

M4RI
1.0.1

Generated by Doxygen 1.6.1

Wed Nov 4 11:47:18 2009

Contents

1	Main Page	1
2	Todo List	1
3	Data Structure Index	1
3.1	Data Structures	1
4	File Index	2
4.1	File List	2
5	Data Structure Documentation	2
5.1	_mm_block Struct Reference	2
5.1.1	Detailed Description	3
5.1.2	Field Documentation	3
5.2	code Struct Reference	3
5.2.1	Detailed Description	3
5.2.2	Field Documentation	4
5.3	mzd_t Struct Reference	4
5.3.1	Detailed Description	4
5.3.2	Field Documentation	4
5.4	mzp_t Struct Reference	5
5.4.1	Detailed Description	5
5.4.2	Field Documentation	5
6	File Documentation	6
6.1	brilliantussian.h File Reference	6
6.1.1	Detailed Description	7
6.1.2	Function Documentation	7
6.2	grayflex.h File Reference	13
6.2.1	Detailed Description	13
6.2.2	Define Documentation	13
6.2.3	Function Documentation	14
6.2.4	Variable Documentation	15
6.3	lqup.h File Reference	15
6.3.1	Detailed Description	16
6.3.2	Define Documentation	16
6.3.3	Function Documentation	16

6.4	m4ri.h File Reference	20
6.4.1	Detailed Description	20
6.5	misc.h File Reference	20
6.5.1	Detailed Description	23
6.5.2	Define Documentation	24
6.5.3	Typedef Documentation	27
6.5.4	Function Documentation	27
6.5.5	Variable Documentation	30
6.6	packedmatrix.h File Reference	31
6.6.1	Detailed Description	34
6.6.2	Define Documentation	34
6.6.3	Function Documentation	35
6.7	parity.h File Reference	48
6.7.1	Detailed Description	49
6.7.2	Define Documentation	49
6.7.3	Function Documentation	50
6.8	permutation.h File Reference	50
6.8.1	Detailed Description	51
6.8.2	Function Documentation	51
6.9	pluq_mmpf.h File Reference	54
6.9.1	Detailed Description	55
6.9.2	Function Documentation	55
6.10	solve.h File Reference	56
6.10.1	Detailed Description	57
6.10.2	Function Documentation	57
6.11	strassen.h File Reference	59
6.11.1	Detailed Description	60
6.11.2	Define Documentation	60
6.11.3	Function Documentation	60
6.12	trsm.h File Reference	63
6.12.1	Detailed Description	64
6.12.2	Function Documentation	64
7	Example Documentation	67
7.1	testsuite/bench_elimination.c	67
7.2	testsuite/test_elimination.c	69
7.3	testsuite/test_lqup.c	70

7.4	testsuite/test_multiplication.c	76
-----	---	----

1 Main Page

M4RI is a library to do fast arithmetic with dense matrices over F_2 . M4RI is available under the GPLv2+ and used by the Sage mathematics software and the PolyBoRI library. See <http://m4ri.sagemath.org> for details.

2 Todo List

Global [_mzd_addmul_even](#) make sure not to overwrite crap after ncols and before width*RADIX

Global [_mzd_mul_even](#) ideally we would use the same Wmk throughout the function but some called function doesn't like that and we end up with a wrong result if we use virtual Wmk matrices. Ideally, this should be fixed not worked around. The check whether the bug has been fixed, use only one Wmk and check if `mzd_mul(4096, 3528, 4096, 2124)` still returns the correct answer.

Global [m4ri_coin_flip](#) Allow user to provide her own `random()` function.

Global [m4ri_die](#) Allow user to register callback which is called on `m4ri_die()`.

Global [m4ri_mm_calloc](#) Allow user to register `calloc` function.

Global [m4ri_mm_free](#) Allow user to register `free` function.

Global [m4ri_mm_malloc](#) Allow user to register `malloc` function.

Global [m4ri_random_word](#) Allow user to provide her own `random()` function.

Global [mzd_combine](#) this `code` is slow if `offset!=0`

Global [mzd_randomize](#) Allow the user to provide a RNG callback.

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

[_mm_block](#)

2

code (Gray codes)	3
mzd_t (Dense matrices over GF(2))	4
mzp_t (Permutations)	5

4 File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

brilliantrussian.h (M4RI and M4RM)	6
grayflex.h (Gray code implementation)	13
lqup.h (PLUQ matrix decomposition routines)	15
m4ri.h (Main include file for the M4RI library)	20
misc.h (Helper functions)	20
packedmatrix.h (Dense matrices over GF(2) represented as a bit field)	31
parity.h (Compute the parity of 64 words in parallel)	48
permutation.h (Permutation matrices)	50
pluq_mmpf.h (LQUP factorization using Gray codes)	54
solve.h (System solving with matrix routines)	56
strassen.h (Matrix operations using Strassen's formulas including Winograd's improvements)	59
trsm.h (Triangular system solving with Matrix routines)	63

5 Data Structure Documentation

5.1 `_mm_block` Struct Reference

```
#include <misc.h>
```

Data Fields

- `size_t size`
- `void * data`

5.1.1 Detailed Description

The mmc memory management functions check a cache for re-usable unused memory before asking the system for it.

5.1.2 Field Documentation

5.1.2.1 void* _mm_block::data

Pointer to buffer of data.

5.1.2.2 size_t _mm_block::size

Size in bytes of the data.

The documentation for this struct was generated from the following file:

- [misc.h](#)

5.2 code Struct Reference

Gray codes.

```
#include <grayflex.h>
```

Data Fields

- int * [ord](#)
- int * [inc](#)

5.2.1 Detailed Description

Gray codes. A codestruct represents one entry in the [code](#) book, i.e. it represents a Gray [code](#) of a given length.

For example the Gray [code](#) table of length 2^3 is:

```
-----
| i | ord | inc |
-----
| 0 | 0 | 0 |
| 1 | 4 | 1 |
| 2 | 6 | 0 |
| 3 | 2 | 2 |
| 4 | 3 | 0 |
| 5 | 7 | 1 |
| 6 | 5 | 0 |
| 7 | 1 | 2 |
-----
```

*

5.2.2 Field Documentation

5.2.2.1 int* code::inc

increment

5.2.2.2 int* code::ord

array of of Gray [code](#) entries

The documentation for this struct was generated from the following file:

- [grayflex.h](#)

5.3 mzd_t Struct Reference

Dense matrices over GF(2).

```
#include <packedmatrix.h>
```

Data Fields

- [mmb_t](#) * [blocks](#)
- [size_t](#) [nrows](#)
- [size_t](#) [ncols](#)
- [size_t](#) [width](#)
- [size_t](#) [offset](#)
- [word](#) ** [rows](#)

5.3.1 Detailed Description

Dense matrices over GF(2). The most fundamental data type in this library.

Examples:

[testsuite/bench_elimination.c](#), [testsuite/test_elimination.c](#), [testsuite/test_lqup.c](#), and [testsuite/test_multiplication.c](#).

5.3.2 Field Documentation

5.3.2.1 mmb_t* mzd_t::blocks

Contains pointers to the actual blocks of memory containing the values packed into words of size RADIX.

5.3.2.2 size_t mzd_t::ncols

Number of columns.

5.3.2.3 size_t mzd_t::nrows

Number of rows.

5.3.2.4 size_t mzd_t::offset

column offset of the first column.

5.3.2.5 word mzd_t::rows**

Address of first word in each row, so the first word of row *i* is `m->rows[i]`

5.3.2.6 size_t mzd_t::width

`width = ceil(ncols/RADIX)`

The documentation for this struct was generated from the following file:

- [packedmatrix.h](#)

5.4 mzp_t Struct Reference

Permutations.

```
#include <permutation.h>
```

Data Fields

- [size_t * values](#)
- [size_t length](#)

5.4.1 Detailed Description

Permutations.

Examples:

[testsuite/test_lqup.c](#).

5.4.2 Field Documentation**5.4.2.1 size_t mzp_t::length**

The length of the swap array.

5.4.2.2 size_t* mzp_t::values

The swap operations in LAPACK format.

The documentation for this struct was generated from the following file:

- [permutation.h](#)

6 File Documentation

6.1 brilliantrussian.h File Reference

```
M4RI and M4RM. #include <math.h>
#include <string.h>
#include <stdlib.h>
#include "misc.h"
#include "packedmatrix.h"
#include "permutation.h"
```

Defines

- #define [M4RM_GRAY8](#)
If defined 8 Gray code tables are used in parallel.

Functions

- void [mzd_make_table](#) ([mzd_t](#) *M, size_t r, size_t c, int k, [mzd_t](#) *T, size_t *L)
Constructs all possible 2^k row combinations using the gray code table.
- void [mzd_process_rows](#) ([mzd_t](#) *M, size_t startrow, size_t endrow, size_t startcol, int k, [mzd_t](#) *T, size_t *L)
The function looks up k bits from position i, startcol in each row and adds the appropriate row from T to the row i.
- void [mzd_process_rows2](#) ([mzd_t](#) *M, size_t startrow, size_t endrow, size_t startcol, int k, [mzd_t](#) *T0, size_t *L0, [mzd_t](#) *T1, size_t *L1)
Same as mzd_process_rows but works with two Gray code tables in parallel.
- void [mzd_process_rows3](#) ([mzd_t](#) *M, size_t startrow, size_t endrow, size_t startcol, int k, [mzd_t](#) *T0, size_t *L0, [mzd_t](#) *T1, size_t *L1, [mzd_t](#) *T2, size_t *L2)
Same as mzd_process_rows but works with three Gray code tables in parallel.
- void [mzd_process_rows4](#) ([mzd_t](#) *M, size_t startrow, size_t endrow, size_t startcol, int k, [mzd_t](#) *T0, size_t *L0, [mzd_t](#) *T1, size_t *L1, [mzd_t](#) *T2, size_t *L2, [mzd_t](#) *T3, size_t *L3)
Same as mzd_process_rows but works with four Gray code tables in parallel.
- size_t [mzd_echelonize_m4ri](#) ([mzd_t](#) *M, int full, int k)
- void [mzd_top_echelonize_m4ri](#) ([mzd_t](#) *M, int k)
Given a matrix in upper triangular form compute the reduced row echelon form of that matrix.
- [mzd_t](#) * [mzd_invert_m4ri](#) ([mzd_t](#) *M, [mzd_t](#) *I, int k)
Invert the matrix M using Konrod's method.

- `mzd_t * mzd_mul_m4rm (mzd_t *C, mzd_t *A, mzd_t *B, int k)`
Matrix multiplication using Konrod's method, i.e. compute C such that C == AB.
- `mzd_t * mzd_addmul_m4rm (mzd_t *C, mzd_t *A, mzd_t *B, int k)`
- `mzd_t * _mzd_mul_m4rm (mzd_t *C, mzd_t *A, mzd_t *B, int k, int clear)`
Matrix multiplication using Konrod's method, i.e. compute C such that C == AB.

6.1.1 Detailed Description

M4RI and M4RM.

Author:

Gregory Bard <bard@fordham.edu>
Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

Note:

For reference see Gregory Bard; Accelerating Cryptanalysis with the Method of Four Russians; 2006;
<http://eprint.iacr.org/2006/251.pdf>

6.1.2 Function Documentation

6.1.2.1 `mzd_t* _mzd_mul_m4rm (mzd_t * C, mzd_t * A, mzd_t * B, int k, int clear)`

Matrix multiplication using Konrod's method, i.e. compute C such that C == AB. This is the actual implementation.

Parameters:

C Preallocated product matrix.
A Input matrix A
B Input matrix B
k M4RI parameter, may be 0 for auto-choose.
clear clear the matrix C first

Author:

Martin Albrecht -- initial implementation
William Hart -- block matrix implementation, use of several Gray [code](#) tables, general speed-ups

Ignores offset attribute of packedmatrix.

Returns:

Pointer to C.

The algorithm proceeds as follows:

Step 1. Make a Gray [code](#) table of all the 2^k linear combinations of the k rows of B_i . Call the x -th row T_x .

Step 2. Read the entries $a_{j,(i-1)k+1}, a_{j,(i-1)k+2}, \dots, a_{j,(i-1)k+k}$.

Let x be the k bit binary number formed by the concatenation of $a_{j,(i-1)k+1}, \dots, a_{j,ik}$.

Step 3. for $h = 1, 2, \dots, c$ do calculate $C_{jh} = C_{jh} + T_{xh}$.

6.1.2.2 `mzd_t* mzd_addmul_m4rm (mzd_t * C, mzd_t * A, mzd_t * B, int k)`

Set C to C + AB using Konrod's method.

Parameters:

C Preallocated product matrix, may be NULL for zero matrix.

A Input matrix A

B Input matrix B

k M4RI parameter, may be 0 for auto-choose.

Ignores offset attribute of packedmatrix.

Returns:

Pointer to C.

Examples:

[testsuite/test_multiplication.c](#).

6.1.2.3 `size_t mzd_echelonize_m4ri (mzd_t * M, int full, int k)`

General algorithm

- Step 1. Denote the first column to be processed in a given iteration as a_i . Then, perform Gaussian elimination on the first $3k$ rows after and including the i -th row to produce an identity matrix in $a_{i,i} \dots a_{i+k-1,i+k-1}$, and zeroes in $a_{i+k,i} \dots a_{i+3k-1,i+k-1}$.
- Step 2. Construct a table consisting of the 2^k binary strings of length k in a Gray [code](#). Thus with only 2^k vector additions, all possible linear combinations of these k rows have been precomputed.
- Step 3. One can rapidly process the remaining rows from $i + 3k$ until row m (the last row) by using the table. For example, suppose the j -th row has entries $a_{j,i} \dots a_{j,i+k-1}$ in the columns being processed. Selecting the row of the table associated with this k -bit string, and adding it to row j will force the k columns to zero, and adjust the remaining columns from $i + k$ to n in the appropriate way, as if Gaussian elimination had been performed.
- Step 4. While the above form of the algorithm will reduce a system of boolean linear equations to unit upper triangular form, and thus permit a system to be solved with back substitution, the M4RI algorithm can also be used to invert a matrix, or put the system into reduced row echelon form (RREF). Simply run Step 3 on rows $0 \dots i - 1$ as well as on rows $i + 3k \dots m$. This only affects the complexity slightly, changing the 2.5 coefficient to 3.

Attention:

This function implements a variant of the algorithm described above.

Examples:

[testsuite/bench_elimination.c](#), and [testsuite/test_elimination.c](#).

6.1.2.4 mzd_t* mzd_invert_m4ri (mzd_t * M , mzd_t * I , int k)

Invert the matrix M using Konrod's method. To avoid recomputing the identity matrix over and over again, I may be passed in as identity parameter.

Parameters:

M Matrix to be reduced.

I Identity matrix.

k M4RI parameter, may be 0 for auto-choose.

Ignores offset attribute of packedmatrix.

Returns:

Inverse of M

Note:

This function allocates a new matrix for the inverse which must be free'd using [mzd_free\(\)](#) once not needed anymore.

