

M4RIE

0.20130416

Generated by Doxygen 1.8.8

Tue Oct 28 2014 17:47:46

Contents

1	Main Page	1
2	Todo List	2
3	Module Documentation	2
3.1	Type definitions	2
3.1.1	Detailed Description	2
3.2	Constructions	3
3.2.1	Detailed Description	4
3.2.2	Function Documentation	4
3.3	Assignment and basic manipulation	9
3.3.1	Detailed Description	9
3.3.2	Function Documentation	9
3.4	Operations on rows	13
3.4.1	Detailed Description	14
3.4.2	Function Documentation	14
3.5	String conversions and I/O	20
3.5.1	Detailed Description	20
3.5.2	Function Documentation	20
3.6	Addition and subtraction	21
3.6.1	Detailed Description	21
3.6.2	Macro Definition Documentation	21
3.6.3	Function Documentation	22
3.7	Multiplication	24
3.7.1	Detailed Description	25
3.7.2	Function Documentation	25
3.8	PLE and PLUQ decomposition	34
3.8.1	Detailed Description	34
3.8.2	Function Documentation	34
3.9	Echelon forms	37
3.9.1	Detailed Description	37
3.9.2	Macro Definition Documentation	37
3.9.3	Function Documentation	37
3.10	Triangular matrices	39
3.10.1	Detailed Description	39
3.10.2	Function Documentation	39

4	Data Structure Documentation	43
4.1	blm_t Struct Reference	43
4.1.1	Detailed Description	43
4.1.2	Field Documentation	43
4.2	gf2e_struct Struct Reference	44
4.2.1	Detailed Description	44
4.2.2	Field Documentation	44
4.3	mzd_poly_t Struct Reference	45
4.3.1	Detailed Description	45
4.3.2	Field Documentation	45
4.4	mzd_slice_t Struct Reference	46
4.4.1	Detailed Description	46
4.4.2	Field Documentation	46
4.5	mzed_t Struct Reference	47
4.5.1	Detailed Description	47
4.5.2	Field Documentation	47
4.6	njt_mzed_t Struct Reference	47
4.6.1	Detailed Description	48
4.6.2	Field Documentation	48
5	File Documentation	48
5.1	blm.h File Reference	48
5.1.1	Detailed Description	49
5.1.2	Function Documentation	49
5.1.3	Variable Documentation	51
5.2	conversion.h File Reference	51
5.2.1	Detailed Description	52
5.2.2	Function Documentation	52
5.3	echelonform.h File Reference	56
5.3.1	Detailed Description	57
5.4	gf2e.h File Reference	57
5.4.1	Detailed Description	58
5.4.2	Function Documentation	58
5.5	gf2x.h File Reference	59
5.5.1	Detailed Description	59
5.5.2	Macro Definition Documentation	60
5.5.3	Typedef Documentation	60

5.5.4	Function Documentation	60
5.6	m4rie.h File Reference	60
5.6.1	Detailed Description	60
5.7	mzd_poly.h File Reference	61
5.7.1	Detailed Description	62
5.7.2	Function Documentation	62
5.8	mzd_slice.h File Reference	62
5.8.1	Detailed Description	64
5.8.2	Function Documentation	64
5.9	mzed.h File Reference	65
5.9.1	Detailed Description	68
5.9.2	Function Documentation	68
5.10	newton_john.h File Reference	69
5.10.1	Detailed Description	70
5.10.2	Function Documentation	70
5.11	permutation.h File Reference	71
5.11.1	Detailed Description	71
5.11.2	Function Documentation	72
5.12	ple.h File Reference	73
5.12.1	Detailed Description	74
5.12.2	Macro Definition Documentation	74
5.13	strassen.h File Reference	74
5.13.1	Detailed Description	75
5.14	trsm.h File Reference	75
5.14.1	Detailed Description	76
5.14.2	Macro Definition Documentation	76
6	Example Documentation	76
6.1	tests/test_multiplication.c	76
Index		80

1 Main Page

M4RIE is a library to do fast computations with dense matrices over \mathbb{F}_{2^e} for small e . M4RIE is available under the GPLv2+.

The two fundamental data types of this library are `mzed_t` and `mzd_slice_t`. For big matrices, i.e., those which do not fit into L2 cache, it is recommended to use `mzd_slice_t` and for smaller matrices `mzed_t` will be slightly faster and use less memory.

Function names follow the pattern

`[_][_type][_what][_algorithm]`

Function names beginning with an underscore perform less consistency checks (matching dimensions, matching fields) than those without, e.g., `_mzed_ple()` is called by `mzed_ple()` after some checks were performed.

For both data types almost all functions are the same, e.g., there is a function `mzd_slice_add()` and there also should be a function `mzed_add()` with the same signature except for the matrix type.

Functions which do not specify an algorithm choose the best available algorithm (based on some heuristic), e.g., `mzed_ple()` might call `mzed_ple_newton_john()`.

2 Todo List

Global `gf2e_t16_init` (`const gf2e *ff, const word a`)

: this is a bit of overkill, we could do better

Global `mzd_poly_randomize` (`mzd_poly_t *A`)

Allow the user to provide a RNG callback.

Global `mzd_slice_add_elem` (`mzd_slice_t *A, const rci_t row, const rci_t col, word elem`)

This function is considerably slower than it needs to be.

Global `mzd_slice_randomize` (`mzd_slice_t *A`)

Allow the user to provide a RNG callback.

Global `mzd_slice_read_elem` (`const mzd_slice_t *A, const rci_t row, const rci_t col`)

This function is considerably slower than it needs to be.

Global `mzd_slice_write_elem` (`mzd_slice_t *A, const rci_t row, const rci_t col, word elem`)

This function is considerably slower than it needs to be.

Global `mzed_echelonize_newton_john` (`mzed_t *A, int full`)

we don't really compute the upper triangular form yet, we need to implement `_mzed_gauss_submatrix()` and a better table creation for that.

Global `mzed_randomize` (`mzed_t *A`)

Allow the user to provide a RNG callback.

3 Module Documentation

3.1 Type definitions

Data Structures

- struct `mzd_slice_t`
Dense matrices over \mathbb{F}_{2^e} represented as slices of matrices over \mathbb{F}_2 .
- struct `mzed_t`
Dense matrices over \mathbb{F}_{2^e} represented as packed matrices.

3.1.1 Detailed Description

3.2 Constructions

Functions

- `mzed_t * mzed_cling (mzed_t *A, const mzd_slice_t *Z)`
Pack a bitslice matrix into a packed representation.
- `mzd_slice_t * mzed_slice (mzd_slice_t *A, const mzed_t *Z)`
Unpack the matrix Z into bitslice representation.
- `static mzd_poly_t * mzd_poly_init (const deg_t d, const rci_t m, const rci_t n)`
Create a new polynomial of degree d with $m \times n$ matrices as coefficients.
- `static void mzd_poly_free (mzd_poly_t *A)`
Free polynomial A.
- `static mzd_poly_t * _mzd_poly_adapt_depth (mzd_poly_t *A, const deg_t new_depth)`
change depth of A to new_depth.
- `static mzd_slice_t * mzd_slice_init (const gf2e *ff, const rci_t m, const rci_t n)`
Create a new matrix of dimension $m \times n$ over ff.
- `static void mzd_slice_free (mzd_slice_t *A)`
Free a matrix created with `mzd_slice_init()`.
- `static mzd_slice_t * mzd_slice_copy (mzd_slice_t *B, const mzd_slice_t *A)`
copy A to B
- `static mzd_slice_t * mzd_slice_concat (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
Concatenate B to A and write the result to C.
- `static mzd_slice_t * mzd_slice_stack (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
Stack A on top of B and write the result to C.
- `static mzd_slice_t * mzd_slice_submatrix (mzd_slice_t *S, const mzd_slice_t *A, const size_t lowr, const size_t lowc, const size_t highr, const size_t highc)`
Copy a submatrix.
- `static mzd_slice_t * mzd_slice_init_window (const mzd_slice_t *A, 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_slice_free_window (mzd_slice_t *A)`
Free a matrix window created with `mzd_slice_init_window()`.
- `mzed_t * mzed_init (const gf2e *ff, const rci_t m, const rci_t n)`
Create a new matrix of dimension $m \times n$ over ff.
- `void mzed_free (mzed_t *A)`
Free a matrix created with `mzed_init()`.
- `static mzed_t * mzed_concat (mzed_t *C, const mzed_t *A, const mzed_t *B)`
Concatenate B to A and write the result to C.
- `static mzed_t * mzed_stack (mzed_t *C, const mzed_t *A, const mzed_t *B)`
Stack A on top of B and write the result to C.
- `static mzed_t * mzed_submatrix (mzed_t *S, const mzed_t *M, const rci_t lowr, const rci_t lowc, const rci_t highr, const rci_t highc)`
Copy a submatrix.
- `static mzed_t * mzed_init_window (const mzed_t *A, const rci_t lowr, const rci_t lowc, const rci_t highr, const rci_t highc)`
Create a window/view into the matrix A.
- `static void mzed_free_window (mzed_t *A)`
Free a matrix window created with `mzed_init_window()`.

3.2.1 Detailed Description

3.2.2 Function Documentation

3.2.2.1 `static mzd_poly_t* _mzd_poly_adapt_depth (mzd_poly_t * A, const deg_t new_depth)` `[inline],`
`[static]`

change depth of A to new_depth.

Parameters

<i>A</i>	Polynomial.
<i>new_depth</i>	New depth (may be <,<=,> than current depth).

3.2.2.2 `static void mzd_poly_free (mzd_poly_t * A)` `[inline],``[static]`

Free polynomial A.

Parameters

<i>A</i>	Polynomial.
----------	-------------

3.2.2.3 `static mzd_poly_t* mzd_poly_init (const deg_t d, const rci_t m, const rci_t n)` `[inline],``[static]`

Create a new polynomial of degree d with m x n matrices as coefficients.

Parameters

<i>d</i>	Degree.
<i>m</i>	Number of rows.
<i>n</i>	Number of columns.

3.2.2.4 `static mzd_slice_t* mzd_slice_concat (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B)`
`[inline],``[static]`

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

<i>C</i>	Matrix, may be NULL for automatic creation.
<i>A</i>	Matrix.
<i>B</i>	Matrix.

Note

This is sometimes called augment.

3.2.2.5 `static mzd_slice_t* mzd_slice_copy (mzd_slice_t * B, const mzd_slice_t * A)` `[inline],``[static]`

copy A to B

Parameters

<i>B</i>	Matrix.
<i>A</i>	Matrix.

3.2.2.6 `static void mzd_slice_free (mzd_slice_t * A) [inline],[static]`

Free a matrix created with [mzd_slice_init\(\)](#).

Parameters

<i>A</i>	Matrix.
----------	---------

Examples:

[tests/test_multiplication.c](#).

3.2.2.7 `static void mzd_slice_free_window (mzd_slice_t * A) [inline],[static]`

Free a matrix window created with [mzd_slice_init_window\(\)](#).

Parameters

<i>A</i>	Matrix
----------	--------

3.2.2.8 `static mzd_slice_t* mzd_slice_init (const gf2e * ff, const rci_t m, const rci_t n) [inline],[static]`

Create a new matrix of dimension $m \times n$ over *ff*.

Use [mzd_slice_free\(\)](#) to free it.

Parameters

<i>ff</i>	Finite field
<i>m</i>	Number of rows
<i>n</i>	Number of columns

3.2.2.9 `static mzd_slice_t* mzd_slice_init_window (const mzd_slice_t * A, const size_t lowr, const size_t lowc, const size_t highr, const size_t highc) [inline],[static]`

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 currently enforced.

Use [mzd_slice_free_window\(\)](#) to free the window.

Parameters

<i>A</i>	Matrix
<i>lowr</i>	Starting row (inclusive)

<i>lowc</i>	Starting column (inclusive)
<i>highr</i>	End row (exclusive)
<i>highc</i>	End column (exclusive)

3.2.2.10 `static mzd_slice_t* mzd_slice_stack (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B)`
`[inline],[static]`

Stack A on top of B and write the result to C.

That is,

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

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

Parameters

<i>C</i>	Matrix, may be NULL for automatic creation
<i>A</i>	Matrix
<i>B</i>	Matrix

3.2.2.11 `static mzd_slice_t* mzd_slice_submatrix (mzd_slice_t * S, const mzd_slice_t * A, const size_t lowr, const size_t lowc, const size_t highr, const size_t highc)` `[inline],[static]`

Copy a submatrix.

Parameters

<i>S</i>	Preallocated space for submatrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>lowr</i>	start rows
<i>lowc</i>	start column
<i>highr</i>	stop row (this row is <i>not</i> included)
<i>highc</i>	stop column (this column is <i>not</i> included)

3.2.2.12 `mzed_t* mzed_cling (mzed_t * A, const mzd_slice_t * Z)`

Pack a bitslice matrix into a packed representation.

Parameters

<i>A</i>	Matrix over \mathbb{F}_{2^e} or NULL
<i>Z</i>	Bitslice matrix over \mathbb{F}_{2^e}

3.2.2.13 `static mzed_t* mzed_concat (mzed_t * C, const mzed_t * A, const mzed_t * B)` `[inline],[static]`

Concatenate B to A and write the result to C.

That is,

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

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

Parameters

<i>C</i>	Matrix, may be NULL for automatic creation
<i>A</i>	Matrix
<i>B</i>	Matrix

Note

This is sometimes called augment.

3.2.2.14 void mzed_free (mzed_t * A)

Free a matrix created with [mzed_init\(\)](#).

Parameters

<i>A</i>	Matrix
----------	--------

3.2.2.15 static void mzed_free_window (mzed_t * A) [inline],[static]

Free a matrix window created with [mzed_init_window\(\)](#).

Parameters

<i>A</i>	Matrix
----------	--------

3.2.2.16 mzed_t* mzed_init (const gf2e * ff, const rci_t m, const rci_t n)

Create a new matrix of dimension m x n over ff.

Use [mzed_free\(\)](#) to kill it.

Parameters

<i>ff</i>	Finite field
<i>m</i>	Number of rows
<i>n</i>	Number of columns

3.2.2.17 static mzed_t* mzed_init_window (const mzed_t * A, const rci_t lowr, const rci_t lowc, const rci_t highr, const rci_t highc) [inline],[static]

Create a window/view into the matrix A.

A matrix window for A is a meta structure on the matrix A. 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 A which is not currently enforced.

Use [mzed_free_window\(\)](#) to free the window.

Parameters

<i>A</i>	Matrix
----------	--------

<i>lowr</i>	Starting row (inclusive)
<i>lowc</i>	Starting column (inclusive)
<i>highr</i>	End row (exclusive)
<i>highc</i>	End column (exclusive)

3.2.2.18 `mzd_slice_t* mzed_slice (mzd_slice_t * A, const mzed_t * Z)`

Unpack the matrix Z into bitslice representation.

Parameters

<i>A</i>	Bitslice matrix or NULL
<i>Z</i>	Input matrix

3.2.2.19 `static mzed_t* mzed_stack (mzed_t * C, const mzed_t * A, const mzed_t * B) [inline],[static]`

Stack A on top of B and write the result to C.

That is,

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

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

Parameters

<i>C</i>	Matrix, may be NULL for automatic creation
<i>A</i>	Matrix
<i>B</i>	Matrix

3.2.2.20 `static mzed_t* mzed_submatrix (mzed_t * S, const mzed_t * M, const rci_t lowr, const rci_t lowc, const rci_t highr, const rci_t highc) [inline],[static]`

Copy a submatrix.

Note that the upper bounds are not included.

Parameters

<i>S</i>	Preallocated space for submatrix, may be NULL for automatic creation.
<i>M</i>	Matrix
<i>lowr</i>	start rows
<i>lowc</i>	start column
<i>highr</i>	stop row (this row is <i>not</i> included)
<i>highc</i>	stop column (this column is <i>not</i> included)

3.3 Assignment and basic manipulation

Functions

- static void `mzd_poly_randomize (mzd_poly_t *A)`
Fill matrix A with random elements.
- void `mzd_slice_set_ui (mzd_slice_t *A, word value)`
Return diagonal matrix with value on the diagonal.
- static word `mzd_slice_read_elem (const mzd_slice_t *A, const rci_t row, const rci_t col)`
Get the element at position (row,col) from the matrix A.
- static void `mzd_slice_add_elem (mzd_slice_t *A, const rci_t row, const rci_t col, word elem)`
At the element elem to the element at position (row,col) in the matrix A.
- static void `mzd_slice_write_elem (mzd_slice_t *A, const rci_t row, const rci_t col, word elem)`
Write the element elem to the position (row,col) in the matrix A.
- static void `mzd_slice_randomize (mzd_slice_t *A)`
Fill matrix A with random elements.
- void `mzed_randomize (mzed_t *A)`
Fill matrix A with random elements.
- `mzed_t * mzed_copy (mzed_t *B, const mzed_t *A)`
Copy matrix A to B.
- void `mzed_set_ui (mzed_t *A, word value)`
Return diagonal matrix with value on the diagonal.
- static word `mzed_read_elem (const mzed_t *A, const rci_t row, const rci_t col)`
Get the element at position (row,col) from the matrix A.
- static void `mzed_add_elem (mzed_t *A, const rci_t row, const rci_t col, const word elem)`
At the element elem to the element at position (row,col) in the matrix A.
- static void `mzed_write_elem (mzed_t *A, const rci_t row, const rci_t col, const word elem)`
Write the element elem to the position (row,col) in the matrix A.

3.3.1 Detailed Description

3.3.2 Function Documentation

3.3.2.1 static void mzd_poly_randomize (mzd_poly_t * A) [inline],[static]

Fill matrix A with random elements.

Parameters

A	Matrix
---	--------

Todo Allow the user to provide a RNG callback.

3.3.2.2 static void mzd_slice_add_elem (mzd_slice_t * A, const rci_t row, const rci_t col, word elem) [inline],[static]

At the element elem to the element at position (row,col) in the matrix A.

