatrix`(double \* max\_relerr, double \* min\_relerr, double \* norm\_relerr, DMatrix mat, DMatrix mat\_true, int kind\_of\_norm) ... Relative errors of mat.
- void `relerr3_ddmatrix`(double max\_relerr[DDSIZE], double min\_relerr[DDSIZE], double norm\_relerr[DDSIZE], DDMatrix mat, DDMatrix mat\_true, int kind\_of\_norm)
- void `relerr3_tdmatrix`(double max\_relerr[TDSIZE], double min\_relerr[TDSIZE], double

- `norm_relerr[TDSIZE], TDMatrix mat, TDMatrix mat_true, int kind_of_norm)`
- `void relerr3_qdmatrix(double max_relerr[QDSIZE], double min_relerr[QDSIZE], double norm_relerr[QDSIZE], QDMatrix mat, QDMatrix mat_true, int kind_of_norm)`
- `void relerr3_mpfmatrix(mpf_t max_relerr, mpf_t min_relerr, mpf_t norm_relerr, MPFMatrix mat, MPFMatrix mat_true, int kind_of_norm)`

### 3.8 Ozaki scheme and related functions

These following functions are necessary for Ozaki scheme and multi-fold precision arithmetic.

- `void extract_dvector(DVector ret_high_vec, DVector ret_low_vec, DVector org_vec, double num_bits) ...  $ret\_high + ret\_low = org\_vec$`
- `void split_dmatrix(DMatrix ret_high_mat, DMatrix ret_low_mat, DMatrix org_mat) ... split  $org\_mat$  to  $ret\_high\_mat$  and  $ret\_low\_mat$  in row direction`
- `void split_dmatrix_t(DMatrix ret_high_mat, DMatrix ret_low_mat, DMatrix org_mat) ... split  $org\_mat$  to  $ret\_high\_mat$  and  $ret\_low\_mat$  in column direction`
- `int split_ddvector_dvec(DVector ret_vec[], int num_div, DDVector org_vec) ... Split  $org\_vec$  to  $ret\_vec[0] + ret\_vec[1] + \dots + ret\_vec[num\_div - 1]$`
- `int split_tdvector_dvec(DVector ret_vec[], int num_div, TDVector org_vec)`
- `void absmax_ddvector(double ret[DDSIZE], long int * max_index, DDVector vec) ... get absolutely maximum of elements in vec.`
- `void absmax_tdvector(double ret[TDSIZE], long int * max_index, TDVector vec)`
- `void absmax_qdvector(double ret[QDSIZE], long int * max_index, QDVector vec)`
- `void absmax_mpfvector(mpf_t ret, long int * max_index, MPFVector vec)`
- `void add_ddvector_dvec(DDVector c, DDVector a, DVector b) ...  $c := a + (double)b$`
- `void add_tdvector_dvec(TDVector c, TDVector a, DVector b)`
- `void add_qdvector_dvec(QDVector c, QDVector a, DVector b)`
- `void add_mpfvector_dvec(MPFVector c, MPFVector a, DVector b) ;`
- `void sub_ddvector_dvec(DDVector c, DDVector a, DVector b) ...  $c := a - (double)b$`
- `void sub_tdvector_dvec(TDVector c, TDVector a, DVector b)`
- `void sub_qdvector_dvec(QDVector c, QDVector a, DVector b)`
- `void sub_mpfvector_dvec(MPFVector c, MPFVector a, DVector b)`
- `void subst_dvector_qdvec(DVector c, QDVector a) ...  $c := (DDVector)a$`
- `void absmax_row_ddmatrix(double mu[DDSIZE], long int * max_j, long int row_index, DDMatrix mat) ... Return the absolutely maximum element in the  $max\_j$ -th row.`
- `void absmax_row_tdmatrix(double mu[TDSIZE], long int * max_j, long int row_index, TDMatrix mat)`
- `void absmax_row_qdmatrix(double mu[QDSIZE], long int * max_j, long int row_index, QDMatrix mat)`
- `void absmax_row_mpfmatrix(mpf_t mu, long int * max_j, long int row_index, MPFMatrix mat)`
- `void add_ddmatrix_dmat(DDMatrix c, DDMatrix a, DMatrix b) ...  $c := a + (DMatrix)b$`
- `void add_tdmatrix_dmat(TDMatrix c, TDMatrix a, DMatrix b)`
- `void add_qdmatrix_dmat(QDMatrix c, QDMatrix a, DMatrix b)`
- `void add_mpfmatrix_dmat(MPFMatrix c, MPFMatrix a, DMatrix b)`
- `void sub_ddmatrix_dmat(DDMatrix c, DDMatrix a, DMatrix b) ...  $c := a - (DMatrix)b$`
- `void sub_tdmatrix_dmat(TDMatrix c, TDMatrix a, DMatrix b)`
- `void sub_qdmatrix_dmat(QDMatrix c, QDMatrix a, DMatrix b)`
- `void sub_mpfmatrix_dmat(MPFMatrix c, MPFMatrix a, DMatrix b)`
- `int split_ddmatrix_dmat(DMatrix ret_mat[], int num_div, DDMatrix org_mat) ... Split  $org\_mat$  to  $ret\_mat[]$  in row direction and return real number of divisions`
- `int split_tdmatrix_dmat(DMatrix ret_mat[], int num_div, TDMatrix org_mat)`
- `int split_qdmatrix_dmat(DMatrix ret_mat[], int num_div, QDMatrix org_mat)`