6.1.2.5 void mzd_make_table (mzd_t * M , size_t r , size_t c , int k , mzd_t * T , size_t * L)

Constructs all possible 2^k row combinations using the gray [code](#) table.

Parameters:

M matrix to operate on

r the starting row

c the starting column (only exact up to block)

k

T preallocated matrix of dimension 2^k x $m \rightarrow$ ncols

L preallocated table of length 2^k

Ignores offset attribute of packedmatrix.

6.1.2.6 `mzd_t* mzd_mul_m4rm (mzd_t * C, mzd_t * A, mzd_t * B, int k)`

Matrix multiplication using Konrod's method, i.e. compute C such that $C == AB$. This is the convenient wrapper function, please see `_mzd_mul_m4rm` for authors and implementation details.

Parameters:

- C* Preallocated product matrix, may be NULL for automatic creation.
- A* Input matrix A
- B* Input matrix B
- k* M4RI parameter, may be 0 for auto-choose.

Ignores offset attribute of packedmatrix.

Returns:

Pointer to C.

Examples:

[testsuite/test_multiplication.c](#).

6.1.2.7 `void mzd_process_rows (mzd_t * M, size_t startrow, size_t endrow, size_t startcol, int k, mzd_t * T, size_t * L)`

The function looks up *k* bits from position *i*, *startcol* in each row and adds the appropriate row from T to the row *i*. This process is iterated for *i* from *startrow* to *stoprow* (exclusive).

Parameters:

- M* Matrix to operate on
- startrow* top row which is operated on
- endrow* bottom row which is operated on
- startcol* Starting column for addition
- k* M4RI parameter
- T* contains the correct row to be added
- L* Contains row number to be added

Ignores offset attribute of packedmatrix.

6.1.2.8 void `mzd_process_rows2` (`mzd_t * M`, `size_t startrow`, `size_t endrow`, `size_t startcol`, `int k`, `mzd_t * T0`, `size_t * L0`, `mzd_t * T1`, `size_t * L1`)

Same as `mzd_process_rows` but works with two Gray [code](#) tables in parallel.

Parameters:

M Matrix to operate on
startrow top row which is operated on
endrow bottom row which is operated on
startcol Starting column for addition
k M4RI parameter
T0 contains the correct row to be added
L0 Contains row number to be added
T1 contains the correct row to be added
L1 Contains row number to be added

Ignores offset attribute of packedmatrix.

6.1.2.9 void `mzd_process_rows3` (`mzd_t * M`, `size_t startrow`, `size_t endrow`, `size_t startcol`, `int k`, `mzd_t * T0`, `size_t * L0`, `mzd_t * T1`, `size_t * L1`, `mzd_t * T2`, `size_t * L2`)

Same as `mzd_process_rows` but works with three Gray [code](#) tables in parallel.

Parameters:

M Matrix to operate on
startrow top row which is operated on
endrow bottom row which is operated on
startcol Starting column for addition
k M4RI parameter
T0 contains the correct row to be added
L0 Contains row number to be added
T1 contains the correct row to be added
L1 Contains row number to be added
T2 contains the correct row to be added
L2 Contains row number to be added

Ignores offset attribute of packedmatrix.

6.1.2.10 void `mzd_process_rows4` (`mzd_t * M`, `size_t startrow`, `size_t endrow`, `size_t startcol`, `int k`, `mzd_t * T0`, `size_t * L0`, `mzd_t * T1`, `size_t * L1`, `mzd_t * T2`, `size_t * L2`, `mzd_t * T3`, `size_t * L3`)

Same as `mzd_process_rows` but works with four Gray [code](#) tables in parallel.

Parameters:

- M* Matrix to operate on
- startrow* top row which is operated on
- endrow* bottom row which is operated on
- startcol* Starting column for addition
- k* M4RI parameter
- T0* contains the correct row to be added
- L0* Contains row number to be added
- T1* contains the correct row to be added
- L1* Contains row number to be added
- T2* contains the correct row to be added
- L2* Contains row number to be added
- T3* contains the correct row to be added
- L3* Contains row number to be added

Ignores offset attribute of packedmatrix.

6.1.2.11 void `mzd_top_echelonize_m4ri` (`mzd_t * M`, `int k`)

Given a matrix in upper triangular form compute the reduced row echelon form of that matrix.

Parameters:

- M* Matrix to be reduced.
- k* M4RI parameter, may be 0 for auto-choose.

Ignores offset attribute of packedmatrix.

Note:

This function isn't as optimized as it should be.

Examples:

[testsuite/test_elimination.c](#).

6.2 grayflex.h File Reference

Gray `code` implementation. `#include "misc.h"`

Data Structures

- struct `code`
Gray codes.

Defines

- `#define MAXKAY 16`

Functions

- int `m4ri_swap_bits` (int v, int l)
- int `m4ri_gray_code` (int i, int l)
- void `m4ri_build_code` (int *ord, int *inc, int l)
- void `m4ri_build_all_codes` (void)
Generates global `code` book.
- void `m4ri_destroy_all_codes` (void)
- int `m4ri_opt_k` (int a, int b, int c)
Return the optimal var k for the given parameters.

Variables

- `code ** codebook`

6.2.1 Detailed Description

Gray `code` implementation. The Gray `code` is a binary numeral system where two successive values differ in only one digit.

Author:

Gregory Bard <bard@fordham.edu>
Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

6.2.2 Define Documentation

6.2.2.1 `#define MAXKAY 16`

Maximum allowed value for k.

6.2.3 Function Documentation

6.2.3.1 void m4ri_build_all_codes (void)

Generates global `code` book. This function is called automatically when the shared library is loaded.

Warning:

Not thread safe!

6.2.3.2 void m4ri_build_code (int * ord, int * inc, int l)

Fills var ord and var inc with Gray `code` data for a Gray `code` of length 2^l .

Parameters:

ord Will hold gray `code` data, must be preallocated with correct size

inc Will hold some increment data, must be preallocated with correct size

l Logarithm of length of Gray `code`.

Note:

Robert Miller had the idea for a non-recursive implementation.

6.2.3.3 void m4ri_destroy_all_codes (void)

Frees memory from the global `code` book.

This function is called automatically when the shared library is unloaded.

Warning:

Not thread safe!

6.2.3.4 int m4ri_gray_code (int i, int l)

Returns the *i*-th gray `code` entry for a gray `code` of length 2^l .

Parameters:

i The index in the Gray `code` table.

l Length of the Gray `code`.

Returns:

i-th Gray `code` entry.

6.2.3.5 int m4ri_opt_k (int a, int b, int c)

Return the optimal var k for the given parameters. If var c != 0 then var k for multiplication is returned, else var k for inversion. The optimal var k here means $0.75\log_2(n)$ where n is $\min(a, b)$ for inversion and b for multiplication.

Parameters:

- a* Number of rows of (first) matrix
- b* Number of columns of (first) matrix
- c* Number of columns of second matrix (may be 0)

Returns:

k

6.2.3.6 int m4ri_swap_bits (int v, int l)

Swaps l bits in v.

Warning:

Upper bits of return value may contain garbage after operation.

6.2.4 Variable Documentation**6.2.4.1 code** codebook**

Global codebook.

Warning:

Not thread safe!

6.3 lqup.h File Reference

```
PLUQ matrix decomposition routines. #include "misc.h"
#include "packedmatrix.h"
```

Defines

- #define [LQUP_CUTOFF](#) MIN(524288,CPU_L2_CACHE>>3)

Functions

- [size_t mzd_pluq](#) ([mzd_t *A](#), [mzp_t *P](#), [mzp_t *Q](#), const int cutoff)
PLUQ matrix decomposition.

- `size_t mzd_lqup (mzd_t *A, mzp_t *P, mzp_t *Q, const int cutoff)`
LQUP matrix decomposition.
- `size_t _mzd_pluq (mzd_t *A, mzp_t *P, mzp_t *Q, const int cutoff)`
PLUQ matrix decomposition.
- `size_t _mzd_lqup (mzd_t *A, mzp_t *P, mzp_t *Qt, const int cutoff)`
LQUP matrix decomposition.
- `size_t _mzd_pluq_naive (mzd_t *A, mzp_t *P, mzp_t *Q)`
PLUQ matrix decomposition (naive base case).
- `size_t _mzd_lqup_naive (mzd_t *A, mzp_t *P, mzp_t *Qt)`
LQUP matrix decomposition (naive base case).
- `size_t mzd_echelonize_pluq (mzd_t *A, int full)`
(Reduced) row echelon form using PLUQ factorisation.

6.3.1 Detailed Description

PLUQ matrix decomposition routines.

Author:

Clement Pernet <clement.pernet@gmail.com>

Note:

This file should be called pluq.h and will be renamed in the future.

6.3.2 Define Documentation

6.3.2.1 `#define LQUP_CUTOFF MIN(524288,CPU_L2_CACHE)>>3)`

Crossover point for PLUQ factorization.

6.3.3 Function Documentation

6.3.3.1 `size_t _mzd_lqup (mzd_t *A, mzp_t *P, mzp_t *Qt, const int cutoff)`

LQUP matrix decomposition. See [mzd_lqup\(\)](#) for details.

Parameters:

A Input matrix

P Output row `mzp_t` matrix

Qt Output column `mzp_t` matrix

cutoff Minimal dimension for Strassen recursion.

See also:

[mzd_lqup\(\)](#)

Ignores offset attribute of packedmatrix.

Returns:

Rank of A.

6.3.3.2 size_t mzd_lqup_naive (mzd_t * A, mzp_t * P, mzp_t * Qt)

LQUP matrix decomposition (naive base case). See [mzd_lqup\(\)](#) for details.

Parameters:

A Input matrix

P Output row [mzp_t](#) matrix

Qt Output column [mzp_t](#) matrix

See also:

[mzd_lqup\(\)](#)

Ignores offset attribute of packedmatrix.

Returns:

Rank of A.

6.3.3.3 size_t mzd_pluq (mzd_t * A, mzp_t * P, mzp_t * Q, const int cutoff)

PLUQ matrix decomposition. See [mzd_pluq\(\)](#) for details.

Parameters:

A Input matrix

P Output row [mzp_t](#) matrix

Q Output column [mzp_t](#) matrix

cutoff Minimal dimension for Strassen recursion.

See also:

[mzd_pluq\(\)](#)

Ignores offset attribute of packedmatrix.

Returns:

Rank of A.

6.3.3.4 `size_t mzd_pluq_naive (mzd_t * A, mzp_t * P, mzp_t * Q)`

PLUQ matrix decomposition (naive base case). See [mzd_pluq\(\)](#) for details.

Parameters:

A Input matrix

P Output row [mzp_t](#) matrix

Q Output column [mzp_t](#) matrix

See also:

[mzd_pluq\(\)](#)

Ignores offset attribute of packedmatrix.

Returns:

Rank of A.

6.3.3.5 `size_t mzd_echelonize_pluq (mzd_t * A, int full)`

(Reduced) row echelon form using PLUQ factorisation.

Parameters:

A Matrix.

full Return the reduced row echelon form, not only upper triangular form.

Ignores offset attribute of packedmatrix.

See also:

[mzd_pluq\(\)](#)

Returns:

Rank of A.

Examples:

[testsuite/bench_elimination.c](#), and [testsuite/test_elimination.c](#).

6.3.3.6 `size_t mzd_lqup (mzd_t * A, mzp_t * P, mzp_t * Q, const int cutoff)`

LQUP matrix decomposition. Computes the transposed LQUP matrix decomposition using a block recursive algorithm.

If (L, Q, U, P) satisfy $LQUP = A^T$, it returns (L, Q^T, U, P) .

P and Q must be preallocated but they don't have to be identity permutations. If `cutoff` is zero a value is chosen automatically. It is recommended to set `cutoff` to zero for most applications.

This is the wrapper function including bounds checks. See [_mzd_lqup\(\)](#) for implementation details.

Parameters:

- A* Input $m \times n$ matrix
- P* Output row permutation of length m
- Q* Output column permutation matrix of length n
- cutoff* Minimal dimension for Strassen recursion.

See also:

[_mzd_lqup\(\)](#) [_mzd_pluq\(\)](#) [_mzd_pluq_mmpf\(\)](#) [mzd_echelonize_pluq\(\)](#)

Ignores offset attribute of packedmatrix.

Returns:

Rank of A .

6.3.3.7 `size_t mzd_pluq (mzd_t * A, mzp_t * P, mzp_t * Q, const int cutoff)`

PLUQ matrix decomposition. If (P, L, U, Q) satisfy $PLUQ = A$, it returns (P, L, U, Q^T) .

P and Q must be preallocated but they don't have to be identity permutations. If `cutoff` is zero a value is chosen automatically. It is recommended to set `cutoff` to zero for most applications.

The row echelon form (not reduced) can be read from the upper triangular matrix U . See [mzd_echelonize_pluq\(\)](#) for details.

This is the wrapper function including bounds checks. See [_mzd_pluq\(\)](#) for implementation details.

Parameters:

- A* Input $m \times n$ matrix
- P* Output row permutation of length m
- Q* Output column permutation matrix of length n
- cutoff* Minimal dimension for Strassen recursion.

See also:

[_mzd_pluq\(\)](#) [_mzd_pluq_mmpf\(\)](#) [mzd_echelonize_pluq\(\)](#)

Ignores offset attribute of packedmatrix.

Returns:

Rank of A.

Examples:

[testsuite/test_lqup.c](#).

6.4 m4ri.h File Reference

Main include file for the M4RI library. #include <stdio.h>

```
#include <stdlib.h>
#include <math.h>
#include "permutation.h"
#include "packedmatrix.h"
#include "brilliantussian.h"
#include "strassen.h"
#include "grayflex.h"
#include "parity.h"
#include "trsm.h"
#include "lqup.h"
#include "pluq_mmpf.h"
#include "solve.h"
```

6.4.1 Detailed Description

Main include file for the M4RI library.