Parameters

<i>A</i>	Target matrix.
<i>row</i>	Starting row.
<i>col</i>	Starting column.
<i>elem</i>	finite field element.

Todo This function is considerably slower than it needs to be.

3.3.2.3 `static void mzd_slice_randomize (mzd_slice_t * A) [inline],[static]`

Fill matrix A with random elements.

Parameters

<i>A</i>	Matrix
----------	--------

Todo Allow the user to provide a RNG callback.

3.3.2.4 `static word mzd_slice_read_elem (const mzd_slice_t * A, const rci_t row, const rci_t col) [inline],[static]`

Get the element at position (row,col) from the matrix A.

Parameters

<i>A</i>	Source matrix.
<i>row</i>	Starting row.
<i>col</i>	Starting column.

Todo This function is considerably slower than it needs to be.

3.3.2.5 `void mzd_slice_set_ui (mzd_slice_t * A, word value)`

Return diagonal matrix with value on the diagonal.

If the matrix is not square then the largest possible square submatrix is used.

Parameters

<i>A</i>	Matrix.
<i>value</i>	Finite Field element.

3.3.2.6 `static void mzd_slice_write_elem (mzd_slice_t * A, const rci_t row, const rci_t col, word elem) [inline],[static]`

Write the element elem to the position (row,col) in the matrix A.

Parameters

<i>A</i>	Target matrix.
<i>row</i>	Starting row.
<i>col</i>	Starting column.

<i>elem</i>	finite field element.
-------------	-----------------------

Todo This function is considerably slower than it needs to be.

3.3.2.7 `static void mzed_add_elem (mzed_t * A, const rci_t row, const rci_t col, const word elem)` `[inline],[static]`

At the element *elem* to the element at position (row,col) in the matrix A.

Parameters

<i>A</i>	Target matrix.
<i>row</i>	Starting row.
<i>col</i>	Starting column.
<i>elem</i>	finite field element.

3.3.2.8 `mzed_t* mzed_copy (mzed_t * B, const mzed_t * A)`

Copy matrix A to B.

Parameters

<i>B</i>	May be NULL for automatic creation.
<i>A</i>	Source matrix.

3.3.2.9 `void mzed_randomize (mzed_t * A)`

Fill matrix A with random elements.

Parameters

<i>A</i>	Matrix
----------	--------

Todo Allow the user to provide a RNG callback.

3.3.2.10 `static word mzed_read_elem (const mzed_t * A, const rci_t row, const rci_t col)` `[inline],[static]`

Get the element at position (row,col) from the matrix A.

Parameters

<i>A</i>	Source matrix.
<i>row</i>	Starting row.
<i>col</i>	Starting column.

3.3.2.11 `void mzed_set_ui (mzed_t * A, word value)`

Return diagonal matrix with value on the diagonal.

If the matrix is not square then the largest possible square submatrix is used.

Parameters

<i>A</i>	Matrix
<i>value</i>	Finite Field element

3.3.2.12 `static void mzed_write_elem (mzed_t * A, const rci_t row, const rci_t col, const word elem)` `[inline]`,
`[static]`

Write the element *elem* to the position (row,col) in the matrix *A*.

Parameters

<i>A</i>	Target matrix.
<i>row</i>	Starting row.
<i>col</i>	Starting column.
<i>elem</i>	finite field element.

3.4 Operations on rows

Functions

- static void `mzd_slice_rescale_row` (`mzd_slice_t` *A, `rci_t` r, `rci_t` c, word x)
Rescale the row r in A by X starting c.
- static void `mzd_slice_row_swap` (`mzd_slice_t` *A, const `rci_t` rowa, const `rci_t` rowb)
Swap the two rows rowa and rowb.
- static void `mzd_slice_copy_row` (`mzd_slice_t` *B, size_t i, const `mzd_slice_t` *A, size_t j)
copy row j from A to row i from B.
- static void `mzd_slice_col_swap` (`mzd_slice_t` *A, const `rci_t` cola, const `rci_t` colb)
Swap the two columns cola and colb.
- static void `mzd_slice_row_add` (`mzd_slice_t` *A, const `rci_t` sourcerow, const `rci_t` destrow)
Add the rows sourcerow and destrow and stores the total in the row destrow.
- void `mzed_add_multiple_of_row` (`mzed_t` *A, `rci_t` ar, const `mzed_t` *B, `rci_t` br, word x, `rci_t` start_col)
- static void `mzed_add_row` (`mzed_t` *A, `rci_t` ar, const `mzed_t` *B, `rci_t` br, `rci_t` start_col)
- static void `mzed_rescale_row` (`mzed_t` *A, `rci_t` r, `rci_t` start_col, const word x)
Rescale the row r in A by X starting c.
- static void `mzed_row_swap` (`mzed_t` *M, const `rci_t` rowa, const `rci_t` rowb)
Swap the two rows rowa and rowb.
- static void `mzed_copy_row` (`mzed_t` *B, `rci_t` i, const `mzed_t` *A, `rci_t` j)
copy row j from A to row i from B.
- static void `mzed_col_swap` (`mzed_t` *M, const `rci_t` cola, const `rci_t` colb)
Swap the two columns cola and colb.
- static void `mzed_col_swap_in_rows` (`mzed_t` *A, const `rci_t` cola, const `rci_t` colb, const `rci_t` start_row, `rci_t` stop_row)
Swap the two columns cola and colb but only between start_row and stop_row.
- static void `mzed_row_add` (`mzed_t` *M, const `rci_t` sourcerow, const `rci_t` destrow)
Add the rows sourcerow and destrow and stores the total in the row destrow.
- static `rci_t` `mzed_first_zero_row` (`mzed_t` *A)
Return the first row with all zero entries.
- static void `mzed_process_rows` (`mzed_t` *M, const `rci_t` startrow, const `rci_t` endrow, `rci_t` startcol, const `njt_mzed_t` *T)
The function looks up 6 entries from position i,startcol in each row and adds the appropriate row from T to the row i.
- static void `mzed_process_rows2` (`mzed_t` *M, const `rci_t` startrow, const `rci_t` endrow, const `rci_t` startcol, const `njt_mzed_t` *T0, const `njt_mzed_t` *T1)
Same as mzed_process_rows but works with two Newton-John tables in parallel.
- static void `mzed_process_rows3` (`mzed_t` *M, const `rci_t` startrow, const `rci_t` endrow, const `rci_t` startcol, const `njt_mzed_t` *T0, const `njt_mzed_t` *T1, const `njt_mzed_t` *T2)
Same as mzed_process_rows but works with three Newton-John tables in parallel.
- static void `mzed_process_rows4` (`mzed_t` *M, const `rci_t` startrow, const `rci_t` endrow, const `rci_t` startcol, const `njt_mzed_t` *T0, const `njt_mzed_t` *T1, const `njt_mzed_t` *T2, const `njt_mzed_t` *T3)
Same as mzed_process_rows but works with four Newton-John tables in parallel.
- static void `mzed_process_rows5` (`mzed_t` *M, const `rci_t` startrow, const `rci_t` endrow, const `rci_t` startcol, const `njt_mzed_t` *T0, const `njt_mzed_t` *T1, const `njt_mzed_t` *T2, const `njt_mzed_t` *T3, const `njt_mzed_t` *T4)
Same as mzed_process_rows but works with five Newton-John tables in parallel.
- static void `mzed_process_rows6` (`mzed_t` *M, const `rci_t` startrow, const `rci_t` endrow, const `rci_t` startcol, const `njt_mzed_t` *T0, const `njt_mzed_t` *T1, const `njt_mzed_t` *T2, const `njt_mzed_t` *T3, const `njt_mzed_t` *T4, const `njt_mzed_t` *T5)
Same as mzed_process_rows but works with six Newton-John tables in parallel.

3.4.1 Detailed Description

3.4.2 Function Documentation

3.4.2.1 `static void mzd_slice_col_swap (mzd_slice_t * A, const rci_t cola, const rci_t colb)` `[inline]`, `[static]`

Swap the two columns cola and colb.

Parameters

<i>A</i>	Matrix.
<i>cola</i>	Column index.
<i>colb</i>	Column index.

3.4.2.2 `static void mzd_slice_copy_row (mzd_slice_t * B, size_t i, const mzd_slice_t * A, size_t j)` `[inline]`, `[static]`

copy row j from A to row i from B.

The number of columns of A must be less than or equal to the number of columns of B.

Parameters

<i>B</i>	Target matrix.
<i>i</i>	Target row index.
<i>A</i>	Source matrix.
<i>j</i>	Source row index.

3.4.2.3 `static void mzd_slice_rescale_row (mzd_slice_t * A, rci_t r, rci_t c, word x)` `[inline]`, `[static]`

Recalc the row r in A by X starting c.

Parameters

<i>A</i>	Matrix
<i>r</i>	Row index.
<i>c</i>	Column index.
<i>x</i>	Multiplier

3.4.2.4 `static void mzd_slice_row_add (mzd_slice_t * A, const rci_t sourcerow, const rci_t destrow)` `[inline]`, `[static]`

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

Parameters

<i>A</i>	Matrix
<i>sourcerow</i>	Index of source row
<i>destrow</i>	Index of target row

Note

this can be done much faster with `mzd_combine`.

3.4.2.5 `static void mzd_slice_row_swap (mzd_slice_t * A, const rci_t rowa, const rci_t rowb)` `[inline]`, `[static]`

Swap the two rows rowa and rowb.

Parameters

<i>A</i>	Matrix
<i>rowa</i>	Row index.
<i>rowb</i>	Row index.

3.4.2.6 `void mzed_add_multiple_of_row (mzed_t * A, rci_t ar, const mzed_t * B, rci_t br, word x, rci_t start_col)`

$A[ar,c] = A[ar,c] + x*B[br,c]$ for all $c \geq \text{startcol}$.

Parameters

<i>A</i>	Matrix.
<i>ar</i>	Row index in A.
<i>B</i>	Matrix.
<i>br</i>	Row index in B.
<i>x</i>	Finite field element.
<i>start_col</i>	Column index.

3.4.2.7 `static void mzed_add_row (mzed_t * A, rci_t ar, const mzed_t * B, rci_t br, rci_t start_col)` `[inline],`
`[static]`

$A[ar,c] = A[ar,c] + B[br,c]$ for all $c \geq \text{startcol}$.

Parameters

<i>A</i>	Matrix.
<i>ar</i>	Row index in A.
<i>B</i>	Matrix.
<i>br</i>	Row index in B.
<i>start_col</i>	Column index.

3.4.2.8 `static void mzed_col_swap (mzed_t * M, const rci_t cola, const rci_t colb)` `[inline], [static]`

Swap the two columns cola and colb.

Parameters

<i>M</i>	Matrix.
<i>cola</i>	Column index.
<i>colb</i>	Column index.

3.4.2.9 `static void mzed_col_swap_in_rows (mzed_t * A, const rci_t cola, const rci_t colb, const rci_t start_row, rci_t stop_row)`
`[inline], [static]`

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

Parameters

<i>A</i>	Matrix.
<i>cola</i>	Column index.
<i>colb</i>	Column index.

<i>start_row</i>	Row index.
<i>stop_row</i>	Row index (exclusive).

3.4.2.10 `static void mzed_copy_row (mzed_t * B, rci_t i, const mzed_t * A, rci_t j)` [inline],[static]

copy row j from A to row i from B.

The the number of columns of A must be less than or equal to the number of columns of B.

Parameters

<i>B</i>	Target matrix.
<i>i</i>	Target row index.
<i>A</i>	Source matrix.
<i>j</i>	Source row index.

3.4.2.11 `static rci_t mzed_first_zero_row (mzed_t * A)` [inline],[static]

Return the first row with all zero entries.

If no such row can be found returns nrow.

Parameters

<i>A</i>	Matrix
----------	--------

3.4.2.12 `static void mzed_process_rows (mzed_t * M, const rci_t startrow, const rci_t endrow, rci_t startcol, const njt_mzed_t * T)` [inline],[static]

The function looks up 6 entries 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

<i>M</i>	Matrix to operate on
<i>startrow</i>	top row which is operated on
<i>endrow</i>	bottom row which is operated on
<i>startcol</i>	Starting column for addition
<i>T</i>	Newton-John table

3.4.2.13 `static void mzed_process_rows2 (mzed_t * M, const rci_t startrow, const rci_t endrow, const rci_t startcol, const njt_mzed_t * T0, const njt_mzed_t * T1)` [inline],[static]

Same as mzed_process_rows but works with two Newton-John tables in parallel.

Parameters

<i>M</i>	Matrix to operate on
<i>startrow</i>	top row which is operated on
<i>endrow</i>	bottom row which is operated on
<i>startcol</i>	Starting column for addition
<i>T0</i>	Newton-John table

<i>T1</i>	Newton-John table
-----------	-------------------

3.4.2.14 `static void mzed_process_rows3 (mzed_t * M, const rci_t startrow, const rci_t endrow, const rci_t startcol, const njt_mzed_t * T0, const njt_mzed_t * T1, const njt_mzed_t * T2)` `[inline],[static]`

Same as `mzed_process_rows` but works with three Newton-John tables in parallel.

Parameters

<i>M</i>	Matrix to operate on
<i>startrow</i>	top row which is operated on
<i>endrow</i>	bottom row which is operated on
<i>startcol</i>	Starting column for addition
<i>T0</i>	Newton-John table
<i>T1</i>	Newton-John table
<i>T2</i>	Newton-John table

3.4.2.15 `static void mzed_process_rows4 (mzed_t * M, const rci_t startrow, const rci_t endrow, const rci_t startcol, const njt_mzed_t * T0, const njt_mzed_t * T1, const njt_mzed_t * T2, const njt_mzed_t * T3)` `[inline],[static]`

Same as `mzed_process_rows` but works with four Newton-John tables in parallel.

Parameters

<i>M</i>	Matrix to operate on
<i>startrow</i>	top row which is operated on
<i>endrow</i>	bottom row which is operated on
<i>startcol</i>	Starting column for addition
<i>T0</i>	Newton-John table
<i>T1</i>	Newton-John table
<i>T2</i>	Newton-John table
<i>T3</i>	Newton-John table

3.4.2.16 `static void mzed_process_rows5 (mzed_t * M, const rci_t startrow, const rci_t endrow, const rci_t startcol, const njt_mzed_t * T0, const njt_mzed_t * T1, const njt_mzed_t * T2, const njt_mzed_t * T3, const njt_mzed_t * T4)` `[inline],[static]`

Same as `mzed_process_rows` but works with five Newton-John tables in parallel.

Parameters

<i>M</i>	Matrix to operate on
<i>startrow</i>	top row which is operated on
<i>endrow</i>	bottom row which is operated on
<i>startcol</i>	Starting column for addition
<i>T0</i>	Newton-John table
<i>T1</i>	Newton-John table
<i>T2</i>	Newton-John table

<i>T3</i>	Newton-John table
<i>T4</i>	Newton-John table

3.4.2.17 `static void mzed_process_rows6 (mzed_t * M, const rci_t startrow, const rci_t endrow, const rci_t startcol, const njt_mzed_t * T0, const njt_mzed_t * T1, const njt_mzed_t * T2, const njt_mzed_t * T3, const njt_mzed_t * T4, const njt_mzed_t * T5) [inline],[static]`

Same as `mzed_process_rows` but works with six Newton-John tables in parallel.

Parameters

<i>M</i>	Matrix to operate on
<i>startrow</i>	top row which is operated on
<i>endrow</i>	bottom row which is operated on
<i>startcol</i>	Starting column for addition
<i>T0</i>	Newton-John table
<i>T1</i>	Newton-John table
<i>T2</i>	Newton-John table
<i>T3</i>	Newton-John table
<i>T4</i>	Newton-John table
<i>T5</i>	Newton-John table

3.4.2.18 `static void mzed_rescale_row (mzed_t * A, rci_t r, rci_t start_col, const word x) [inline],[static]`

Rescale the row *r* in *A* by *X* starting *c*.

Parameters

<i>A</i>	Matrix
<i>r</i>	Row index.
<i>start_col</i>	Column index.
<i>x</i>	Multiplier

3.4.2.19 `static void mzed_row_add (mzed_t * M, const rci_t sourcerow, const rci_t destrow) [inline],[static]`

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

Parameters

<i>M</i>	Matrix
<i>sourcerow</i>	Index of source row
<i>destrow</i>	Index of target row

Note

this can be done much faster with `mzed_combine`.

3.4.2.20 `static void mzed_row_swap (mzed_t * M, const rci_t rowa, const rci_t rowb) [inline],[static]`

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

Parameters

M	Matrix
$rowa$	Row index.
$rowb$	Row index.

3.5 String conversions and I/O

Functions

- void [mzd_slice_print](#) (const [mzd_slice_t](#) *A)
Print a matrix to stdout.
- void [mzed_print](#) (const [mzed_t](#) *M)
Print a matrix to stdout.

3.5.1 Detailed Description

3.5.2 Function Documentation

3.5.2.1 void [mzd_slice_print](#) (const [mzd_slice_t](#) * A)

Print a matrix to stdout.

Parameters

<i>A</i>	Matrix
----------	--------

3.5.2.2 void [mzed_print](#) (const [mzed_t](#) * M)

Print a matrix to stdout.

