BNCmatmul User's Guide

Tomonori Kouya
`https://github.com/tkouya/bncmatmul`

Version 0.21: May 31, 2023

Contents

Chapter 1

What's BNCmatmul?

BNCmatmul is one of optimized BLAS(Basic Linear Algebra Subprograms) libaries 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/libncmatmul*.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 $\mathbf{x} = [x_i]_{i=1,2,..,n} \in \mathbb{R}^n$, and $(A)_{ij} (= a_{ij})$ as the (i,j)-th element of $A = [a_{ij}]_{i=1,2,...,m,j=1,2,...,n} \in \mathbb{R}^{m\times n}$.

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 α 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 [?] [?] 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.

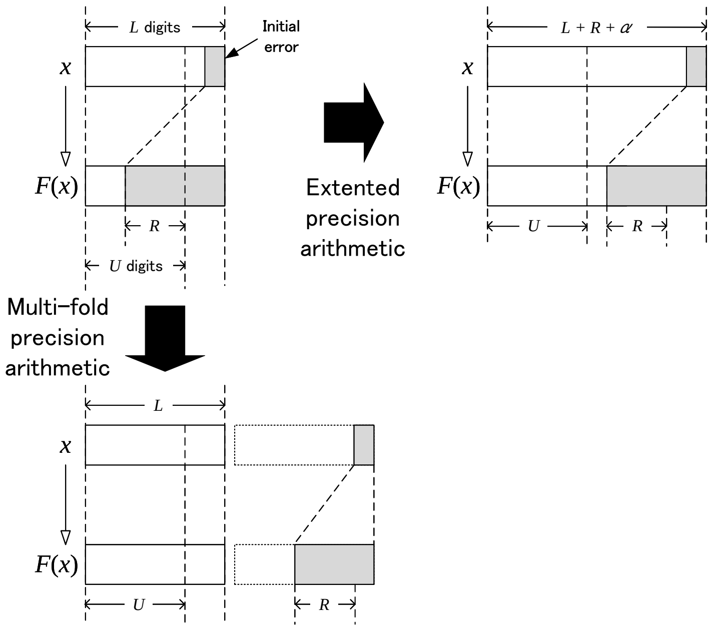
A conceptual diagram of the two aforementioned approaches is presented in Fig. **??**.

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[?][?]. 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[?]. Subsequently, owing to current popularity of multi-core



L digits
Initial error
x
F(x)
R
U digits
Extented precision arithmetic
L + R + α
U
R
Multi-fold precision arithmetic
L
x
F(x)
U
R

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 **??** 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: $C :=$ Strassen$(A, B) = AB \in \mathbb{R}^{m\times n}$
if $m < m_0$ && $n < n_0$ then
    $C := AB$
end if
$P_1 :=$ Strassen$(A_{11} + A_{22}, B_{11} + B_{22})$
$P_2 :=$ Strassen$(A_{21} + A_{22}, B_{11})$
$P_3 :=$ Strassen$(A_{11}, B_{12} - B_{22})$
$P_4 :=$ Strassen$(A_{22}, B_{21} - B_{11})$
$P_5 :=$ Strassen$(A_{11} + A_{12}, B_{22})$
$P_6 :=$ Strassen$(A_{21} - A_{11}, B_{11} + B_{12})$
$P_7 :=$ 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[?]. 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 ($S << L$), as depicted in Algorithm **??**. 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.