Author:

Gregory Bard <bard@fordham.edu>
Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

6.5 misc.h File Reference

Helper functions. #include <stdlib.h>

```
#include <assert.h>
#include <string.h>
```

Data Structures

- [struct _mm_block](#)

Defines

- #define **RADIX** (sizeof(word)<<3)
The number of bits in a word.
- #define **ONE** ((word)1)
The number one as a word.
- #define **FFFF** ((word)0xfffffffffull)
The number $2^{64}-1$ as a word.
- #define **MAX**(x, y) (((x) > (y))?(x):(y))
Return the maximal element of x and y.
- #define **MIN**(x, y) (((x) < (y))?(x):(y))
Return the minimal element of x and y.
- #define **DIV_CEIL**(x, y) (((x)%(y))?(x)/(y)+1:(x)/(y))
Return r such that x elements fit into r blocks of length y.
- #define **TRUE** 1
Pretty for 1.
- #define **FALSE** 0
Pretty for 0.
- #define **TWOPOW**(i) (ONE<<(i))
 2^i
- #define **CLR_BIT**(w, spot) ((w) &= ~(ONE<<(RADIX - (spot) - 1)))
Clear the bit spot (counting from the left) in the word w.
- #define **SET_BIT**(w, spot) ((w) |= (ONE<<(RADIX - (spot) - 1)))
Set the bit spot (counting from the left) in the word w.
- #define **GET_BIT**(w, spot) (((w) & (ONE<<(RADIX - (spot) - 1))) >> (RADIX - (spot) - 1))
Get the bit spot (counting from the left) in the word w.
- #define **WRITE_BIT**(w, spot, value) ((w) = (((w) & ~(ONE<<(RADIX - (spot) - 1))) | (((word)(value))<<(RADIX - (spot) - 1))))
Write the value to the bit spot in the word w.
- #define **FLIP_BIT**(w, spot) ((w) ^= (ONE<<(RADIX - (spot) - 1)))
Flip the spot in the word w.
- #define **LEFTMOST_BITS**(w, n) ((w) & ~((ONE<<(RADIX-(n))-1))>>(RADIX-(n)))
Return the n leftmost bits of the word w.
- #define **RIGHTMOST_BITS**(w, n) (((w)<<(RADIX-(n)-1))>>(RADIX-(n)-1))
Return the n rightmost bits of the word w.

- #define `LEFT_BITMASK(n)` ($\sim((\text{ONE} \ll ((\text{RADIX} - (n \% \text{RADIX}))\% \text{RADIX})) - 1)$)
creat a bit mask to zero out all but the nRADIX leftmost bits.
- #define `RIGHT_BITMASK(n)` ($\text{FFFF} \gg ((\text{RADIX} - (n \% \text{RADIX}))\% \text{RADIX})$)
creat a bit mask to zero out all but the nRADIX rightmost bits.
- #define `BITMASK(n)` ($\text{ONE} \ll (\text{RADIX} - ((n \% \text{RADIX}) - 1)$)
creat a bit mask to zero out all but the nRADIX bit.
- #define `ALIGNMENT(addr, n)` ($((\text{unsigned long})(\text{addr})) \% (n)$)
Return alignment of addr w.r.t. n. For example the address 17 would be 1 aligned w.r.t. 16.
- #define `CPU_L2_CACHE` 524288
- #define `CPU_L1_CACHE` 16384
- #define `MM_MAX_MALLOC` ($(1\text{ULL}) \ll 30$)
Maximum number of bytes allocated in one malloc() call.
- #define `ENABLE_MMC`
Enable memory block cache (default: disabled).
- #define `M4RI_MMC_NBLOCKS` 16
Number of blocks that are cached.
- #define `M4RI_MMC_THRESHOLD` `CPU_L2_CACHE`
Maximal size of blocks stored in cache.

Typedefs

- typedef unsigned long long `word`
- typedef unsigned char `BIT`
Pretty for unsigned char.
- typedef struct `_mm_block` `mmb_t`

Functions

- static int `leftmost_bit` (`word` a)
- void `m4ri_die` (`const char *errormessage,...`)
Print error message and abort().
- void `m4ri_word_to_str` (`char *destination, word data, int colon`)
Write a sting representing the word data to destination.
- static `BIT` `m4ri_coin_flip` ()
Return 1 or 0 uniformly randomly distributed.
- `word` `m4ri_random_word` ()

Return uniformly randomly distributed random word.

- void [m4ri_init](#) (void)
Initialize global data structures for the M4RI library.
- void [m4ri_fini](#) (void)
De-initialize global data structures from the M4RI library.
- static void * [m4ri_mm_calloc](#) (int count, int size)
Calloc wrapper.
- static void * [m4ri_mm_malloc](#) (int size)
Malloc wrapper.
- static void [m4ri_mm_free](#) (void *condemned,...)
Free wrapper.
- static [mmb_t](#) * [m4ri_mmc_handle](#) (void)
Return handle for locale memory management cache.
- static void * [m4ri_mmc_malloc](#) (size_t size)
Allocate size bytes.
- static void * [m4ri_mmc_calloc](#) (size_t size, size_t count)
Allocate size times count zeroed bytes.
- static void [m4ri_mmc_free](#) (void *condemned, size_t size)
Free the data pointed to by condemned of the given size.
- static void [m4ri_mmc_cleanup](#) (void)
Cleans up the cache.

Variables

- [mmb_t m4ri_mmc_cache](#) [M4RI_MMC_NBLOCKS]

6.5.1 Detailed Description

Helper functions.

Author:

Gregory Bard <bard@fordham.edu>
Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

6.5.2 Define Documentation

6.5.2.1 #define ALIGNMENT(addr, n) (((unsigned long)(addr))%(n))

Return alignment of *addr* w.r.t. *n*. For example the address 17 would be 1 aligned w.r.t. 16.

Parameters:

addr

n

6.5.2.2 #define BITMASK(n) (ONE<<(RADIX-((n)%RADIX)-1))

creat a bit mask to zero out all but the *n*RADIX bit.

Parameters:

n Integer

6.5.2.3 #define CLR_BIT(w, spot) ((w) &= ~(ONE<<(RADIX - (spot) - 1)))

Clear the bit *spot* (counting from the left) in the word *w*.

Parameters:

w Word

spot Integer with $0 \leq \text{spot} < \text{RADIX}$

6.5.2.4 #define CPU_L1_CACHE 16384

Fix some standard value for L1 cache size if it couldn't be determined by configure.

6.5.2.5 #define CPU_L2_CACHE 524288

Fix some standard value for L2 cache size if it couldn't be determined by configure.

6.5.2.6 #define DIV_CEIL(x, y) (((x)%(y))?((x)/(y)+1):(x)/(y))

Return *r* such that *x* elements fit into *r* blocks of length *y*.

Parameters:

x Number of elements

y Block size

6.5.2.7 #define FLIP_BIT(*w*, *spot*) $((w) \wedge= (ONE \ll (RADIX - (spot) - 1)))$

Flip the spot in the word *w*.

Parameters:

w Word.
spot Integer with $0 \leq spot < RADIX$.

6.5.2.8 #define GET_BIT(*w*, *spot*) $((w) \& (ONE \ll (RADIX - (spot) - 1))) \gg (RADIX - (spot) - 1)$

Get the bit spot (counting from the left) in the word *w*.

Parameters:

w Word
spot Integer with $0 \leq spot < RADIX$

6.5.2.9 #define LEFT_BITMASK(*n*) $(\sim((ONE \ll ((RADIX - (n \% RADIX)) \% RADIX)) - 1))$

creat a bit mask to zero out all but the *n*RADIX leftmost bits.

Parameters:

n Integer

6.5.2.10 #define LEFTMOST_BITS(*w*, *n*) $((w) \& \sim((ONE \ll (RADIX-(n)))-1)) \gg (RADIX-(n))$

Return the *n* leftmost bits of the word *w*.

Parameters:

w Word
n Integer with $0 \leq spot < RADIX$

6.5.2.11 #define MAX(*x*, *y*) $((x) > (y))?(x):(y)$

Return the maximal element of *x* and *y*.

Parameters:

x Word
y Word

6.5.2.12 #define MIN(x, y) (((x) < (y))?(x):(y))

Return the minimal element of x and y.

Parameters:

x Word
y Word

6.5.2.13 #define RIGHT_BITMASK(n) (FFFF>>(RADIX - (n%RADIX))%RADIX)

creat a bit mask to zero out all but the nRADIX rightmost bits.

Parameters:

n Integer

Warning:

Does not handle multiples of RADIX correctly

6.5.2.14 #define RIGHTMOST_BITS(w, n) (((w)<<(RADIX-(n)-1))>>(RADIX-(n)-1))

Return the n rightmost bits of the word w.

Parameters:

w Word
n Integer with $0 \leq \text{spot} < \text{RADIX}$

6.5.2.15 #define SET_BIT(w, spot) ((w) |= (ONE<<(RADIX - (spot) - 1)))

Set the bit spot (counting from the left) in the word w.

Parameters:

w Word
spot Integer with $0 \leq \text{spot} < \text{RADIX}$

6.5.2.16 #define TWOPOW(i) (ONE<<(i))

2^i

Parameters:

i Integer.

```
6.5.2.17 #define WRITE_BIT(w, spot, value) (((w) = (((w) &~(ONE<<(RADIX - (spot) - 1))) |  
(((word)(value))<<(RADIX - (spot) - 1))))
```

Write the value to the bit spot in the word w.

Parameters:

w Word.

spot Integer with $0 \leq \text{spot} < \text{RADIX}$.

value Either 0 or 1.

6.5.3 Typedef Documentation

6.5.3.1 typedef struct _mm_block mmb_t

The mmc memory management functions check a cache for re-usable unused memory before asking the system for it.

6.5.3.2 typedef unsigned long long word

A word is the typical packed data structure to represent packed bits.

6.5.4 Function Documentation

6.5.4.1 static int leftmost_bit (word *a*) [*inline*, *static*]

Return the index of the leftmost bit in *a* for a nonzero.

Parameters:

a Word

6.5.4.2 static BIT m4ri_coin_flip () [*inline*, *static*]

Return 1 or 0 uniformly randomly distributed.

Todo

Allow user to provide her own random() function.

6.5.4.3 void m4ri_die (const char * *errormessage*, ...)

Print error message and abort(). The function accepts additional parameters like printf, so e.g. m4ri_die("foo %d bar %f\n",1,2.0) is valid and will print the string "foo 1 bar 2.0" before dying.

Parameters:

errormessage a string to be printed.

Todo

Allow user to register callback which is called on [m4ri_die\(\)](#).

Warning:

The provided string is not free'd.

Examples:

[testsuite/bench_elimination.c](#).

6.5.4.4 void m4ri_fini (void)

De-initialize global data structures from the M4RI library. On Linux/Solaris this is called automatically when the shared library is unloaded, but it doesn't harm if it is called twice.

6.5.4.5 void m4ri_init (void)

Initialize global data structures for the M4RI library. On Linux/Solaris this is called automatically when the shared library is loaded, but it doesn't harm if it is called twice.

6.5.4.6 static void* m4ri_mm_calloc (int count, int size) [inline, static]

Calloc wrapper.

Parameters:

count Number of elements.

size Size of each element.

Todo

Allow user to register calloc function.

6.5.4.7 static void m4ri_mm_free (void *condemned, ...) [inline, static]

Free wrapper.

Parameters:

condemned Pointer.

Todo

Allow user to register free function.

6.5.4.8 `static void* m4ri_mm_malloc (int size) [inline, static]`

Malloc wrapper.

Parameters:

size Size in bytes.

Todo

Allow user to register malloc function.

6.5.4.9 `static void* m4ri_mmc_calloc (size_t size, size_t count) [inline, static]`

Allocate *size* times *count* zeroed bytes.

Parameters:

size Number of bytes per block.

count Number of blocks.

Warning:

Not thread safe.

6.5.4.10 `static void m4ri_mmc_cleanup (void) [inline, static]`

Cleans up the cache. This function is called automatically when the shared library is loaded.

Warning:

Not thread safe.

6.5.4.11 `static void m4ri_mmc_free (void * condemned, size_t size) [inline, static]`

Free the data pointed to by *condemned* of the given *size*.

Parameters:

condemned Pointer to memory.

size Number of bytes.

Warning:

Not thread safe.

6.5.4.12 static mmb_t* m4ri_mmc_handle (void) [inline, static]

Return handle for locale memory management cache.

Attention:

Not thread safe.

6.5.4.13 static void* m4ri_mmc_malloc (size_t size) [inline, static]

Allocate size bytes.

Parameters:

size Number of bytes.

6.5.4.14 word m4ri_random_word ()

Return uniformly randomly distributed random word.

Todo

Allow user to provide her own random() function.

6.5.4.15 void m4ri_word_to_str (char * destination, word data, int colon)

Write a sting representing the word data to destination.

Parameters:

destination Address of buffer of length at least RADIX*1.3

data Source word

colon Insert a Colon after every 4-th bit.

Warning:

Assumes destination has RADIX*1.3 bytes available

6.5.5 Variable Documentation**6.5.5.1 mmb_t m4ri_mmc_cache[M4RI_MMC_NBLOCKS]**

The actual memory block cache.

6.6 packedmatrix.h File Reference

Dense matrices over GF(2) represented as a bit field. `#include <math.h>`

`#include <assert.h>`

`#include <stdio.h>`

`#include "misc.h"`

Data Structures

- struct `mzd_t`
Dense matrices over GF(2).

Defines

- `#define MZD_MUL_BLOCKSIZE` `MIN(((int)sqrt(((double)(4*CPU_L2_CACHE))))/2,2048)`
Matrix multiplication block-ing dimension.
- `#define mzd_free_window` `mzd_free`
Free a matrix window created with `mzd_init_window`.
- `#define mzd_sub` `mzd_add`
Same as `mzd_add`.
- `#define _mzd_sub` `_mzd_add`
Same as `mzd_sub` but without any checks on the input.

Functions

- `mzd_t * mzd_init` (`const size_t r`, `const size_t c`)
Create a new matrix of dimension $r \times c$.
- `void mzd_free` (`mzd_t *A`)
Free a matrix created with `mzd_init`.
- `mzd_t * mzd_init_window` (`const mzd_t *M`, `const size_t lowr`, `const size_t lowc`, `const size_t highr`, `const size_t highc`)
Create a window/view into the matrix M .
- `static void mzd_row_swap` (`mzd_t *M`, `const size_t rowa`, `const size_t rowb`)
Swap the two rows $rowa$ and $rowb$.
- `void mzd_copy_row` (`mzd_t *B`, `size_t i`, `const mzd_t *A`, `size_t j`)
copy row j from A to row i from B .
- `void mzd_col_swap` (`mzd_t *M`, `const size_t cola`, `const size_t colb`)
Swap the two columns $cola$ and $colb$.

