

Matrix2D

Generated by Doxygen 1.9.1

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 Mat2D Struct Reference	5
3.1.1 Detailed Description	5
3.1.2 Member Data Documentation	5
3.1.2.1 cols	6
3.1.2.2 elements	6
3.1.2.3 rows	6
3.1.2.4 stride_r	7
3.2 Mat2D_Minor Struct Reference	7
3.2.1 Detailed Description	8
3.2.2 Member Data Documentation	8
3.2.2.1 cols	8
3.2.2.2 cols_list	8
3.2.2.3 ref_mat	8
3.2.2.4 rows	9
3.2.2.5 rows_list	9
3.2.2.6 stride_r	9
3.3 Mat2D_uint32 Struct Reference	9
3.3.1 Detailed Description	10
3.3.2 Member Data Documentation	10
3.3.2.1 cols	10
3.3.2.2 elements	10
3.3.2.3 rows	10
3.3.2.4 stride_r	10
4 File Documentation	11
4.1 examples/example1.c File Reference	11
4.1.1 Macro Definition Documentation	11
4.1.1.1 MATRIX2D_IMPLEMENTATION	11
4.1.2 Function Documentation	12
4.1.2.1 main()	12
4.2 example1.c	12
4.3 Matrix2D.h File Reference	13
4.3.1 Detailed Description	17
4.3.2 Macro Definition Documentation	18
4.3.2.1 MAT2D_ASSERT	18
4.3.2.2 MAT2D_AT	19

4.3.2.3 MAT2D_AT_UINT32	19
4.3.2.4 mat2D_dprintDOUBLE	19
4.3.2.5 mat2D_dprintINT	19
4.3.2.6 mat2D_dprintSIZE_T	20
4.3.2.7 MAT2D_EPS	20
4.3.2.8 MAT2D_FREE	20
4.3.2.9 MAT2D_IS_ZERO	20
4.3.2.10 MAT2D_MALLOC	20
4.3.2.11 MAT2D_MAX_POWER_ITERATION	21
4.3.2.12 MAT2D_MINOR_AT	21
4.3.2.13 MAT2D_MINOR_PRINT	21
4.3.2.14 mat2D_normalize	21
4.3.2.15 mat2D_normalize_inf	22
4.3.2.16 MAT2D_PI	22
4.3.2.17 MAT2D_PRINT	22
4.3.2.18 MAT2D_PRINT_AS_COL	22
4.3.3 Enumeration Type Documentation	22
4.3.3.1 mat2D_upper_triangulate_flag	22
4.3.4 Function Documentation	23
4.3.4.1 mat2D_add()	23
4.3.4.2 mat2D_add_col_to_col()	23
4.3.4.3 mat2D_add_row_time_factor_to_row()	24
4.3.4.4 mat2D_add_row_to_row()	24
4.3.4.5 mat2D_alloc()	25
4.3.4.6 mat2D_alloc_uint32()	26
4.3.4.7 mat2D_calc_col_norma()	26
4.3.4.8 mat2D_calc_norma()	27
4.3.4.9 mat2D_calc_norma_inf()	27
4.3.4.10 mat2D_col_is_all_digit()	28
4.3.4.11 mat2D_copy()	29
4.3.4.12 mat2D_copy_col_from_src_to_des()	29
4.3.4.13 mat2D_copy_row_from_src_to_des()	30
4.3.4.14 mat2D_copy_src_to_des_window()	30
4.3.4.15 mat2D_copy_src_window_to_des()	31
4.3.4.16 mat2D_create_col_ref()	32
4.3.4.17 mat2D_cross()	32
4.3.4.18 mat2D_det()	33
4.3.4.19 mat2D_det_2x2_mat()	33
4.3.4.20 mat2D_det_2x2_mat_minor()	34
4.3.4.21 mat2D_dot()	34
4.3.4.22 mat2D_dot_product()	35
4.3.4.23 mat2D_eig_check()	36