Parameters

<i>M</i>	Matrix
----------	--------

3.6 Addition and subtraction

Macros

- `#define mzd_slice_sub mzd_slice_add`
 $C = A + B.$
- `#define _mzd_slice_sub _mzd_slice_add`
 $C = A + B.$
- `#define mzed_sub mzed_add`
 $C = A + B.$
- `#define _mzed_sub _mzed_add`
 $C = A + B.$

Functions

- static `mzd_poly_t * _mzd_poly_add` (`mzd_poly_t *C`, const `mzd_poly_t *A`, const `mzd_poly_t *B`, unsigned int offset)
 $C += (A+B)*x^{\text{offset}}.$
- static `mzd_poly_t * mzd_poly_add` (`mzd_poly_t *C`, const `mzd_poly_t *A`, const `mzd_poly_t *B`)
 $C += (A+B)$
- static `mzd_slice_t * _mzd_slice_add` (`mzd_slice_t *C`, const `mzd_slice_t *A`, const `mzd_slice_t *B`)
 $C = A + B.$
- static `mzd_slice_t * mzd_slice_add` (`mzd_slice_t *C`, const `mzd_slice_t *A`, const `mzd_slice_t *B`)
 $C = A + B.$
- `mzed_t * mzed_add` (`mzed_t *C`, const `mzed_t *A`, const `mzed_t *B`)
 $C = A + B.$
- `mzed_t * _mzed_add` (`mzed_t *C`, const `mzed_t *A`, const `mzed_t *B`)
 $C = A + B.$

3.6.1 Detailed Description

3.6.2 Macro Definition Documentation

3.6.2.1 `#define _mzd_slice_sub _mzd_slice_add`

$C = A + B.$

Parameters

<i>C</i>	Preallocated sum matrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.6.2.2 `#define _mzed_sub _mzed_add`

$C = A + B.$

Parameters

<i>C</i>	Preallocated difference matrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.6.2.3 #define mzd_slice_sub mzd_slice_add

$$C = A + B.$$

C is also returned. If C is NULL then a new matrix is created which must be freed by [mzd_slice_free\(\)](#).

Parameters

<i>C</i>	Preallocated sum matrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.6.2.4 #define mzed_sub mzed_add

$$C = A + B.$$

Parameters

<i>C</i>	Preallocated difference matrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.6.3 Function Documentation

3.6.3.1 static mzd_poly_t* _mzd_poly_add (mzd_poly_t * C, const mzd_poly_t * A, const mzd_poly_t * B, unsigned int offset) [inline], [static]

$$C += (A+B)*x^{\text{offset}}.$$

Parameters

<i>C</i>	Target polynomial.
<i>A</i>	Source polynomial.
<i>B</i>	Source polynomial.
<i>offset</i>	The result is shifted offset entries upwards.

Warning

No bounds checks are performed.

3.6.3.2 static mzd_slice_t* _mzd_slice_add (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B) [inline], [static]

$$C = A + B.$$

Parameters

<i>C</i>	Preallocated sum matrix.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.6.3.3 `mzed_t* _mzed_add (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = A + B$.

Parameters

<i>C</i>	Preallocated sum matrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.6.3.4 `static mzd_poly_t* mzd_poly_add (mzd_poly_t * C, const mzd_poly_t * A, const mzd_poly_t * B)`
`[inline], [static]`

$C += (A+B)$

Parameters

<i>C</i>	Target polynomial.
<i>A</i>	Source polynomial.
<i>B</i>	Source polynomial.

3.6.3.5 `static mzd_slice_t* mzd_slice_add (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B)`
`[inline], [static]`

$C = A + B$.

C is also returned. If C is NULL then a new matrix is created which must be freed by [mzd_slice_free\(\)](#).

Parameters

<i>C</i>	Preallocated sum matrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.6.3.6 `mzed_t* mzed_add (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = A + B$.

C is also returned. If C is NULL then a new matrix is created which must be freed by [mzed_free\(\)](#).

Parameters

<i>C</i>	Preallocated sum matrix, may be NULL for automatic creation.
<i>A</i>	Matrix
<i>B</i>	Matrix

3.7 Multiplication

Functions

- static `mzd_poly_t * _mzd_poly_addmul_naive` (`mzd_poly_t *C`, `const mzd_poly_t *A`, `const mzd_poly_t *B`)
 $C += A \cdot B$ using naive polynomial multiplication.
- static `mzd_poly_t * _mzd_poly_addmul_karatsuba_balanced` (`mzd_poly_t *C`, `const mzd_poly_t *A`, `const mzd_poly_t *B`)
 $C += A \cdot B$ using Karatsuba multiplication on balanced inputs.
- void `_mzd_ptr_addmul_karatsuba2` (`const gf2e **X`, `const mzd_t **A`, `const mzd_t **B`)
 $X += A \cdot B$ over \mathbb{F}_{2^2} using 3 multiplications over \mathbb{F}_2 and 2 temporary \mathbb{F}_2 matrices.
- `mzd_slice_t * _mzd_slice_addmul_naive` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`)
 $C = A \cdot B$ using quadratic polynomial multiplication with matrix coefficients.
- static `mzd_slice_t * _mzd_slice_addmul_karatsuba` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`)
 $C = C + A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .
- static `mzd_slice_t * mzd_slice_mul_karatsuba` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`)
 $C = A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .
- static `mzd_slice_t * mzd_slice_addmul_karatsuba` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`)
 $C = C + A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .
- static `mzd_slice_t * _mzd_slice_mul_blm` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`, `blm_t *f`)
 $C = A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .
- static `mzd_slice_t * mzd_slice_mul_blm` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`, `blm_t *f`)
 $C = A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .
- static `mzd_slice_t * mzd_slice_addmul_blm` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`, `blm_t *f`)
 $C = C + A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .
- `mzd_slice_t * mzd_slice_mul_scalar` (`mzd_slice_t *C`, `const word a`, `const mzd_slice_t *B`)
 $C = a \cdot B$.
- `mzd_slice_t * mzd_slice_addmul_scalar` (`mzd_slice_t *C`, `const word a`, `const mzd_slice_t *B`)
 $C += a \cdot B$.
- static `mzd_slice_t * mzd_slice_mul` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`)
 $C = A \cdot B$.
- static `mzd_slice_t * mzd_slice_addmul` (`mzd_slice_t *C`, `const mzd_slice_t *A`, `const mzd_slice_t *B`)
 $C = C + A \cdot B$.
- `mzed_t * mzed_mul` (`mzed_t *C`, `const mzed_t *A`, `const mzed_t *B`)
 $C = A \cdot B$.
- `mzed_t * mzed_addmul` (`mzed_t *C`, `const mzed_t *A`, `const mzed_t *B`)
 $C = C + A \cdot B$.
- `mzed_t * _mzed_mul` (`mzed_t *C`, `const mzed_t *A`, `const mzed_t *B`)
 $C = A \cdot B$.
- `mzed_t * _mzed_addmul` (`mzed_t *C`, `const mzed_t *A`, `const mzed_t *B`)
 $C = C + A \cdot B$.
- `mzed_t * mzed_addmul_naive` (`mzed_t *C`, `const mzed_t *A`, `const mzed_t *B`)
 $C = C + A \cdot B$ using naive cubic multiplication.
- `mzed_t * mzed_mul_naive` (`mzed_t *C`, `const mzed_t *A`, `const mzed_t *B`)
 $C = A \cdot B$ using naive cubic multiplication.

- `mzed_t * _mzed_mul_naive (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = C + A \cdot B$ using naive cubic multiplication.
- `mzed_t * mzed_mul_scalar (mzed_t *C, const word a, const mzed_t *B)`
 $C = a \cdot B$.
- `mzed_t * mzed_mul_newton_john (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = A \cdot B$ using Newton-John tables.
- `mzed_t * mzed_addmul_newton_john (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = C + A \cdot B$ using Newton-John tables.
- `mzed_t * _mzed_mul_newton_john0 (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = C + A \cdot B$ using Newton-John tables.
- `mzed_t * _mzed_mul_newton_john (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = C + A \cdot B$ using Newton-John tables.
- `mzed_t * mzed_mul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`
 $C = A \cdot B$ using Strassen-Winograd.
- `mzed_t * mzed_addmul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`
 $C = C + A \cdot B$ using Strassen-Winograd.
- `mzed_t * _mzed_mul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`
 $C = A \cdot B$ using Strassen-Winograd.
- `mzed_t * _mzed_addmul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`
 $C = A \cdot B$ using Strassen-Winograd.
- `rci_t _mzed_strassen_cutoff (const mzed_t *C, const mzed_t *A, const mzed_t *B)`
Return heuristic choice for crossover parameter for Strassen-Winograd multiplication given A, B and C.

3.7.1 Detailed Description

3.7.2 Function Documentation

3.7.2.1 `static mzd_poly_t* _mzd_poly_addmul_karatsubs_balanced (mzd_poly_t * C, const mzd_poly_t * A, const mzd_poly_t * B)` `[inline], [static]`

$C += A \cdot B$ using Karatsuba multiplication on balanced inputs.

Parameters

<i>C</i>	Target polynomial.
<i>A</i>	Source polynomial.
<i>B</i>	Source polynomial.

3.7.2.2 `static mzd_poly_t* _mzd_poly_addmul_naive (mzd_poly_t * C, const mzd_poly_t * A, const mzd_poly_t * B)` `[inline], [static]`

$C += A \cdot B$ using naive polynomial multiplication.

Parameters

<i>C</i>	Target polynomial.
<i>A</i>	Source polynomial.

B	Source polynomial.
-----	--------------------

3.7.2.3 `void _mzd_ptr_addmul_karatsuba2 (const gf2e * ff, mzd_t ** X, const mzd_t ** A, const mzd_t ** B)`

$X += A \cdot B$ over \mathbb{F}_2 using 3 multiplications over \mathbb{F}_2 and 2 temporary \mathbb{F}_2 matrices.

karatsuba.c If no finite field is given, polynomial arithmetic with polynomials of degree 1 is performed. In this case, X is expected to have at least length 3. If a finite field is given, then C is expected to have at least length 2.

The formula was taken from Peter L. Montgomery. "Five, Six, and Seven-Term Karatsuba-Like Formulae" in IEEE TRANSACTIONS ON COMPUTERS, VOL. 54, NO. 3, MARCH 2005/

Parameters

ff	Finite Field, may be NULL for polynomial arithmetic.
X	Preallocated return matrix, of length ≥ 2 ($ff \neq \text{NULL}$) or ≥ 3 ($ff == \text{NULL}$)
A	Input matrix A, preallocated of length ≥ 2 .
B	Input matrix B, preallocated of length ≥ 2 .

See also

`_mzd_ptr_addmul_karatsuba()`

3.7.2.4 `static mzd_slice_t* _mzd_slice_addmul_karatsuba (mzd_slice_t* C, const mzd_slice_t* A, const mzd_slice_t* B) [inline], [static]`

$C = C + A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .

This function reduces matrix multiplication over \mathbb{F}_{2^e} to matrix multiplication over \mathbb{F}_2 .

As an example consider \mathbb{F}_4 . The minimal polynomial is $x^2 + x + 1$. The matrix A can be represented as $A0 \cdot x + A1$ and the matrix B can be represented as $B0 \cdot x + B1$. Their product C is

$$A0 \cdot B0 \cdot x^2 + (A0 \cdot B1 + A1 \cdot B0) \cdot x + A1 \cdot B1.$$

Reduction modulo $x^2 + x + 1$ gives

$$(A0 \cdot B0 + A0 \cdot B1 + A1 \cdot B0) \cdot x + A1 \cdot B1 + A0 \cdot B0.$$

This can be re-written as

$$((A0 + A1) \cdot (B0 + B1) + A1 \cdot B1) \cdot x + A1 \cdot B1 + A0 \cdot B0$$

and thus this multiplication costs 3 matrix multiplications over \mathbb{F}_2 and 4 matrix additions over \mathbb{F}_2 .

This technique was proposed in Tomas J. Boothby and Robert W. Bradshaw; Bitslicing and the Method of Four Russians Over Larger Finite Fields; 2009; <http://arxiv.org/abs/0901.1413>

Parameters

C	Preallocated return matrix, may be NULL for automatic creation.
A	Input matrix A.
B	Input matrix B.

See also

[mzed_mul\(\)](#) [mzd_slice_mul\(\)](#) [mzd_slice_addmul_karatsuba\(\)](#)

3.7.2.5 `mzd_slice_t* _mzd_slice_addmul_naive (mzd_slice_t* C, const mzd_slice_t* A, const mzd_slice_t* B)`

$C = A \cdot B$ using quadratic polynomial multiplication with matrix coefficients.

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

3.7.2.6 `static mzd_slice_t* _mzd_slice_mul_blm (mzd_slice_t* C, const mzd_slice_t* A, const mzd_slice_t* B, blm_t* f)` `[inline], [static]`

$C = A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.
<i>f</i>	Bilinear map such that $C == H*((F*A) \times (G*B))$, if NULL it will be created and destroyed

Note

Calling `_mzd_slice_addmul_karatsuba` will be more efficient

3.7.2.7 `mzed_t* _mzed_addmul (mzed_t* C, const mzed_t* A, const mzed_t* B)`

$C = C + A \cdot B$.

Parameters

<i>C</i>	Preallocated product matrix.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

3.7.2.8 `mzed_t* _mzed_addmul_strassen (mzed_t* C, const mzed_t* A, const mzed_t* B, int cutoff)`

$C = A \cdot B$ using Strassen-Winograd.

This function uses Strassen-Winograd multiplication (Bodrato variant) recursively until it reaches the cutoff, where it switches to Newton-John table based multiplication or naive multiplication.

Parameters

<i>C</i>	Preallocated product matrix.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.
<i>cutoff</i>	Crossover to basecase dimension > 64

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.

3.7.2.9 `mzed_t* _mzed_mul (mzed_t* C, const mzed_t* A, const mzed_t* B)`

$C = A \cdot B$.

Parameters

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

3.7.2.10 `mzed_t* _mzed_mul_naive (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = C + A \cdot B$ using naive cubic multiplication.

Parameters

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

3.7.2.11 `mzed_t* _mzed_mul_newton_john (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = C + A \cdot B$ using Newton-John tables.

This is an optimised implementation.

Parameters

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

See also

[mzed_mul\(\)](#)

3.7.2.12 `mzed_t* _mzed_mul_newton_john0 (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = C + A \cdot B$ using Newton-John tables.

This is a simple implementation for clarity of presentation. Do not call, it is slow.

Parameters

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

See also

[mzed_mul_newton_john\(\)](#) [mzed_mul\(\)](#)

3.7.2.13 `mzed_t* _mzed_mul_strassen (mzed_t * C, const mzed_t * A, const mzed_t * B, int cutoff)`

$C = A \cdot B$ using Strassen-Winograd.

This function uses Strassen-Winograd multiplication (Bodrato variant) recursively until it reaches the cutoff, where it switches to Newton-John table based multiplication or naive multiplication.

Parameters

<i>C</i>	Preallocated product matrix.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.
<i>cutoff</i>	Crossover to basecase dimension > 64

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.

3.7.2.14 `rci_t_mzed_strassen_cutoff (const mzed_t * C, const mzed_t * A, const mzed_t * B)`

Return heuristic choice for crossover parameter for Strassen-Winograd multiplication given A, B and C.

Parameters

<i>C</i>	Matrix (ignored)
<i>A</i>	Matrix
<i>B</i>	Matrix (ignored)

3.7.2.15 `static mzd_slice_t* mzd_slice_addmul (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B)`
[inline], [static]

$$C = C + A \cdot B.$$

Parameters

<i>C</i>	Preallocated return matrix.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

See also

`_mzd_slice_addmul_karatsuba(n)`

3.7.2.16 `static mzd_slice_t* mzd_slice_addmul_blm (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B, blm_t * f)` [inline], [static]

$C = C + A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.
<i>f</i>	Bilinear map such that $C == C + H*((F*A) \times (G*B))$, if NULL it will be created and destroyed

Note

Calling `mzd_slice_addmul_karatsuba` will be more efficient

3.7.2.17 `static mzd_slice_t* mzd_slice_addmul_karatsuba (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B)` [inline], [static]

$C = C + A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .

Parameters

C	Preallocated return matrix.
A	Input matrix A.
B	Input matrix B.

See also

[_mzd_slice_addmul_karatsuba\(\)](#)

3.7.2.18 `mzd_slice_t* mzd_slice_addmul_scalar (mzd_slice_t * C, const word a, const mzd_slice_t * B)`

$$C += a \cdot B.$$

Parameters

C	Preallocated product matrix.
a	finite field element.
B	Input matrix B.

3.7.2.19 `static mzd_slice_t* mzd_slice_mul (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B)` [inline], [static]

$$C = A \cdot B.$$

Parameters

C	Preallocated return matrix, may be NULL for automatic creation.
A	Input matrix A.
B	Input matrix B.

See also

[_mzd_slice_addmul_karatsuba\(\)](#)

3.7.2.20 `static mzd_slice_t* mzd_slice_mul_blm (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B, blm_t * f)` [inline], [static]

$$C = A \cdot B \text{ using bilinear maps over matrices over } \mathbb{F}_2.$$

Parameters

C	Preallocated return matrix, may be NULL for automatic creation.
A	Input matrix A.
B	Input matrix B.
f	Bilinear map such that $C == H*((F*A) \times (G*B))$, if NULL it will be created and destroyed

Note

Calling `mzd_slice_mul_karatsuba` will be more efficient

3.7.2.21 `static mzd_slice_t* mzd_slice_mul_karatsuba (mzd_slice_t * C, const mzd_slice_t * A, const mzd_slice_t * B)` [inline], [static]

$$C = A \cdot B \text{ using Karatsuba multiplication of polynomials over matrices over } \mathbb{F}_2.$$

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

See also

[_mzd_slice_addmul_karatsuba\(\)](#)

3.7.2.22 `mzd_slice_t* mzd_slice_mul_scalar (mzd_slice_t* C, const word a, const mzd_slice_t* B)`

$$C = a \cdot B.$$

Parameters

<i>C</i>	Preallocated product matrix or NULL.
<i>a</i>	finite field element.
<i>B</i>	Input matrix B.

3.7.2.23 `mzed_t* mzed_addmul (mzed_t* C, const mzed_t* A, const mzed_t* B)`

$$C = C + A \cdot B.$$

Parameters

<i>C</i>	Preallocated product matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

3.7.2.24 `mzed_t* mzed_addmul_naive (mzed_t* C, const mzed_t* A, const mzed_t* B)`

$$C = C + A \cdot B \text{ using naive cubic multiplication.}$$

Parameters

<i>C</i>	Preallocated product matrix.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

Note

There is no reason to call this function except for checking the correctness of other algorithms. It is very slow.