- static void `mzd_col_swap_in_rows` (`mzd_t *M`, `const size_t cola`, `const size_t colb`, `const size_t start_row`, `const size_t stop_row`)
Swap the two columns cola and colb but only between start_row and stop_row.
- static `BIT mzd_read_bit` (`const mzd_t *M`, `const size_t row`, `const size_t col`)
Read the bit at position M[row,col].
- static void `mzd_write_bit` (`mzd_t *M`, `const size_t row`, `const size_t col`, `const BIT value`)
Write the bit value to position M[row,col].
- void `mzd_print` (`const mzd_t *M`)
Print a matrix to stdout.
- void `mzd_print_tight` (`const mzd_t *M`)
Print the matrix to stdout.
- static void `mzd_row_add_offset` (`mzd_t *M`, `size_t dstrow`, `size_t srcrow`, `size_t coloffset`)
Add the rows sourcerow and destrow and stores the total in the row destrow, but only begins at the column coloffset.
- void `mzd_row_add` (`mzd_t *M`, `const size_t sourcerow`, `const size_t destrow`)
Add the rows sourcerow and destrow and stores the total in the row destrow.
- `mzd_t * mzd_transpose` (`mzd_t *DST`, `const mzd_t *A`)
Transpose a matrix.
- `mzd_t * mzd_mul_naive` (`mzd_t *C`, `const mzd_t *A`, `const mzd_t *B`)
Naive cubic matrix multiplication.
- `mzd_t * mzd_addmul_naive` (`mzd_t *C`, `const mzd_t *A`, `const mzd_t *B`)
Naive cubic matrix multiplication and addition.
- `mzd_t * _mzd_mul_naive` (`mzd_t *C`, `const mzd_t *A`, `const mzd_t *B`, `const int clear`)
Naive cubic matrix multiplication with the pre-transposed B.
- `mzd_t * _mzd_mul_va` (`mzd_t *C`, `const mzd_t *v`, `const mzd_t *A`, `const int clear`)
*Matrix multiplication optimized for v*A where v is a vector.*
- void `mzd_randomize` (`mzd_t *M`)
Fill matrix M with uniformly distributed bits.
- void `mzd_set_ui` (`mzd_t *M`, `const unsigned value`)
Set the matrix M to the value equivalent to the integer value provided.
- int `mzd_gauss_delayed` (`mzd_t *M`, `const size_t startcol`, `const int full`)
Gaussian elimination.
- int `mzd_echelonize_naive` (`mzd_t *M`, `const int full`)
Gaussian elimination.

- **BIT** `mzd_equal` (const `mzd_t` *A, const `mzd_t` *B)
Return TRUE if A == B.
- **int** `mzd_cmp` (const `mzd_t` *A, const `mzd_t` *B)
Return -1,0,1 if if A < B, A == B or A > B respectively.
- `mzd_t` * `mzd_copy` (`mzd_t` *DST, const `mzd_t` *A)
Copy matrix A to DST.
- `mzd_t` * `mzd_concat` (`mzd_t` *C, const `mzd_t` *A, const `mzd_t` *B)
Concatenate B to A and write the result to C.
- `mzd_t` * `mzd_stack` (`mzd_t` *C, const `mzd_t` *A, const `mzd_t` *B)
Stack A on top of B and write the result to C.
- `mzd_t` * `mzd_submatrix` (`mzd_t` *S, const `mzd_t` *M, const `size_t` lowr, const `size_t` lowc, const `size_t` highr, const `size_t` highc)
Copy a submatrix.
- `mzd_t` * `mzd_invert_naive` (`mzd_t` *INV, `mzd_t` *A, const `mzd_t` *I)
Invert the matrix target using Gaussian elimination.
- `mzd_t` * `mzd_add` (`mzd_t` *C, const `mzd_t` *A, const `mzd_t` *B)
Set C = A+B.
- `mzd_t` * `_mzd_add` (`mzd_t` *C, const `mzd_t` *A, const `mzd_t` *B)
Same as mzd_add but without any checks on the input.
- **void** `mzd_combine` (`mzd_t` *DST, const `size_t` row3, const `size_t` startblock3, const `mzd_t` *SC1, const `size_t` row1, const `size_t` startblock1, const `mzd_t` *SC2, const `size_t` row2, const `size_t` startblock2)
 $row3[col3:] = row1[col1:] + row2[col2:]$
- **static** `word` `mzd_read_bits` (const `mzd_t` *M, const `size_t` x, const `size_t` y, const `int` n)
- **static** **void** `mzd_xor_bits` (const `mzd_t` *M, const `size_t` x, const `size_t` y, const `int` n, `word` values)
XOR n bits from values to M starting a position (x,y).
- **static** **void** `mzd_and_bits` (const `mzd_t` *M, const `size_t` x, const `size_t` y, const `int` n, `word` values)
AND n bits from values to M starting a position (x,y).
- **static** **void** `mzd_clear_bits` (const `mzd_t` *M, const `size_t` x, const `size_t` y, const `int` n)
Clear n bits in M starting a position (x,y).
- **int** `mzd_is_zero` (`mzd_t` *A)
Zero test for matrix.
- **void** `mzd_row_clear_offset` (`mzd_t` *M, const `size_t` row, const `size_t` coloffset)
Clear the given row, but only begins at the column coloffset.

- int `mzd_find_pivot` (`mzd_t *M`, `size_t start_row`, `size_t start_col`, `size_t *r`, `size_t *c`)
Find the next nonzero entry in M starting at start_row and start_col.
- double `mzd_density` (`mzd_t *A`, `int res`)
*Return the number of nonzero entries divided by nrows * ncols.*
- `size_t mzd_first_zero_row` (`mzd_t *A`)
Return the first row with all zero entries.

6.6.1 Detailed Description

Dense matrices over GF(2) represented as a bit field.

Author:

Gregory Bard <bard@fordham.edu>
Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

6.6.2 Define Documentation

6.6.2.1 #define _mzd_sub _mzd_add

Same as `mzd_sub` but without any checks on the input.

Parameters:

C Preallocated difference matrix, may be NULL for automatic creation.
A Matrix
B Matrix

Ignores offset attribute of `packedmatrix`.

6.6.2.2 #define mzd_free_window mzd_free

Free a matrix window created with `mzd_init_window`.

Parameters:

A Matrix

6.6.2.3 #define MZD_MUL_BLOCKSIZE MIN(((int)sqrt(((double)(4*CPU_L2_CACHE)))/2,2048)

Matrix multiplication block-ing dimension. Defines the number of rows of the matrix *A* that are processed as one block during the execution of a multiplication algorithm.

6.6.2.4 #define mzd_sub mzd_add

Same as mzd_add.

Parameters:

C Preallocated difference matrix, may be NULL for automatic creation.

A Matrix

B Matrix

Ignores offset attribute of packedmatrix.

6.6.3 Function Documentation

6.6.3.1 mzd_t* _mzd_add (mzd_t * C, const mzd_t * A, const mzd_t * B)

Same as mzd_add but without any checks on the input.

Parameters:

C Preallocated sum matrix, may be NULL for automatic creation.

A Matrix

B Matrix

Ignores offset attribute of packedmatrix.

6.6.3.2 mzd_t* _mzd_mul_naive (mzd_t * C, const mzd_t * A, const mzd_t * B, const int clear)

Naive cubic matrix multiplication with the pre-transposed B. That is, compute C such that $C == AB^t$.

Parameters:

C Preallocated product matrix.

A Input matrix A.

B Pre-transposed input matrix B.

clear Whether to clear C before accumulating AB

6.6.3.3 mzd_t* _mzd_mul_va (mzd_t * C, const mzd_t * v, const mzd_t * A, const int clear)

Matrix multiplication optimized for $v*A$ where v is a vector.

Parameters:

- C* Preallocated product matrix.
- v* Input matrix v.
- A* Input matrix A.
- clear* If set clear C first, otherwise add result to C.

6.6.3.4 mzd_t* mzd_add (mzd_t * C, const mzd_t * A, const mzd_t * B)

Set $C = A+B$. C is also returned. If C is NULL then a new matrix is created which must be freed by `mzd_free`.

Parameters:

- C* Preallocated sum matrix, may be NULL for automatic creation.
- A* Matrix
- B* Matrix

Examples:

[testsuite/test_multiplication.c](#).

6.6.3.5 mzd_t* mzd_addmul_naive (mzd_t * C, const mzd_t * A, const mzd_t * B)

Naive cubic matrix multiplication and addition. That is, compute C such that $C == C + AB$.

Parameters:

- C* Preallocated product matrix.
- A* Input matrix A.
- B* Input matrix B.

Note:

Normally, if you will multiply several times by b, it is smarter to calculate b^T yourself, and keep it, and then use the function called `_mzd_mul_naive`

6.6.3.6 static void mzd_and_bits (const mzd_t * M, const size_t x, const size_t y, const int n, word values) [inline, static]

AND n bits from values to M starting a position (x,y).

Parameters:

- M* Source matrix.

x Starting row.
y Starting column.
n Number of bits (\leq RADIX);
values Word with values;

6.6.3.7 `static void mzd_clear_bits (const mzd_t * M, const size_t x, const size_t y, const int n)`
`[inline, static]`

Clear *n* bits in *M* starting a position (*x*,*y*).

Parameters:

M Source matrix.
x Starting row.
y Starting column.
n Number of bits (\leq RADIX);

6.6.3.8 `int mzd_cmp (const mzd_t * A, const mzd_t * B)`

Return -1,0,1 if if $A < B$, $A == B$ or $A > B$ respectively.

Parameters:

A Matrix.
B Matrix.

Note:

This comparison is not well defined mathematically and relatively arbitrary since elements of GF(2) don't have an ordering.

Ignores offset attribute of packedmatrix.

6.6.3.9 `void mzd_col_swap (mzd_t * M, const size_t cola, const size_t colb)`

Swap the two columns *cola* and *colb*.

Parameters:

M Matrix.
cola Column index.
colb Column index.

6.6.3.10 `static void mzd_col_swap_in_rows (mzd_t * M, const size_t cola, const size_t colb, const size_t start_row, const size_t stop_row) [inline, static]`

Swap the two columns cola and colb but only between start_row and stop_row.

Parameters:

M Matrix.
cola Column index.
colb Column index.
start_row Row index.
stop_row Row index (exclusive).

6.6.3.11 `void mzd_combine (mzd_t * DST, const size_t row3, const size_t startblock3, const mzd_t * SC1, const size_t row1, const size_t startblock1, const mzd_t * SC2, const size_t row2, const size_t startblock2)`

row3[col3:] = row1[col1:] + row2[col2:] Adds row1 of SC1, starting with startblock1 to the end, to row2 of SC2, starting with startblock2 to the end. This gets stored in DST, in row3, starting with startblock3.

Parameters:

DST destination matrix
row3 destination row for matrix dst
startblock3 starting block to work on in matrix dst
SC1 source matrix
row1 source row for matrix sc1
startblock1 starting block to work on in matrix sc1
SC2 source matrix
startblock2 starting block to work on in matrix sc2
row2 source row for matrix sc2

Ignores offset attribute of packedmatrix.

Todo

this `code` is slow if offset!=0

6.6.3.12 `mzd_t* mzd_concat (mzd_t * C, const mzd_t * A, const mzd_t * B)`

Concatenate B to A and write the result to C. That is,

$$[A], [B] \rightarrow [A \ B] = C$$

The inputs are not modified but a new matrix is created.

Parameters:

C Matrix, may be NULL for automatic creation

A Matrix

B Matrix

Note:

This is sometimes called augment.

Ignores offset attribute of packedmatrix.

6.6.3.13 mzd_t* mzd_copy (mzd_t * DST, const mzd_t * A)

Copy matrix A to DST.

Parameters:

DST May be NULL for automatic creation.

A Source matrix.

Ignores offset attribute of packedmatrix.

Examples:

[testsuite/test_elimination.c](#), [testsuite/test_lqup.c](#), and [testsuite/test_multiplication.c](#).

6.6.3.14 void mzd_copy_row (mzd_t * B, size_t i, const mzd_t * A, size_t j)

copy row *j* from A to row *i* from B. The offsets of A and B must match and the number of columns of A must be less than or equal to the number of columns of B.

Parameters:

B Target matrix.

i Target row index.

A Source matrix.

j Source row index.

6.6.3.15 double mzd_density (mzd_t * *A*, int *res*)

Return the number of nonzero entries divided by `nrows * ncols`. If `res = 0` then 100 samples per row are made, if `res > 0` the function takes `res` sized steps within each row (`res = 1` uses every word).

Parameters:

A Matrix

res Resolution of sampling

6.6.3.16 int mzd_echelonize_naive (mzd_t * *M*, const int *full*)

Gaussian elimination. This will do Gaussian elimination on the matrix `m`. If `full=FALSE`, then it will do triangular style elimination, and if `full=TRUE`, it will do Gauss-Jordan style, or full elimination.

Parameters:

M Matrix

full Gauss-Jordan style or upper triangular form only.

Ignores offset attribute of packedmatrix.

See also:

[mzd_echelonize_m4ri\(\)](#), [mzd_echelonize_pluq\(\)](#)

Examples:

[testsuite/bench_elimination.c](#), and [testsuite/test_elimination.c](#).

6.6.3.17 BIT mzd_equal (const mzd_t * *A*, const mzd_t * *B*)

Return TRUE if `A == B`.

Parameters:

A Matrix

B Matrix

Ignores offset attribute of packedmatrix.

Examples:

[testsuite/test_elimination.c](#), and [testsuite/test_multiplication.c](#).

6.6.3.18 int mzd_find_pivot (mzd_t * *M*, size_t *start_row*, size_t *start_col*, size_t * *r*, size_t * *c*)

Find the next nonzero entry in *M* starting at *start_row* and *start_col*. This function walks down rows in the inner loop and columns in the outer loop. If a nonzero entry is found this function returns 1 and zero otherwise.

If and only if a nonzero entry is found *r* and *c* are updated.

Parameters:

- M* Matrix
- start_row* Index of row where to start search
- start_col* Index of column where to start search
- r* Row index updated if pivot is found
- c* Column index updated if pivot is found

6.6.3.19 size_t mzd_first_zero_row (mzd_t * *A*)

Return the first row with all zero entries. If no such row can be found returns *nrows*.

Parameters:

- A* Matrix

6.6.3.20 void mzd_free (mzd_t * *A*)

Free a matrix created with *mzd_init*.

Parameters:

- A* Matrix

Examples:

[testsuite/bench_elimination.c](#), [testsuite/test_elimination.c](#), [testsuite/test_lqup.c](#), and [testsuite/test_multiplication.c](#).

6.6.3.21 int mzd_gauss_delayed (mzd_t * *M*, const size_t *startcol*, const int *full*)

Gaussian elimination. This will do Gaussian elimination on the matrix *m* but will start not at column 0 necc but at column *startcol*. If *full*=FALSE, then it will do triangular style elimination, and if *full*=TRUE, it will do Gauss-Jordan style, or full elimination.

Parameters:

- M* Matrix

startcol First column to consider for reduction.

full Gauss-Jordan style or upper triangular form only.

Ignores offset attribute of packedmatrix.

6.6.3.22 `mzd_t* mzd_init (const size_t r, const size_t c)`

Create a new matrix of dimension $r \times c$. Use `mzd_free` to kill it.

Parameters:

r Number of rows

c Number of columns

Examples:

[testsuite/bench_elimination.c](#), [testsuite/test_elimination.c](#), [testsuite/test_lqpc.c](#), and [testsuite/test_multiplication.c](#).

6.6.3.23 `mzd_t* mzd_init_window (const mzd_t * M, const size_t lowr, const size_t lowc, const size_t highr, const size_t highc)`

Create a window/view into the matrix M . A matrix window for M is a meta structure on the matrix M . It is setup to point into the matrix so M *must not* be freed while the matrix window is used.

This function puts the restriction on the provided parameters that all parameters must be within range for M which is not enforced currently .

Use `mzd_free_window` to free the window.