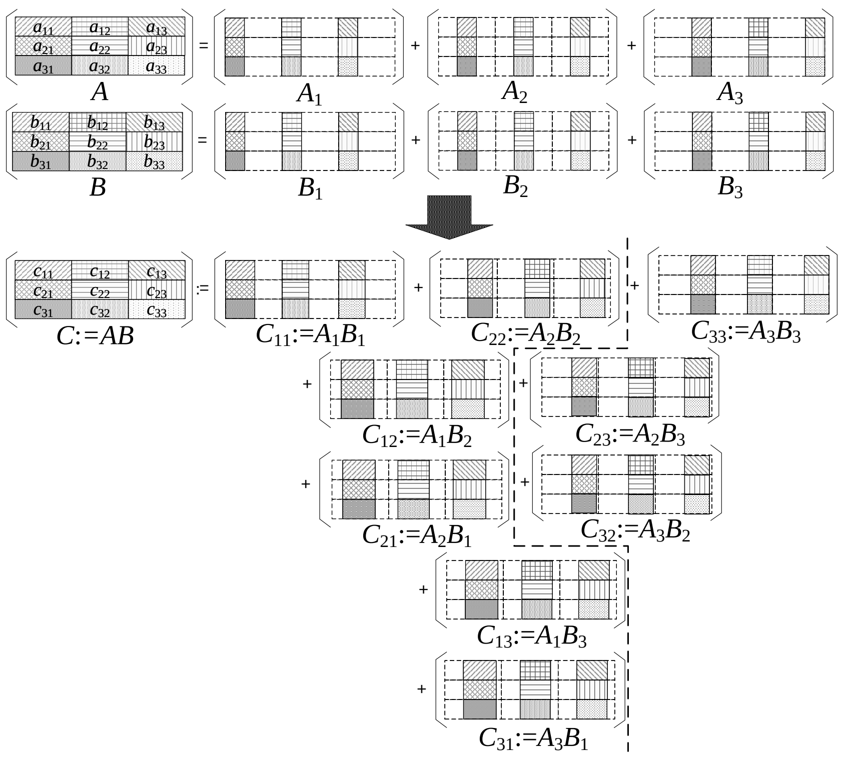
Algorithm 2 Ozaki scheme for multiple-precision matrix multiplication

```
  Input: $A \in \mathbb{F}_{bL}^{m\times l}, B \in \mathbb{F}_{bL}^{l\times n}$
  Output: $C \in \mathbb{F}_{bL}^{m\times n}$
$A^{(S)} := A,\ B^{(S)} := B :\ A^{(S)} \in \mathbb{F}_{bS}^{m\times l},\ B^{(S)} \in \mathbb{F}_{bS}^{l\times n}$
$\mathbf{e} := [1\ 1\ ...\ 1]^T \in \mathbb{F}_{bS}^{l}$
$\alpha := 1$
while $\alpha < D$ do
    $\boldsymbol{\mu}_A := [\max_{1\le p\le l} |(A^{(S)})_{ip}|]_{i=1,2,...,m} \in \mathbb{F}_{bS}^{m}$
    $\boldsymbol{\mu}_B := [\max_{1\le q\le l} |(B^{(S)})_{qj}|]_{j=1,2,...,n} \in \mathbb{F}_{bS}^{n}$
    $\boldsymbol{\tau}_A := [2^{\lceil \log_2((\boldsymbol{\mu}_A)_i)\rceil + \lceil (S+\log_2(l))/2\rceil}]_{i=1,2,...,m} \in \mathbb{F}_{bS}^{m}$
    $\boldsymbol{\tau}_B := [2^{\lceil \log_2((\boldsymbol{\mu}_B)_j)\rceil + \lceil (S+\log_2(l))/2\rceil}]_{j=1,2,...,n} \in \mathbb{F}_{bS}^{n}$
    $S_A := \boldsymbol{\tau}_A \mathbf{e}^T$
    $S_B := \mathbf{e}\boldsymbol{\tau}_B^T$
    $A_\alpha := (A^{(S)} + S_A) - S_A\colon\ A_\alpha \in \mathbb{F}_{bS}^{m\times l}$
    $B_\alpha := (B^{(S)} + S_B) - S_B\colon\ B_\alpha \in \mathbb{F}_{bS}^{l\times n}$
    $A := A - A_\alpha,\ B := B - B_\alpha$ : $L$-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 $\alpha = 1, 2, ..., D$ do
    for $\beta = 1, 2, ..., D - \alpha + 1$ do
        $C_{\alpha\beta} := A_\alpha B_\beta$
    end for
    $C := C + \sum_{\beta=1}^{D-\alpha+1} C_{\alpha\beta}$ : $L$-bit FP arithmetic
end for
```