3.7.2.25 `mzed_t* mzed_addmul_newton_john (mzed_t* C, const mzed_t* A, const mzed_t* B)`

$$C = C + A \cdot B \text{ using Newton-John tables.}$$

Parameters

<i>C</i>	Preallocated product matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.

<i>B</i>	Input matrix B.
----------	-----------------

See also

[_mzed_mul_newton_john\(\)](#) [mzed_mul\(\)](#)

3.7.2.26 `mzed_t* mzed_addmul_strassen (mzed_t * C, const mzed_t * A, const mzed_t * B, int cutoff)`

$C = C + A \cdot B$ using Strassen-Winograd.

This function uses Strassen-Winograd multiplication (Bodrato variant) recursively until it reaches the cutoff, where it switches to Newton-John table based multiplication or naive multiplication.

Parameters

<i>C</i>	Preallocated product matrix, may be NULL for allocation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.
<i>cutoff</i>	Crossover to basecase dimension > 64 or 0 for heuristic choice.

3.7.2.27 `mzed_t* mzed_mul (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = A \cdot B$.

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

3.7.2.28 `mzed_t* mzed_mul_naive (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = A \cdot B$ using naive cubic multiplication.

Parameters

<i>C</i>	Preallocated product matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

Note

There is no reason to call this function except for checking the correctness of other algorithms. It is very slow.

3.7.2.29 `mzed_t* mzed_mul_newton_john (mzed_t * C, const mzed_t * A, const mzed_t * B)`

$C = A \cdot B$ using Newton-John tables.

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.

<i>B</i>	Input matrix B.
----------	-----------------

See also

[mzed_mul_mzed_mul_newton_john0\(\)](#)

3.7.2.30 `mzed_t* mzed_mul_scalar (mzed_t * C, const word a, const mzed_t * B)`

$C = a \cdot B$.

Parameters

<i>C</i>	Preallocated product matrix or NULL.
<i>a</i>	finite field element.
<i>B</i>	Input matrix B.

The algorithm proceeds as follows:

0) If a direct approach would need less lookups we use that.

1) We generate a lookup table of 16-bit wide entries

2) We use that lookup table to do 4 lookups per word

3.7.2.31 `mzed_t* mzed_mul_strassen (mzed_t * C, const mzed_t * A, const mzed_t * B, int cutoff)`

$C = A \cdot B$ using Strassen-Winograd.

This function uses Strassen-Winograd multiplication (Bodrato variant) recursively until it reaches the cutoff, where it switches to Newton-John table based multiplication or naive multiplication.

Parameters

<i>C</i>	Preallocated product matrix, may be NULL for allocation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.
<i>cutoff</i>	Crossover to basecase dimension > 64 or 0 for heuristic choice

3.8 PLE and PLUQ decomposition

Functions

- `rci_t mzed_ple_newton_john (mzed_t *A, mzp_t *P, mzp_t *Q)`
PLE decomposition: $L \cdot E = P \cdot A$ using Newton-John tables.
- `rci_t mzed_ple_naive (mzed_t *A, mzp_t *P, mzp_t *Q)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `rci_t _mzd_slice_ple (mzd_slice_t *A, mzp_t *P, mzp_t *Q, rci_t cutoff)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `static rci_t mzd_slice_ple (mzd_slice_t *A, mzp_t *P, mzp_t *Q)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `rci_t _mzd_slice_pluq (mzd_slice_t *A, mzp_t *P, mzp_t *Q, rci_t cutoff)`
PLUQ decomposition: $L \cdot U \cdot Q = P \cdot A$.
- `static rci_t mzd_slice_pluq (mzd_slice_t *A, mzp_t *P, mzp_t *Q)`
PLUQ decomposition: $L \cdot U \cdot Q = P \cdot A$.
- `rci_t _mzed_ple (mzed_t *A, mzp_t *P, mzp_t *Q, rci_t cutoff)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `static rci_t mzed_ple (mzed_t *A, mzp_t *P, mzp_t *Q)`
PLE decomposition: $L \cdot E = P \cdot A$.

3.8.1 Detailed Description

3.8.2 Function Documentation

3.8.2.1 `rci_t _mzd_slice_ple (mzd_slice_t * A, mzp_t * P, mzp_t * Q, rci_t cutoff)`

PLE decomposition: $L \cdot E = P \cdot A$.

Modifies A in place to store lower triangular L below (and on) the main diagonal and E – an echelon form of A – above the main diagonal (pivots are stored in Q). P and Q are updated with row and column permutations respectively.

This function uses either asymptotically fast PLE decomposition by reducing it to matrix multiplication or naive cubic PLE decomposition depending on the size of the underlying field. If asymptotically fast PLE decomposition is used, then the algorithm switches to `mzed_ple_newton_john` if $e * ncols * nrows$ is \leq cutoff where e is the exponent of the finite field.

Parameters

<i>A</i>	Matrix
<i>P</i>	Permutation vector of length A->nrows
<i>Q</i>	Permutation vector of length A->ncols
<i>cutoff</i>	Integer

See also

[mzed_ple_naive\(\)](#) [mzed_ple_newton_john\(\)](#) [mzed_ple\(\)](#)

3.8.2.2 `rci_t _mzd_slice_pluq (mzd_slice_t * A, mzp_t * P, mzp_t * Q, rci_t cutoff)`

PLUQ decomposition: $L \cdot U \cdot Q = P \cdot A$.

This function implements asymptotically fast PLE decomposition by reducing it to matrix multiplication. From PLE the PLUQ decomposition is then obtained.

Parameters

<i>A</i>	Matrix
<i>P</i>	Permutation vector of length $A \rightarrow \text{nrows}$
<i>Q</i>	Permutation vector of length $A \rightarrow \text{ncols}$
<i>cutoff</i>	Crossover to base case if $\text{mzed_t::w} * \text{mzed_t::ncols} * \text{mzed_t::nrows} < \text{cutoff}$.

3.8.2.3 `rci_t_mzed_ple (mzed_t * A, mzp_t * P, mzp_t * Q, rci_t cutoff)`

PLE decomposition: $L \cdot E = P \cdot A$.

Modifies *A* in place to store lower triangular *L* below (and on) the main diagonal and *E* – an echelon form of *A* – above the main diagonal (pivots are stored in *Q*). *P* and *Q* are updated with row and column permutations respectively.

This function uses either asymptotically fast PLE decomposition by reducing it to matrix multiplication or naive cubic PLE decomposition depending on the size of the underlying field. If asymptotically fast PLE decomposition is used, then the algorithm switches to `mzed_ple_newton_john` if $e * \text{ncols} * \text{nrows}$ is $\leq \text{cutoff}$ where *e* is the exponent of the finite field.

Parameters

<i>A</i>	Matrix
<i>P</i>	Permutation vector of length $A \rightarrow \text{nrows}$
<i>Q</i>	Permutation vector of length $A \rightarrow \text{ncols}$
<i>cutoff</i>	Integer ≥ 0

See also

[mzed_ple_naive\(\)](#) [mzed_ple_newton_john\(\)](#) [_mzed_ple\(\)](#)

3.8.2.4 `static rci_t mzd_slice_ple (mzd_slice_t * A, mzp_t * P, mzp_t * Q) [inline], [static]`

PLE decomposition: $L \cdot E = P \cdot A$.

Modifies *A* in place to store lower triangular *L* below (and on) the main diagonal and *E* – an echelon form of *A* – above the main diagonal (pivots are stored in *Q*). *P* and *Q* are updated with row and column permutations respectively.

This function implements asymptotically fast PLE decomposition by reducing it to matrix multiplication.

Parameters

<i>A</i>	Matrix
<i>P</i>	Permutation vector of length $A \rightarrow \text{nrows}$
<i>Q</i>	Permutation vector of length $A \rightarrow \text{ncols}$

See also

[mzed_ple_naive\(\)](#) [mzed_ple_newton_john\(\)](#) [_mzd_slice_ple\(\)](#)

3.8.2.5 `static rci_t mzd_slice_pluq (mzd_slice_t * A, mzp_t * P, mzp_t * Q) [inline], [static]`

PLUQ decomposition: $L \cdot U \cdot Q = P \cdot A$.

This function implements asymptotically fast PLE decomposition by reducing it to matrix multiplication. From PLE the PLUQ decomposition is then obtained.

Parameters

A	Matrix
P	Permutation vector of length $A \rightarrow \text{nrows}$
Q	Permutation vector of length $A \rightarrow \text{ncols}$

3.8.2.6 `static rci_t mzed_ple (mzed_t * A, mzp_t * P, mzp_t * Q)` `[inline], [static]`

PLE decomposition: $L \cdot E = P \cdot A$.

Modifies A in place to store lower triangular L below (and on) the main diagonal and E – an echelon form of A – above the main diagonal (pivots are stored in Q). P and Q are updated with row and column permutations respectively.

This function uses either asymptotically fast PLE decomposition by reducing it to matrix multiplication or naive cubic PLE decomposition depending on the size of the underlying field.

Parameters

A	Matrix
P	Permutation vector of length $A \rightarrow \text{nrows}$
Q	Permutation vector of length $A \rightarrow \text{ncols}$

See also

[mzed_ple_naive\(\)](#) [mzed_ple_newton_john\(\)](#) [_mzed_ple\(\)](#)

3.8.2.7 `rci_t mzed_ple_naive (mzed_t * A, mzp_t * P, mzp_t * Q)`

PLE decomposition: $L \cdot E = P \cdot A$.

Modifies A in place to store lower triangular L below (and on) the main diagonal and E – an echelon form of A – above the main diagonal (pivots are stored in Q). P and Q are updated with row and column permutations respectively.

This function uses naive cubic PLE decomposition depending on the size of the underlying field.

Parameters

A	Matrix
P	Permutation vector of length $A \rightarrow \text{nrows}$
Q	Permutation vector of length $A \rightarrow \text{ncols}$

See also

[mzed_ple_newton_john\(\)](#) [mzed_ple\(\)](#)

3.9 Echelon forms

Macros

- `#define mzd_slice_echelonize mzd_slice_echelonize_ple`
Compute row echelon forms.

Functions

- `rci_t mzd_slice_echelonize_ple (mzd_slice_t *A, int full)`
Compute row echelon forms using PLE decomposition.
- `static rci_t mzed_echelonize_ple (mzed_t *A, int full)`
Compute row echelon forms using PLE decomposition.
- `rci_t mzed_echelonize (mzed_t *A, int full)`
Compute row echelon forms.
- `rci_t mzed_echelonize_naive (mzed_t *A, int full)`
Gaussian elimination.
- `rci_t mzed_echelonize_newton_john (mzed_t *A, int full)`
Reduce matrix A to row echelon form using Gauss-Newton-John elimination.

3.9.1 Detailed Description

3.9.2 Macro Definition Documentation

3.9.2.1 `#define mzd_slice_echelonize mzd_slice_echelonize_ple`

Compute row echelon forms.

Compute the (reduced) row echelon form of the matrix A. If full=0, then return the reduced REF.

Parameters

<i>A</i>	Matrix
<i>full</i>	REF or RREF.

3.9.3 Function Documentation

3.9.3.1 `rci_t mzd_slice_echelonize_ple (mzd_slice_t * A, int full)`

Compute row echelon forms using PLE decomposition.

Compute the (reduced) row echelon form of the matrix A. If full=0, then return the reduced row echelon. This function reduces echelon forms to PLE (or PLUQ) decomposition.

Parameters

<i>A</i>	Matrix
<i>full</i>	REF or RREF.

3.9.3.2 `rci_t mzed_echelonize (mzed_t * A, int full)`

Compute row echelon forms.

Compute the (reduced) row echelon form of the matrix A. If full=0, then return the reduced row echelon form.

Parameters

<i>A</i>	Matrix
<i>full</i>	REF or RREF.

3.9.3.3 `rci_t mzed_echelonize_naive (mzed_t * A, int full)`

Gaussian elimination.

Perform Gaussian elimination on the matrix A. If full=0, then it will do triangular style elimination, and if full=1, it will do Gauss-Jordan style, or full elimination.

Parameters

<i>A</i>	Matrix
<i>full</i>	Gauss-Jordan style or upper unit-triangular form only.

3.9.3.4 `rci_t mzed_echelonize_newton_john (mzed_t * A, int full)`

Reduce matrix A to row echelon form using Gauss-Newton-John elimination.

Parameters

<i>A</i>	Matrix to be reduced.
<i>full</i>	If set to true, the reduced row echelon form will be computed.

Todo we don't really compute the upper triangular form yet, we need to implement `_mzed_gauss_submatrix()` and a better table creation for that.

3.9.3.5 `static rci_t mzed_echelonize_ple (mzed_t * A, int full) [inline],[static]`

Compute row echelon forms using PLE decomposition.

Compute the (reduced) row echelon form of the matrix A. If full=0, then return the reduced REF. This function reduces echelon forms to PLE (or PLUQ) decomposition.

Parameters

<i>A</i>	Matrix
<i>full</i>	REF or RREF.

Note

This function converts A to bitslice representation and back. Hence, it uses more memory than using [mzed_echelonize_newton_john\(\)](#) or [mzd_slice_echelonize_ple\(\)](#)

3.10 Triangular matrices

Functions

- void `mzed_trsm_lower_left_newton_john` (const `mzed_t` *L, `mzed_t` *B)
 $B = L^{-1} \cdot B$ using Newton-John tables.
- void `mzd_slice_trsm_lower_left_newton_john` (const `mzd_slice_t` *L, `mzd_slice_t` *B)
 $B = L^{-1} \cdot B$ using Newton-John tables.
- void `mzed_trsm_upper_left_newton_john` (const `mzed_t` *U, `mzed_t` *B)
 $B = U^{-1} \cdot B$ using Newton-John tables.
- void `mzd_slice_trsm_upper_left_newton_john` (const `mzd_slice_t` *U, `mzd_slice_t` *B)
 $B = U^{-1} \cdot B$ using Newton-John tables.
- void `_mzed_trsm_upper_left` (const `mzed_t` *U, `mzed_t` *B, const `rci_t` cutoff)
 $B = U^{-1} \cdot B$
- void `mzed_trsm_upper_left_naive` (const `mzed_t` *U, `mzed_t` *B)
 $B = U^{-1} \cdot B$
- static void `mzed_trsm_upper_left` (const `mzed_t` *U, `mzed_t` *B)
 $B = U^{-1} \cdot B$
- void `_mzd_slice_trsm_upper_left` (const `mzd_slice_t` *U, `mzd_slice_t` *B, const `rci_t` cutoff)
 $B = U^{-1} \cdot B$
- void `mzd_slice_trsm_upper_left_naive` (const `mzd_slice_t` *U, `mzd_slice_t` *B)
 $B = U^{-1} \cdot B$
- static void `mzd_slice_trsm_upper_left` (const `mzd_slice_t` *U, `mzd_slice_t` *B)
 $B = U^{-1} \cdot B$
- void `_mzed_trsm_lower_left` (const `mzed_t` *L, `mzed_t` *B, const `rci_t` cutoff)
 $B = L^{-1} \cdot B$
- void `mzed_trsm_lower_left_naive` (const `mzed_t` *L, `mzed_t` *B)
 $B = L^{-1} \cdot B$
- static void `mzed_trsm_lower_left` (const `mzed_t` *L, `mzed_t` *B)
 $B = L^{-1} \cdot B$
- void `_mzd_slice_trsm_lower_left` (const `mzd_slice_t` *L, `mzd_slice_t` *B, const `rci_t` cutoff)
 $B = L^{-1} \cdot B$
- void `mzd_slice_trsm_lower_left_naive` (const `mzd_slice_t` *L, `mzd_slice_t` *B)
 $B = L^{-1} \cdot B$
- static void `mzd_slice_trsm_lower_left` (const `mzd_slice_t` *L, `mzd_slice_t` *B)
 $B = L^{-1} \cdot B$

3.10.1 Detailed Description

3.10.2 Function Documentation

3.10.2.1 void `_mzd_slice_trsm_lower_left` (const `mzd_slice_t` * L, `mzd_slice_t` * B, const `rci_t` cutoff)

$$B = L^{-1} \cdot B$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.
$cutoff$	Crossover dimension to base case.

3.10.2.2 `void _mzd_slice_trsm_upper_left (const mzd_slice_t * U , mzd_slice_t * B , const rci_t $cutoff$)`

$$B = U^{-1} \cdot B$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.
$cutoff$	Crossover dimension to base case.

3.10.2.3 `void _mzed_trsm_lower_left (const mzed_t * L , mzed_t * B , const rci_t $cutoff$)`

$$B = L^{-1} \cdot B$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.
$cutoff$	Crossover dimension to base case.

3.10.2.4 `void _mzed_trsm_upper_left (const mzed_t * U , mzed_t * B , const rci_t $cutoff$)`

$$B = U^{-1} \cdot B$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.
$cutoff$	Crossover dimension to base case.