- `int split_mpfmatrix_dmat(DMatrix ret_mat[], int num_div, MPFMatrix org_mat)`
- `void absmax_col_ddmatrix(double mu[DDSIZE], long int * max_i, long int col_index, DDMatrix mat)` ... return absolutely maximum element and its  $(i, j)$ -index in each column of mat.
- `void absmax_col_tdmatrix(double mu[TDSIZE], long int * max_i, long int col_index, TDMatrix mat)`
- `void absmax_col_qdmatrix(double mu[QDSIZE], long int * max_i, long int col_index, QDMatrix-mat)`
- `void absmax_col_mpfmatrix(mpf_t mu, long int * max_i, long int col_index, MPFMatrix mat)`
- `int split_ddmatrix_t_dmat(DMatrix ret_mat[], int num_div, DDMatrix org_mat)` ... Split `org_mat` to `ret_mat[]` in column direction and return real number of divisions.
- `int split_tdmatrix_t_dmat(DMatrix ret_mat[], int num_div, TDMatrix org_mat)`
- `int split_qdvector_dvec(DVector ret_vec[], int num_div, QDVector org_vec)`
- `int split_qdmatrix_t_dmat(DMatrix ret_mat[], int num_div, QDMatrix org_mat)`
- `int split_mpfmatrix_t_dmat(DMatrix ret_mat[], int num_div, MPFMatrix org_mat)`
- `void subst_qdmatrix_dmat(QDMatrix c, DMatrix a)`
- `void subst_dmatrix_qdmat(DMatrix c, QDMatrix a)`
- `void mul_ddmatrix_oz(DDMatrix ret, DDMatrix a, int max_num_div_a, DDMatrix b, int max_num_div_b)` *cdots* Matrix multiplication based on Ozaki scheme
- `void mul_tdmatrix_oz(TDMatrix ret, TDMatrix a, int max_num_div_a, TDMatrix b, int max_num_div_b)`
- `void mul_qdmatrix_oz(QDMatrix ret, QDMatrix a, int max_num_div_a, QDMatrix b, int max_num_div_b)`
- `void mul_mpfmatrix_oz(MPFMatrix ret, MPFMatrix a, int max_num_div_a, MPFMatrix b, int max_num_div_b)`
- `void mul_ddmatrix_ddvec_oz(DDVector ret, DDMatrix a, int max_num_div_a, DDVector vb, int max_num_div_vb)` ... Matrix-Vector multiplication based on Ozaki scheme
- `void mul_tdmatrix_tdvec_oz(TDVector ret, TDMatrix a, int max_num_div_a, TDVector vb, int max_num_div_vb)`
- `void mul_qdmatrix_qdvec_oz(QDVector ret, QDMatrix a, int max_num_div_a, QDVector vb, int max_num_div_vb)`
- `void mul_mpfmatrix_dvec_oz(MPFVector ret, MPFMatrix a, int max_num_div_a, MPFVector vb, int max_num_div_vb)`

## Chapter 4

# LU decomposition

BNCmatmul has various LU decompositions and solvers for the corresponding decomposed linear system of equations.

### 4.1 List of functions

- `int DLUdecomp(DMatrix A)` ... LU decompose the matrix  $A$  without any types of pivoting.
- `int DDLUdecomp(DDMatrix A)`
- `int TDLUdecomp(TDMatrix A)`
- `int QDLUdecomp(QDMatrix A)`
- `int MPFLUdecomp(MPFMatrix A)`
- `int SolveDLS(DVector x, DMatrix LU, DVector b)` ... Solve  $(LU)\mathbf{x} = \mathbf{b}$  for  $\mathbf{x}$ .
- `int SolveDDLSP(DDVector x, DDMatrix LU, DDVector b)`
- `int SolveTDLSP(TDVector x, TDMatrix LU, TDVector b)`
- `int SolveQDLSP(QDVector x, QDMatrix LU, QDVector b)`
- `int SolveMPFLSP(MPFVector x, MPFMatrix LU, MPFVector b)`
- `int DLUdecompP(DMatrix A, long int ch[])` ... LU decompose the matrix  $A$  with partial pivoting, order of rows is stored in `ch[]`.
- `int DDLUdecompP(DDMatrix A, long int ch[])`
- `int TDLUdecompP(TDMatrix A, long int ch[])`
- `int QDLUdecompP(QDMatrix A, long int ch[])`
- `int MPFLUdecompP(MPFMatrix A, long int ch[])`
- `int SolveDLSP(DVector x, DMatrix LU, DVector b, long int ch[])` ... Solve  $(LU)\mathbf{x} = \mathbf{b}$  for  $\mathbf{x}$  using `ch[]` as row numbering.
- `int SolveDDLSP(DDVector x, DDMatrix LU, DDVector b, long int ch[])`
- `int SolveTDLSP(TDVector x, TDMatrix LU, TDVector b, long int ch[])`
- `int SolveQDLSP(QDVector x, QDMatrix LU, QDVector b, long int ch[])`
- `int SolveMPFLSP(MPFVector x, MPFMatrix LU, MPFVector b, long int ch[])`
- `int DLUdecompC(DMatrix A, long int row_ch[], long int col_ch[])` ... LU decompose the matrix  $A$  with complete pivoting, order of rows is stored in `row_ch[]`, and order of columns in `col_ch[]`.
- `int DDLUdecompC(DDMatrix A, long int row_ch[], long int col_ch[])`
- `int TDLUdecompC(TDMatrix A, long int row_ch[], long int col_ch[])`
- `int QDLUdecompC(QDMatrix A, long int row_ch[], long int col_ch[])`
- `int MPFLUdecompC(MPFMatrix A, long int row_ch[], long int col_ch[])`
- `int SolveDLSC(DVector x, DMatrix LU, DVector b, long int row_ch[], long int col_ch[])` ... Solve  $(LU)\mathbf{x} = \mathbf{b}$  for  $\mathbf{x}$  using `row_ch[]` and `col_ch[]` as row and column numbering, respectively.
- `int SolveDDLSC(DDVector x, DDMatrix LU, DDVector b, long int row_ch[], long int col_ch[])`
- `int SolveTDLSC(TDVector x, TDMatrix LU, TDVector b, long int row_ch[], long int col_ch[])`
- `int SolveQDLSC(QDVector x, QDMatrix LU, QDVector b, long int row_ch[], long int col_ch[])`
- `int SolveMPFLSC(MPFVector x, MPFMatrix LU, MPFVector b, long int row_ch[], long int col_ch[])`