Parameters:

M Matrix

lowr Starting row (inclusive)

lowc Starting column (inclusive)

highr End row (exclusive)

highc End column (exclusive)

6.6.3.24 `mzd_t* mzd_invert_naive (mzd_t * INV, mzd_t * A, const mzd_t * I)`

Invert the matrix target using Gaussian elimination. To avoid recomputing the identity matrix over and over again, I may be passed in as identity parameter.

Parameters:

INV Preallocated space for inversion matrix, may be NULL for automatic creation.

A Matrix to be reduced.

I Identity matrix.

Ignores offset attribute of packedmatrix.

6.6.3.25 int mzd_is_zero (mzd_t * A)

Zero test for matrix.

Parameters:

A Input matrix.

6.6.3.26 mzd_t* mzd_mul_naive (mzd_t * C, const mzd_t * A, const mzd_t * B)

Naive cubic matrix multiplication. That is, compute C such that $C == AB$.

Parameters:

C Preallocated product matrix, may be NULL for automatic creation.

A Input matrix A.

B Input matrix B.

Note:

Normally, if you will multiply several times by b, it is smarter to calculate bT yourself, and keep it, and then use the function called `_mzd_mul_naive`

Examples:

[testsuite/test_multiplication.c](#).

6.6.3.27 void mzd_print (const mzd_t * M)

Print a matrix to stdout. The output will contain colons between every 4-th column.

Parameters:

M Matrix

6.6.3.28 void mzd_print_tight (const mzd_t * *M*)

Print the matrix to stdout.

Parameters:

M Matrix

6.6.3.29 void mzd_randomize (mzd_t * *M*)

Fill matrix *M* with uniformly distributed bits.

Parameters:

M Matrix

Todo

Allow the user to provide a RNG callback.

Ignores offset attribute of packedmatrix.

Examples:

[testsuite/bench_elimination.c](#), [testsuite/test_elimination.c](#), [testsuite/test_lqup.c](#), and [testsuite/test_multiplication.c](#).

**6.6.3.30 static BIT mzd_read_bit (const mzd_t * *M*, const size_t *row*, const size_t *col*)
[inline, static]**

Read the bit at position *M*[*row*,*col*].

Parameters:

M Matrix

row Row index

col Column index

Note:

No bounds checks whatsoever are performed.

Examples:

[testsuite/test_lqup.c](#).

6.6.3.31 static word mzd_read_bits (const mzd_t * *M*, const size_t *x*, const size_t *y*, const int *n*)
[inline, static]

Get *n* bits starting a position (*x*,*y*) from the matrix *M*.

Parameters:

M Source matrix.
x Starting row.
y Starting column.
n Number of bits (<= RADIX);

6.6.3.32 void mzd_row_add (mzd_t * *M*, const size_t *sourcerow*, const size_t *destrow*)

Add the rows *sourcerow* and *destrow* and stores the total in the row *destrow*.

Parameters:

M Matrix
sourcerow Index of source row
destrow Index of target row

Note:

this can be done much faster with `mzd_combine`.

6.6.3.33 static void mzd_row_add_offset (mzd_t * *M*, size_t *dstrow*, size_t *srcrow*, size_t *coloffset*)
[inline, static]

Add the rows *sourcerow* and *destrow* and stores the total in the row *destrow*, but only begins at the column *coloffset*.

Parameters:

M Matrix
dstrow Index of target row
srcrow Index of source row
coloffset Column offset

6.6.3.34 void mzd_row_clear_offset (mzd_t * *M*, const size_t *row*, const size_t *coloffset*)

Clear the given row, but only begins at the column *coloffset*.

Parameters:

M Matrix
row Index of row
coloffset Column offset

6.6.3.35 `static void mzd_row_swap (mzd_t * M, const size_t rowa, const size_t rowb)` [`inline`, `static`]

Swap the two rows *rowa* and *rowb*.

Parameters:

M Matrix

rowa Row index.

rowb Row index.

6.6.3.36 `void mzd_set_ui (mzd_t * M, const unsigned value)`

Set the matrix *M* to the value equivalent to the integer value provided. Specifically, this function does nothing if *value* == 0 and returns the identity matrix if *value* == 1.

If the matrix is not square then the largest possible square submatrix is set to the identity matrix.

Parameters:

M Matrix

value Either 0 or 1

6.6.3.37 `mzd_t* mzd_stack (mzd_t * C, const mzd_t * A, const mzd_t * B)`

Stack *A* on top of *B* and write the result to *C*. That is,

$$\begin{bmatrix} A \\ B \end{bmatrix}, \begin{bmatrix} B \end{bmatrix} \rightarrow \begin{bmatrix} A \\ B \end{bmatrix} = C$$

The inputs are not modified but a new matrix is created.

Parameters:

C Matrix, may be NULL for automatic creation

A Matrix

B Matrix

Ignores offset attribute of packedmatrix.

6.6.3.38 `mzd_t* mzd_submatrix (mzd_t * S, const mzd_t * M, const size_t lowr, const size_t lowc, const size_t highr, const size_t highc)`

Copy a submatrix. Note that the upper bounds are not included.

Parameters:

S Preallocated space for submatrix, may be NULL for automatic creation.

M Matrix

lowr start rows

lowc start column

highr stop row (this row is *not* included)

highc stop column (this column is *not* included)

6.6.3.39 `mzd_t* mzd_transpose (mzd_t * DST, const mzd_t * A)`

Transpose a matrix. This function uses the fact that:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^T = \begin{bmatrix} A^T & C^T \\ B^T & D^T \end{bmatrix}$$

and thus rearranges the blocks recursively.

Parameters:

DST Preallocated return matrix, may be NULL for automatic creation.

A Matrix

6.6.3.40 `static void mzd_write_bit (mzd_t * M, const size_t row, const size_t col, const BIT value)`
`[inline, static]`

Write the bit value to position M[row,col].

Parameters:

M Matrix

row Row index

col Column index

value Either 0 or 1

Note:

No bounds checks whatsoever are performed.

Examples:

[testsuite/bench_elimination.c](#), and [testsuite/test_lqup.c](#).

6.6.3.41 `static void mzd_xor_bits (const mzd_t * M, const size_t x, const size_t y, const int n, word values) [inline, static]`

XOR n bits from values to M starting a position (x,y) .

Parameters:

- M* Source matrix.
- x* Starting row.
- y* Starting column.
- n* Number of bits (\leq RADIX);
- values* Word with values;

6.7 parity.h File Reference

Compute the parity of 64 words in parallel.

Defines

- #define `MIX32(a, b)`
Step for mixing two 64-bit words to compute their parity.
- #define `MIX16(a, b)`
Step for mixing two 64-bit words to compute their parity.
- #define `MIX8(a, b)`
Step for mixing two 64-bit words to compute their parity.
- #define `MIX4(a, b)`
Step for mixing two 64-bit words to compute their parity.
- #define `MIX2(a, b)`
Step for mixing two 64-bit words to compute their parity.
- #define `MIX1(a, b)`
Step for mixing two 64-bit words to compute their parity.

Functions

- static `word _parity64_helper (word *buf)`
See parity64.
- static `word parity64 (word *buf)`
Computes parity of each of `buf[0]`, `buf[1]`, ..., `buf[63]`. Returns single word whose bits are the parities of `buf[0]`, ..., `buf[63]`. Assumes 64-bit machine unsigned long.

6.7.1 Detailed Description

Compute the parity of 64 words in parallel.

Author:

David Harvey

6.7.2 Define Documentation

6.7.2.1 #define MIX1(a, b)

Value:

```
(((a) >> 1) ^ (a)) & 0x5555555555555551) + \
  (((b) << 1) ^ (b)) & 0xAAAAAAAAAAAAAAAA11);
```

Step for mixing two 64-bit words to compute their parity.

6.7.2.2 #define MIX16(a, b)

Value:

```
(((a) >> 16) ^ (a)) & 0x0000FFFF0000FFFF1) + \
  (((b) << 16) ^ (b)) & 0xFFFF0000FFFF000011);
```

Step for mixing two 64-bit words to compute their parity.

6.7.2.3 #define MIX2(a, b)

Value:

```
(((a) >> 2) ^ (a)) & 0x3333333333333331) + \
  (((b) << 2) ^ (b)) & 0xCCCCCCCCCCCCCC11);
```

Step for mixing two 64-bit words to compute their parity.

6.7.2.4 #define MIX32(a, b)

Value:

```
(((a) << 32) ^ (a)) >> 32) + \
  (((b) >> 32) ^ (b)) << 32)
```

Step for mixing two 64-bit words to compute their parity.

6.7.2.5 #define MIX4(a, b)

Value:

```
(((a) >> 4) ^ (a)) & 0x0F0F0F0F0F0F0F1) + \
  (((b) << 4) ^ (b)) & 0xF0F0F0F0F0F0F11);
```

Step for mixing two 64-bit words to compute their parity.

6.7.2.6 #define MIX8(a, b)

Value:

```

((((a) >> 8) ^ (a)) & 0x00FF00FF00FF00FF11) + \
      (((b) << 8) ^ (b)) & 0xFF00FF00FF00FF011);

```

Step for mixing two 64-bit words to compute their parity.

6.7.3 Function Documentation

6.7.3.1 static word parity64 (word * buf) [inline, static]

Computes parity of each of buf[0], buf[1], ..., buf[63]. Returns single word whose bits are the parities of buf[0], ..., buf[63]. Assumes 64-bit machine unsigned long.

Parameters:

buf buffer of words of length 64

6.8 permutation.h File Reference

Permutation matrices. #include "misc.h"

#include "packedmatrix.h"

Data Structures

- struct [mzp_t](#)
Permutations.

Functions

- [mzp_t * mzp_init](#) (size_t length)
- void [mzp_free](#) (mzp_t *P)
- [mzp_t * mzp_init_window](#) (mzp_t *P, size_t begin, size_t end)
Create a window/view into the permutation P.
- void [mzp_free_window](#) (mzp_t *condemned)
Free a permutation window created with mzp_init_mzp_t_window().
- void [mzp_set_ui](#) (mzp_t *P, unsigned int value)
Set the permutation P to the identity permutation. The only allowed value is 1.
- void [mzd_apply_p_left](#) (mzd_t *A, mzp_t *P)
- void [mzd_apply_p_left_trans](#) (mzd_t *A, mzp_t *P)
- void [mzd_apply_p_right](#) (mzd_t *A, mzp_t *P)
- void [mzd_apply_p_right_trans](#) (mzd_t *A, mzp_t *P)
- void [mzd_apply_p_right_even_capped](#) (mzd_t *A, mzp_t *P, size_t start_row, size_t start_col)

- void `mzd_apply_p_right_trans_even_capped` (`mzd_t *A`, `mzp_t *P`, `size_t start_row`, `size_t start_col`)
- void `mzd_apply_p_right_tri` (`mzd_t *A`, `mzp_t *Q`)
- void `mzp_print` (`mzp_t *P`)

6.8.1 Detailed Description

Permutation matrices.

Author:

Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

6.8.2 Function Documentation

6.8.2.1 void `mzd_apply_p_left` (`mzd_t *A`, `mzp_t *P`)

Apply the permutation P to A from the left.

This is equivalent to row swaps walking from 0 to length-1.

Parameters:

A Matrix.

P Permutation.

Examples:

[testsuite/test_lqup.c](#).

6.8.2.2 void `mzd_apply_p_left_trans` (`mzd_t *A`, `mzp_t *P`)

Apply the permutation P to A from the left but transpose P before.

This is equivalent to row swaps walking from length-1 to 0.

Parameters:

A Matrix.

P Permutation.

6.8.2.3 void `mzd_apply_p_right` (`mzd_t *A`, `mzp_t *P`)

Apply the permutation P to A from the right.

This is equivalent to column swaps walking from length-1 to 0.

Parameters:

A Matrix.

P Permutation.

6.8.2.4 void mzd_apply_p_right_even_capped (mzd_t * A, mzp_t * P, size_t start_row, size_t start_col)

Apply the permutation P to A from the right starting at start_row.

This is equivalent to column swaps walking from length-1 to 0.

Parameters:

A Matrix.

P Permutation.

start_row Start swapping at this row.

start_col Start swapping at this column.

Ignores offset attribute of packedmatrix.

6.8.2.5 void mzd_apply_p_right_trans (mzd_t * A, mzp_t * P)

Apply the permutation P to A from the right but transpose P before.

This is equivalent to column swaps walking from 0 to length-1.

Parameters:

A Matrix.

P Permutation.

Apply the [mzp_t](#) P to A from the right but transpose P before.

This is equivalent to column swaps walking from 0 to length-1.

Parameters:

A Matrix.

P Permutation.

Examples:

[testsuite/test_lqup.c](#).

6.8.2.6 void mzd_apply_p_right_trans_even_capped (mzd_t * A, mzp_t * P, size_t start_row, size_t start_col)

Apply the permutation P^T to A from the right starting at start_row.

This is equivalent to column swaps walking from 0 to length-1.

Parameters:

A Matrix.

P Permutation.

start_row Start swapping at this row.

start_col Start swapping at this column.

Ignores offset attribute of packedmatrix.

6.8.2.7 void mzd_apply_p_right_tri (mzd_t * A, mzp_t * Q)

Apply the permutation P to A from the right, but only on the lower triangular part of the matrix A. This is equivalent to column swaps walking from length-1 to 0.

Parameters:

A Matrix.

Q Permutation.

6.8.2.8 void mzp_free (mzp_t * P)

Free a [mzp_t](#).

Parameters:

P Permutation to free.

Examples:

[testsuite/test_lqup.c](#).

6.8.2.9 void mzp_free_window (mzp_t * condemned)

Free a permutation window created with `mzp_init_mzp_t_window()`.

Parameters:

condemned Permutation Matrix

6.8.2.10 mzp_t* mzp_init (size_t length)

Construct an identity permutation.

Parameters:

length Length of the permutation.

Examples:

[testsuite/test_lqup.c](#).

6.8.2.11 `mzp_t* mzp_init_window (mzp_t * P, size_t begin, size_t end)`

Create a window/view into the permutation P. Use `mzp_free_mzp_t_window()` to free the window.

Parameters:

- P* Permutaiton matrix
- begin* Starting index (inclusive)
- end* Ending index (exclusive)

6.8.2.12 `void mzp_print (mzp_t * P)`

Print the `mzp_t` P

Parameters:

- P* Permutation.

6.8.2.13 `void mzp_set_ui (mzp_t * P, unsigned int value)`

Set the permutation P to the identity permutation. The only allowed value is 1.

Parameters:

- P* Permutation
- value* 1

Note:

This interface was chosen to be similar to [mzd_set_ui\(\)](#).