Fig. **??** 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[**?**] as multi-component-type fixed precision C++ class library, CAMPARY[**?**] as multi-component-type arbitrary precision library, and MPFR[**?**] baased on GNU MP[**?**]'s MPN kernel. MPLAPACK/MPBLAS[**?**] is greeding these all reliable MP libraries and providing the coresponding APIs of LAPACK/BLAS[**?**]. Our BNCmatmul developed aims to get better performance than MPBLAS



a_{11} a_{12} a_{13} a_{21} a_{22} a_{23} a_{31} a_{32} a_{33}
A
A_1
A_2
A_3
b_{11} b_{12} b_{13} b_{21} b_{22} b_{23} b_{31} b_{32} b_{33}
B
B_1
B_2
B_3
c_{11} c_{12} c_{13} c_{21} c_{22} c_{23} c_{31} c_{32} c_{33}
$C:=AB$
$C_{11}:=A_1B_1$
$C_{22}:=A_2B_2$
$C_{33}:=A_3B_3$
$C_{12}:=A_1B_2$
$C_{23}:=A_2B_3$
$C_{21}:=A_2B_1$
$C_{32}:=A_3B_2$
$C_{13}:=A_1B_3$
$C_{31}:=A_3B_1$

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 wriiten in the traditional way of ANSI C, not C++.
- Supporting native multi-component-type DD, TD, and QD precision arithmetic, and also accelelating 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 algorightms 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. **??**.

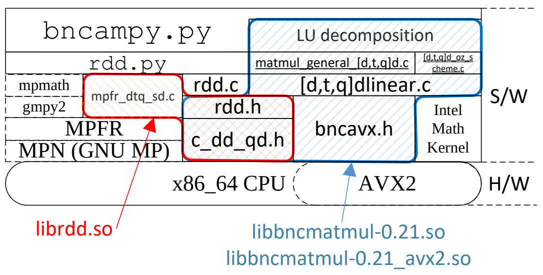


Fig. 1.3 Software layer of BNCmatmul library

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. **??**) 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 artibrary 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-precsion 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 (53bits $\times$ 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`) $\cdots$ Computes $\mathrm{fl}(a+b)$ and $\mathrm{err}(a+b)$. Assumes $|a| \geq |b|$.
- double `quick_two_diff`(double `a`, double `b`, double * `err`) $\cdots$ Computes $\mathrm{fl}(a-b)$ and $\mathrm{err}(a-b)$. Assumes $|a| >= |b|$.
- double `two_sum`(double `a`, double `b`, double * `err`) $\cdots$ Computes $\mathrm{fl}(a+b)$ and $\mathrm{err}(a+b)$.
- double `two_diff`(double `a`, double `b`, double * `err`) $\cdots$ Computes $\mathrm{fl}(a-b)$ and $\mathrm{err}(a-b)$.
- void `split`(double `a`, double *hi, double *lo) $\cdots$ divide a to hi and lo without error.
- double `two_prod`(double `a`, double `b`, double * `err`) $\cdots$ Computes $\mathrm{fl}(ab)$ and $\mathrm{err}(ab)$.
- double `two_sqr`(double `a`, double * `err`) $\cdots$ Computes $\mathrm{fl}(a^2)$ and $\mathrm{err}(a^2)$. Faster than the above method.