- `int DLUdecomp_strassen(DMatrix A, long int min_dim)` ... LU decompose, using Strassen matrix multiplication.
- `int DDLUdecomp_strassen(DDMatrix A, long int min_dim)`
- `int TDLUdecomp_strassen(TDMatrix A, long int min_dim)`
- `int QDLUdecomp_strassen(QDMatrix A, long int min_dim)`
- `int MPFLUdecomp_strassen(MPFMatrix A, long int min_dim)`
- `int DLUdecomp_strassenPM(DMatrix A, long int ch[], long int min_dim)` ... LU decompose, using Strassen matrix multiplication, the matrix  $A$  with partial pivoting, order of rows is stored in `ch[]`.
- `int DDLUdecomp_strassenPM(DDMatrix A, long int ch[], long int min_dim)`
- `int TDLUdecomp_strassenPM(TDMatrix A, long int ch[], long int min_dim)`
- `int QDLUdecomp_strassenPM(QDMatrix A, long int ch[], long int min_dim)`
- `int MPFLUdecomp_strassenPM(MPFMatrix A, long int ch[], long int min_dim)`
- `int DLUdecomp_oz(DMatrix A, long int min_dim, int max_num_div)` ... LU decompose, using Ozaki scheme setting `max_num_div` as maximum number of divisions.
- `int DDLUdecomp_oz(DDMatrix A, long int min_dim, int max_num_div)`
- `int TDLUdecomp_oz(DMatrix A, long int min_dim, int max_num_div)`
- `int QDLUdecomp_oz(DMatrix A, long int min_dim, int max_num_div)`
- `int MPFLUdecomp_oz(MPFMatrix A, long int min_dim, int max_num_div)`
- `int DLUdecomp_ozPM(DMatrix A, long int ch[], long int min_dim, int max_num_div)` ... LU decompose, using partial pivoting and Ozaki scheme setting `max_num_div` as maximum number of divisions.
- `int DDLUdecomp_ozPM(DDMatrix A, long int ch[], long int min_dim, int max_num_div)`
- `int TDLUdecomp_ozPM(TDMatrix A, long int ch[], long int min_dim, int max_num_div)`
- `int QDLUdecomp_ozPM(QDMatrix A, long int ch[], long int min_dim, int max_num_div)`
- `int MPFLUdecomp_ozPM(MPFMatrix A, long int ch[], long int min_dim, int max_num_div)`



# Bibliography

- [1] D.H. Bailey. QD. <https://www.davidhbailey.com/dhbsoftware/>.
- [2] T. Granlaud and GMP development team. The GNU Multiple Precision arithmetic library. <https://gmplib.org/>.
- [3] M. Jolders, J.-M. Muller, V. Popescu, and W. Tucker. Campary: Cuda mutiple precision arithmetic library and applications. *5th ICMS*, 2016.
- [4] Tomonori Kouya. BNCpack. <https://na-inet.jp/na/bnc/>.
- [5] LAPACK. <http://www.netlib.org/lapack/>.
- [6] MPLAPACK/MPBLAS. Multiple precision arithmetic LAPACK and BLAS. <https://github.com/nakatamaho/mplapack>.
- [7] Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Toshiyuki Imamura. Accurate matrix multiplication on binary128 format accelerated by ozaki scheme. In *50th International Conference on Parallel Processing*, ICPP 2021, New York, NY, USA, 2021. Association for Computing Machinery.
- [8] MPFR Project. The MPFR library. <https://www.mpfr.org/>.
- [9] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, Vol. 31, No. 1, pp. 189–224, 2008.
- [10] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, k-fold faithful and rounding to nearest. *SIAM Journal on Scientific Computing*, Vol. 31, No. 2, pp. 1269–1302, 2008.
- [11] Taiga Utsugiri and Tomonori Kouya. Acceleration of multiple precision matrix multiplication using ozaki scheme, 2023.

# Index

- \_\_m256, 13
- \_\_m256d, 14–16
- \_\_m256d \*, 14, 16
- \_\_m256d a, 14
- \_\_m256d\*, 16
- \_bncavx2\_dadd, 14
- \_bncavx2\_ddiv, 14
- \_bncavx2\_ddotp, 14
- \_bncavx2\_dfma, 14
- \_bncavx2\_dmul, 14
- \_bncavx2\_dneg, 14
- \_bncavx2\_dquick\_two\_diff, 14
- \_bncavx2\_dquick\_two\_sum, 14
- \_bncavx2\_dsub, 14
- \_bncavx2\_dtwo\_diff, 14
- \_bncavx2\_dtwo\_prod, 14
- \_bncavx2\_dtwo\_sum, 14
- \_bncavx2\_fabs, 14
- \_bncavx2\_fabsf, 13
- \_bncavx2\_ffma, 13
- \_bncavx2\_fmulp, 13
- \_bncavx2\_fneg, 13
- \_bncavx2\_get\_dd\_m256d\_i, 14
- \_bncavx2\_get\_qd\_m256d\_i, 15
- \_bncavx2\_get\_td\_m256d\_i, 15
- \_bncavx2\_merge, 15
- \_bncavx2\_rdd\_abs, 14
- \_bncavx2\_rdd\_absmax256d, 14
- \_bncavx2\_rdd\_abssum256d, 14
- \_bncavx2\_rdd\_add, 14
- \_bncavx2\_rdd\_div, 14
- \_bncavx2\_rdd\_mulp, 14
- \_bncavx2\_rdd\_norm256d, 14
- \_bncavx2\_rdd\_set0, 14
- \_bncavx2\_rdd\_sub, 14
- \_bncavx2\_rdd\_sum256d, 14
- \_bncavx2\_renorm, 16
- \_bncavx2\_renorm4, 16
- \_bncavx2\_rqd\_absmax256d, 15
- \_bncavx2\_rqd\_abssum256d, 15
- \_bncavx2\_rqd\_add, 16
- \_bncavx2\_rqd\_div, 16
- \_bncavx2\_rqd\_mulp, 16
- \_bncavx2\_rqd\_mulp\_d, 16
- \_bncavx2\_rqd\_norm256d, 16
- \_bncavx2\_rqd\_sub, 16
- \_bncavx2\_rqd\_sum256d, 15
- \_bncavx2\_rtd\_abs, 15
- \_bncavx2\_rtd\_absmax256d, 15
- \_bncavx2\_rtd\_abssum256d, 15
- \_bncavx2\_rtd\_add, 15
- \_bncavx2\_rtd\_addq, 15, 16
- \_bncavx2\_rtd\_addt, 15
- \_bncavx2\_rtd\_div, 15
- \_bncavx2\_rtd\_divq, 15, 16
- \_bncavx2\_rtd\_divt, 15
- \_bncavx2\_rtd\_divtq, 15
- \_bncavx2\_rtd\_mulp, 15

- \_bncavx2\_rtd\_mulp\_d, 15
- \_bncavx2\_rtd\_mulp\_dd, 15
- \_bncavx2\_rtd\_mulpq, 15, 16
- \_bncavx2\_rtd\_neg, 15
- \_bncavx2\_rtd\_norm256d, 15
- \_bncavx2\_rtd\_sub, 15
- \_bncavx2\_rtd\_subq, 15
- \_bncavx2\_rtd\_sum256d, 15
- \_bncavx2\_set0\_dd, 14
- \_bncavx2\_set0\_qd, 15
- \_bncavx2\_set0\_td, 15
- \_bncavx2\_three\_sum, 16
- \_bncavx2\_three\_sum2, 16
- \_bncavx2\_to\_td, 15
- \_bncavx2\_vec\_sum, 15
- \_bncavx2\_vseb, 15
- \_bncomp\_mulp\_ddmatrix\_block, 26
- \_bncomp\_mulp\_ddmatrix\_simple, 26
- \_bncomp\_mulp\_ddmatrix\_strassen, 26
- \_bncomp\_mulp\_mpfmatrix\_simple, 26
- \_bncomp\_mulp\_mpfmatrix\_block, 26
- \_bncomp\_mulp\_mpfmatrix\_strassen, 26
- \_bncomp\_mulp\_qdmatrix\_block, 26
- \_bncomp\_mulp\_qdmatrix\_simple, 26
- \_bncomp\_mulp\_qdmatrix\_strassen, 26
- \_bncomp\_mulp\_tdmatrix\_block, 26
- \_bncomp\_mulp\_tdmatrix\_simple, 26
- \_bncomp\_mulp\_tdmatrix\_strassen, 26