6.9 `pluq_mmpf.h` File Reference

```
LQUP factorization using Gray codes. #include "packedmatrix.h"
#include "permutation.h"
```

Functions

- `size_t _mzd_lqup_mmpf (mzd_t *A, mzp_t *P, mzp_t *Q, int k)`
LQUP matrix decomposition of A using Gray codes.
- `size_t _mzd_pluq_mmpf (mzd_t *A, mzp_t *P, mzp_t *Q, int k)`
PLUQ matrix decomposition of A using Gray codes.
- `size_t _mzd_lqup_submatrix (mzd_t *A, size_t start_row, size_t stop_row, size_t start_col, int k, mzp_t *P, mzp_t *Q, size_t *done, size_t *done_row)`
LQUP matrix decomposition of a submatrix for up to k columns starting at (r,c).

6.9.1 Detailed Description

LQUP factorization using Gray codes.

Author:

Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

6.9.2 Function Documentation

6.9.2.1 `size_t _mzd_lqup_mmpf (mzd_t * A, mzp_t * P, mzp_t * Q, int k)`

LQUP matrix decomposition of A using Gray codes. If (L,Q,U,P) satisfy $LQUP = A^T$, this function returns (L,Q^T,U,P). The matrix L and U are stored in place over A.

Parameters:

- A* Matrix.
- P* Preallocated row permutation.
- Q* Preallocated column permutation.
- k* Size of Gray code tables.

Ignores offset attribute of packedmatrix.

Returns:

Rank of A.

6.9.2.2 `size_t _mzd_lqup_submatrix (mzd_t * A, size_t start_row, size_t stop_row, size_t start_col, int k, mzp_t * P, mzp_t * Q, size_t * done, size_t * done_row)`

LQUP matrix decomposition of a submatrix for up to k columns starting at (r,c). Updates P and Q and modifies A in place. The buffer done afterwards holds how far a particular row was already added.

Parameters:

- A* Matrix.
- start_row* Row Offset.
- stop_row* Up to which row the matrix should be processed (exclusive).
- start_col* Column Offset.
- k* Size of Gray code tables.
- P* Preallocated row permutation.
- Q* Preallocated column permutation.
- done* Preallocated temporary buffer.
- done_row* Stores the last row which is already reduced processed after function execution.

Return values:

kbar Maximum k for which a pivot could be found.

6.9.2.3 size_t _mzd_pluq_mmpf (mzd_t *A, mzp_t *P, mzp_t *Q, int k)

PLUQ matrix decomposition of A using Gray codes. If (P,L,U,Q) satisfy $PLUQ = A$, it returns (P, L, U, Q^T).

Parameters:

- A* Matrix.
- P* Preallocated row permutation.
- Q* Preallocated column permutation.
- k* Size of Gray code tables.

Ignores offset attribute of packedmatrix.

Returns:

- Rank of A.

6.10 solve.h File Reference

System solving with matrix routines. #include <stdio.h>

```
#include "misc.h"
```

```
#include "permutation.h"
```

```
#include "packedmatrix.h"
```

Functions

- void `mzd_solve_left` (mzd_t *A, mzd_t *B, const int cutoff, const int inconsistency_check)
Solves $A X = B$ with A and B matrices.
- void `mzd_pluq_solve_left` (mzd_t *A, size_t rank, mzp_t *P, mzp_t *Q, mzd_t *B, const int cutoff, const int inconsistency_check)
Solves $(P L U Q) X = B$.
- void `_mzd_pluq_solve_left` (mzd_t *A, size_t rank, mzp_t *P, mzp_t *Q, mzd_t *B, const int cutoff, const int inconsistency_check)
Solves $(P L U Q) X = B$.
- void `_mzd_solve_left` (mzd_t *A, mzd_t *B, const int cutoff, const int inconsistency_check)
Solves $A X = B$ with A and B matrices.
- `mzd_t * mzd_kernel_left_pluq` (mzd_t *A, const int cutoff)
Solve X for $A X = 0$.

6.10.1 Detailed Description

System solving with matrix routines.

Author:

Jean-Guillaume Dumas <Jean-Guillaume.Dumas@imag.fr>

Attention:

This file is currently broken.

6.10.2 Function Documentation

6.10.2.1 void _mzd_pluq_solve_left (mzd_t * A, size_t rank, mzp_t * P, mzp_t * Q, mzd_t * B, const int cutoff, const int inconsistency_check)

Solves (P L U Q) X = B. A is an input matrix supposed to store both:

- an upper right triangular matrix U
- a lower left unitary triangular matrix L.

The solution X is stored inplace on B.

This version assumes that the matrices are at an even position on the RADIX grid and that their dimension is a multiple of RADIX.

Parameters:

A Input upper/lower triangular matrices.

rank is rank of A.

P Input row permutation matrix.

Q Input column permutation matrix.

B Input matrix, being overwritten by the solution matrix X.

cutoff Minimal dimension for Strassen recursion (default: 0).

inconsistency_check decide whether or not to check for inconsistency (faster without but output not defined if system is not consistent).

A is supposed to store L lower triangular and U upper triangular B is modified in place (Bi's in the comments are just modified versions of B) PLUQ = A 1) P B2 = B1 2) L B3 = B2 3) U B4 = B3 4) Q B5 = B4

FASTER without this check

update with the lower part of L

Default is to set the undefined bits to zero if inconsistency has been checked then Y2 bits are already all zeroes thus this clearing is not needed

6.10.2.2 void `_mzd_solve_left` (`mzd_t * A`, `mzd_t * B`, `const int cutoff`, `const int inconsistency_check`)

Solves $A X = B$ with A and B matrices. The solution X is stored inplace on B .

This version assumes that the matrices are at an even position on the RADIX grid and that their dimension is a multiple of RADIX.

Parameters:

A Input matrix.

B Input matrix, being overwritten by the solution matrix X .

cutoff Minimal dimension for Strassen recursion (default: 0).

inconsistency_check decide whether or not to check for inconsistency (faster without but output not defined if system is not consistent).

B is modified in place (B_i 's in the comments are just modified versions of B) 1) $PLUQ = A$ 2) $P B_2 = B_1$ 3) $L B_3 = B_2$ 4) $U B_4 = B_3$ 5) $Q B_5 = B_4$

6.10.2.3 `mzd_t* mzd_kernel_left_pluq` (`mzd_t * A`, `const int cutoff`)

Solve X for $A X = 0$. If r is the rank of the $n_r \times n_c$ matrix A , return the $n_c \times (n_c - r)$ matrix X such that $A * X == 0$ and that the columns of X are linearly independent.

Parameters:

A Matrix.

cutoff Minimal dimension for Strassen recursion (default: 0).

Ignores offset attribute of packedmatrix.

See also:

[mzd_pluq\(\)](#)

Returns:

X

6.10.2.4 void `mzd_pluq_solve_left` (`mzd_t * A`, `size_t rank`, `mzp_t * P`, `mzp_t * Q`, `mzd_t * B`, `const int cutoff`, `const int inconsistency_check`)

Solves $(P L U Q) X = B$. A is an input matrix supposed to store both:

- an upper right triangular matrix U
- a lower left unitary triangular matrix L .

The solution X is stored inplace on B

This version assumes that the matrices are at an even position on the RADIX grid and that their dimension is a multiple of RADIX.

Parameters:

A Input upper/lower triangular matrices.

rank is rank of A .

P Input row permutation matrix.

Q Input column permutation matrix.

B Input matrix, being overwritten by the solution matrix X .

cutoff Minimal dimension for Strassen recursion (default: 0).

inconsistency_check decide whether or not to check for inconsistency (faster without but output not defined if system is not consistent).

6.10.2.5 void mzd_solve_left (mzd_t * A , mzd_t * B , const int *cutoff*, const int *inconsistency_check*)

Solves $A X = B$ with A and B matrices. The solution X is stored inplace on B .

Parameters:

A Input matrix (overwritten).

B Input matrix, being overwritten by the solution matrix X

cutoff Minimal dimension for Strassen recursion (default: 0).

inconsistency_check decide whether or not to check for inconsistency (faster without but output not defined if system is not consistent).

6.11 strassen.h File Reference

Matrix operations using Strassen's formulas including Winograd's improvements. `#include <math.h>`

```
#include "misc.h"
```

```
#include "packedmatrix.h"
```

```
#include "brilliantussian.h"
```

Defines

- `#define STRASSEN_MUL_CUTOFF MIN(((int)sqrt((double)(4*CPU_L2_CACHE))),4096)`

Functions

- `mzd_t * mzd_mul (mzd_t * C , mzd_t * A , mzd_t * B , int cutoff)`

Matrix multiplication via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = AB$.

- `mzd_t * mzd_addmul (mzd_t *C, mzd_t *A, mzd_t *B, int cutoff)`
Matrix multiplication and in-place addition via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = C + AB$.
- `mzd_t * _mzd_mul_even (mzd_t *C, mzd_t *A, mzd_t *B, int cutoff)`
Matrix multiplication via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = AB$.
- `mzd_t * _mzd_addmul_even (mzd_t *C, mzd_t *A, mzd_t *B, int cutoff)`
Matrix multiplication and in-place addition via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = C + AB$.
- `mzd_t * _mzd_addmul (mzd_t *C, mzd_t *A, mzd_t *B, int cutoff)`
Matrix multiplication and in-place addition via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = C + AB$.
- `mzd_t * _mzd_addmul_weird_weird (mzd_t *C, mzd_t *A, mzd_t *B, int cutoff)`
- `mzd_t * _mzd_addmul_weird_even (mzd_t *C, mzd_t *A, mzd_t *B, int cutoff)`
- `mzd_t * _mzd_addmul_even_weird (mzd_t *C, mzd_t *A, mzd_t *B, int cutoff)`

6.11.1 Detailed Description

Matrix operations using Strassen's formulas including Winograd's improvements.

Author:

Gregory Bard <bard@fordham.edu>
 Martin Albrecht <M.R.Albrecht@rhul.ac.uk>

6.11.2 Define Documentation

6.11.2.1 `#define STRASSEN_MUL_CUTOFF MIN(((int)sqrt((double)(4*CPU_L2_CACHE))),4096)`

The default cutoff for Strassen-Winograd multiplication. It should hold hold that $2 * (n^2)/8$ fits into the L2 cache.

6.11.3 Function Documentation

6.11.3.1 `mzd_t* _mzd_addmul (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)`

Matrix multiplication and in-place addition via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = C + AB$. The matrices A and B are respectively $m \times k$ and $k \times n$, and can be not aligned on the RADIX grid.

Parameters:

- C** Preallocated product matrix, may be NULL for automatic creation.
- A** Input matrix A
- B** Input matrix B

cutoff Minimal dimension for Strassen recursion.

Assumes that B and C are aligned in the same manner (as in a Schur complement)

6.11.3.2 `mzd_t* _mzd_addmul_even (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)`

Matrix multiplication and in-place addition via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = C + AB$. This is the actual implementation. Any matrix where either the number of rows or the number of columns is smaller than *cutoff* is processed using the M4RM algorithm.

Parameters:

C Preallocated product matrix, may be NULL for automatic creation.

A Input matrix A

B Input matrix B

cutoff Minimal dimension for Strassen recursion.

Note:

This implementation is heavily inspired by the function `strassen_window_multiply_c` in Sage 3.0; For reference see <http://www.sagemath.org>

Todo

make sure not to overwrite `crap` after `ncols` and before `width*RADIX`

Note:

See Marco Bodrato; "A Strassen-like Matrix Multiplication Suited for Squaring and Highest Power Computation"; <http://bodrato.it/papres/#CIVV2008> for reference on the used sequence of operations.

6.11.3.3 `mzd_t* _mzd_addmul_even_weird (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)`

$C = A*B + C$ for A with offset $\neq 0$ and B with offset $== 0$.

This is scratch [code](#).

6.11.3.4 `mzd_t* _mzd_addmul_weird_even (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)`

$C = A*B + C$ for A with offset $== 0$ and B with offset $\neq 0$.

This is scratch [code](#).

6.11.3.5 `mzd_t* _mzd_addmul_weird_weird (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)`

$C = A*B + C$ for matrices with offsets $\neq 0$

This is scratch [code](#).

6.11.3.6 `mzd_t* _mzd_mul_even (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)`

Matrix multiplication via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = AB$. This is the actual implementation. Any matrix where either the number of rows or the number of columns is smaller than cutoff is processed using the M4RM algorithm.

Parameters:

- C* Preallocated product matrix, may be NULL for automatic creation.
- A* Input matrix A
- B* Input matrix B
- cutoff* Minimal dimension for Strassen recursion.

Note:

This implementation is heavily inspired by the function `strassen_window_multiply_c` in Sage 3.0; For reference see <http://www.sagemath.org>

Note:

See Marco Bodrato; "A Strassen-like Matrix Multiplication Suited for Squaring and Highest Power Computation"; <http://bodrato.it/papres/#CIVV2008> for reference on the used sequence of operations.

Todo

ideally we would use the same Wmk throughout the function but some called function doesn't like that and we end up with a wrong result if we use virtual Wmk matrices. Ideally, this should be fixed not worked around. The check whether the bug has been fixed, use only one Wmk and check if `mzd_mul(4096, 3528, 4096, 2124)` still returns the correct answer.

6.11.3.7 `mzd_t* mzd_addmul (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)`

Matrix multiplication and in-place addition via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = C + AB$. This is the wrapper function including bounds checks. See `_mzd_addmul_even` for implementation details.

Parameters:

- C* product matrix
- A* Input matrix A
- B* Input matrix B
- cutoff* Minimal dimension for Strassen recursion.

Examples:

[testsuite/test_lqqr.c](#), and [testsuite/test_multiplication.c](#).

6.11.3.8 mzd_t* mzd_mul (mzd_t * C, mzd_t * A, mzd_t * B, int cutoff)

Matrix multiplication via the Strassen-Winograd matrix multiplication algorithm, i.e. compute $C = AB$. This is the wrapper function including bounds checks. See `_mzd_mul_even` for implementation details.

Parameters:

- C* Preallocated product matrix, may be NULL for automatic creation.
- A* Input matrix A
- B* Input matrix B
- cutoff* Minimal dimension for Strassen recursion.

Examples:

[testsuite/bench_elimination.c](#), [testsuite/test_lqup.c](#), and [testsuite/test_multiplication.c](#).

6.12 trsm.h File Reference

Triangular system solving with Matrix routines. `#include "misc.h"`
`#include "packedmatrix.h"`

Functions

- void `mzd_trsm_upper_right` (mzd_t *U, mzd_t *B, const int cutoff)
Solves $XU = B$ with X and B matrices and U upper triangular.
- void `_mzd_trsm_upper_right` (mzd_t *U, mzd_t *B, const int cutoff)
Solves $XU = B$ with X and B matrices and U upper triangular.
- void `mzd_trsm_lower_right` (mzd_t *L, mzd_t *B, const int cutoff)
Solves $XL = B$ with X and B matrices and L lower triangular.
- void `_mzd_trsm_lower_right` (mzd_t *L, mzd_t *B, const int cutoff)
Solves $XL = B$ with X and B with matrices and L lower triangular.
- void `mzd_trsm_lower_left` (mzd_t *L, mzd_t *B, const int cutoff)
Solves $LX = B$ with X and B matrices and L lower triangular.
- void `_mzd_trsm_lower_left` (mzd_t *L, mzd_t *B, const int cutoff)
Solves $LX = B$ with X and B matrices and L lower triangular.
- void `mzd_trsm_upper_left` (mzd_t *U, mzd_t *B, const int cutoff)
Solves $UX = B$ with X and B matrices and U upper triangular.
- void `_mzd_trsm_upper_left` (mzd_t *U, mzd_t *B, const int cutoff)
Solves $UX = B$ with X and B matrices and U upper triangular.