3.10.2.5 `static void mzd_slice_trsm_lower_left (const mzd_slice_t * L , mzd_slice_t * B) [inline],[static]`

$$B = L^{-1} \cdot B$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.6 `void mzd_slice_trsm_lower_left_naive (const mzd_slice_t * L , mzd_slice_t * B)`

$$B = L^{-1} \cdot B$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.7 `void mzd_slice_trsm_lower_left_newton_john (const mzd_slice_t * L , mzd_slice_t * B)`

$$B = L^{-1} \cdot B \text{ using Newton-John tables.}$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.8 `static void mzd_slice_trsm_upper_left (const mzd_slice_t * U , mzd_slice_t * B)` `[inline], [static]`

$$B = U^{-1} \cdot B$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.9 `void mzd_slice_trsm_upper_left_naive (const mzd_slice_t * U , mzd_slice_t * B)`

$$B = U^{-1} \cdot B$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.10 `void mzd_slice_trsm_upper_left_newton_john (const mzd_slice_t * U , mzd_slice_t * B)`

$$B = U^{-1} \cdot B \text{ using Newton-John tables.}$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.11 `static void mzed_trsm_lower_left (const mzed_t * L , mzed_t * B)` `[inline], [static]`

$$B = L^{-1} \cdot B$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.12 `void mzed_trsm_lower_left_naive (const mzed_t * L , mzed_t * B)`

$$B = L^{-1} \cdot B$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.13 `void mzed_trsm_lower_left_newton_john (const mzed_t * L , mzed_t * B)`

$$B = L^{-1} \cdot B \text{ using Newton-John tables.}$$

Parameters

L	Lower-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.14 `static void mzed_trsm_upper_left (const mzed_t * U , mzed_t * B)` `[inline],[static]`

$$B = U^{-1} \cdot B$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.15 `void mzed_trsm_upper_left_naive (const mzed_t * U , mzed_t * B)`

$$B = U^{-1} \cdot B$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.

3.10.2.16 `void mzed_trsm_upper_left_newton_john (const mzed_t * U , mzed_t * B)`

$$B = U^{-1} \cdot B \text{ using Newton-John tables.}$$

Parameters

U	Upper-triangular matrix (other entries are ignored).
B	Matrix.

4 Data Structure Documentation

4.1 blm_t Struct Reference

Bilinear Maps on Matrices over GF(2).

```
#include <blm.h>
```

Data Fields

- mzd_t * [H](#)
- djb_t * [h](#)
- mzd_t * [F](#)
- djb_t * [f](#)
- mzd_t * [G](#)
- djb_t * [g](#)

4.1.1 Detailed Description

Bilinear Maps on Matrices over GF(2).

Encodes the bilinear map $H((F*A) \times (G*B))$ where A,B are vectors of mzd_t, "*" is matrix-vector multiplication and "x" is pointwise multiplication.

If a DJB map is not NULL, it will be used instead its matrix representant.

4.1.2 Field Documentation

4.1.2.1 mzd_t* blm_t::F

lineatr map on A

4.1.2.2 djb_t* blm_t::f

lineatr map on N (DJB encoding)

4.1.2.3 mzd_t* blm_t::G

lineatr map on B

4.1.2.4 djb_t* blm_t::g

lineatr map on B (DJB encoding)

4.1.2.5 mzd_t* blm_t::H

final linear map H

4.1.2.6 djb_t* blm_t::h

final linear map H (DJB encoding)

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

- [blm.h](#)

4.2 gf2e_struct Struct Reference

\mathbb{F}_{2^e}

```
#include <gf2e.h>
```

Data Fields

- [deg_t degree](#)
- [word minpoly](#)
- [word * pow_gen](#)
- [word * red](#)
- [word ** _mul](#)
- [word\(* inv\)\(const \[gf2e\]\(#\) *ff, const word a\)](#)
- [word\(* mul\)\(const \[gf2e\]\(#\) *ff, const word a, const word b\)](#)

4.2.1 Detailed Description

\mathbb{F}_{2^e}

Examples:

[tests/test_multiplication.c](#).

4.2.2 Field Documentation

4.2.2.1 [word** gf2e_struct::_mul](#)

`mul[a][b]` holds $a \cdot b$ for small fields.

4.2.2.2 [deg_t gf2e_struct::degree](#)

The degree e .

Examples:

[tests/test_multiplication.c](#).

4.2.2.3 [word\(* gf2e_struct::inv\)\(const \[gf2e\]\(#\) *ff, const word a\)](#)

implements a^{-1} for a in \mathbb{F}_{2^e}

4.2.2.4 [word gf2e_struct::minpoly](#)

Irreducible polynomial of degree e .

Examples:

[tests/test_multiplication.c](#).

4.2.2.5 [word\(* gf2e_struct::mul\)\(const \[gf2e\]\(#\) *ff, const word a, const word b\)](#)

implements $a \cdot b$ for a in \mathbb{F}_{2^e} .

4.2.2.6 word* gf2e_struct::pow_gen

pow_gen[i] holds $a^i / \langle f \rangle$ for a a generator of this field.

4.2.2.7 word* gf2e_struct::red

red[i] holds precomputed reducers for the minpoly.

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

- [gf2e.h](#)

4.3 mzd_poly_t Struct Reference

will be the data type for matrices over $\mathbb{F}_2[x]$ in the future

```
#include <mzd_poly.h>
```

Data Fields

- mzd_t** x
- rci_t nrows
- rci_t ncols
- deg_t depth

4.3.1 Detailed Description

will be the data type for matrices over $\mathbb{F}_2[x]$ in the future

4.3.2 Field Documentation

4.3.2.1 deg_t mzd_poly_t::depth

Degree +1

4.3.2.2 rci_t mzd_poly_t::ncols

Number of columns.

4.3.2.3 rci_t mzd_poly_t::nrows

Number of rows.

4.3.2.4 mzd_t** mzd_poly_t::x

Coefficients.

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

- [mzd_poly.h](#)

4.4 mzd_slice_t Struct Reference

Dense matrices over \mathbb{F}_{2^e} represented as slices of matrices over \mathbb{F}_2 .

```
#include <mzd_slice.h>
```

Data Fields

- `mzd_t * x[16]`
- `rci_t nrows`
- `rci_t ncols`
- `unsigned int depth`
- `const gf2e * finite_field`

4.4.1 Detailed Description

Dense matrices over \mathbb{F}_{2^e} represented as slices of matrices over \mathbb{F}_2 .

This is one of two fundamental data types of this library, the other being `mzed_t`. For large matrices ($m \times n \times e > L2$) it is advisable to use this data type because multiplication is faster in this representation. Hence, compared to `mzed_t` one saves the time to convert between representations and - more importantly - memory.

Examples:

[tests/test_multiplication.c](#).

4.4.2 Field Documentation

4.4.2.1 unsigned int mzd_slice_t::depth

Number of slices *

Note

This value may be greater than `finite_field->degree` in some situations

4.4.2.2 const gf2e* mzd_slice_t::finite_field

A finite field \mathbb{F}_{2^e} .

4.4.2.3 rci_t mzd_slice_t::ncols

Number of columns.

4.4.2.4 rci_t mzd_slice_t::nrows

Number of rows.

4.4.2.5 mzd_t* mzd_slice_t::x[16]

`mzd_slice_t::x[e][i,j]` is the e -th bit of the entry $A[i,j]$.

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

- [mzd_slice.h](#)

4.5 mzed_t Struct Reference

Dense matrices over \mathbb{F}_{2^e} represented as packed matrices.

```
#include <mzed.h>
```

Data Fields

- mzd_t * [x](#)
- const [gf2e](#) * [finite_field](#)
- rci_t [nrows](#)
- rci_t [ncols](#)
- wi_t [w](#)

4.5.1 Detailed Description

Dense matrices over \mathbb{F}_{2^e} represented as packed matrices.

Examples:

[tests/test_multiplication.c](#).

4.5.2 Field Documentation

4.5.2.1 const gf2e* mzed_t::finite_field

A finite field \mathbb{F}_{2^e} .

4.5.2.2 rci_t mzed_t::ncols

Number of columns.

4.5.2.3 rci_t mzed_t::nrows

Number of rows.

4.5.2.4 wi_t mzed_t::w

The internal width of elements (must divide 64).

4.5.2.5 mzd_t* mzed_t::x

$m \times n$ matrices over \mathbb{F}_{2^e} are represented as $m \times (en)$ matrices over \mathbb{F}_2 .

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

- [mzed.h](#)

4.6 njt_mzed_t Struct Reference

Newton-John table.

```
#include <newton_john.h>
```

Data Fields

- `rci_t * L`
- `mzed_t * M`
- `mzed_t * T`

4.6.1 Detailed Description

Newton-John table.

4.6.2 Field Documentation

4.6.2.1 `rci_t* njt_mzed_t::L`

A map such that `L[a]` points to the row where the first entry is `a`.

4.6.2.2 `mzed_t* njt_mzed_t::M`

Table of length e with multiples of the input s.t. a^i is the first entry of row i .

4.6.2.3 `mzed_t* njt_mzed_t::T`

Actual table of length 2^e of all linear combinations of `T`.

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

- `newton_john.h`

5 File Documentation

5.1 `blm.h` File Reference

Bilinear Maps on Matrices over $GF(2)$.

```
#include <m4ri/m4ri.h>
#include "m4rie/gf2e.h"
```

Data Structures

- struct `blm_t`

Bilinear Maps on Matrices over $GF(2)$.

Macros

- `#define M4RIE_CRT_LEN (M4RIE_MAX_DEGREE + 1)`

We consider at most polynomials of degree `M4RIE_MAX_DEGREE` in CRT.

Functions

- static int `blm_cost_crt` (const int p[M4RIE_CRT_LEN])
- int * `crt_init` (const `deg_t` f_len, const `deg_t` g_len)
- `blm_t` * `blm_init_crt` (const `gf2e` *ff, const `deg_t` f_ncols, const `deg_t` g_ncols, const int *p, int djb)
- `blm_t` * `_blm_finish_polymult` (const `gf2e` *ff, `blm_t` *f)
- void `blm_free` (`blm_t` *f)
- `blm_t` * `_blm_djb_compile` (`blm_t` *f)
Compile DJB map for f.
- void `_mzd_ptr_apply_blm_mzd` (`mzd_t` **X, const `mzd_t` **A, const `mzd_t` **B, const `blm_t` *f)
Apply f (stored as a matrix) on A and B, writing to X.
- void `_mzd_ptr_apply_blm_djb` (`mzd_t` **X, const `mzd_t` **A, const `mzd_t` **B, const `blm_t` *f)
Apply f (stored as a DJB map) on A and B, writing to X.
- static void `_mzd_ptr_apply_blm` (`mzd_t` **X, const `mzd_t` **A, const `mzd_t` **B, const `blm_t` *f)
Apply f on A and B, writing to X.

Variables

- const int `costs` [17]

5.1.1 Detailed Description

Bilinear Maps on Matrices over GF(2).

This is used to realise `mzd_poly_t` multiplication.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.1.2 Function Documentation

5.1.2.1 `blm_t* _blm_djb_compile (blm_t * f)`

Compile DJB map for f.

Parameters

<code>f</code>	Bilinear map
----------------	--------------

5.1.2.2 `blm_t* _blm_finish_polymult (const gf2e * ff, blm_t * f)`

Given F and G compute H.

Parameters

<code>ff</code>	finite field for modular reduction
<code>f</code>	Bilinear Map with F and G already computed.

5.1.2.3 `static void _mzd_ptr_apply_blm (mzd_t ** X, const mzd_t ** A, const mzd_t ** B, const blm_t * f) [inline], [static]`

Apply f on A and B, writing to X.

Parameters

X	Array of matrices
A	Array of matrices
B	Array of matrices
f	Bilinear map

5.1.2.4 `void _mzd_ptr_apply_blm_djb (mzd_t ** X , const mzd_t ** A , const mzd_t ** B , const blm_t * f)`

Apply f (stored as a DJB map) on A and B , writing to X .

Parameters

X	Array of matrices
A	Array of matrices
B	Array of matrices
f	Bilinear map

5.1.2.5 `void _mzd_ptr_apply_blm_mzd (mzd_t ** X , const mzd_t ** A , const mzd_t ** B , const blm_t * f)`

Apply f (stored as a matrix) on A and B , writing to X .

Parameters

X	Array of matrices
A	Array of matrices
B	Array of matrices
f	Bilinear map

5.1.2.6 `static int blm_cost_crt (const int $p[M4RIE_CRT_LEN]$) [inline],[static]`

Return the multiplication cost of the multiplication scheme p

5.1.2.7 `void blm_free (blm_t * f)`

Free bilinear map f .

5.1.2.8 `blm_t* blm_init_crt (const gf2e * ff , const deg_t f_ncols , const deg_t g_ncols , const int * p , int djb)`

Compute H, F, G such that $\text{vec}(c) = H \cdot (F \cdot \text{vec}(a) \times G \cdot \text{vec}(b))$ with $\text{poly}(c) = \text{poly}(a) \cdot \text{poly}(b)$, $\deg(\text{poly}(a)) = a_ncols - 1$, $\deg(\text{poly}(b)) = b_ncols - 1$ and " x " being pointwise multiplication

This is realised by a multi-modular computation modulo the primes up to degree \deg (which may not be irreducible polynomials, but merely co-prime). 1) We construct maps F, G which combine modular reduction to degree d and the linear map required for multiplying modulo a polynomial of degree d .

1.1) We deal with $(x + \infty)^\omega$ first

1.2) We deal with regular polynomial which are co-prime

the minimal polynomial is a square

the minimal polynomial is a fourth power

the minimal polynomial is an eighth power

2) We solve for H as we know $\text{poly}(c)$ and $(F \cdot \text{vec}(a) \times G \cdot \text{vec}(b))$. We pick points $\text{poly}(a) = x^v$, $\text{poly}(b) = x^w$ (hence: $\text{poly}(c) = x^{(v+w)}$).

3) We compile DJB maps if asked for.

5.1.2.9 `int* crt_init (const deg_t f_len, const deg_t g_len)`

Find a list of co-prime polynomials p_i such that $\deg(\text{prod}(p_i)) \geq f_len * g_len - 1$.

We store the number of polynomials of degree d in $p[d]$. We store the degree w of $(x - \infty)^w$ in $p[0]$.

5.1.3 Variable Documentation

5.1.3.1 `const int costs[17]`

$\text{costs}[i]$ = cost of multiplying two polynomials of length i over \mathbb{F}_2 .

5.2 conversion.h File Reference

Conversion between `mzed_t` and `mzd_slice_t`.

```
#include <m4ri/m4ri.h>
#include <m4rie/mzed.h>
#include <m4rie/mzd_slice.h>
```

Functions

- `mzed_t * mzed_cling (mzed_t *A, const mzd_slice_t *Z)`
Pack a bitslice matrix into a packed representation.
- `mzd_slice_t * mzed_slice (mzd_slice_t *A, const mzed_t *Z)`
Unpack the matrix Z into bitslice representation.
- `mzd_slice_t * _mzed_slice2 (mzd_slice_t *A, const mzed_t *Z)`
Unpack the matrix Z over $GF(2^2)$ into bitslice representation.
- `mzd_slice_t * _mzed_slice4 (mzd_slice_t *A, const mzed_t *Z)`
Unpack the matrix Z over \mathbb{F}_{2^e} into bitslice representation.
- `mzd_slice_t * _mzed_slice8 (mzd_slice_t *A, const mzed_t *Z)`
Unpack the matrix Z over \mathbb{F}_{2^e} into bitslice representation.
- `mzd_slice_t * _mzed_slice16 (mzd_slice_t *A, const mzed_t *Z)`
Unpack the matrix Z over \mathbb{F}_{2^e} into bitslice representation.
- `mzed_t * _mzed_cling2 (mzed_t *A, const mzd_slice_t *Z)`
Pack a bitslice matrix into a classical representation over $GF(2^2)$.
- `mzed_t * _mzed_cling4 (mzed_t *A, const mzd_slice_t *Z)`
Pack a bitslice matrix into a classical representation over \mathbb{F}_{2^e} for $2 < e \leq 4$.
- `mzed_t * _mzed_cling8 (mzed_t *A, const mzd_slice_t *Z)`
Pack a bitslice matrix into a classical representation over \mathbb{F}_{2^e} for $4 < e \leq 8$.
- `mzed_t * _mzed_cling16 (mzed_t *A, const mzd_slice_t *Z)`
Pack a bitslice matrix into a classical representation over \mathbb{F}_{2^e} for $8 < e \leq 16$.
- `static mzed_t * _mzed_addmul_karatsuba (mzed_t *C, const mzed_t *A, const mzed_t *B)`
Compute $C += A*B$ using Karatsuba multiplication of polynomials over $GF(2)$.
- `static mzed_t * mzed_mul_karatsuba (mzed_t *C, const mzed_t *A, const mzed_t *B)`
Compute $C = A*B$.

- static `mzed_t * mzed_addmul_karatsuba (mzed_t *C, const mzed_t *A, const mzed_t *B)`
*Compute $C += A*B$.*
- static `mzed_t * _mzed_addmul_blm (mzed_t *C, const mzed_t *A, const mzed_t *B)`
*Compute $C += A*B$ using Bilinear Maps over $GF(2)$.*
- static `mzed_t * mzed_mul_blm (mzed_t *C, const mzed_t *A, const mzed_t *B)`
*Compute $C = A*B$.*
- static `mzed_t * mzed_addmul_blm (mzed_t *C, const mzed_t *A, const mzed_t *B)`
*Compute $C += A*B$.*
- static void `mzd_slice_rescale_row (mzd_slice_t *A, rci_t r, rci_t c, word x)`
Rescale the row r in A by X starting c .

5.2.1 Detailed Description

Conversion between `mzed_t` and `mzd_slice_t`.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.2.2 Function Documentation

5.2.2.1 static `mzed_t* _mzed_addmul_blm (mzed_t * C, const mzed_t * A, const mzed_t * B)` [inline],
[static]

Compute $C += A*B$ using Bilinear Maps over $GF(2)$.

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

See also

[_mzd_slice_addmul_blm](#)

5.2.2.2 static `mzed_t* _mzed_addmul_karatsuba (mzed_t * C, const mzed_t * A, const mzed_t * B)` [inline],
[static]

Compute $C += A*B$ using Karatsuba multiplication of polynomials over $GF(2)$.

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

See also

[_mzd_slice_addmul_karatsuba](#)

5.2.2.3 `mzed_t* _mzed_cling16 (mzed_t * A, const mzd_slice_t * Z)`

Pack a bitslice matrix into a classical representation over \mathbb{F}_{2^e} for $8 < e \leq 16$.

Parameters

A	Matrix over \mathbb{F}_{2^e} , must be zero
Z	Bitslice matrix over \mathbb{F}_{2^e}

5.2.2.4 `mzed_t* _mzed_cling2 (mzed_t * A, const mzd_slice_t * Z)`

Pack a bitslice matrix into a classical representation over $\text{GF}(2^2)$.