4.3.4.24 mat2D_eig_power_iteration()	37
4.3.4.25 mat2D_fill()	37
4.3.4.26 mat2D_fill_sequence()	38
4.3.4.27 mat2D_fill_uint32()	38
4.3.4.28 mat2D_find_first_non_zero_value()	39
4.3.4.29 mat2D_free()	39
4.3.4.30 mat2D_free_uint32()	40
4.3.4.31 mat2D_inner_product()	40
4.3.4.32 mat2D_invert()	41
4.3.4.33 mat2D_LUP_decomposition_with_swap()	42
4.3.4.34 mat2D_make_orthogonal_Gaussian_elimination()	43
4.3.4.35 mat2D_make_orthogonal_modified_Gram_Schmidt()	43
4.3.4.36 mat2D_mat_is_all_digit()	44
4.3.4.37 mat2D_minor_alloc_fill_from_mat()	45
4.3.4.38 mat2D_minor_alloc_fill_from_mat_minor()	45
4.3.4.39 mat2D_minor_det()	46
4.3.4.40 mat2D_minor_free()	46
4.3.4.41 mat2D_minor_print()	47
4.3.4.42 mat2D_mult()	47
4.3.4.43 mat2D_mult_row()	48
4.3.4.44 mat2D_offset2d()	48
4.3.4.45 mat2D_offset2d_uint32()	49
4.3.4.46 mat2D_outer_product()	49
4.3.4.47 mat2D_power_iterate()	50
4.3.4.48 mat2D_print()	51
4.3.4.49 mat2D_print_as_col()	51
4.3.4.50 mat2D_rand()	52
4.3.4.51 mat2D_rand_double()	52
4.3.4.52 mat2D_reduce()	53
4.3.4.53 mat2D_row_is_all_digit()	53
4.3.4.54 mat2D_set_DCM_zyx()	54
4.3.4.55 mat2D_set_identity()	55
4.3.4.56 mat2D_set_rot_mat_x()	55
4.3.4.57 mat2D_set_rot_mat_y()	55
4.3.4.58 mat2D_set_rot_mat_z()	56
4.3.4.59 mat2D_shift()	56
4.3.4.60 mat2D_solve_linear_sys_LUP_decomposition()	57
4.3.4.61 mat2D_sub()	57
4.3.4.62 mat2D_sub_col_to_col()	58
4.3.4.63 mat2D_sub_row_time_factor_to_row()	58
4.3.4.64 mat2D_sub_row_to_row()	59
4.3.4.65 mat2D_SVD_full()	60

4.3.4.66 mat2D_SVD_thin()	61
4.3.4.67 mat2D_swap_rows()	62
4.3.4.68 mat2D_transpose()	63
4.3.4.69 mat2D_upper_triangularize()	63
4.4 Matrix2D.h	64
4.5 temp.c File Reference	80
4.5.1 Macro Definition Documentation	81
4.5.1.1 MATRIX2D_IMPLEMENTATION	81
4.5.2 Function Documentation	81
4.5.2.1 main()	81
4.6 temp.c	81
4.7 test_matrix2d.c File Reference	82
4.7.1 Macro Definition Documentation	83
4.7.1.1 MATRIX2D_IMPLEMENTATION	83
4.7.1.2 RUN_TEST	83
4.7.2 Function Documentation	84
4.7.2.1 assert_identity_close()	84
4.7.2.2 assert_inverse_identity_both_sides()	84
4.7.2.3 assert_mat_close()	84
4.7.2.4 assert_permutation_matrix()	85
4.7.2.5 close_rel_abs()	85
4.7.2.6 det_by_minors_first_col()	85
4.7.2.7 fill_mat_from_array()	85
4.7.2.8 fill_strictly_diag_dominant()	86
4.7.2.9 main()	86
4.7.2.10 nearly_equal()	86
4.7.2.11 rng_range()	87
4.7.2.12 rng_unit01()	87
4.7.2.13 test_alloc_fill_copy_add_sub()	87
4.7.2.14 test_copy_row_and_col_helpers()	87
4.7.2.15 test_copy_windows()	88
4.7.2.16 test_DCM_zyx_matches_product()	88
4.7.2.17 test_det_2x2_and_upper_triangularize_sign()	88
4.7.2.18 test_det_and_minor_det_agree_3x3()	88
4.7.2.19 test_det_early_zero_row_and_zero_col_paths()	89
4.7.2.20 test_deterministic_fuzz_loop()	89
4.7.2.21 test_dot_product_and_vector_variants()	89
4.7.2.22 test_dot_product_matrix_multiply()	89
4.7.2.23 test_invert()	90
4.7.2.24 test_LUP_decomposition_identity_P_no_swap_case()	90
4.7.2.25 test_LUP_decomposition_swap_required_case()	90
4.7.2.26 test_mat_is_all_digit()	90