- absmax\_col\_ddmatrix, 28
- absmax\_col\_mpfmatrix, 28
- absmax\_col\_qdmatrix, 28
- absmax\_col\_tdmatrix, 28
- absmax\_ddvector, 27
- absmax\_mpfvector, 27
- absmax\_qdvector, 27
- absmax\_row\_ddmatrix, 27
- absmax\_row\_mpfmatrix, 27
- absmax\_row\_qdmatrix, 27
- absmax\_row\_tdmatrix, 27
- absmax\_tdvector, 27
- add\_cmulp\_ddvector, 20
- add\_cmulp\_mpfvector, 20
- add\_cmulp\_qdvector, 20
- add\_cmulp\_tdvector, 20
- add\_ddmatrix, 22
- add\_ddmatrix\_dmat, 27
- add\_ddvector, 19
- add\_ddvector\_dvec, 27
- add\_mpfmatrix, 22
- add\_mpfmatrix\_dmat, 27
- add\_mpfvector, 19
- add\_mpfvector\_dvec, 27
- add\_qdmatrix, 22
- add\_qdmatrix\_dmat, 27
- add\_qdvector, 19
- add\_qdvector\_dvec, 27
- add\_tdmatrix, 22

add\_tdmatrix\_dmat, 27  
 add\_tdvector, 19  
 add\_tdvector\_dvec, 27  
 add2\_ddvector, 19  
 add2\_mpfvector, 19  
 add2\_qdvector, 19  
 add2\_tdvector, 19  
  
 c\_d\_add, 8  
 c\_d\_div, 8  
 c\_d\_mul, 8  
 c\_d\_sqr, 9  
 c\_d\_sub, 8  
 c\_dd\_abs, 9  
 c\_dd\_add, 8  
 c\_dd\_add\_d\_dd, 8  
 c\_dd\_add\_dd\_d, 8  
 c\_dd\_add\_sloppy, 8  
 c\_dd\_ceil, 9  
 c\_dd\_comp, 8  
 c\_dd\_comp\_d\_dd, 8  
 c\_dd\_comp\_dd\_d, 8  
 c\_dd\_copy, 9  
 c\_dd\_copy\_d, 9  
 c\_dd\_div, 8  
 c\_dd\_div\_d\_dd, 9  
 c\_dd\_div\_dd\_d, 8  
 c\_dd\_floor, 9  
 c\_dd\_mul, 8  
 c\_dd\_mul\_d\_dd, 8  
 c\_dd\_mul\_dd\_d, 8  
 c\_dd\_neg, 8  
 c\_dd\_pi, 8  
 c\_dd\_set, 8  
 c\_dd\_set\_dd\_d, 8  
 c\_dd\_set0, 8  
 c\_dd\_setnan, 8  
 c\_dd\_sloppy\_div, 8  
 c\_dd\_sqr, 9  
 c\_dd\_sqr\_d, 9  
 c\_dd\_sqrt, 9  
 c\_dd\_sub, 8  
 c\_dd\_sub\_d\_dd, 8  
 c\_dd\_sub\_dd\_d, 8  
 c\_dd\_sub\_sloppy, 8  
 c\_qd\_abs, 10  
 c\_qd\_add, 9  
 c\_qd\_add\_qd\_dd, 9  
 c\_qd\_add\_sloppy, 9  
 c\_qd\_ceil, 10  
 c\_qd\_comp, 10  
 c\_qd\_comp\_d\_qd, 10  
 c\_qd\_comp\_qd\_d, 10  
 c\_qd\_copy, 9  
 c\_qd\_copy\_d, 9  
 c\_qd\_copy\_dd, 9  
 c\_qd\_copy\_td, 10  
 c\_qd\_div\_accurate, 10  
 c\_qd\_div\_d\_qd, 10  
 c\_qd\_div\_qd\_d, 10  
 c\_qd\_div\_qd\_dd, 10  
 c\_qd\_div\_sloppy, 10  
 c\_qd\_floor, 10  
 c\_qd\_mul, 9  
 c\_qd\_mul\_d\_qd, 9  
 c\_qd\_mul\_dd\_qd, 9

c\_qd\_mul\_pwr2, 10  
 c\_qd\_mul\_qd\_d, 9  
 c\_qd\_mul\_qd\_dd, 9  
 c\_qd\_mul\_sloppy, 9  
 c\_qd\_neg, 9  
 c\_qd\_neg\_d, 9  
 c\_qd\_neg\_dd, 9  
 c\_qd\_nint, 10  
 c\_qd\_selfadd, 9  
 c\_qd\_selfadd\_d, 9  
 c\_qd\_selfadd\_dd, 9  
 c\_qd\_selfdiv, 10  
 c\_qd\_selfdiv\_d, 10  
 c\_qd\_selfdiv\_dd, 10  
 c\_qd\_selfmul, 9  
 c\_qd\_selfmul\_d, 9  
 c\_qd\_selfmul\_dd, 9  
 c\_qd\_selfsub, 9  
 c\_qd\_selfsub\_d, 9  
 c\_qd\_selfsub\_dd, 9  
 c\_qd\_set, 10  
 c\_qd\_set0double \*qdval, 10  
 c\_qd\_sqr, 9  
 c\_qd\_sqrt, 10  
 c\_qd\_sub, 9  
 c\_qd\_sub\_d\_qd, 9  
 c\_qd\_sub\_dd\_qd, 9  
 c\_qd\_sub\_qd\_d, 9  
 c\_qd\_sub\_qd\_dd, 9  
 c\_td\_2mtw\_dd\_td, 11  
 c\_td\_abs, 11  
 c\_td\_add, 10  
 c\_td\_add\_td\_d, 10  
 c\_td\_addq, 9  
 c\_td\_comp, 11  
 c\_td\_comp\_d\_td, 11  
 c\_td\_comp\_td\_d, 11  
 c\_td\_copy, 10  
 c\_td\_copy\_d, 10  
 c\_td\_copy\_dd, 10  
 c\_td\_copy\_qd, 10  
 c\_td\_div, 11  
 c\_td\_div\_td\_d, 11  
 c\_td\_divq, 11  
 c\_td\_divt, 11  
 c\_td\_divtq, 11  
 c\_td\_mul\_accurate, 10  
 c\_td\_mul\_d\_td, 11  
 c\_td\_mul\_dd\_td\_accurate, 10  
 c\_td\_mul\_dd\_td\_sloppy, 10  
 c\_td\_mul\_sloppy, 10  
 c\_td\_mul\_td\_d, 11  
 c\_td\_neg, 10  
 c\_td\_reciprocal, 11  
 c\_td\_sqrt, 11  
 c\_td\_sqrt\_d, 11  
 c\_td\_sub, 10  
 c\_td\_sub\_d\_td, 10  
 c\_td\_sub\_td\_d, 10  
 c\_td\_subq, 10  
 c\_to\_td, 10  
 cmul\_ddmatrix, 22  
 cmul\_ddvector, 19  
 cmul\_mpfmatrix, 22  
 cmul\_mpfvector, 19