2.3 Double-double precision arithmetic

- dd_bool `dd_is_zero`(const double * `x`) $\cdots$ Check if *x is zero.
- dd_bool `dd_is_one`(const double * `x`) $\cdots$ Check if *x is one.
- void `c_dd_set`(const double * `x`, double *`ret`) $\cdots$ $ret := x$
- void `c_dd_set_dd_d`(const double `x`, double * `ret`) $\cdots$ $ret := (double)x$
- void `c_dd_set0`(double * `ret`) $\cdots$ $ret := 0$
- void `c_dd_setnan`(double * `ret`) $\cdots$ $ret := NaN$
- void `c_dd_neg`(const double * `a`, double * `b`) $\cdots$ $b := -a$
- dd_bool `dd_is_positive`(const double * `x`) $\cdots$ *x > 0 ?
- dd_bool `dd_is_negative`(const double * `x`) $\cdots$ *x < 0 ?
- void `c_dd_comp`(const double * `a`, const double * `b`, int * `result`) $\cdots$ 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`) $\cdots$ double-double := double + double
- void `c_dd_add`(const double * `a`, const double * `b`, double * `c`) $\cdots$ $c := a + b$
- 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`) $\cdots$ double-double = double - double
- void `c_dd_sub`(const double * `a`, const double * `b`, double * `c`) $\cdots$ $c := a - b$
- 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`) $\cdots$ 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`) $\cdots$ $c := a/b$
- void `c_dd_div`(const double * `a`, const double * `b`, double * `c`) $\cdots$ $c := a/b$
- 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) $\cdots$ $b := a^2$
- 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) $\cdots$ $b := \sqrt(a)$
- void c_dd_abs(const double * a, double * b) $\cdots$ $b := |a|$
- void c_dd_floor(const double * a, double * b) $\cdots$ $b := \mathrm{floor}(a)$
- void c_dd_ceil(const double * a, double * b) $\cdots$ $b := \mathrm{ceil}(a)$

2.4 Quadruple-double precision arithmetic

- void c_qd_copy(const double * a, double * b) $\cdots$ $b := a$
- 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) $\cdots$ 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) $\cdots$ $c := a + b$
- 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) $\cdots$ not so accurate but fast addition $c := a + b$
- void c_td_addq(const double * a, const double * b, double * c) $\cdots$ triple double addition based on quad-double way
- void c_qd_selfadd(const double * a, double * b) $\cdots$ $b := b + a$
- 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) $\cdots$ $b := -a$
- 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) $\cdots$ $c := a - b$
- 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) $\cdots$ $b := b + (-a) = b - a$
- 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) $\cdots$ $c := a \times b$
- 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) $\cdots$ $c = a^2$
- void c_qd_selfmul(const double * a, double * b) $\cdots$ $b := a * b$
- 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`) $\cdots$ $c := a/b$
- void `c_qd_div_sloppy`(const double * `a`, const double * `b`, double * `c`) $\cdots$ not so accurate but fast division (default)

(Macro) `USE_QD_DIV_ACCURATE()` $\cdots$ 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`) $\cdots$ $b := b/a$
- 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`) $\cdots$ $c := (a[0] * b, a[1] * b, a[2] * b, a[3] * b)$
- void `c_qd_set`(const double * `x`, double * `qdval`) $\cdots$ $qdval := c$
- void `c_qd_set0double *qdval`() $\cdots$ $qdval := 0$
- void `c_qd_sqrt`(const double * `a`, double * `b`) $\cdots$ $b := \sqrt{a}$
- void `c_qd_abs`(const double * `a`, double * `b`) $\cdots$ $b := |a|$

(Macro) `nint`(a) $\cdots$ round(a) as integer

- void `c_qd_nint`(const double * `a`, double * `b`)
- void `c_qd_floor`(const double * `a`, double * `b`) $\cdots$ $b :=$ floor(a)
- void `c_qd_ceil`(const double * `a`, double * `b`) $\cdots$ $b :=$ ceil(a)
- void `c_qd_comp`(const double * `a`, const double * `b`, int * `result`) $\cdots$ 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`) $\cdots$ $b := a$
- 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`) $\cdots$ $e[n] := vec\_sum(x[n])$
- void `vseb`(double * `y`, int `ny`, const double * `e`, int `ne`) $\cdots$ $y[n] := vec\_sum\_err\_branch(vseb)(k)(e[n])$
- void `c_to_td`(double * `r`, double `a`, double `b`, double `c`) $\cdots$ $r[3] := to\_td(a, b, c)$
- void `merge`(double * `c`, double * `a`, int na, double * `b`, int `nb`) $\cdots$ Merge a[na] and b[nb] into c[na + nb]
- void `c_td_add`(double * `a`, double * `b`, double * `c`) $\cdots$ $c := a + b$
- void `c_td_add_td_d`(double * `a`, double `b`, double * `c`)
- void `c_td_neg`(const double * `a`, double * `c`) $\cdots$ $c := -a$
  //
- void `c_td_sub`(double * `c`, double * `a`, double * `b`) $\cdots$ $c := a - b$
- 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`) $\cdots$ $c := a \times b$
- void `c_td_mul_sloppy`(double * `a`, double * `b`, double * `c`) $\cdots$ not so accurate but fast $c := a \times b$