4.7.2.27 test_minor_det_matches_gauss_4x4_known()	91
4.7.2.28 test_non_contiguous_stride_views()	91
4.7.2.29 test_norms_and_normalize()	91
4.7.2.30 test_offset2d_and_stride()	91
4.7.2.31 test_outer_product_and_cross()	92
4.7.2.32 test_outer_product_row_vector_path()	92
4.7.2.33 test_power_iterate_and_eig_helpers()	92
4.7.2.34 test_rand_range()	92
4.7.2.35 test_reduce_rank()	93
4.7.2.36 test_rotation_matrices_orthonormal()	93
4.7.2.37 test_row_col_ops_and_scaling()	93
4.7.2.38 test_shift_and_identity()	93
4.7.2.39 test_solve_linear_system_LUP()	94
4.7.2.40 test_transpose()	94
4.7.2.41 test_uint32_alloc_fill_and_at()	94
4.7.2.42 xorshift64star()	94
4.8 test_matrix2d.c	95

Index

109

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Mat2D	Dense row-major matrix of double	5
Mat2D_Minor	A minor "view" into a reference matrix	7
Mat2D_uint32	Dense row-major matrix of uint32_t	9

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Matrix2D.h		
Lightweight 2D matrix helpers (double / uint32_t)	13
temp.c	80
test_matrix2d.c	82
examples/ example1.c	11

Chapter 3

Class Documentation

3.1 Mat2D Struct Reference

Dense row-major matrix of double.

```
#include <Matrix2D.h>
```

Public Attributes

- `size_t` [rows](#)
- `size_t` [cols](#)
- `size_t` [stride_r](#)
- `double *` [elements](#)

3.1.1 Detailed Description

Dense row-major matrix of double.

- `rows` Number of rows (height).
- `cols` Number of columns (width).
- `stride_r` Number of elements between successive rows in memory. For contiguous storage, `stride_r == cols`.
- `elements` Pointer to a contiguous buffer of `rows * cols` doubles.

Note

This type is a shallow handle; copying [Mat2D](#) copies the pointer, not the underlying data.

Definition at line [117](#) of file [Matrix2D.h](#).

3.1.2 Member Data Documentation

3.1.2.1 cols

```
size_t Mat2D::cols
```

Definition at line 119 of file [Matrix2D.h](#).

Referenced by [assert_identity_close\(\)](#), [assert_inverse_identity_both_sides\(\)](#), [assert_mat_close\(\)](#), [assert_permutation_matrix\(\)](#), [det_by_minors_first_col\(\)](#), [fill_mat_from_array\(\)](#), [fill_strictly_diag_dominant\(\)](#), [mat2D_add\(\)](#), [mat2D_add_col_to_col\(\)](#), [mat2D_add_row_time_factor_to_row\(\)](#), [mat2D_add_row_to_row\(\)](#), [mat2D_alloc\(\)](#), [mat2D_calc_col_norma\(\)](#), [mat2D_calc_norma\(\)](#), [mat2D_calc_norma_inf\(\)](#), [mat2D_copy\(\)](#), [mat2D_copy_col_from_src_to_des\(\)](#), [mat2D_copy_row_from_src_to_des\(\)](#), [mat2D_copy_src_to_des_window\(\)](#), [mat2D_copy_src_window_to_des\(\)](#), [mat2D_create_col_ref\(\)](#), [mat2D_cross\(\)](#), [mat2D_det\(\)](#), [mat2D_det_2x2_mat\(\)](#), [mat2D_dot\(\)](#), [mat2D_dot_product\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_fill\(\)](#), [mat2D_fill_sequence\(\)](#), [mat2D_find_first_non_zero_value\(\)](#), [mat2D_inner_product\(\)](#), [mat2D_invert\(\)](#), [mat2D_LUP_decomposition_with_swap\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_mat_is_all_digit\(\)](#), [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_mult\(\)](#), [mat2D_mult_row\(\)](#), [mat2D_offset2d\(\)](#), [mat2D_outer_product\(\)](#), [mat2D_power_iterate\(\)](#), [mat2D_print\(\)](#), [mat2D_print_as_col\(\)](#), [mat2D_rand\(\)](#), [mat2D_reduce\(\)](#), [mat2D_row_is_all_digit\(\)](#), [mat2D_set_identity\(\)](#), [mat2D_set_rot_mat_x\(\)](#), [mat2D_set_rot_mat_y\(\)](#), [mat2D_set_rot_mat_z\(\)](#), [mat2D_shift\(\)](#), [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), [mat2D_sub\(\)](#), [mat2D_sub_col_to_col\(\)](#), [mat2D_sub_row_time_factor_to_row\(\)](#), [mat2D_SVD_full\(\)](#), [mat2D_SVD_thin\(\)](#), [mat2D_swap_rows\(\)](#), [mat2D_transpose\(\)](#), [mat2D_upper_triangularize\(\)](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_rand_range\(\)](#), [test_shift_and_identity\(\)](#), and [test_transpose\(\)](#).