6.12.2.4 void _mzd_trsm_upper_right (mzd_t * *U*, mzd_t * *B*, const int *cutoff*)

Solves $XU = B$ with X and B matrices and U upper triangular. X is stored inplace on B .

Attention:

Note, that the 'right' variants of TRSM are slower than the 'left' variants.

Parameters:

U Input upper triangular matrix.

B Input matrix, being overwritten by the solution matrix X

cutoff Minimal dimension for Strassen recursion.



- $U00$ and $B0$ are possibly located at uneven locations.
- Their column dimension is lower than 64.
- The first column of $U01$, $U11$, $B1$ are aligned at words.

6.12.2.5 void mzd_trsm_lower_left (mzd_t * *L*, mzd_t * *B*, const int *cutoff*)

Solves $LX = B$ with X and B matrices and L lower triangular. X is stored inplace on B .

This is the wrapper function including bounds checks. See [_mzd_trsm_lower_left\(\)](#) for implementation details.

Parameters:

L Input lower triangular matrix.

B Input matrix, being overwritten by the solution matrix X

cutoff Minimal dimension for Strassen recursion.

6.12.2.6 void mzd_trsm_lower_right (mzd_t * *L*, mzd_t * *B*, const int *cutoff*)

Solves $XL = B$ with X and B matrices and L lower triangular. X is stored inplace on B .

This is the wrapper function including bounds checks. See [_mzd_trsm_upper_right\(\)](#) for implementation details.

Attention:

Note, that the 'right' variants of TRSM are slower than the 'left' variants.

Parameters:

L Input upper triangular matrix.
B Input matrix, being overwritten by the solution matrix X
cutoff Minimal dimension for Strassen recursion.

6.12.2.7 void mzd_trsm_upper_left (mzd_t * U, mzd_t * B, const int cutoff)

Solves $U X = B$ with X and B matrices and U upper triangular. X is stored inplace on B.

This is the wrapper function including bounds checks. See [_mzd_trsm_upper_left\(\)](#) for implementation details.

Parameters:

U Input upper triangular matrix.
B Input matrix, being overwritten by the solution matrix X
cutoff Minimal dimension for Strassen recursion.

6.12.2.8 void mzd_trsm_upper_right (mzd_t * U, mzd_t * B, const int cutoff)

Solves $X U = B$ with X and B matrices and U upper triangular. X is stored inplace on B.

Attention:

Note, that the 'right' variants of TRSM are slower than the 'left' variants.

This is the wrapper function including bounds checks. See [_mzd_trsm_upper_right\(\)](#) for implementation details.

Parameters:

U Input upper triangular matrix.
B Input matrix, being overwritten by the solution matrix X
cutoff Minimal dimension for Strassen recursion.

7 Example Documentation

7.1 testsuite/bench_elimination.c

Ignores offset attribute of packedmatrix.

Returns:

Rank of A.

```

#include <stdlib.h>

#include "m4ri/m4ri.h"
#include "cpucycles.h"
#include "walltime.h"

int main(int argc, char **argv) {
    int halfrank = 0;
    size_t m, n, r;
    const char *algorithm;
    unsigned long long t;
    double wt;
    double clockZero = 0.0;

    if (argc < 3) {
        m4ri_die("Parameters m,n expected.\n");
    }
    if (argc == 4)
        algorithm = argv[3];
    else
        algorithm = "m4ri";
    m = atoi(argv[1]);
    n = atoi(argv[2]);
    mzd_t *A = mzd_init(m, n);
    if (!halfrank) {
        mzd_randomize(A);
    } else {
        mzd_t *L, *U;
        L = mzd_init(m, m);
        U = mzd_init(m, n);
        mzd_randomize(U);
        mzd_randomize(L);
        size_t i, j;
        for (i=0; i<m; ++i){
            mzd_write_bit(U, i, i, 1);
            for (j=0; j<i; ++j)
                mzd_write_bit(U, i, j, 0);
            if (i%2)
                for (j=i; j<n; ++j)
                    mzd_write_bit(U, i, j, 0);
            for (j=i+1; j<m; ++j)
                mzd_write_bit(L, i, j, 0);
            mzd_write_bit(L, i, i, 1);
        }
        mzd_mul(A, L, U, 0);
        mzd_free(L);
        mzd_free(U);
    }
    wt = walltime(&clockZero);
    t = cpucycles();
    if (strcmp(algorithm, "m4ri")==0)
        r = mzd_echelonize_m4ri(A, 1, 0);
    else if (strcmp(algorithm, "pluq")==0)
        r = mzd_echelonize_pluq(A, 1);
    else if (strcmp(algorithm, "naive")==0)
        r = mzd_echelonize_naive(A, 1);
    printf("m: %5d, n: %5d, r: %5d, cpu cycles: %llu wall time: %lf\n", m, n, r, cpu
        cycles() - t, walltime(&wt));

    mzd_free(A);
}

```

7.2 testsuite/test_elimination.c

Matrix elimination using the 'Method of the Four Russians' (M4RI).

Parameters:

M Matrix to be reduced.

full Return the reduced row echelon form, not only upper triangular form.

k M4RI parameter, may be 0 for auto-choose.

```
#include <stdlib.h>
#include "m4ri/m4ri.h"

int elim_test_equality(int nr, int nc) {
    int ret = 0;

    printf("elim: m: %4d, n: %4d ",nr,nc);

    mzd_t *A = mzd_init(nr, nc);
    mzd_randomize(A);
    mzd_t *B = mzd_copy(NULL, A);
    mzd_t *C = mzd_copy(NULL, A);
    mzd_t *D = mzd_copy(NULL, A);
    mzd_t *E = mzd_copy(NULL, A);
    mzd_t *F = mzd_copy(NULL, A);
    mzd_t *G = mzd_copy(NULL, A);

    /* M4RI k=auto */
    mzd_echelonize_m4ri(A, 1, 0);

    /* M4RI k=8 */
    mzd_echelonize_m4ri(B, 1, 8);

    /* M4RI Upper Triangular k=auto*/
    mzd_echelonize_m4ri(C, 0, 0);
    mzd_top_echelonize_m4ri(C, 0);

    /* M4RI Upper Triangular k=4*/
    mzd_echelonize_m4ri(D, 0, 4);
    mzd_top_echelonize_m4ri(D, 4);

    /* Gauss */
    mzd_echelonize_naive(E, 1);

    /* Gauss Upper Triangular */
    mzd_echelonize_naive(F, 0);
    mzd_top_echelonize_m4ri(F, 0);

    /* PLUQ */
    mzd_echelonize_pluq(G, 1);

    if(mzd_equal(A, B) != TRUE) {
        printf("A != B ");
        ret -= 1;
    }

    if(mzd_equal(B, C) != TRUE) {
        printf("B != C ");
        ret -= 1;
    }

    if(mzd_equal(D, E) != TRUE) {
        printf("D != E ");
        ret -= 1;
    }
}
```

```

if(mzd_equal(E, F) != TRUE) {
    printf("E != F ");
    ret -= 1;
}

if(mzd_equal(F, G) != TRUE) {
    printf("F != G ");
    ret -= 1;
}
if(mzd_equal(G, A) != TRUE) {
    printf("G != A ");
    ret -= 1;
}

mzd_free(A);
mzd_free(B);
mzd_free(C);
mzd_free(D);
mzd_free(E);
mzd_free(F);
mzd_free(G);

if(ret == 0) {
    printf(" ... passed\n");
} else {
    printf(" ... FAILED\n");
}
return ret;
}

int main(int argc, char **argv) {
    int status = 0;
    status += elim_test_equality(100, 100);
    status += elim_test_equality(21, 171);
    status += elim_test_equality(31, 121);
    status += elim_test_equality(193, 65);
    status += elim_test_equality(1025, 1025);
    status += elim_test_equality(2048, 2048);
    status += elim_test_equality(64, 64);
    status += elim_test_equality(128, 128);
    status += elim_test_equality(4096, 3528);
    status += elim_test_equality(1024, 1025);
    status += elim_test_equality(1000,1000);
    status += elim_test_equality(1000,10);
    status += elim_test_equality(1710,1290);
    status += elim_test_equality(1290, 1710);
    status += elim_test_equality(1290, 1710);
    status += elim_test_equality(1290, 1290);
    status += elim_test_equality(1000, 210);

    if (status == 0) {
        printf("All tests passed.\n");
        return 0;
    } else {
        return -1;
    }
}

```

7.3 testsuite/test_lqup.c

```

#include <stdlib.h>
#include "m4ri/m4ri.h"

int test_lqup_full_rank (size_t m, size_t n){

```

```

printf("pluq: testing full rank m: %5zu, n: %5zu",m,n);

mzd_t* U = mzd_init (m,n);
mzd_t* L = mzd_init (m,m);
mzd_t* U2 = mzd_init (m,n);
mzd_t* L2 = mzd_init (m,m);
mzd_t* A = mzd_init (m,n);
mzd_randomize (U);
mzd_randomize (L);

size_t i,j;
for (i=0; i<m; ++i){
    for (j=0; j<i && j<n;++j)
        mzd_write_bit(U,i,j, 0);
    for (j=i+1; j<m;++j)
        mzd_write_bit(L,i,j, 0);
    if(i<n)
        mzd_write_bit(U,i,i, 1);
        mzd_write_bit(L,i,i, 1);
}

mzd_mul(A, L, U, 2048);

mzd_t* Acopy = mzd_copy (NULL,A);

mzp_t* P = mzp_init(m);
mzp_t* Q = mzp_init(n);
mzd_pluq(A, P, Q, 2048);

for (i=0; i<m; ++i){
    for (j=0; j<i && j <n;++j)
        mzd_write_bit (L2, i, j, mzd_read_bit(A,i,j));
    for (j=i+1; j<n;++j)
        mzd_write_bit (U2, i, j, mzd_read_bit(A,i,j));
}

for (i=0; i<n && i<m; ++i){
    mzd_write_bit(L2,i,i, 1);
    mzd_write_bit(U2,i,i, 1);
}
mzd_addmul(Acopy,L2,U2,0);
int status = 0;
for ( i=0; i<m; ++i)
    for ( j=0; j<n; ++j){
        if (mzd_read_bit (Acopy,i,j)){
            status = 1;
        }
    }
if (status){
    printf(" ... FAILED\n");
} else
    printf (" ... passed\n");
mzd_free(U);
mzd_free(L);
mzd_free(U2);
mzd_free(L2);
mzd_free(A);
mzd_free(Acopy);
mzp_free(P);
mzp_free(Q);
return status;
}

int test_lqup_half_rank(size_t m, size_t n) {
    printf("pluq: testing half rank m: %5zd, n: %5zd",m,n);

    mzd_t* U = mzd_init(m, n);

```

```

mzd_t* L = mzd_init(m, m);
mzd_t* U2 = mzd_init(m, n);
mzd_t* L2 = mzd_init(m, m);
mzd_t* A = mzd_init(m, n);
mzd_randomize (U);
mzd_randomize (L);

size_t i, j;
for (i=0; i<m && i<n; ++i){
    mzd_write_bit(U,i,i, 1);
    for (j=0; j<i;++j)
        mzd_write_bit(U,i,j, 0);
    if (i%2)
        for (j=i; j<n;++j)
            mzd_write_bit(U,i,j, 0);
    for (j=i+1; j<m;++j)
        mzd_write_bit(L,i,j, 0);
    mzd_write_bit(L,i,i, 1);
}

mzd_mul(A, L, U, 0);

mzd_t* Acopy = mzd_copy (NULL,A);

mzp_t* Pt = mzp_init(m);
mzp_t* Q = mzp_init(n);
int r = mzd_pluq(A, Pt, Q, 0);

for (i=0; i<r; ++i){
    for (j=0; j<i;++j)
        mzd_write_bit (L2, i, j, mzd_read_bit(A,i,j));
    for (j=i+1; j<n;++j)
        mzd_write_bit (U2, i, j, mzd_read_bit(A,i,j));
}
for (i=r; i<m; i++)
    for (j=0; j<r;++j)
        mzd_write_bit (L2, i, j, mzd_read_bit(A,i,j));
for (i=0; i<r; ++i){
    mzd_write_bit(L2,i,i, 1);
    mzd_write_bit(U2,i,i, 1);
}

mzd_apply_p_left(Acopy, Pt);
mzd_apply_p_right_trans(Acopy, Q);
mzd_admmul(Acopy,L2,U2,0);

int status = 0;
for ( i=0; i<m; ++i) {
    for ( j=0; j<n; ++j){
        if (mzd_read_bit(Acopy,i,j)){
            status = 1;
        }
    }
    if(status)
        break;
}
if (status)
    printf(" ... FAILED\n");
else
    printf (" ... passed\n");
mzd_free(U);
mzd_free(L);
mzd_free(U2);
mzd_free(L2);
mzd_free(A);

```

```

    mzd_free(Acopy);
    mzp_free(Pt);
    mzp_free(Q);
    return status;
}

int test_lqup_structured(size_t m, size_t n) {

    printf("pluq: testing structured m: %5zd, n: %5zd", m, n);

    size_t i, j;
    mzd_t* A = mzd_init(m, n);
    mzd_t* L = mzd_init(m, m);
    mzd_t* U = mzd_init(m, n);

    for(i=0; i<m; i+=2)
        for (j=i; j<n; j++)
            mzd_write_bit(A, i, j, 1);

    mzd_t* Acopy = mzd_copy (NULL,A);

    mzp_t* P = mzp_init(m);
    mzp_t* Q = mzp_init(n);
    int r;
    r=mzd_pluq(A, P, Q, 0);
    printf(", rank: %5d ",r);

    for (i=0; i<r; ++i){
        for (j=0; j<i; ++j)
            mzd_write_bit(L, i, j, mzd_read_bit(A,i,j));
        for (j=i+1; j<n; ++j)
            mzd_write_bit(U, i, j, mzd_read_bit(A,i,j));
    }
    for (i=r; i<m; i++)
        for (j=0; j<r; ++j)
            mzd_write_bit(L, i, j, mzd_read_bit(A,i,j));
    for (i=0; i<r; ++i){
        mzd_write_bit(L,i,i, 1);
        mzd_write_bit(U,i,i, 1);
    }
}

mzd_apply_p_left(Acopy, P);
mzd_apply_p_right_trans(Acopy, Q);

mzd_addmul(Acopy, L, U, 0);
int status = 0;
for ( i=0; i<m; ++i)
    for ( j=0; j<n; ++j){
        if (mzd_read_bit (Acopy,i,j)){
            status = 1;
            break;
        }
    }

if (status) {
    printf("\n");
    printf(" ... FAILED\n");
} else
    printf (" ... passed\n");
mzd_free(U);
mzd_free(L);
mzd_free(A);
mzd_free(Acopy);
mzp_free(P);
mzp_free(Q);
return status;
}