Elements in $\text{GF}(2^2)$ can be represented as $c_1*a + c_0$ where a is a root of $x^2 + x + 1$. $A1$ contains the coefficients for c_1 while $A0$ contains the coefficients for c_0 .

Parameters

A	Matrix over $\text{GF}(2^2)$, must be zero
Z	Bitslice matrix over $\text{GF}(2^2)$

5.2.2.5 `mzed_t* _mzed_cling4 (mzed_t * A, const mzd_slice_t * Z)`

Pack a bitslice matrix into a classical representation over \mathbb{F}_{2^e} for $2 < e \leq 4$.

Parameters

A	Matrix over \mathbb{F}_{2^e} , must be zero
Z	Bitslice matrix over \mathbb{F}_{2^e}

5.2.2.6 `mzed_t* _mzed_cling8 (mzed_t * A, const mzd_slice_t * Z)`

Pack a bitslice matrix into a classical representation over \mathbb{F}_{2^e} for $4 < e \leq 8$.

Parameters

A	Matrix over \mathbb{F}_{2^e} , must be zero
Z	Bitslice matrix over \mathbb{F}_{2^e}

5.2.2.7 `mzd_slice_t* _mzed_slice16 (mzd_slice_t * A, const mzed_t * Z)`

Unpack the matrix Z over \mathbb{F}_{2^e} into bitslice representation.

Parameters

A	Zero bitslice matrix over \mathbb{F}_{2^e}
Z	Matrix over \mathbb{F}_{2^e}

5.2.2.8 `mzd_slice_t* _mzed_slice2 (mzd_slice_t * A, const mzed_t * Z)`

Unpack the matrix Z over $\text{GF}(2^2)$ into bitslice representation.

Elements in $\text{GF}(2^2)$ can be represented as $x*a + y$ where a is a root of $x^2 + x + 1$. $A0$ contains the coefficients for x while $A1$ contains the coefficients for y .

Parameters

A	Zero bitslice matrix over $\text{GF}(2^2)$
-----	--

Z	Matrix over $\text{GF}(2^2)$
-----	------------------------------

5.2.2.9 `mzd_slice_t* _mzed_slice4 (mzd_slice_t * A, const mzed_t * Z)`

Unpack the matrix Z over \mathbb{F}_{2^e} into bitslice representation.

Parameters

A	Zero bitslice matrix over \mathbb{F}_{2^e}
Z	Matrix over \mathbb{F}_{2^e}

5.2.2.10 `mzd_slice_t* _mzed_slice8 (mzd_slice_t * A, const mzed_t * Z)`

Unpack the matrix Z over \mathbb{F}_{2^e} into bitslice representation.

Parameters

A	Zero bitslice matrix over \mathbb{F}_{2^e}
Z	Matrix over \mathbb{F}_{2^e}

5.2.2.11 `static mzed_t* mzed_addmul_blm (mzed_t * C, const mzed_t * A, const mzed_t * B) [inline], [static]`

Compute $C += A*B$.

Parameters

C	Preallocated return matrix.
A	Input matrix A.
B	Input matrix B.

Examples:

[tests/test_multiplication.c](#).

5.2.2.12 `static mzed_t* mzed_addmul_karatsuba (mzed_t * C, const mzed_t * A, const mzed_t * B) [inline], [static]`

Compute $C += A*B$.

Parameters

C	Preallocated return matrix.
A	Input matrix A.
B	Input matrix B.

Examples:

[tests/test_multiplication.c](#).

5.2.2.13 `static mzed_t* mzed_mul_blm (mzed_t * C, const mzed_t * A, const mzed_t * B) [inline], [static]`

Compute $C = A*B$.

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

See also

[_mzd_slice_mul_blm](#)

Examples:

[tests/test_multiplication.c.](#)

```
5.2.2.14 static mzed_t* mzed_mul_karatsuba ( mzed_t * C, const mzed_t * A, const mzed_t * B ) [inline],
[static]
```

Compute $C = A * B$.

Parameters

<i>C</i>	Preallocated return matrix, may be NULL for automatic creation.
<i>A</i>	Input matrix A.
<i>B</i>	Input matrix B.

See also

[_mzd_slice_mul_karatsuba](#)

Examples:

[tests/test_multiplication.c.](#)

5.3 echelonform.h File Reference

Echelon forms.

```
#include <m4rie/mzed.h>
#include <m4rie/mzd_slice.h>
#include <m4rie/conversion.h>
```

Macros

- `#define mzd_slice_echelonize mzd_slice_echelonize_ple`
Compute row echelon forms.

Functions

- `rci_t mzd_slice_echelonize_ple (mzd_slice_t *A, int full)`
Compute row echelon forms using PLE decomposition.
- `static rci_t mzed_echelonize_ple (mzed_t *A, int full)`
Compute row echelon forms using PLE decomposition.
- `rci_t mzed_echelonize (mzed_t *A, int full)`
Compute row echelon forms.

5.3.1 Detailed Description

Echelon forms.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.4 gf2e.h File Reference

\mathbb{F}_{2^e}

```
#include <m4ri/m4ri.h>
#include <m4rie/gf2x.h>
```

Data Structures

- struct [gf2e_struct](#)

\mathbb{F}_{2^e}

Macros

- #define [M4RIE_MAX_DEGREE](#) 16
maximal supported degree

Typedefs

- typedef struct [gf2e_struct](#) [gf2e](#)

\mathbb{F}_{2^e}

Functions

- [gf2e * gf2e_init](#) (const word minpoly)
- void [gf2e_free](#) ([gf2e](#) *ff)
- static word [gf2e_inv](#) (const [gf2e](#) *ff, word a)
 $a^{(-1)} \% \text{minpoly}$
- static word [_gf2e_mul_table](#) (const [gf2e](#) *ff, const word a, const word b)
 $a*b$ in \mathbb{F}_{2^e} using a table lookups.
- static word [_gf2e_mul_arith](#) (const [gf2e](#) *ff, const word a, const word b)
 $a*b$ in \mathbb{F}_{2^e} using a [gf2x_mul\(\)](#) lookups.
- static word [gf2e_mul](#) (const [gf2e](#) *ff, const word a, const word b)
 $a*b$ in \mathbb{F}_{2^e} .
- static size_t [gf2e_degree_to_w](#) (const [gf2e](#) *ff)
- static word * [gf2e_t16_init](#) (const [gf2e](#) *ff, const word a)
- static void [gf2e_t16_free](#) (word *mul)
Free multiplication table.

Variables

- const word * [irreducible_polynomials](#) [17]
all Irreducible polynomials over GF(2) up to degree 16.

5.4.1 Detailed Description

 \mathbb{F}_{2^e}

Author

Martin Albrecht martinralbrecht@gmail.com

5.4.2 Function Documentation

5.4.2.1 static size_t gf2e_degree_to_w (const gf2e * ff) [inline],[static]

Return the width used for storing elements of ff

Parameters

<i>ff</i>	Finite field.
-----------	---------------

5.4.2.2 void gf2e_free (gf2e * ff)

Free ff

Parameters

<i>ff</i>	Finite field.
-----------	---------------

Examples:

[tests/test_multiplication.c.](#)

5.4.2.3 gf2e* gf2e_init (const word minpoly)

Create finite field from minimal polynomial

Parameters

<i>minpoly</i>	Polynomial represented as series of bits.
----------------	---

red

pow_gen: X^i

mul tables

Examples:

[tests/test_multiplication.c.](#)

5.4.2.4 static void gf2e_t16_free (word * mul) [inline],[static]

Free multiplication table.

Parameters

<i>mul</i>	Multiplication table
------------	----------------------

5.4.2.5 `static word* gf2e_t16_init (const gf2e * ff, const word a)` `[inline], [static]`

Compute all multiples by *a* of vectors fitting into 16 bits.

Parameters

<i>ff</i>	Finite field.
<i>a</i>	Finite field element.

Todo : this is a bit of overkill, we could do better

5.5 gf2x.h File Reference

\mathbb{F}_{2^x} for degrees < 64

```
#include <m4rie/m4rie.h>
```

Macros

- `#define __M4RIE_1tF(X) ~((X)-1)`

Typedefs

- `typedef int deg_t`

Functions

- `static word gf2x_mul (const word a, const word b, deg_t d)`
 $a * b$ in \mathbb{F}_{2^x} with $\deg(a)$ and $\deg(b) < d$.
- `static deg_t gf2x_deg (word a)`
 $\deg(a)$ in \mathbb{F}_{2^x} .
- `static word gf2x_div (word a, word b)`
 a / b in \mathbb{F}_{2^x} .
- `static word gf2x_mod (word a, word b)`
 $a \bmod b$ in \mathbb{F}_{2^x} .
- `static word gf2x_divmod (word a, word b, word *rem)`
 a / b and $a \bmod b$ in \mathbb{F}_{2^x} .
- `static word gf2x_invmod (word a, word b, const deg_t d)`
 $a^{-1} \bmod b$ with $\deg(a), \deg(b) \leq d$.

5.5.1 Detailed Description

\mathbb{F}_{2^x} for degrees < 64

Author

Martin Albrecht martinralbrecht@googlemail.com

Warning

Do not rely on these functions for high performance, they are not fully optimised.

5.5.2 Macro Definition Documentation**5.5.2.1 #define __M4RIE_1tF(X) ~((X)-1)**

Maps 1 to word with all ones and 0 to 0.

5.5.3 Typedef Documentation**5.5.3.1 typedef int deg_t**

degree type

5.5.4 Function Documentation**5.5.4.1 static deg_t gf2x_deg(word a) [inline], [static]**

deg(a) in \mathbb{F}_2^x .

Parameters

<i>a</i>	Polynomial of degree ≤ 64 .
----------	----------------------------------

5.6 m4rie.h File Reference

Main include file for the M4RIE library.

```
#include <m4rie/gf2e.h>
#include <m4rie/mzed.h>
#include <m4rie/newton_john.h>
#include <m4rie/echelonform.h>
#include <m4rie/strassen.h>
#include <m4rie/mzd_slice.h>
#include <m4rie/trsm.h>
#include <m4rie/ple.h>
#include <m4rie/conversion.h>
#include <m4rie/permutation.h>
#include <m4rie/mzd_poly.h>
```

5.6.1 Detailed Description

Main include file for the M4RIE library.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.7 mzd_poly.h File Reference

Matrices over $\mathbb{F}_2[x]$.

```
#include <m4ri/m4ri.h>
#include "mzd_ptr.h"
#include "gf2x.h"
#include "blm.h"
```

Data Structures

- struct `mzd_poly_t`
will be the data type for matrices over $\mathbb{F}_2[x]$ in the future

Functions

- static `mzd_poly_t * _mzd_poly_add` (`mzd_poly_t *C`, const `mzd_poly_t *A`, const `mzd_poly_t *B`, unsigned int `offset`)
 *$C += (A+B)*x^{\text{offset}}$.*
- static `mzd_poly_t * mzd_poly_add` (`mzd_poly_t *C`, const `mzd_poly_t *A`, const `mzd_poly_t *B`)
 $C += (A+B)$
- static `mzd_poly_t * mzd_poly_init` (const `deg_t d`, const `rci_t m`, const `rci_t n`)
Create a new polynomial of degree d with $m \times n$ matrices as coefficients.
- static void `mzd_poly_free` (`mzd_poly_t *A`)
Free polynomial A .
- static `mzd_poly_t * _mzd_poly_adapt_depth` (`mzd_poly_t *A`, const `deg_t new_depth`)
change depth of A to new_depth .
- static `mzd_poly_t * _mzd_poly_addmul_naive` (`mzd_poly_t *C`, const `mzd_poly_t *A`, const `mzd_poly_t *B`)
 *$C += A*B$ using naive polynomial multiplication.*
- static `mzd_poly_t * _mzd_poly_addmul_karatsubs_balanced` (`mzd_poly_t *C`, const `mzd_poly_t *A`, const `mzd_poly_t *B`)
 *$C += A*B$ using Karatsuba multiplication on balanced inputs.*
- static `mzd_poly_t * _mzd_poly_addmul_blm` (`mzd_poly_t *C`, `mzd_poly_t *A`, `mzd_poly_t *B`, const `blm_t *f`)
 *$C += A*B$ by applying the bilinear maps f , i.e. $f \mapsto H*((f \mapsto F*A) \times (f \mapsto G*B))$.*
- static `mzd_poly_t * _mzd_poly_addmul_crt` (`mzd_poly_t *C`, `mzd_poly_t *A`, `mzd_poly_t *B`)
 *$C += A*B$ using the Chinese Remainder Theorem.*
- `mzd_poly_t * _mzd_poly_addmul_ext1` (`mzd_poly_t *C`, `mzd_poly_t *A`, `mzd_poly_t *B`)
 *$C += A*B$ using arithmetic in $GF(2^{\log_2(d)})$ if C has degree d .*
- static int `mzd_poly_cmp` (`mzd_poly_t *A`, `mzd_poly_t *B`)
Return -1,0,1 if $A < B$, $A == B$ or $A > B$ respectively.
- static void `mzd_poly_randomize` (`mzd_poly_t *A`)
Fill matrix A with random elements.

5.7.1 Detailed Description

Matrices over $\mathbb{F}_2[x]$.

Warning

This code is experimental.

5.7.2 Function Documentation

5.7.2.1 static int mzd_poly_cmp (mzd_poly_t * A, mzd_poly_t * B) [inline], [static]

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

Parameters

<i>A</i>	Matrix.
<i>B</i>	Matrix.

Note

This comparison is not well defined (except for !=0) mathematically and relatively arbitrary.

5.8 mzd_slice.h File Reference

Matrices using a bitsliced representation.

```
#include <m4ri/m4ri.h>
#include <m4rie/mzd_poly.h>
#include <m4rie/mzed.h>
#include <m4rie/blm.h>
```

Data Structures

- struct [mzd_slice_t](#)
Dense matrices over \mathbb{F}_{2^e} represented as slices of matrices over \mathbb{F}_2 .

Macros

- #define [mzd_slice_sub](#) [mzd_slice_add](#)
 $C = A + B.$
- #define [_mzd_slice_sub](#) [_mzd_slice_add](#)
 $C = A + B.$

Functions

- static [mzd_slice_t](#) * [mzd_slice_init](#) (const [gf2e](#) *ff, const rci_t m, const rci_t n)
Create a new matrix of dimension $m \times n$ over ff.
- void [mzd_slice_set_ui](#) ([mzd_slice_t](#) *A, word value)
Return diagonal matrix with value on the diagonal.

- static `mzd_slice_t * _mzd_slice_adapt_depth (mzd_slice_t *A, const unsigned int new_depth)`
Extend or truncate the depth of A to depth new_depth.
- static void `mzd_slice_free (mzd_slice_t *A)`
Free a matrix created with `mzd_slice_init()`.
- static `mzd_slice_t * mzd_slice_copy (mzd_slice_t *B, const mzd_slice_t *A)`
copy A to B
- static word `mzd_slice_read_elem (const mzd_slice_t *A, const rci_t row, const rci_t col)`
Get the element at position (row,col) from the matrix A.
- static void `mzd_slice_add_elem (mzd_slice_t *A, const rci_t row, const rci_t col, word elem)`
At the element elem to the element at position (row,col) in the matrix A.
- static void `mzd_slice_write_elem (mzd_slice_t *A, const rci_t row, const rci_t col, word elem)`
Write the element elem to the position (row,col) in the matrix A.
- static int `mzd_slice_cmp (mzd_slice_t *A, mzd_slice_t *B)`
Return -1,0,1 if if $A < B$, $A == B$ or $A > B$ respectively.
- static int `mzd_slice_is_zero (const mzd_slice_t *A)`
Zero test for matrix.
- static void `mzd_slice_row_swap (mzd_slice_t *A, const rci_t rowa, const rci_t rowb)`
Swap the two rows rowa and rowb.
- static void `mzd_slice_copy_row (mzd_slice_t *B, size_t i, const mzd_slice_t *A, size_t j)`
copy row j from A to row i from B.
- static void `mzd_slice_col_swap (mzd_slice_t *A, const rci_t cola, const rci_t colb)`
Swap the two columns cola and colb.
- static void `mzd_slice_col_swap_in_rows (mzd_slice_t *A, const rci_t cola, const rci_t colb, const rci_t start_row, rci_t stop_row)`
Swap the two columns cola and colb but only between start_row and stop_row.
- static void `mzd_slice_row_add (mzd_slice_t *A, const rci_t sourcerow, const rci_t destrow)`
Add the rows sourcerow and destrow and stores the total in the row destrow.
- void `mzd_slice_print (const mzd_slice_t *A)`
Print a matrix to stdout.
- static void `_mzd_slice_compress_l (mzd_slice_t *A, const rci_t r1, const rci_t n1, const rci_t r2)`
Move the submatrix L of rank r2 starting at column n1 to the left to column r1.
- static `mzd_slice_t * mzd_slice_concat (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
Concatenate B to A and write the result to C.
- static `mzd_slice_t * mzd_slice_stack (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
Stack A on top of B and write the result to C.
- static `mzd_slice_t * mzd_slice_submatrix (mzd_slice_t *S, const mzd_slice_t *A, const size_t lowr, const size_t lowc, const size_t highr, const size_t highc)`
Copy a submatrix.
- static `mzd_slice_t * mzd_slice_init_window (const mzd_slice_t *A, 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_slice_free_window (mzd_slice_t *A)`
Free a matrix window created with `mzd_slice_init_window()`.
- static `mzd_slice_t * _mzd_slice_add (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = A + B.$
- static `mzd_slice_t * mzd_slice_add (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = A + B.$

- `mzd_slice_t * _mzd_slice_addmul_naive (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = A \cdot B$ using quadratic polynomial multiplication with matrix coefficients.
- `static mzd_slice_t * _mzd_slice_addmul_karatsuba (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = C + A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .
- `static mzd_slice_t * mzd_slice_mul_karatsuba (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .
- `static mzd_slice_t * mzd_slice_addmul_karatsuba (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = C + A \cdot B$ using Karatsuba multiplication of polynomials over matrices over \mathbb{F}_2 .
- `static mzd_slice_t * _mzd_slice_mul_blm (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B, blm_t *f)`
 $C = A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .
- `static mzd_slice_t * mzd_slice_mul_blm (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B, blm_t *f)`
 $C = A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .
- `static mzd_slice_t * mzd_slice_addmul_blm (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B, blm_t *f)`
 $C = C + A \cdot B$ using bilinear maps over matrices over \mathbb{F}_2 .
- `mzd_slice_t * mzd_slice_mul_scalar (mzd_slice_t *C, const word a, const mzd_slice_t *B)`
 $C = a \cdot B$.
- `mzd_slice_t * mzd_slice_addmul_scalar (mzd_slice_t *C, const word a, const mzd_slice_t *B)`
 $C += a \cdot B$.
- `static mzd_slice_t * mzd_slice_mul (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = A \cdot B$.
- `static mzd_slice_t * mzd_slice_addmul (mzd_slice_t *C, const mzd_slice_t *A, const mzd_slice_t *B)`
 $C = C + A \cdot B$.
- `static void mzd_slice_randomize (mzd_slice_t *A)`
Fill matrix A with random elements.