cmul\_qdvector, 19  
 cmul\_tdmatrix, 22  
 cmul\_tdvector, 19  
 cmul2\_ddvector, 19, 20  
 const \_\_m256d, 15  
 const char \*, 19, 21, 23, 24  
 const double, 8, 9, 11  
 const double \*, 8–11  
  
 DD\_FALSE, 7  
 DD\_HI, 7  
 dd\_is\_negative, 8  
 dd\_is\_one, 8  
 dd\_is\_positive, 8  
 dd\_is\_zero, 8  
 DD\_ISINF, 7  
 DD\_ISNAN, 7  
 DD\_ISNEGATIVE, 7  
 DD\_ISONE, 7  
 DD\_ISZERO, 7  
 DD\_LOW, 7  
 DD\_NAN, 8  
 DD\_TRUE, 7  
 ddfloat, 19, 21  
 DDLUdecomp, 29  
 DDLUdecomp\_oz, 30  
 DDLUdecomp\_ozPM, 30  
 DDLUdecomp\_strassen, 30  
 DDLUdecomp\_strassenPM, 30  
 DDLUdecompC, 29  
 DDLUdecompP, 29  
 DDMatrix, 20–30  
 DDSIZE, 7  
 DDVector, 18–20, 22–24, 26–29  
 DFMA, 7  
 diag\_ddmatrix, 24  
 diag\_mpfmatrix, 24  
 diag\_qdmatrix, 24  
 diag\_tdmatrix, 24  
 DLUdecomp, 29  
 DLUdecomp\_oz, 30  
 DLUdecomp\_ozPM, 30  
 DLUdecomp\_strassen, 30  
 DLUdecomp\_strassenPM, 30  
 DLUdecompC, 29  
 DLUdecompP, 29  
 DMatrix, 22, 23, 25–30  
 DMatrix org\_mat, 27  
 DMatrix ret\_high\_mat, 27  
 double, 8–12, 14–16, 18–28  
 double \*, 8–11, 18, 20, 26  
 DVector, 22, 26–29  
  
 extract\_dvector, 27  
  
 FILE \*, 11, 23  
 float, 13  
 frank\_ddmatrix, 24  
 frank\_mpfmatrix, 24  
 frank\_qdmatrix, 24  
 frank\_tdmatrix, 24  
 fread\_ddmatrix, 23  
 fread\_ddmatrix\_fname, 23  
 fread\_ddvector, 23  
 fread\_ddvector\_fname, 23  
 fread\_mpfmatrix, 23

fread\_mpfmatrix\_fname, 23  
 fread\_mpfvector, 23  
 fread\_mpfvector\_fname, 23  
 fread\_qdmatrix, 23  
 fread\_qdmatrix\_fname, 23  
 fread\_qdvector, 23  
 fread\_qdvector\_fname, 23  
 fread\_tdmatrix, 23  
 fread\_tdmatrix\_fname, 23  
 fread\_tdvector, 23  
 fread\_tdvector\_fname, 23  
 free\_ddmatrix, 21  
 free\_ddvector, 19  
 free\_mpfmatrix, 21  
 free\_mpfvector, 19  
 free\_qdmatrix, 21  
 free\_qdvector, 19  
 free\_tdmatrix, 21  
 free\_tdvector, 19  
 fwrite\_ddmatrix, 23  
 fwrite\_ddmatrix\_fname, 23  
 fwrite\_ddvector, 23  
 fwrite\_ddvector\_fname, 23  
 fwrite\_mpfmatrix, 23  
 fwrite\_mpfmatrix\_fname, 23  
 fwrite\_mpfvector, 23  
 fwrite\_mpfvector\_fname, 23  
 fwrite\_qdmatrix, 23  
 fwrite\_qdmatrix\_fname, 23  
 fwrite\_qdvector, 23  
 fwrite\_qdvector\_fname, 23  
 fwrite\_tdmatrix, 23  
 fwrite\_tdmatrix\_fname, 23  
 fwrite\_tdvector, 23  
 fwrite\_tdvector\_fname, 23  
  
 GET\_DDMATRIX\_IJ, 20  
 get\_ddmatrix\_ij, 20  
 get\_ddmatrix\_ij\_ddfloat, 20  
 GET\_DDVECTOR\_I, 18  
 get\_ddvector\_i, 18  
 get\_ddvector\_i\_ddfloat, 18  
 GET\_MPFMATRIX\_IJ, 20  
 get\_mpfmatrix\_ij, 20  
 GET\_MPFVECTOR\_I, 18  
 get\_mpfvector\_i, 18  
 GET\_QDMATRIX\_IJ, 20  
 get\_qdmatrix\_ij, 20  
 get\_qdmatrix\_ij\_qdfloat, 20  
 GET\_QDVECTOR\_I, 18  
 get\_qdvector\_i, 18  
 get\_qdvector\_i\_qdfloat, 18  
 GET\_TDMATRIX\_IJ, 20  
 get\_tdmatrix\_ij, 20  
 get\_tdmatrix\_ij\_tdfloat, 20  
 GET\_TDVECTOR\_I, 18  
 get\_tdvector\_i, 18  
 get\_tdvector\_i\_tdfloat, 18  
  
 hilbert\_ddmatrix, 24  
 hilbert\_mpfmatrix, 24  
 hilbert\_qdmatrix, 24  
 hilbert\_tdmatrix, 24  
  