(Macro) `USE_TD_MUL_ACCURATE()` $\cdots$ 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()` $\cdots$ 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) $\cdots$ $b := |a|$
- void `c_td_comp`(const double * a, const double * b, int * `result`) $\cdots$ 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()` $\cdots$ $1 + 2\times$ DBL_EPSILON $= (1.00000000000000044e + 00)$

(Macro) `ONE_M_2DBL_EPS()` $\cdots$ $1 - 2\times$ DBL_EPSILON $= (9.99999999999999556e - 01)$

- void `c_td_reci`(double * a, double * c)
- void `c_td_divt`(double * a, double * b, double * c)

(Macro) `c_td_div()` $\cdots$ the same as c_td_divtq

- void `c_td_divq`(const double * a, const double * b, double * c) $\cdots$ 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) $\cdots$ $c := \sqrt{a}$
- void `c_td_sqrt_d`(double a, double * c)
- void `c_td_sqr`(double * a, double * c) $\cdots$ $c := a^2$

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) $\cdots$ $val := 0$ 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]`) $\cdots$ DD print(no appending CR)
- int `rdd_cmp`(double `a[DDSIZE]`, double `b[DDSIZE]`) $\cdots$ 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) $\cdots$ return $\sqrt{ret}$
- void `rdd_fma`(double `ret[DDSIZE]`, double `a[DDSIZE]`, double `b[DDSIZE]`, double c[DDSIZE]) $\cdots$ $ret := a \times b + c$
- void `rdd_pow`(double `ret[DDSIZE]`, double `base[DDSIZE]`, double power[DDSIZE]) $\cdots$ ret := base$^{\text{power}}$
- void `rqd_out_str_base`(FILE *fp, int base, int length, double val[QDSIZE]) $\cdots$ print(no appending CR)
- int `rqd_cmp`(double `a[QDSIZE]`, double `b[QDSIZE]`) $\cdots$ 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) $\cdots$ return $\sqrt{ret}$
- void `rqd_fma`(double ret[QDSIZE], double `a[QDSIZE]`, double `b[QDSIZE]`, double c[QDSIZE]) $\cdots$ $ret := a \times b + c$
- void `rqd_pow`(double ret[QDSIZE], double base[QDSIZE], double power[QDSIZE]) $\cdots$ ret := 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]`) $\cdots$ return -1 if $a < b$, 0 if $a == b$, and -1 if

$a < b$.

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

(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) $\cdots$ $c := a/b$
(Macro) `rtd_add_d`(ret, a, b) $\cdots$ $c := a + b$
(Macro) `rtd_sub_d`(ret, a, b) $\cdots$ $c := a - b$
(Macro) `rtd_mul_d`(ret, a, b) $\cdots$ $c := a \times b$
(Macro) `rtd_div_ui`(ret, a, b) $\cdots$ $c := a/b$
(Macro) `rtd_add_ui`(ret, a, b) $\cdots$ $c := a + b$
(Macro) `rtd_sub_ui`(ret, a, b) $\cdots$ $c := a - b$
(Macro) `rtd_mul_ui`(ret, a, b) $\cdots$ $c := a \times b$
(Macro) `set0_qd`(val) $\cdots$ $val := 0$
(Macro) `rqd_set0`(val)
(Macro) `rqd_add`(ret, a, b) $\cdots$ $c := a + b$
(Macro) `rqd_sub`(ret, a, b) $\cdots$ $c := a - b$
(Macro) `rqd_mul`(ret, a, b) $\cdots$ $c := a \times b$
(Macro) `rqd_div`(ret, a, b) $\cdots$ $c := a/b$
(Macro) `rqd_sqrt`(ret, a) $\cdots$ $ret := \sqrt{a}$
(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) $\cdots$ $c := a/b$
(Macro) `rqd_add_d`(ret, a, b) $\cdots$ $c := a + b$
(Macro) `rqd_sub_d`(ret, a, b) $\cdots$ $c := a - b$
(Macro) `rqd_mul_d`(ret, a, b) $\cdots$ $c := a \times b$
(Macro) `rqd_div_ui`(ret, a, b) $\cdots$ $c := a/b$
(Macro) `rqd_add_ui`(ret, a, b) $\cdots$ $c := a + b$
(Macro) `rqd_sub_ui`(ret, a, b) $\cdots$ $c := a - b$
(Macro) `rqd_mul_ui`(ret, a, b) $\cdots$ $c := a \times b$