5.8.1 Detailed Description

Matrices using a bitsliced representation.

Matrices over \mathbb{F}_{2^e} can be represented as polynomials with matrix coefficients where the matrices are in \mathbb{F}_2 .

In this file, matrices over \mathbb{F}_{2^e} are implemented as e slices of matrices over \mathbb{F}_2 where each slice holds the coefficients of one degree when viewing elements of \mathbb{F}_{2^e} as polynomials over \mathbb{F}_2 .

Author

Martin Albrecht martinralbrecht@googlemail.com

5.8.2 Function Documentation

5.8.2.1 `static mzd_slice_t* _mzd_slice_adapt_depth (mzd_slice_t * A, const unsigned int new_depth) [inline], [static]`

Extend or truncate the depth of A to depth new_depth.

We may think of `mzd_slice_t` as polynomials over matrices over \mathbb{F}_2 . This function then truncates/extends these polynomials to degree new_depth-1. In case of extension, all newly created coefficients are zero, hence the mathematical content of A is not changed. In case of truncation higher degree terms are simply deleted and A's mathematical content modified.

Parameters

<i>A</i>	Matrix, modified in place.
<i>new_depth</i>	Integer \geq <code>mzd_slice_t::finite_field::degree</code> .

5.8.2.2 `static void _mzd_slice_compress_l (mzd_slice_t * A, const rci_t r1, const rci_t n1, const rci_t r2) [inline], [static]`

Move the submatrix L of rank r2 starting at column n1 to the left to column r1.

Parameters

<i>A</i>	Matrix
<i>r1</i>	Integer $<$ n1
<i>n1</i>	Integer $>$ r1
<i>r2</i>	Integer \leq A->ncols - n1

5.8.2.3 `static int mzd_slice_cmp (mzd_slice_t * A, mzd_slice_t * B) [inline], [static]`

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

Parameters

<i>A</i>	Matrix.
<i>B</i>	Matrix.

Note

This comparison is not well defined (except for $k=0$) mathematically and relatively arbitrary since elements of $G \leftarrow F(2^k)$ don't have an ordering.

5.8.2.4 `static void mzd_slice_col_swap_in_rows (mzd_slice_t * A, const rci_t cola, const rci_t colb, const rci_t start_row, rci_t stop_row) [inline], [static]`

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

Parameters

<i>A</i>	Matrix.
<i>cola</i>	Column index.
<i>colb</i>	Column index.
<i>start_row</i>	Row index.
<i>stop_row</i>	Row index (exclusive).

5.8.2.5 `static int mzd_slice_is_zero (const mzd_slice_t * A) [inline], [static]`

Zero test for matrix.

Parameters

<i>A</i>	Input matrix.
----------	---------------

5.9 mzed.h File Reference

Dense matrices over \mathbb{F}_{2^e} represented as packed matrices.

```
#include <m4ri/m4ri.h>
#include <m4rie/gf2e.h>
#include <m4rie/m4ri_functions.h>
```

Data Structures

- struct [mzed_t](#)
Dense matrices over \mathbb{F}_{2^e} represented as packed matrices.

Macros

- #define [mzed_sub mzed_add](#)
 $C = A + B.$
- #define [_mzed_sub _mzed_add](#)
 $C = A + B.$

Functions

- [mzed_t * mzed_init](#) (const [gf2e](#) *ff, const rci_t m, const rci_t n)
Create a new matrix of dimension $m \times n$ over ff.
- void [mzed_free](#) ([mzed_t](#) *A)
Free a matrix created with [mzed_init\(\)](#).
- static [mzed_t * mzed_concat](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
Concatenate B to A and write the result to C.
- static [mzed_t * mzed_stack](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
Stack A on top of B and write the result to C.
- static [mzed_t * mzed_submatrix](#) ([mzed_t](#) *S, const [mzed_t](#) *M, const rci_t lowr, const rci_t lowc, const rci_t highr, const rci_t highc)
Copy a submatrix.
- static [mzed_t * mzed_init_window](#) (const [mzed_t](#) *A, const rci_t lowr, const rci_t lowc, const rci_t highr, const rci_t highc)
Create a window/view into the matrix A.
- static void [mzed_free_window](#) ([mzed_t](#) *A)
Free a matrix window created with [mzed_init_window\(\)](#).
- [mzed_t * mzed_add](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = A + B.$
- [mzed_t * _mzed_add](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = A + B.$
- [mzed_t * mzed_mul](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = A \cdot B.$
- [mzed_t * mzed_addmul](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = C + A \cdot B.$
- [mzed_t * _mzed_mul](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = A \cdot B.$
- [mzed_t * _mzed_addmul](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = C + A \cdot B.$

- `mzed_t * mzed_addmul_naive (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = C + A \cdot B$ using naive cubic multiplication.
- `mzed_t * mzed_mul_naive (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = A \cdot B$ using naive cubic multiplication.
- `mzed_t * _mzed_mul_naive (mzed_t *C, const mzed_t *A, const mzed_t *B)`
 $C = C + A \cdot B$ using naive cubic multiplication.
- `mzed_t * mzed_mul_scalar (mzed_t *C, const word a, const mzed_t *B)`
 $C = a \cdot B$.
- `mzed_t * _mzed_mul_init (mzed_t *C, const mzed_t *A, const mzed_t *B, int clear)`
- `void mzed_randomize (mzed_t *A)`
Fill matrix A with random elements.
- `mzed_t * mzed_copy (mzed_t *B, const mzed_t *A)`
Copy matrix A to B.
- `void mzed_set_ui (mzed_t *A, word value)`
Return diagonal matrix with value on the diagonal.
- `static word mzed_read_elem (const mzed_t *A, const rci_t row, const rci_t col)`
Get the element at position (row,col) from the matrix A.
- `static void mzed_add_elem (mzed_t *A, const rci_t row, const rci_t col, const word elem)`
Add the element elem to the element at position (row,col) in the matrix A.
- `static void mzed_write_elem (mzed_t *A, const rci_t row, const rci_t col, const word elem)`
Write the element elem to the position (row,col) in the matrix A.
- `static int mzed_cmp (mzed_t *A, mzed_t *B)`
Return -1,0,1 if if $A < B$, $A == B$ or $A > B$ respectively.
- `static int mzed_is_zero (const mzed_t *A)`
Zero test for matrix.
- `void mzed_add_multiple_of_row (mzed_t *A, rci_t ar, const mzed_t *B, rci_t br, word x, rci_t start_col)`
- `static void mzed_add_row (mzed_t *A, rci_t ar, const mzed_t *B, rci_t br, rci_t start_col)`
- `static void mzed_rescale_row (mzed_t *A, rci_t r, rci_t start_col, const word x)`
Rescale the row r in A by X starting c.
- `static void mzed_row_swap (mzed_t *M, const rci_t rowa, const rci_t rowb)`
Swap the two rows rowa and rowb.
- `static void mzed_copy_row (mzed_t *B, rci_t i, const mzed_t *A, rci_t j)`
copy row j from A to row i from B.
- `static void mzed_col_swap (mzed_t *M, const rci_t cola, const rci_t colb)`
Swap the two columns cola and colb.
- `static void mzed_col_swap_in_rows (mzed_t *A, const rci_t cola, const rci_t colb, const rci_t start_row, rci_t stop_row)`
Swap the two columns cola and colb but only between start_row and stop_row.
- `static void mzed_row_add (mzed_t *M, const rci_t sourcerow, const rci_t destrow)`
Add the rows sourcerow and destrow and stores the total in the row destrow.
- `static rci_t mzed_first_zero_row (mzed_t *A)`
Return the first row with all zero entries.
- `rci_t mzed_echelonize_naive (mzed_t *A, int full)`
Gaussian elimination.
- `void mzed_print (const mzed_t *M)`
Print a matrix to stdout.

5.9.1 Detailed Description

Dense matrices over \mathbb{F}_{2^e} represented as packed matrices.

This file implements the data type `mzed_t`. That is, matrices over \mathbb{F}_{2^e} in row major representation.

For example, let $a = \sum a_i x_i / \langle f \rangle$ and $b = \sum b_i x_i / \langle f \rangle$ be elements in \mathbb{F}_{2^6} with minimal polynomial f . Then, the 1×2 matrix `[b a]` would be stored as

```
[...| 0 0 b5 b4 b3 b2 b1 b0 | 0 0 a5 a4 a3 a2 a1 a0]
```

Internally M4RI matrices are used to store bits with allows to re-use existing M4RI methods (such as `mzd_add`) when implementing functions for `mzed_t`.

This data type is preferable when Newton-John tables ought be used or when the matrix is small ($m \times n \times e < L2$).

Author

Martin Albrecht martinralbrecht@googlemail.com

5.9.2 Function Documentation

5.9.2.1 `mzed_t* _mzed_mul_init (mzed_t* C, const mzed_t* A, const mzed_t* B, int clear)`

Check whether C, A and B match in sizes and fields for multiplication

Parameters

<i>C</i>	Output matrix, if NULL a new matrix is created.
<i>A</i>	Input matrix.
<i>B</i>	Input matrix.
<i>clear</i>	Write zeros to C or not.

5.9.2.2 `static int mzed_cmp (mzed_t* A, mzed_t* B) [inline],[static]`

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

Parameters

<i>A</i>	Matrix.
<i>B</i>	Matrix.

Note

This comparison is not well defined mathematically and relatively arbitrary since elements of \mathbb{F}_{2^e} don't have an ordering.

Examples:

[tests/test_multiplication.c](#).

5.9.2.3 `static int mzed_is_zero (const mzed_t* A) [inline],[static]`

Zero test for matrix.

Parameters

A	Input matrix.
-----	---------------

5.10 newton_john.h File Reference

Newton-John table based algorithms.

```
#include <m4rie/gf2e.h>
#include <m4rie/mzed.h>
#include <m4rie/mzd_slice.h>
```

Data Structures

- struct [njt_mzed_t](#)
Newton-John table.

Functions

- [njt_mzed_t](#) * [njt_mzed_init](#) (const [gf2e](#) *ff, const [rci_t](#) ncols)
*Allocate Newton-John table of dimension [gf2e::degree](#) < 1 * ncols.*
- void [njt_mzed_free](#) ([njt_mzed_t](#) *t)
Free Newton-John table.
- [njt_mzed_t](#) * [mzed_make_table](#) ([njt_mzed_t](#) *T, const [mzed_t](#) *A, const [rci_t](#) r, const [rci_t](#) c)
Construct Newton-John table T for row r of A, and element A[r,c].
- [mzed_t](#) * [mzed_mul_newton_john](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = A \cdot B$ using Newton-John tables.
- [mzed_t](#) * [mzed_addmul_newton_john](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = C + A \cdot B$ using Newton-John tables.
- [mzed_t](#) * [_mzed_mul_newton_john0](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = C + A \cdot B$ using Newton-John tables.
- [mzed_t](#) * [_mzed_mul_newton_john](#) ([mzed_t](#) *C, const [mzed_t](#) *A, const [mzed_t](#) *B)
 $C = C + A \cdot B$ using Newton-John tables.
- [rci_t](#) [mzed_echelonize_newton_john](#) ([mzed_t](#) *A, int full)
Reduce matrix A to row echelon form using Gauss-Newton-John elimination.
- [mzed_t](#) * [mzed_invert_newton_john](#) ([mzed_t](#) *B, const [mzed_t](#) *A)
Invert the matrix A using Gauss-Newton-John elimination.
- void [mzed_trsm_lower_left_newton_john](#) (const [mzed_t](#) *L, [mzed_t](#) *B)
 $B = L^{-1} \cdot B$ using Newton-John tables.
- void [mzd_slice_trsm_lower_left_newton_john](#) (const [mzd_slice_t](#) *L, [mzd_slice_t](#) *B)
 $B = L^{-1} \cdot B$ using Newton-John tables.
- void [mzed_trsm_upper_left_newton_john](#) (const [mzed_t](#) *U, [mzed_t](#) *B)
 $B = U^{-1} \cdot B$ using Newton-John tables.
- void [mzd_slice_trsm_upper_left_newton_john](#) (const [mzd_slice_t](#) *U, [mzd_slice_t](#) *B)
 $B = U^{-1} \cdot B$ using Newton-John tables.
- [rci_t](#) [mzed_ple_newton_john](#) ([mzed_t](#) *A, [mzp_t](#) *P, [mzp_t](#) *Q)
PLE decomposition: $L \cdot E = P \cdot A$ using Newton-John tables.

- static void [mzed_process_rows](#) ([mzed_t](#) *M, const [rci_t](#) startrow, const [rci_t](#) endrow, [rci_t](#) startcol, const [njt_mzed_t](#) *T)

The function looks up 6 entries from position i,startcol in each row and adds the appropriate row from T to the row i.

- static void [mzed_process_rows2](#) ([mzed_t](#) *M, const [rci_t](#) startrow, const [rci_t](#) endrow, const [rci_t](#) startcol, const [njt_mzed_t](#) *T0, const [njt_mzed_t](#) *T1)

Same as mzed_process_rows but works with two Newton-John tables in parallel.

- static void [mzed_process_rows3](#) ([mzed_t](#) *M, const [rci_t](#) startrow, const [rci_t](#) endrow, const [rci_t](#) startcol, const [njt_mzed_t](#) *T0, const [njt_mzed_t](#) *T1, const [njt_mzed_t](#) *T2)

Same as mzed_process_rows but works with three Newton-John tables in parallel.

- static void [mzed_process_rows4](#) ([mzed_t](#) *M, const [rci_t](#) startrow, const [rci_t](#) endrow, const [rci_t](#) startcol, const [njt_mzed_t](#) *T0, const [njt_mzed_t](#) *T1, const [njt_mzed_t](#) *T2, const [njt_mzed_t](#) *T3)

Same as mzed_process_rows but works with four Newton-John tables in parallel.

- static void [mzed_process_rows5](#) ([mzed_t](#) *M, const [rci_t](#) startrow, const [rci_t](#) endrow, const [rci_t](#) startcol, const [njt_mzed_t](#) *T0, const [njt_mzed_t](#) *T1, const [njt_mzed_t](#) *T2, const [njt_mzed_t](#) *T3, const [njt_mzed_t](#) *T4)

Same as mzed_process_rows but works with five Newton-John tables in parallel.

- static void [mzed_process_rows6](#) ([mzed_t](#) *M, const [rci_t](#) startrow, const [rci_t](#) endrow, const [rci_t](#) startcol, const [njt_mzed_t](#) *T0, const [njt_mzed_t](#) *T1, const [njt_mzed_t](#) *T2, const [njt_mzed_t](#) *T3, const [njt_mzed_t](#) *T4, const [njt_mzed_t](#) *T5)

Same as mzed_process_rows but works with six Newton-John tables in parallel.

5.10.1 Detailed Description

Newton-John table based algorithms.

Note

These tables were formally known as Travolta tables.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.10.2 Function Documentation

5.10.2.1 [mzed_t](#)* [mzed_invert_newton_john](#) ([mzed_t](#) * B, const [mzed_t](#) * A)

Invert the matrix A using Gauss-Newton-John elimination.

Parameters

B	Preallocated space for inversion matrix, may be NULL for automatic creation.
A	Matrix to be inverted.

5.10.2.2 [njt_mzed_t](#)* [mzed_make_table](#) ([njt_mzed_t](#) * T, const [mzed_t](#) * A, const [rci_t](#) r, const [rci_t](#) c)

Construct Newton-John table T for row r of A, and element A[r,c].

Parameters

<i>T</i>	Preallocated Newton-John table or NULL.
<i>A</i>	Matrix.
<i>r</i>	Row index.
<i>c</i>	Column index.

5.10.2.3 void njt_mzed_free (njt_mzed_t * t)

Free Newton-John table.

Parameters

<i>t</i>	Table
----------	-------

5.10.2.4 njt_mzed_t* njt_mzed_init (const gf2e * ff, const rci_t ncols)

Allocate Newton-John table of dimension [gf2e::degree](#) << 1 * ncols.

Parameters

<i>ff</i>	Finite field.
<i>ncols</i>	Integer > 0.