 im\_rand\_ddmatrix, 25  
 im\_rand\_mpfmatrix, 25

im\_rand\_qdmatrix, 25  
 im\_rand\_tdmatrix, 25  
 in, 18  
 init\_ddmatrix, 21  
 init\_ddvector, 18  
 init\_mpfmatrix, 21  
 init\_mpfvector, 18  
 init\_qdmatrix, 21  
 init\_qdvector, 18  
 init\_tdmatrix, 21  
 init\_tdvector, 18  
 init2\_mpfmatrix, 21  
 init2\_mpfvector, 19  
 int, 10, 11, 13–15, 19, 21–23, 26–28, 30  
 int \*, 8, 10, 11  
 int num\_div, 28  
 int\_sym\_rand\_ddmatrix, 24  
 int\_sym\_rand\_mpfmatrix, 24  
 int\_sym\_rand\_qdmatrix, 24  
 int\_sym\_rand\_tdmatrix, 24  
 int\_unsym\_rand\_ddmatrix, 24  
 int\_unsym\_rand\_mpfmatrix, 24  
 int\_unsym\_rand\_qdmatrix, 24  
 int\_unsym\_rand\_tdmatrix, 24  
 inv\_ddmatrix, 22  
 inv\_ddmatrix\_strassen\_even, 26  
 inv\_mpfmatrix, 22  
 inv\_qdmatrix, 22  
 inv\_qdmatrix\_strassen\_even, 26  
 inv\_tdmatrix, 22  
 inv\_tdmatrix\_strassen\_even, 26  
 ip\_ddvector, 20  
 ip\_mpfvector, 20  
 ip\_qdvector, 20  
 ip\_tdvector, 20  
  
 long int, 18–21, 23–30  
 long int \*, 27, 28  
 long int dim, 23, 24  
 lotkin\_ddmatrix, 24  
 lotkin\_mpfmatrix, 24  
 lotkin\_qdmatrix, 24  
 lotkin\_tdmatrix, 24  
  
 merge, 10  
 MFPMatrix, 29  
 mpf\_t, 19–22, 25–28  
 MPFLUdecomp, 29  
 MPFLUdecomp\_oz, 30  
 MPFLUdecomp\_ozPM, 30  
 MPFLUdecomp\_strassen, 30  
 MPFLUdecomp\_strassenPM, 30  
 MPFLUdecompC, 29  
 MPFLUdecompP, 29  
 MPFMatrix, 20–30  
 MPFMatrix b, 25  
 MPFVector, 19, 20, 22–24, 26–29  
 mul\_ddmatrix, 21, 25  
 mul\_ddmatrix\_block, 25  
 mul\_ddmatrix\_ddvec, 22  
 mul\_ddmatrix\_ddvec\_oz, 28  
 mul\_ddmatrix\_oz, 28  
 mul\_ddmatrix\_simple, 25  
 mul\_ddmatrix\_strassen, 25  
 mul\_ddmatrix\_strassen\_even, 25  
 mul\_ddmatrix\_winograd\_even, 25

mul\_ddmatrixt\_ddvec, 22  
 mul\_dmatrix\_block, 25  
 mul\_dmatrix\_strassen, 25  
 mul\_dmatrix\_strassen\_even, 25  
 mul\_dmatrix\_winograd\_even, 25  
 mul\_mpfmatrix, 21  
 mul\_mpfmatrix\_block, 25  
 mul\_mpfmatrix\_dvec\_oz, 28  
 mul\_mpfmatrix\_mpfvec, 22  
 mul\_mpfmatrix\_oz, 28  
 mul\_mpfmatrix\_simple, 25  
 mul\_mpfmatrix\_strassen, 25  
 mul\_mpfmatrix\_strassen\_even, 25  
 mul\_mpfmatrix\_winograd\_even, 26  
 mul\_mpfmatrixt\_mpfvec, 22  
 mul\_qdmatrix, 21, 25  
 mul\_qdmatrix\_block, 25  
 mul\_qdmatrix\_oz, 28  
 mul\_qdmatrix\_qdvec, 22  
 mul\_qdmatrix\_qdvec\_oz, 28  
 mul\_qdmatrix\_simple, 25  
 mul\_qdmatrix\_strassen, 25  
 mul\_qdmatrix\_strassen\_even, 25  
 mul\_qdmatrix\_winograd\_even, 26  
 mul\_qdmatrixt\_qdvec, 22  
 mul\_tdmatrix, 21, 25  
 mul\_tdmatrix\_block, 25  
 mul\_tdmatrix\_oz, 28  
 mul\_tdmatrix\_simple, 25  
 mul\_tdmatrix\_strassen, 25  
 mul\_tdmatrix\_strassen\_even, 25  
 mul\_tdmatrix\_tdvec, 22  
 mul\_tdmatrix\_tdvec\_oz, 28  
 mul\_tdmatrix\_winograd\_even, 25  
 mul\_tdmatrixt\_tdvec, 22  
  
 neg\_ddvector, 20  
 neg\_mpfvector, 20  
 neg\_qdvector, 20  
 neg\_tdvector, 20  
 nint, 10  
 norm1\_ddmatrix, 21  
 norm1\_ddvector, 20  
 norm1\_mpfmatrix, 22  
 norm1\_mpfvector, 20  
 norm1\_qdmatrix, 21  
 norm1\_qdvector, 20  
 norm1\_tdmatrix, 21  
 norm1\_tdvector, 20  
 norm2\_ddvector, 20  
 norm2\_mpfvector, 20  
 norm2\_qdvector, 20  
 norm2\_tdvector, 20  
 normf\_ddmatrix, 21  
 normf\_mpfmatrix, 21  
 normf\_qdmatrix, 21  
 normf\_tdmatrix, 21  
 normi\_ddmatrix, 21  
 normi\_ddvector, 20  
 normi\_mpfmatrix, 21  
 normi\_mpfvector, 20  
 normi\_qdmatrix, 21  
 normi\_qdvector, 20  
 normi\_tdmatrix, 21  
 normi\_tdvector, 20  
  
 ONE\_M\_2DBL\_EPS, 11

ONE\_P\_2DBL\_EPS, 11

pascal\_ddmatrix, 25  
pascal\_mpfmatrix, 25  
pascal\_qdmatrix, 25  
pascal\_tdmatrix, 25  
print\_ddmatrix, 21  
print\_ddvector, 19  
print\_mpfmatrix, 21  
print\_mpfvector, 19  
print\_normf\_ddmatrix, 21  
print\_qdmatrix, 21  
print\_qdvector, 19  
print\_tdmatrix, 21  
print\_tdvector, 19