2.7 AVX2

2.7.1 float

- __m256 `_bncavx2_fabsf`(__m256 `val8`) $\cdots$ $\mathrm{val8} := \max(\mathrm{val8}, 0 - \mathrm{val8})$
- __m256 `_bncavx2_fneg`(__m256 `a`) $\cdots$ return $-a$
- void `_bncavx2_ffma`(float `ret[]`, float `a[]`, float `b[]`, float `c[]`, int `dim`) $\cdots$ ret $:= a \times b + c$
- void `_bncavx2_fmul`(float `ret[]`, float `a[]`, float `b[]`, int `dim`) $\cdots$ ret $:= a \times b$

2.7.2 double

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

2.7.3 Error-free transformation

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

2.7.4 Double-double precision arithmetic

- `void _bncavx2_set0_dd(__m256d) ret[DDSIZE]` $\cdots$ ret := 0

(Macro) `_bncavx2_rdd_set0(ret)` $\cdots$ is the same as `_bncavx2_set0_dd(ret)`

- `void _bncavx2_get_dd_m256d_i(ddfloat *ret, __m256d ret4[DDSIZE], int avx_index)` $\cdots$ ret := ret4[][avx_index]
- `void _bncavx2_rdd_sum256d(double ret[DDSIZE], __m256d ret4[DDSIZE])` $\cdots$ ret := ret4[0] + ret4[1] + ret4[2] + ret4[3]
- `void _bncavx2_rdd_abssum256d(double ret[DDSIZE], __m256d ret4[DDSIZE])` $\cdots$ ret := |ret4[0]| + |ret4[1]| + |ret4[2]| + |ret4[3]|
- `void _bncavx2_rdd_absmax256d(double ret[DDSIZE], __m256d ret4[DDSIZE])` $\cdots$ ret := max(|ret4[0]|, |ret4[1]|, |ret4[2]|, |ret4[3]|)
- `void _bncavx2_rdd_norm256d(double ret[DDSIZE], __m256d ret4[DDSIZE])` $\cdots$ ret := $\|\mathrm{ret4}[0]^2 + \mathrm{ret4}[1]^2 + \mathrm{ret4}[2]^2 + \mathrm{ret4}[3]^2\|_2$
- `void _bncavx2_rdd_add(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE])` $\cdots$ ret := $a + b$
- `void _bncavx2_rdd_sub(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE])` $\cdots$ ret := $a - b$
- `void _bncavx2_rdd_mul(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE])` $\cdots$ ret := $a \times b$
- `void _bncavx2_rdd_div(__m256d ret[DDSIZE], __m256d a[DDSIZE], __m256d b[DDSIZE])` $\cdots$ ret := a/b
- `void _bncavx2_rdd_abs(__m256d ret[DDSIZE], __m256d a[DDSIZE])` $\cdots$ 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 := $a \times b$
- 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])` ⋯ $c := a \times b$
- void `_bncavx2_rtd_mul_dd(__m256d c[TDSIZE], __m256d a[DDSIZE], __m256d b[TDSIZE])` ⋯ $c := a \times b$

(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(|ret4[0]|, |ret4[1]|, |ret4[2]|, |ret4[3]|)