3.1.2.2 elements

```
double* Mat2D::elements
```

Definition at line 121 of file [Matrix2D.h](#).

Referenced by [mat2D_alloc\(\)](#), [mat2D_free\(\)](#), and [mat2D_print_as_col\(\)](#).

3.1.2.3 rows

```
size_t Mat2D::rows
```

Definition at line 118 of file [Matrix2D.h](#).

Referenced by [assert_identity_close\(\)](#), [assert_inverse_identity_both_sides\(\)](#), [assert_mat_close\(\)](#), [assert_permutation_matrix\(\)](#), [det_by_minors_first_col\(\)](#), [fill_mat_from_array\(\)](#), [fill_strictly_diag_dominant\(\)](#), [main\(\)](#), [mat2D_add\(\)](#), [mat2D_add_col_to_col\(\)](#), [mat2D_add_row_to_row\(\)](#), [mat2D_alloc\(\)](#), [mat2D_calc_col_norma\(\)](#), [mat2D_calc_norma\(\)](#), [mat2D_calc_norma_inf\(\)](#), [mat2D_col_is_all_digit\(\)](#), [mat2D_copy\(\)](#), [mat2D_copy_col_from_src_to_des\(\)](#), [mat2D_copy_row_from_src_to_des\(\)](#), [mat2D_copy_src_to_des_window\(\)](#), [mat2D_copy_src_window_to_des\(\)](#), [mat2D_create_col_ref\(\)](#), [mat2D_cross\(\)](#), [mat2D_det\(\)](#), [mat2D_det_2x2_mat\(\)](#), [mat2D_dot\(\)](#), [mat2D_dot_product\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_fill\(\)](#), [mat2D_fill_sequence\(\)](#), [mat2D_inner_product\(\)](#), [mat2D_invert\(\)](#), [mat2D_LUP_decomposition_with_swap\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_mat_is_all_digit\(\)](#), [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_mult\(\)](#), [mat2D_offset2d\(\)](#), [mat2D_outer_product\(\)](#), [mat2D_power_iterate\(\)](#), [mat2D_print\(\)](#), [mat2D_print_as_col\(\)](#), [mat2D_rand\(\)](#), [mat2D_reduce\(\)](#), [mat2D_set_identity\(\)](#), [mat2D_set_rot_mat_x\(\)](#), [mat2D_set_rot_mat_y\(\)](#), [mat2D_set_rot_mat_z\(\)](#), [mat2D_shift\(\)](#), [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), [mat2D_sub\(\)](#), [mat2D_sub_col_to_col\(\)](#), [mat2D_sub_row_to_row\(\)](#), [mat2D_SVD_full\(\)](#), [mat2D_SVD_thin\(\)](#), [mat2D_transpose\(\)](#), [mat2D_upper_triangularize\(\)](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_non_contiguous_stride_views\(\)](#), [test_rand_range\(\)](#), [test_shift_and_identity\(\)](#), and [test_transpose\(\)](#).

3.1.2.4 stride_r

```
size_t Mat2D::stride_r
```

Definition at line 120 of file [Matrix2D.h](#).

Referenced by [mat2D_alloc\(\)](#), [mat2D_create_col_ref\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), and [mat2D_offset2d\(\)](#).