5.11 permutation.h File Reference

Permutation matrices.

```
#include <m4ri/mzp.h>
#include <m4rie/mzed.h>
#include <m4rie/mzd_slice.h>
```

Functions

- static void [mzed_apply_p_left](#) (mzed_t *A, mzp_t const *P)
- static void [mzed_apply_p_left_trans](#) (mzed_t *A, mzp_t const *P)
- static void [mzed_apply_p_right](#) (mzed_t *A, mzp_t const *P)
- static void [mzed_apply_p_right_trans](#) (mzed_t *A, mzp_t const *P)
- static void [mzd_slice_apply_p_left](#) (mzd_slice_t *A, mzp_t const *P)
- static void [mzd_slice_apply_p_left_trans](#) (mzd_slice_t *A, mzp_t const *P)
- static void [mzd_slice_apply_p_right](#) (mzd_slice_t *A, mzp_t const *P)
- static void [mzd_slice_apply_p_right_trans](#) (mzd_slice_t *A, mzp_t const *P)
- static void [mzd_slice_apply_p_right_trans_tri](#) (mzd_slice_t *A, mzp_t const *P)

5.11.1 Detailed Description

Permutation matrices.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.11.2 Function Documentation

5.11.2.1 `static void mzd_slice_apply_p_left (mzd_slice_t * A, mzp_t const * P) [inline],[static]`

Apply the permutation P to A from the left.

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

Parameters

<i>A</i>	Matrix.
<i>P</i>	Permutation.

5.11.2.2 `static void mzd_slice_apply_p_left_trans (mzd_slice_t * A, mzp_t const * P) [inline],[static]`

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

<i>A</i>	Matrix.
<i>P</i>	Permutation.

5.11.2.3 `static void mzd_slice_apply_p_right (mzd_slice_t * A, mzp_t const * P) [inline],[static]`

Apply the permutation P to A from the right.

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

Parameters

<i>A</i>	Matrix.
<i>P</i>	Permutation.

5.11.2.4 `static void mzd_slice_apply_p_right_trans (mzd_slice_t * A, mzp_t const * P) [inline],[static]`

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

<i>A</i>	Matrix.
<i>P</i>	Permutation.

5.11.2.5 `static void mzd_slice_apply_p_right_trans_tri (mzd_slice_t * A, mzp_t const * P) [inline],[static]`

Apply the permutation P to A from the right, but only on the upper the matrix A above the main diagonal.

This is equivalent to column swaps walking from 0 to length-1 and is used to compress PLE to PLUQ.

Parameters

<i>A</i>	Matrix.
----------	---------

P	Permutation.
-----	--------------

5.11.2.6 `static void mzed_apply_p_left (mzed_t * A, mzp_t const * P) [inline],[static]`

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.

5.11.2.7 `static void mzed_apply_p_left_trans (mzed_t * A, mzp_t const * P) [inline],[static]`

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.

5.11.2.8 `static void mzed_apply_p_right (mzed_t * A, mzp_t const * P) [inline],[static]`

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.

5.11.2.9 `static void mzed_apply_p_right_trans (mzed_t * A, mzp_t const * P) [inline],[static]`

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.

5.12 ple.h File Reference

PLE decomposition: $L \cdot E = P \cdot A$.

```
#include <m4ri/m4ri.h>
#include <m4rie/mzed.h>
#include <m4rie/mzd_slice.h>
#include <m4rie/conversion.h>
```

Macros

- `#define __M4RIE_PLE_CUTOFF (__M4RI_CPU_L2_CACHE<<2)`

Functions

- `rci_t mzed_ple_naive (mzed_t *A, mzp_t *P, mzp_t *Q)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `rci_t _mzd_slice_ple (mzd_slice_t *A, mzp_t *P, mzp_t *Q, rci_t cutoff)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `static rci_t mzd_slice_ple (mzd_slice_t *A, mzp_t *P, mzp_t *Q)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `rci_t _mzd_slice_pluq (mzd_slice_t *A, mzp_t *P, mzp_t *Q, rci_t cutoff)`
PLUQ decomposition: $L \cdot U \cdot Q = P \cdot A$.
- `static rci_t mzd_slice_pluq (mzd_slice_t *A, mzp_t *P, mzp_t *Q)`
PLUQ decomposition: $L \cdot U \cdot Q = P \cdot A$.
- `rci_t _mzed_ple (mzed_t *A, mzp_t *P, mzp_t *Q, rci_t cutoff)`
PLE decomposition: $L \cdot E = P \cdot A$.
- `static rci_t mzed_ple (mzed_t *A, mzp_t *P, mzp_t *Q)`
PLE decomposition: $L \cdot E = P \cdot A$.

5.12.1 Detailed Description

PLE decomposition: $L \cdot E = P \cdot A$.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.12.2 Macro Definition Documentation

5.12.2.1 `#define __M4RIE_PLE_CUTOFF (__M4RI_CPU_L2_CACHE<<2)`

Default crossover to PLE base case (Newton-John based).

5.13 strassen.h File Reference

Strassen-Winograd multiplication for `mzed_t`.

Functions

- `mzed_t * mzed_mul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`
 $C = A \cdot B$ using Strassen-Winograd.
- `mzed_t * mzed_addmul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`
 $C = C + A \cdot B$ using Strassen-Winograd.
- `mzed_t * _mzed_mul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`
 $C = A \cdot B$ using Strassen-Winograd.
- `mzed_t * _mzed_addmul_strassen (mzed_t *C, const mzed_t *A, const mzed_t *B, int cutoff)`

$C = A \cdot B$ using Strassen-Winograd.

- `rci_t mzed_strassen_cutoff` (const `mzed_t` *C, const `mzed_t` *A, const `mzed_t` *B)

Return heuristic choice for crossover parameter for Strassen-Winograd multiplication given A, B and C.

5.13.1 Detailed Description

Strassen-Winograd multiplication for `mzed_t`.

Author

Martin Albrecht martinralbrecht@googlemail.com

5.14 trsm.h File Reference

Triangular System Solving with Matrices (TRSM).

```
#include <m4rie/mzed.h>
#include <m4rie/mzd_slice.h>
```

Macros

- `#define MZED_TRSM_CUTOFF` 512

Functions

- void `_mzed_trsm_upper_left` (const `mzed_t` *U, `mzed_t` *B, const `rci_t` cutoff)
 $B = U^{-1} \cdot B$
- void `mzed_trsm_upper_left_naive` (const `mzed_t` *U, `mzed_t` *B)
 $B = U^{-1} \cdot B$
- static void `mzed_trsm_upper_left` (const `mzed_t` *U, `mzed_t` *B)
 $B = U^{-1} \cdot B$
- void `_mzd_slice_trsm_upper_left` (const `mzd_slice_t` *U, `mzd_slice_t` *B, const `rci_t` cutoff)
 $B = U^{-1} \cdot B$
- void `mzd_slice_trsm_upper_left_naive` (const `mzd_slice_t` *U, `mzd_slice_t` *B)
 $B = U^{-1} \cdot B$
- static void `mzd_slice_trsm_upper_left` (const `mzd_slice_t` *U, `mzd_slice_t` *B)
 $B = U^{-1} \cdot B$
- void `_mzed_trsm_lower_left` (const `mzed_t` *L, `mzed_t` *B, const `rci_t` cutoff)
 $B = L^{-1} \cdot B$
- void `mzed_trsm_lower_left_naive` (const `mzed_t` *L, `mzed_t` *B)
 $B = L^{-1} \cdot B$
- static void `mzed_trsm_lower_left` (const `mzed_t` *L, `mzed_t` *B)
 $B = L^{-1} \cdot B$
- void `_mzd_slice_trsm_lower_left` (const `mzd_slice_t` *L, `mzd_slice_t` *B, const `rci_t` cutoff)
 $B = L^{-1} \cdot B$
- void `mzd_slice_trsm_lower_left_naive` (const `mzd_slice_t` *L, `mzd_slice_t` *B)
 $B = L^{-1} \cdot B$
- static void `mzd_slice_trsm_lower_left` (const `mzd_slice_t` *L, `mzd_slice_t` *B)
 $B = L^{-1} \cdot B$

5.14.1 Detailed Description

Triangular System Solving with Matrices (TRSM).

Author

Martin Albrecht martinralbrecht@googlemail.com

5.14.2 Macro Definition Documentation

5.14.2.1 #define MZED_TRSM_CUTOFF 512

Crossover dimension to TRSM base cases

6 Example Documentation

6.1 tests/test_multiplication.c

```

/*****
 *
 *          M4RIE: Linear Algebra over GF(2^e)
 *
 *
 *   Copyright (C) 2010-2012 Martin Albrecht <martinralbrecht@googlemail.com>
 *
 *   Distributed under the terms of the GNU General Public License (GEL)
 *   version 2 or higher.
 *
 *   This code is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 *   General Public License for more details.
 *
 *   The full text of the GPL is available at:
 *
 *           http://www.gnu.org/licenses/
 *****/

#include "testing.h"

int test_addmul(gf2e *ff, rci_t m, rci_t n, rci_t l) {
    int fail_ret = 0;

    mzed_t *A = random_mzed_t(ff, m, l);
    mzed_t *B = random_mzed_t(ff, l, n);

    mzed_t *C0 = random_mzed_t(ff, m, n);
    mzed_t *C1 = mzed_copy(NULL, C0);
    mzed_t *C2 = mzed_copy(NULL, C0);
    mzed_t *C3 = mzed_copy(NULL, C0);
    mzed_t *C4 = mzed_copy(NULL, C0);
    mzed_t *C5 = mzed_copy(NULL, C0);

    mzed_set_canary(C1);
    mzed_set_canary(C2);
    mzed_set_canary(C3);
    mzed_set_canary(C4);
    mzed_set_canary(C5);

    mzed_addmul_newton_john(C0, A, B);
    mzed_addmul_naive(C1, A, B);
    mzed_addmul_strassen(C2, A, B, 64);
    mzed_addmul(C3, A, B);
    mzed_addmul_karatsuba(C4, A, B);
    mzed_addmul_blm(C5, A, B);

    m4rie_check( mzed_cmp(C0, C1) == 0);
    m4rie_check( mzed_cmp(C1, C2) == 0);
    m4rie_check( mzed_cmp(C2, C3) == 0);

```

```

m4rie_check( mzed_cmp(C3, C4) == 0);
m4rie_check( mzed_cmp(C4, C5) == 0);

m4rie_check( mzed_canary_is_alive(A) );
m4rie_check( mzed_canary_is_alive(B) );
m4rie_check( mzed_canary_is_alive(C1) );
m4rie_check( mzed_canary_is_alive(C2) );
m4rie_check( mzed_canary_is_alive(C3) );
m4rie_check( mzed_canary_is_alive(C4) );
m4rie_check( mzed_canary_is_alive(C5) );

mzed_free(A);
mzed_free(B);
mzed_free(C0);
mzed_free(C1);
mzed_free(C2);
mzed_free(C3);
mzed_free(C4);
mzed_free(C5);

return fail_ret;
}

int test_mul(gf2e *ff, rci_t m, rci_t n, rci_t l) {
    int fail_ret = 0;
    const mzed_t *A = random_mzed_t(ff, m, l);
    const mzed_t *B = random_mzed_t(ff, l, n);

    mzed_t *C0 = random_mzed_t(ff, m, n);
    mzed_t *C1 = random_mzed_t(ff, m, n);
    mzed_t *C2 = random_mzed_t(ff, m, n);
    mzed_t *C3 = random_mzed_t(ff, m, n);
    mzed_t *C4 = random_mzed_t(ff, m, n);
    mzed_t *C5 = random_mzed_t(ff, m, n);

    mzed_mul_newton_john(C0, A, B);
    mzed_mul_naive(C1, A, B);
    mzed_mul_strassen(C2, A, B, 64);
    mzed_mul(C3, A, B);
    mzed_mul_karatsuba(C4, A, B);
    mzed_mul_blm(C5, A, B);

    m4rie_check( mzed_cmp(C0, C1) == 0);
    m4rie_check( mzed_cmp(C1, C2) == 0);
    m4rie_check( mzed_cmp(C2, C3) == 0);
    m4rie_check( mzed_cmp(C3, C4) == 0);
    m4rie_check( mzed_cmp(C4, C5) == 0);

    m4rie_check( mzed_canary_is_alive((mzed_t*)A) );
    m4rie_check( mzed_canary_is_alive((mzed_t*)B) );
    m4rie_check( mzed_canary_is_alive(C1) );
    m4rie_check( mzed_canary_is_alive(C2) );
    m4rie_check( mzed_canary_is_alive(C3) );
    m4rie_check( mzed_canary_is_alive(C4) );
    m4rie_check( mzed_canary_is_alive(C5) );

    mzed_free((mzed_t*)A);
    mzed_free((mzed_t*)B);
    mzed_free(C0);
    mzed_free(C1);
    mzed_free(C2);
    mzed_free(C3);
    mzed_free(C4);
    mzed_free(C5);

    return fail_ret;
}

int test_scalar(gf2e *ff, rci_t m, rci_t n) {
    int fail_ret = 0;

    word a = random() & ((1<<ff->degree)-1);
    while (!a)
        a = random() & ((1<<ff->degree)-1);
    mzed_t *B = random_mzed_t(ff, m, n);

    mzed_t *C0 = mzed_init(ff, m, n);
    mzed_t *C1 = random_mzed_t(ff, m, n);
    mzed_t *C2 = NULL;

    C0 = mzed_mul_scalar(C0, a, B);

```

```

C1 = mzed_mul_scalar(C1, a, B);
C2 = mzed_mul_scalar(C2, a, B);

m4rie_check( mzed_cmp(C0, C1) == 0);
m4rie_check( mzed_cmp(C1, C2) == 0);

mzed_t *C3 = NULL;
mzd_slice_t *BB = mzed_slice(NULL, B);
mzd_slice_t *CC = mzd_slice_mul_scalar(NULL, a, BB);
C3 = mzed_cling(C3, CC);
mzd_slice_free(BB);
mzd_slice_free(CC);
m4rie_check( mzed_cmp(C2, C3) == 0);
mzed_free(C3);

const word a_inv = gf2e_inv(ff, a);

mzed_t *B0 = mzed_init(ff, m, n);
mzed_t *B1 = random_mzed_t(ff, m, n);
mzed_t *B2 = NULL;

B0 = mzed_mul_scalar(B0, a_inv, C0);
B1 = mzed_mul_scalar(B1, a_inv, C1);
B2 = mzed_mul_scalar(B2, a_inv, C2);

m4rie_check( mzed_cmp(B, B0) == 0);
m4rie_check( mzed_cmp(B, B1) == 0);
m4rie_check( mzed_cmp(B, B2) == 0);

mzed_free(C0);
mzed_free(C1);
mzed_free(C2);

mzed_free(B0);
mzed_free(B1);
mzed_free(B2);

mzed_free(B);

return fail_ret;
}

int test_batch(gf2e *ff, rci_t m, rci_t l, rci_t n) {
    int fail_ret = 0;
    printf("mul: k: %2d, minpoly: 0x%05x m: %5d, l: %5d, n: %5d ", (int)ff->degree, (unsigned int)ff->
        minpoly, (int)m, (int)l, (int)n);

    m4rie_check(test_scalar(ff, m, m) == 0); printf("."); fflush(0);
    m4rie_check(test_scalar(ff, l, l) == 0); printf("."); fflush(0);
    m4rie_check(test_scalar(ff, n, n) == 0); printf("."); fflush(0);
    m4rie_check(test_scalar(ff, m, l) == 0); printf("."); fflush(0);
    m4rie_check(test_scalar(ff, l, n) == 0); printf("."); fflush(0);
    m4rie_check(test_scalar(ff, m, n) == 0); printf("."); fflush(0);
    m4rie_check(test_scalar(ff, l, m) == 0); printf("."); fflush(0);

    if(m == l && m == n) {
        m4rie_check( test_mul(ff, m, l, n) == 0); printf("."); fflush(0);
        m4rie_check(test_addmul(ff, m, l, n) == 0); printf("."); fflush(0);
        printf("
    ")
    } else {
        m4rie_check( test_mul(ff, m, l, n) == 0); printf("."); fflush(0);
        m4rie_check( test_mul(ff, m, n, l) == 0); printf("."); fflush(0);
        m4rie_check( test_mul(ff, n, m, l) == 0); printf("."); fflush(0);
        m4rie_check( test_mul(ff, n, l, m) == 0); printf("."); fflush(0);
        m4rie_check( test_mul(ff, l, m, n) == 0); printf("."); fflush(0);
        m4rie_check( test_mul(ff, l, n, m) == 0); printf("."); fflush(0);
        m4rie_check(test_addmul(ff, m, l, n) == 0); printf("."); fflush(0);
        m4rie_check(test_addmul(ff, m, n, l) == 0); printf("."); fflush(0);
        m4rie_check(test_addmul(ff, n, m, l) == 0); printf("."); fflush(0);
        m4rie_check(test_addmul(ff, n, l, m) == 0); printf("."); fflush(0);
        m4rie_check(test_addmul(ff, l, m, n) == 0); printf("."); fflush(0);
        m4rie_check(test_addmul(ff, l, n, m) == 0); printf("."); fflush(0);
    }

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

    return fail_ret;
}

```



```
}

int main(int argc, char **argv) {
    srand(17);

    int runlong = parse_parameters(argc, argv);

    gf2e *ff;
    int fail_ret = 0;

    for(int k=2; k<=16; k++) {
        ff = gf2e_init(irreducible_polynomials[k][1]);

        fail_ret += test_batch(ff, 1, 1, 1);
        fail_ret += test_batch(ff, 1, 2, 3);
        fail_ret += test_batch(ff, 11, 12, 13);
        fail_ret += test_batch(ff, 21, 22, 23);
        fail_ret += test_batch(ff, 13, 2, 90);
        fail_ret += test_batch(ff, 32, 33, 34);
        fail_ret += test_batch(ff, 63, 64, 65);
        if(k<=12 || runlong) {
            fail_ret += test_batch(ff, 127, 128, 129);
            fail_ret += test_batch(ff, 200, 20, 112);
        }

        gf2e_free(ff);
    }

    return fail_ret;
}
```

Index

Addition and subtraction, [21](#)
Assignment and basic manipulation, [9](#)

Constructions, [3](#)

Echelon forms, [37](#)

Multiplication, [24](#)

Operations on rows, [13](#)

Triangular matrices, [39](#)
Type definitions, [2](#)