```

```

int test_lqup_random(size_t m, size_t n) {
    printf("pluq: testing random m: %5zd, n: %5zd",m,n);

    size_t i,j;
    mzd_t* U = mzd_init(m, n);
    mzd_t* L = mzd_init(m, m);
    mzd_t* A = mzd_init(m, n);
    mzd_randomize(A);

    mzd_t* Acopy = mzd_copy (NULL,A);

    mzp_t* P = mzp_init(m);
    mzp_t* Q = mzp_init(n);
    int r;
    r=mzd_pluq(A, P, Q, 0);
    printf(", rank: %5d ",r);

    for (i=0; i<r; ++i){
        for (j=0; j<i;++j)
            mzd_write_bit(L, i, j, mzd_read_bit(A,i,j));
        for (j=i+1; j<n;++j)
            mzd_write_bit(U, i, j, mzd_read_bit(A,i,j));
    }
    for (i=r; i<m; i++)
        for (j=0; j<r;++j)
            mzd_write_bit(L, i, j, mzd_read_bit(A,i,j));
    for (i=0; i<r; ++i){
        mzd_write_bit(L,i,i, 1);
        mzd_write_bit(U,i,i, 1);
    }

    mzd_apply_p_left(Acopy, P);
    mzd_apply_p_right_trans(Acopy, Q);

    mzd_admmul(Acopy, L, U, 0);

    int status = 0;
    for ( i=0; i<m; ++i)
        for ( j=0; j<n; ++j){
            if (mzd_read_bit (Acopy,i,j)){
                status = 1;
                break;
            }
        }
    if (status) {
        printf(" ... FAILED\n");
    } else
        printf (" ... passed\n");
    mzd_free(U);
    mzd_free(L);
    mzd_free(A);
    mzd_free(Acopy);
    mzp_free(P);
    mzp_free(Q);
    return status;
}

int main(int argc, char **argv) {
    int status = 0;

    status += test_lqup_structured(37, 37);
    status += test_lqup_structured(63, 63);
    status += test_lqup_structured(64, 64);
    status += test_lqup_structured(65, 65);
    status += test_lqup_structured(128, 128);
}

```

```
status += test_lqup_structured(37, 137);
status += test_lqup_structured(65, 5);
status += test_lqup_structured(128, 18);

status += test_lqup_full_rank(13,13);
status += test_lqup_full_rank(37,37);
status += test_lqup_full_rank(63,63);
status += test_lqup_full_rank(64,64);
status += test_lqup_full_rank(65,65);
status += test_lqup_full_rank(97,97);
status += test_lqup_full_rank(128,128);
status += test_lqup_full_rank(150,150);
status += test_lqup_full_rank(256,256);
status += test_lqup_full_rank(1024,1024);

status += test_lqup_full_rank(13,11);
status += test_lqup_full_rank(37,39);
status += test_lqup_full_rank(64,164);
status += test_lqup_full_rank(97,92);
status += test_lqup_full_rank(128,121);
status += test_lqup_full_rank(150,153);
status += test_lqup_full_rank(256,258);
status += test_lqup_full_rank(1024,1023);

status += test_lqup_half_rank(64,64);
status += test_lqup_half_rank(65,65);
status += test_lqup_half_rank(66,66);
status += test_lqup_half_rank(127,127);
status += test_lqup_half_rank(129,129);
status += test_lqup_half_rank(148,148);
status += test_lqup_half_rank(132,132);
status += test_lqup_half_rank(256,256);
status += test_lqup_half_rank(1024,1024);

status += test_lqup_half_rank(129,127);
status += test_lqup_half_rank(132,136);
status += test_lqup_half_rank(256,251);
status += test_lqup_half_rank(1024,2100);

status += test_lqup_random(63,63);
status += test_lqup_random(64,64);
status += test_lqup_random(65,65);

status += test_lqup_random(128,128);
status += test_lqup_random(128, 131);
status += test_lqup_random(132, 731);
status += test_lqup_random(150,150);
status += test_lqup_random(252, 24);
status += test_lqup_random(256,256);
status += test_lqup_random(1024,1022);
status += test_lqup_random(1024,1024);

status += test_lqup_random(128,1280);
status += test_lqup_random(128, 130);
status += test_lqup_random(132, 132);
status += test_lqup_random(150,151);
status += test_lqup_random(252, 2);
status += test_lqup_random(256,251);
status += test_lqup_random(1024,1025);
status += test_lqup_random(1024,1021);

if (!status) {
    printf("All tests passed.\n");
    return 0;
} else {
    return -1;
}
```

```

}
}

```

7.4 testsuite/test_multiplication.c

```

#include <stdlib.h>
#include "m4ri/m4ri.h"

int mul_test_equality(int m, int l, int n, int k, int cutoff) {
    int ret = 0;
    mzd_t *A, *B, *C, *D, *E;

    printf("    mul: m: %4d, l: %4d, n: %4d, k: %2d, cutoff: %4d",m,l,n,k,cutoff);

    /* we create two random matrices */
    A = mzd_init(m, l);
    B = mzd_init(l, n);
    mzd_randomize(A);
    mzd_randomize(B);

    /* C = A*B via Strassen */
    C = mzd_mul(NULL, A, B, cutoff);

    /* D = A*B via M4RM, temporary buffers are managed internally */
    D = mzd_mul_m4rm(    NULL, A, B, k);

    /* E = A*B via naive cubic multiplication */
    E = mzd_mul_naive(    NULL, A, B);

    mzd_free(A);
    mzd_free(B);

    if (mzd_equal(C, D) != TRUE) {
        printf(" Strassen != M4RM");
        ret -=1;
    }

    if (mzd_equal(D, E) != TRUE) {
        printf(" M4RM != Naiv");
        ret -= 1;
    }

    if (mzd_equal(C, E) != TRUE) {
        printf(" Strassen != Naiv");
        ret -= 1;
    }

    mzd_free(C);
    mzd_free(D);
    mzd_free(E);

    if(ret==0) {
        printf(" ... passed\n");
    } else {
        printf(" ... FAILED\n");
    }

    return ret;
}

int sqr_test_equality(int m, int k, int cutoff) {
    int ret = 0;
    mzd_t *A, *C, *D, *E;

    printf("    sqr: m: %4d, k: %2d, cutoff: %4d",m,k,cutoff);

```

```

/* we create one random matrix */
A = mzd_init(m, m);
mzd_randomize(A);

/* C = A*A via Strassen */
C = mzd_mul(NULL, A, A, cutoff);

/* D = A*A via M4RM, temporary buffers are managed internally */
D = mzd_mul_m4rm( NULL, A, A, k);

/* E = A*A via naive cubic multiplication */
E = mzd_mul_naive( NULL, A, A);

mzd_free(A);

if (mzd_equal(C, D) != TRUE) {
    printf(" Strassen != M4RM");
    ret -=1;
}

if (mzd_equal(D, E) != TRUE) {
    printf(" M4RM != Naiv");
    ret -= 1;
}

if (mzd_equal(C, E) != TRUE) {
    printf(" Strassen != Naiv");
    ret -= 1;
}

mzd_free(C);
mzd_free(D);
mzd_free(E);

if(ret==0) {
    printf(" ... passed\n");
} else {
    printf(" ... FAILED\n");
}

return ret;
}

int addmul_test_equality(int m, int l, int n, int k, int cutoff) {
    int ret = 0;
    mzd_t *A, *B, *C, *D, *E, *F;

    printf("addmul: m: %4d, l: %4d, n: %4d, k: %2d, cutoff: %4d",m,l,n,k,cutoff);

    /* we create two random matrices */
    A = mzd_init(m, l);
    B = mzd_init(l, n);
    C = mzd_init(m, n);
    mzd_randomize(A);
    mzd_randomize(B);
    mzd_randomize(C);

    /* D = C + A*B via M4RM, temporary buffers are managed internally */
    D = mzd_copy(NULL, C);
    D = mzd_addmul_m4rm(D, A, B, k);

    /* E = C + A*B via naiv cubic multiplication */
    E = mzd_mul_m4rm(NULL, A, B, k);
    mzd_add(E, E, C);

    /* F = C + A*B via naiv cubic multiplication */

```

```

F = mzd_copy(NULL, C);
F = mzd_addmul(F, A, B, cutoff);

mzd_free(A);
mzd_free(B);
mzd_free(C);

if (mzd_equal(D, E) != TRUE) {
    printf(" M4RM != add,mul");
    ret -=1;
}
if (mzd_equal(E, F) != TRUE) {
    printf(" add,mul = addmul");
    ret -=1;
}
if (mzd_equal(F, D) != TRUE) {
    printf(" M4RM != addmul");
    ret -=1;
}

if (ret==0)
    printf(" ... passed\n");
else
    printf(" ... FAILED\n");

mzd_free(D);
mzd_free(E);
mzd_free(F);
return ret;
}

int addsqr_test_equality(int m, int k, int cutoff) {
    int ret = 0;
    mzd_t *A, *C, *D, *E, *F;

    printf("addsqr: m: %4d, k: %2d, cutoff: %4d",m,k,cutoff);

    /* we create two random matrices */
    A = mzd_init(m, m);
    C = mzd_init(m, m);
    mzd_randomize(A);
    mzd_randomize(C);

    /* D = C + A*B via M4RM, temporary buffers are managed internally */
    D = mzd_copy(NULL, C);
    D = mzd_addmul_m4rm(D, A, A, k);

    /* E = C + A*B via naive cubic multiplication */
    E = mzd_mul_m4rm(NULL, A, A, k);
    mzd_add(E, E, C);

    /* F = C + A*B via naive cubic multiplication */
    F = mzd_copy(NULL, C);
    F = mzd_addmul(F, A, A, cutoff);

    mzd_free(A);
    mzd_free(C);

    if (mzd_equal(D, E) != TRUE) {
        printf(" M4RM != add,mul");
        ret -=1;
    }
    if (mzd_equal(E, F) != TRUE) {
        printf(" add,mul = addmul");
        ret -=1;
    }
}

```

```
if (mzd_equal(F, D) != TRUE) {
    printf(" M4RM != addmul");
    ret -=1;
}

if (ret==0)
    printf(" ... passed\n");
else
    printf(" ... FAILED\n");

mzd_free(D);
mzd_free(E);
mzd_free(F);
return ret;
}

int main(int argc, char **argv) {
    int status = 0;

    status += mul_test_equality(1, 1, 1, 0, 1024);
    status += mul_test_equality(1, 128, 128, 0, 0);
    status += mul_test_equality(3, 131, 257, 0, 0);
    status += mul_test_equality(64, 64, 64, 0, 64);
    status += mul_test_equality(128, 128, 128, 0, 64);
    status += mul_test_equality(21, 171, 31, 0, 63);
    status += mul_test_equality(21, 171, 31, 0, 131);
    status += mul_test_equality(193, 65, 65, 10, 64);
    status += mul_test_equality(1025, 1025, 1025, 3, 256);
    status += mul_test_equality(2048, 2048, 4096, 0, 1024);
    status += mul_test_equality(4096, 3528, 4096, 0, 1024);
    status += mul_test_equality(1024, 1025, 1, 0, 1024);
    status += mul_test_equality(1000,1000,1000, 0, 256);
    status += mul_test_equality(1000,10,20, 0, 64);
    status += mul_test_equality(1710,1290,1000, 0, 256);
    status += mul_test_equality(1290, 1710, 200, 0, 64);
    status += mul_test_equality(1290, 1710, 2000, 0, 256);
    status += mul_test_equality(1290, 1290, 2000, 0, 64);
    status += mul_test_equality(1000, 210, 200, 0, 64);

    status += addmul_test_equality(1, 128, 128, 0, 0);
    status += addmul_test_equality(3, 131, 257, 0, 0);
    status += addmul_test_equality(64, 64, 64, 0, 64);
    status += addmul_test_equality(128, 128, 128, 0, 64);
    status += addmul_test_equality(21, 171, 31, 0, 63);
    status += addmul_test_equality(21, 171, 31, 0, 131);
    status += addmul_test_equality(193, 65, 65, 10, 64);
    status += addmul_test_equality(1025, 1025, 1025, 3, 256);
    status += addmul_test_equality(4096, 4096, 4096, 0, 2048);
    status += addmul_test_equality(1000,1000,1000, 0, 256);
    status += addmul_test_equality(1000,10,20, 0, 64);
    status += addmul_test_equality(1710,1290,1000, 0, 256);
    status += addmul_test_equality(1290, 1710, 200, 0, 64);
    status += addmul_test_equality(1290, 1710, 2000, 0, 256);
    status += addmul_test_equality(1290, 1290, 2000, 0, 64);
    status += addmul_test_equality(1000, 210, 200, 0, 64);

    status += sqr_test_equality(1, 0, 1024);
    status += sqr_test_equality(128, 0, 0);
    status += sqr_test_equality(131, 0, 0);
    status += sqr_test_equality(64, 0, 64);
    status += sqr_test_equality(128, 0, 64);
    status += sqr_test_equality(171, 0, 63);
    status += sqr_test_equality(171, 0, 131);
    status += sqr_test_equality(193, 10, 64);
    status += sqr_test_equality(1025, 3, 256);
    status += sqr_test_equality(2048, 0, 1024);
```

```
status += sqr_test_equality(3528, 0, 1024);
status += sqr_test_equality(1000, 0, 256);
status += sqr_test_equality(1000, 0, 64);
status += sqr_test_equality(1710, 0, 256);
status += sqr_test_equality(1290, 0, 64);
status += sqr_test_equality(2000, 0, 256);
status += sqr_test_equality(2000, 0, 64);
status += sqr_test_equality(210, 0, 64);

status += addsqr_test_equality(1, 0, 0);
status += addsqr_test_equality(131, 0, 0);
status += addsqr_test_equality(64, 0, 64);
status += addsqr_test_equality(128, 0, 64);
status += addsqr_test_equality(171, 0, 63);
status += addsqr_test_equality(171, 0, 131);
status += addsqr_test_equality(193, 10, 64);
status += addsqr_test_equality(1025, 3, 256);
status += addsqr_test_equality(4096, 0, 2048);
status += addsqr_test_equality(1000, 0, 256);
status += addsqr_test_equality(1000, 0, 64);
status += addsqr_test_equality(1710, 0, 256);
status += addsqr_test_equality(1290, 0, 64);
status += addsqr_test_equality(2000, 0, 256);
status += addsqr_test_equality(2000, 0, 64);
status += addsqr_test_equality(210, 0, 64);

if (status == 0) {
    printf("All tests passed.\n");
    return 0;
} else {
    return -1;
}
}
```