- void _bncavx2_rqd_norm256d(double ret[QDSIZE], __m256d ret4[QDSIZE]) $\cdots$ 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) $\cdots$ renorm(double *c0, double *c1, double *c2, double *c3)
- void _bncavx2_renorm4(__m256d * c0, __m256d * c1, __m256d * c2, __m256d* c3, __m256d * c4) $\cdots$ renorm4(double *c0, double *c1, double *c2, double *c3, double *c4)
- void _bncavx2_three_sum(__m256d * a, __m256d * b, __m256d * c) $\cdots$ three_sum(double *a, double *b, double *c)
- void _bncavx2_three_sum2(__m256d * a, __m256d * b, __m256d * c) $\cdots$ void three_sum2(double *a, double *b, double *c)
- void _bncavx2_rqd_add(__m256d ret[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE]) $\cdots$ ret $:= a + b$
- void _bncavx2_rtd_addq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) $\cdots$ ret $:= a + b$
- void _bncavx2_rtd_mulq(__m256d ret[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) $\cdots$ ret $:= a \times b$
- void _bncavx2_rqd_mul(__m256d ret[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE]) $\cdots$ ret $:= a \times b$
- void _bncavx2_rqd_sub(__m256d c[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE]) $\cdots$ ret $:= a - b$
- void _bncavx2_rqd_mul_d(__m256d c[QDSIZE], const __m256d a[QDSIZE], __m256d b) $\cdots$ ret $:= a \times b$
- void _bncavx2_rtd_divq(__m256d c[TDSIZE], __m256d a[TDSIZE], __m256d b[TDSIZE]) $\cdots$ $c := a/b$
- void _bncavx2_rqd_div(__m256d c[QDSIZE], __m256d a[QDSIZE], __m256d b[QDSIZE])

Chapter 3

Basic linear computation

3.1 Detatypes 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`(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`) $cdots\ c := c + a$
- 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`) ··· $c := \mathrm{val} \times a$
- 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_ddvector`(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) ⋯ $c := a + val * b$
- 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) ⋯ 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) ⋯ $c := -a$
- 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) ⋯ $\|a\|_1$
- 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) ⋯ $\|a\|_\infty$
- 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) ⋯ $\|a\|_2$
- 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) ⋯ 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_tdmat(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_ddmatrix(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) $\cdots$ E lementwise 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) $\cdots$ $c := (double)a$
- void relerr_ddvector(double relerr[DDSIZE], DDVector approx_vec, DDVector true_vec, int norm_type) $\cdots$ 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) $\cdots$ E lementwise 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) $\cdots$ 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) $\cdots$ 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, QDMatrix mat)
- void fread_ddmatrix_fname(const char * fname, DDMatrix mat) $\cdots$ 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) $\cdots$ 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) $\cdots$ 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) $\cdots$ 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) $\cdots$ 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) $\cdots$ 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) $\cdots$ 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 benhmark 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 multiplicaiton
- 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`) ··· Computattion 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_mpfdmatrix_simple`(c, a, b) _bncomp_mul_mpfdmatrix(c, a, b)