QD\_FALSE, 7  
QD\_FMA, 7  
QD\_FMS, 7  
QD\_ISINF, 7  
QD\_ISNAN, 7  
QD\_ISNEGATIVE, 7  
QD\_ISONE, 7  
QD\_ISZERO, 7  
QD\_NAN, 8  
QD\_TRUE, 7  
qdfloat, 19, 21  
qdfloat \*, 15  
QDLUdecomp, 29  
QDLUdecomp\_oz, 30  
QDLUdecomp\_ozPM, 30  
QDLUdecomp\_strassen, 30  
QDLUdecomp\_strassenPM, 30  
QDLUdecompC, 29  
QDLUdecompP, 29  
QDMatrix, 20–30  
QDMatrixmat, 28  
QDSIZE, 7  
QDVector, 18–20, 22–24, 26–29  
quick\_renorm, 9  
quick\_two\_diff, 8  
quick\_two\_sum, 8

rdd\_abs, 12  
rdd\_add, 12  
rdd\_add\_d, 12  
rdd\_add\_ui, 12  
rdd\_cmp, 11  
rdd\_cmp\_d, 11  
rdd\_cmp\_ui, 12  
rdd\_div, 12  
rdd\_div\_d, 12  
rdd\_div\_ui, 12  
rdd\_fma, 11  
rdd\_get\_d, 12  
rdd\_mul, 12  
rdd\_mul\_d, 12  
rdd\_mul\_ui, 12  
rdd\_neg, 12  
rdd\_out\_str\_base, 11  
rdd\_pow, 11  
rdd\_set, 12  
rdd\_set\_d, 12  
rdd\_set\_ui, 12  
rdd\_set0, 12  
rdd\_sqrt, 12

rdd\_sqrt\_d, 11, 12  
rdd\_sqrt\_ui, 12  
rdd\_sub, 12  
rdd\_sub\_d, 12  
rdd\_sub\_ui, 12  
rdd\_ui\_div, 12  
rdd\_ui\_sub, 12  
read\_test\_linear\_eq\_dd, 24  
read\_test\_linear\_eq\_mpf, 24  
read\_test\_linear\_eq\_qd, 24  
read\_test\_linear\_eq\_td, 24  
relerr\_ddvector, 23  
relerr\_ddvector\_mpfvec, 22  
relerr\_element\_ddvector, 23  
relerr\_element\_ddvector\_mpf, 22  
relerr3\_ddmatrix, 26  
relerr3\_ddvector, 26  
relerr3\_dmatrix, 26  
relerr3\_dvector, 26  
relerr3\_mpfmatrix, 27  
relerr3\_mpfvector, 26  
relerr3\_qdmatrix, 27  
relerr3\_qdvector, 26  
relerr3\_tdmatrix, 27  
relerr3\_tdvector, 26  
renorm, 9  
renorm4, 9  
row\_swap\_ddmatrix, 23  
rqd\_abs, 13  
rqd\_add, 13  
rqd\_add\_d, 13  
rqd\_add\_ui, 13  
rqd\_cmp, 11  
rqd\_cmp\_d, 11  
rqd\_cmp\_ui, 13  
rqd\_div, 13  
rqd\_div\_d, 13  
rqd\_div\_ui, 13  
rqd\_fma, 11  
rqd\_get\_d, 13  
rqd\_mul, 13  
rqd\_mul\_d, 13  
rqd\_mul\_ui, 13  
rqd\_neg, 13  
rqd\_out\_str\_base, 11  
rqd\_pow, 11  
rqd\_set, 13  
rqd\_set\_d, 13  
rqd\_set\_ui, 13  
rqd\_set0, 13  
rqd\_sqrt, 13  
rqd\_sqrt\_d, 11, 13  
rqd\_sqrt\_ui, 13  
rqd\_sub, 13  
rqd\_sub\_d, 13  
rqd\_sub\_ui, 13  
rqd\_ui\_div, 13  
rqd\_ui\_sub, 13  
rtd\_abs, 13  
rtd\_add, 12  
rtd\_add\_d, 13  
rtd\_add\_ui, 13  
rtd\_addq, 12  
rtd\_addt, 12  
rtd\_cmp, 11  
rtd\_cmp\_d, 12

rtd\_cmp\_ui, 13  
 rtd\_div, 12  
 rtd\_div\_d, 13  
 rtd\_div\_ui, 13  
 rtd\_divq, 12  
 rtd\_divt, 12  
 rtd\_divtq, 12  
 rtd\_fma, 12  
 rtd\_get\_d, 12  
 rtd\_mul, 12  
 rtd\_mul\_d, 13  
 rtd\_mul\_ui, 13  
 rtd\_neg, 13  
 rtd\_out\_str\_base, 11  
 rtd\_pow, 12  
 rtd\_set, 13  
 rtd\_set\_d, 12  
 rtd\_set\_ui, 12  
 rtd\_set0, 12  
 rtd\_sqrt, 12  
 rtd\_sqrt\_d, 12  
 rtd\_sqrt\_ui, 12  
 rtd\_sub, 12  
 rtd\_sub\_d, 13  
 rtd\_sub\_ui, 13  
 rtd\_subq, 12  
 rtd\_subt, 12  
 rtd\_ui\_div, 13  
 rtd\_ui\_sub, 13

set\_ddfloat\_ddmat, 21  
 set\_ddfloat\_ddvec, 19  
 set\_ddmatrix\_ddfloat, 21  
 SET\_DDMATRIX\_IJ, 20  
 set\_ddmatrix\_ij, 20  
 SET\_DDMATRIX\_IJ\_D, 21  
 set\_ddmatrix\_ij\_d, 20  
 SET\_DDMATRIX\_IJ\_UI, 21  
 set\_ddmatrix\_ij\_ui, 21  
 set\_ddvector\_ddfloat, 19  
 SET\_DDVECTOR\_I, 18  
 set\_ddvector\_i, 18  
 SET\_DDVECTOR\_I\_D, 18  
 set\_ddvector\_i\_d, 18  
 set\_ddvector\_i\_str, 19  
 SET\_MPFMATRIX\_IJ, 20  
 set\_mpfmatrix\_ij, 20  
 set\_mpfvector\_i\_str, 19  
 set\_qdfloat\_qdmat, 21  
 set\_qdfloat\_qdvec, 19  
 SET\_QDMATRIX\_IJ, 20  
 set\_qdmatrix\_ij, 20  
 set\_qdmatrix\_qdfloat, 21  
 SET\_QDVECTOR\_I, 18  
 set\_qdvector\_i, 18  
 SET\_QDVECTOR\_I\_D, 18  
 set\_qdvector\_i\_d, 18  
 set\_qdvector\_i\_str, 19  
 set\_qdvector\_qdfloat, 19  
 set\_tdfloat\_tdmatrix, 21  
 set\_tdfloat\_tdvec, 19  
 SET\_TDMATRIX\_IJ, 20  
 set\_tdmatrix\_ij, 20  
 set\_tdmatrix\_tdfloat, 21  
 SET\_TDVECTOR\_I, 18  
 set\_tdvector\_i, 18

SET\_TDVECTOR\_I\_D, 18  
 set\_tdvector\_i\_d, 18  
 set\_tdvector\_i\_str, 19  
 set\_tdvector\_tdfloat, 19  
 SET0\_DD, 11  
 set0\_dd, 12  
 set0\_ddmatrix, 21  
 SET0\_DDMATRIX\_IJ, 21  
 set0\_ddmatrix\_ij, 21  
 set0\_ddvector, 19  
 SET0\_DDVECTOR\_I, 18  
 set0\_ddvector\_i, 18  
 set0\_mpfmatrix, 21  
 set0\_mpfvector, 19  
 SET0\_QD, 11  
 set0\_qd, 13  
 set0\_qdmatrix, 21  
 set0\_qdvector, 19  
 SET0\_QDVECTOR\_I, 18  
 set0\_qdvector\_i, 18  
 SET0\_TD, 11  
 set0\_td, 12  
 set0\_tdmatrix, 21  
 set0\_tdvector, 19  
 SET0\_TDVECTOR\_I, 18  
 set0\_tdvector\_i, 18  
 setI\_ddmatrix, 22  
 setI\_mpfmatrix, 22  
 setI\_qdmatrix, 22  
 setI\_tdmatrix, 22  
 SFMA, 7  
 SolveDDLs, 29  
 SolveDDLSC, 29  
 SolveDDLSP, 29  
 SolveDLS, 29  
 SolveDLSC, 29  
 SolveDLSP, 29  
 SolveMPFLS, 29  
 SolveMPFLSC, 29  
 SolveMPFLSP, 29  
 SolveQDLS, 29  
 SolveQDLSC, 29  
 SolveQDLSP, 29  
 SolveTDLS, 29  
 SolveTDLSC, 29  
 SolveTDLSP, 29  
 split, 8  
 split\_ddmatrix\_dmat, 27  
 split\_ddmatrix\_t\_dmat, 28  
 split\_ddvector\_dvec, 27  
 split\_dmatrix, 27  
 split\_dmatrix\_t, 27  
 split\_mpfmatrix\_dmat, 28  
 split\_mpfmatrix\_t\_dmat, 28  
 split\_qdmatrix\_dmat, 27  
 split\_qdmatrix\_t\_dmat, 28  
 split\_qdvector\_dvec, 28  
 split\_tdmatrix\_dmat, 27  
 split\_tdmatrix\_t\_dmat, 28  
 split\_tdvector\_dvec, 27  
 sub\_ddmatrix, 22  
 sub\_ddmatrix\_dmat, 27  
 sub\_ddvector, 19  
 sub\_ddvector\_dvec, 27  
 sub\_mpfmatrix, 22  
 sub\_mpfmatrix\_dmat, 27

sub\_mpfvector, 19  
 sub\_mpfvector\_dvec, 27  
 sub\_qdmatrix, 22  
 sub\_qdmatrix\_dmat, 27  
 sub\_qdvector, 19  
 sub\_qdvector\_dvec, 27  
 sub\_tdmatrix, 22  
 sub\_tdmatrix\_dmat, 27  
 sub\_tdvector, 19  
 sub\_tdvector\_dvec, 27  
 sub2\_ddvector, 19  
 sub2\_mpfvector, 19  
 sub2\_qdvector, 19  
 sub2\_tdvector, 19  
 subst\_ddmatrix, 22  
 subst\_ddmatrix\_dmat, 22  
 subst\_ddmatrix\_mpfmat, 22  
 subst\_ddvector\_dvec, 22  
 subst\_ddvector\_mpfvec, 22  
 subst\_dmatrix\_ddmat, 23  
 subst\_dmatrix\_qdmat, 28  
 subst\_dvector\_ddvec, 22  
 subst\_dvector\_qdvec, 27  
 subst\_mpfmatrix, 22  
 subst\_mpfmatrix\_ddmat, 22  
 subst\_mpfvector\_ddvec, 22  
 subst\_qdmatrix, 22  
 subst\_qdmatrix\_dmat, 28  
 subst\_tdmatrix, 22  
  
 TD\_FALSE, 7  
 TD\_ISINF, 7  
 TD\_ISNAN, 7  
 TD\_ISNEGATIVE, 7  
 TD\_ISONE, 7  
 TD\_ISZERO, 7  
 TD\_NAN, 8  
 TD\_TRUE, 7  
 tdfloat, 19, 21  
 tdfloat \*, 15  
 TDLUdecomp, 29  
 TDLUdecomp\_oz, 30  
 TDLUdecomp\_ozPM, 30  
 TDLUdecomp\_strassen, 30  
 TDLUdecomp\_strassenPM, 30  
 TDLUdecompC, 29  
 TDLUdecompP, 29  
 TDMatrix, 20–30  
 TDSIZE, 7  
 TDVector, 18–20, 22–24, 26–29  
 three\_sum, 9  
 three\_sum2, 9  
 toeplitz\_ddmatrix, 24  
 toeplitz\_mpfmatrix, 25  
 toeplitz\_qdmatrix, 25  
 toeplitz\_tdmatrix, 25  
 transpose\_ddmatrix, 22  
 transpose\_mpfmatrix, 22  
 transpose\_qdmatrix, 22  
 transpose\_tdmatrix, 22  
 tridiag\_ddmatrix, 24  
 tridiag\_mpfmatrix, 24  
 tridiag\_qdmatrix, 24  
 tridiag\_tdmatrix, 24  
 two\_diff, 8  
 two\_prod, 8

two\_sqr, 8  
 two\_sum, 8  
  
 unsigned long, 19, 21, 25  
 USE\_QD\_DIV\_ACCURATE, 10  
 USE\_TD\_MUL\_ACCURATE, 10  
 USE\_TD\_MUL\_DD\_TD\_ACCURATE, 10  
  
 vec\_sum, 10  
 void, 15  
 vseb, 10