- void `_bncomp_mul_ddmatrix_block`(DDMatrix `ret`, DDMatrix `mat_a`, DDMatrix `mat_b`, long int `min_dim`) ··· Parallelized block matrix multiplicaiton.
- 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) $\cdots$ ret_high + ret_low = org_vec
- void split_dmatrix(DMatrix ret_high_mat , DMatrix ret_low_mat, DMatrix org_mat) $\cdots$ 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) $\cdots$ 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) $\cdots$ Split org_vec to ret_vec[0] + ret_vec[1] +...+ 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) $\cdots$ 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) $\cdots$ $c := a + (double)b$
- 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) $\cdots$ $c := a - (double)b$
- 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) $\cdots$ $c := (DDVector)a$
- void absmax_row_ddmatrix(double mu[DDSIZE], long int * max_j, long int row_index, DDMatrix mat) $\cdots$ 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) $\cdots$ $c := a + (DMatrix)b$
- 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) $\cdots$ $c := a - (DMatrix)b$
- 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) *cdots* 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) $\cdots$ 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) $\cdots$ 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) $\cdots$ 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) $\cdots$ LU decompose the matrix A without any types of pivoting.
- int `DDLUdecomp`(DDMatrix A)
- int `TDLUdecomp`(TDMatrix A)
- int `QDLUdecomp`(QDMatrix A)
- int `MPFLUdecomp`(MFPMatrix A)
- int `SolveDLS`(DVector $\mathbf{x}$, DMatrix LU, DVector $\mathbf{b}$) $\cdots$ Solve $(LU)\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$.
- int `SolveDDLS`(DDVector $\mathbf{x}$, DDMatrix LU, DDVector $\mathbf{b}$)
- int `SolveTDLS`(TDVector $\mathbf{x}$, TDMatrix LU, TDVector $\mathbf{b}$)
- int `SolveQDLS`(QDVector $\mathbf{x}$, QDMatrix `l`LU, QDVector $\mathbf{b}$)
- int `SolveMPFLS`(MPFVector $\mathbf{x}$, MPFMatrix LU, MPFVector $\mathbf{b}$)
- int `DLUdecompP`(DMatrix A, long int `ch[]`) $\cdots$ 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`(DDMatrix A, long int `ch[]`)
- int `MPFLUdecompP`(MPFMatrix A, long int `ch[]`)
- int `SolveDLSP`(DVector $\mathbf{x}$, DMatrix LU, DVector $\mathbf{b}$, long int `ch[]`) $\cdots$ Solve $(LU)\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ using `ch[]` as row numbering.
- int `SolveDDLSP`(DDVector $\mathbf{x}$, DDMatrix LU, DDVector $\mathbf{b}$, long int `ch[]`)
- int `SolveTDLSP`(TDVector $\mathbf{x}$, TDMatrix LU, TDVector $\mathbf{b}$, long int `ch[]`)
- int `SolveQDLSP`(DDVector $\mathbf{x}$, QDMatrix LU, QDVector $\mathbf{b}$, long int `ch[]`)
- int `SolveMPFLSP`(MPFVector $\mathbf{x}$, MPFMatrix LU, MPFVector $\mathbf{b}$, long int `ch[]`)
- int `DLUdecompC`(DMatrix A, long int `row_ch[]`, long int `col_ch[]`) $\cdots$ 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 $\mathbf{x}$, DMatrix LU, DVector $\mathbf{b}$, long int `row_ch[]`, long int `col_ch[]`) $\cdots$ Solve $(LU)\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ using `row_ch[]` and `col_ch[]` as row and column numbering, respectively.
- int `SolveDDLSC`(DDVector $\mathbf{x}$, DDMatrix LU, DDVector $\mathbf{b}$, long int `row_ch[]`, long int `col_ch[]`)
- int `SolveTDLSC`(TDVector $\mathbf{x}$, TDMatrix LU, TDVector $\mathbf{b}$, long int `row_ch[]`, long int `col_ch[]`)
- int `SolveQDLSC`(QDVector $\mathbf{x}$, QDMatrix LU, QDVector $\mathbf{b}$, long int `row_ch[]`, long int `col_ch[]`)
- int `SolveMPFLSC`(MPFVector $\mathbf{x}$, MPFMatrix LU, MPFVector $\mathbf{b}$, long int `row_ch[]`, long int `col_ch[]`)

- int `DLUdecomp_strassen`(DMatrix A, long int `min_dim`) $\cdots$ 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`) $\cdots$ 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`) $\cdots$ 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`) $\cdots$ LU decompose, using partial pivoting and Ozaki scheme setting `max_num_div` as maximum number of divisions.
- int `DDLUdecomp_ozPM`(DMatrix A, long int `ch[]`, long int `min_dim`, int `max_num_div`)
- int `TDLUdecomp_ozPM`(DMatrix A, long int `ch[]`, long int `min_dim`, int `max_num_div`)
- int `QDLUdecomp_ozPM`(DMatrix 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.