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

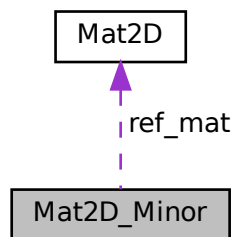
- [Matrix2D.h](#)

3.2 Mat2D_Minor Struct Reference

A minor "view" into a reference matrix.

```
#include <Matrix2D.h>
```

Collaboration diagram for Mat2D_Minor:



Public Attributes

- `size_t` [rows](#)
- `size_t` [cols](#)
- `size_t` [stride_r](#)
- `size_t *` [rows_list](#)
- `size_t *` [cols_list](#)
- [Mat2D](#) [ref_mat](#)

3.2.1 Detailed Description

A minor "view" into a reference matrix.

Represents a minor by excluding one row and one column of a reference matrix. The minor does not own the reference matrix data; instead it stores two index arrays (`rows_list`, `cols_list`) mapping minor coordinates to the reference matrix coordinates.

Memory ownership:

- `rows_list` and `cols_list` are heap-allocated by the minor allocators and must be freed with [mat2D_minor_free\(\)](#).
- `ref_mat.elements` is not owned by the minor and must not be freed by [mat2D_minor_free\(\)](#).

Definition at line 152 of file [Matrix2D.h](#).

3.2.2 Member Data Documentation

3.2.2.1 cols

```
size_t Mat2D_Minor::cols
```

Definition at line 154 of file [Matrix2D.h](#).

Referenced by [det_by_minors_first_col\(\)](#), [mat2D_det\(\)](#), [mat2D_det_2x2_mat_minor\(\)](#), [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_minor_alloc_fill_from_mat_minor\(\)](#), [mat2D_minor_det\(\)](#), and [mat2D_minor_print\(\)](#).

3.2.2.2 cols_list

```
size_t* Mat2D_Minor::cols_list
```

Definition at line 157 of file [Matrix2D.h](#).

Referenced by [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_minor_alloc_fill_from_mat_minor\(\)](#), and [mat2D_minor_free\(\)](#).

3.2.2.3 ref_mat

```
Mat2D Mat2D_Minor::ref_mat
```

Definition at line 158 of file [Matrix2D.h](#).

Referenced by [mat2D_minor_alloc_fill_from_mat\(\)](#), and [mat2D_minor_alloc_fill_from_mat_minor\(\)](#).

3.2.2.4 rows

```
size_t Mat2D_Minor::rows
```

Definition at line 153 of file [Matrix2D.h](#).

Referenced by [det_by_minors_first_col\(\)](#), [mat2D_det\(\)](#), [mat2D_det_2x2_mat_minor\(\)](#), [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_minor_alloc_fill_from_mat_minor\(\)](#), [mat2D_minor_det\(\)](#), and [mat2D_minor_print\(\)](#).

3.2.2.5 rows_list

```
size_t* Mat2D_Minor::rows_list
```

Definition at line 156 of file [Matrix2D.h](#).

Referenced by [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_minor_alloc_fill_from_mat_minor\(\)](#), and [mat2D_minor_free\(\)](#).

3.2.2.6 stride_r

```
size_t Mat2D_Minor::stride_r
```

Definition at line 155 of file [Matrix2D.h](#).

Referenced by [mat2D_minor_alloc_fill_from_mat\(\)](#), and [mat2D_minor_alloc_fill_from_mat_minor\(\)](#).

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

- [Matrix2D.h](#)

3.3 Mat2D_uint32 Struct Reference

Dense row-major matrix of `uint32_t`.

```
#include <Matrix2D.h>
```

Public Attributes

- `size_t` [rows](#)
- `size_t` [cols](#)
- `size_t` [stride_r](#)
- `uint32_t *` [elements](#)

3.3.1 Detailed Description

Dense row-major matrix of `uint32_t`.

Same layout rules as [Mat2D](#), but with `uint32_t` elements.

Definition at line 130 of file [Matrix2D.h](#).

3.3.2 Member Data Documentation

3.3.2.1 cols

```
size_t Mat2D_uint32::cols
```

Definition at line 132 of file [Matrix2D.h](#).

Referenced by [mat2D_alloc_uint32\(\)](#), [mat2D_fill_uint32\(\)](#), and [mat2D_offset2d_uint32\(\)](#).

3.3.2.2 elements

```
uint32_t* Mat2D_uint32::elements
```

Definition at line 134 of file [Matrix2D.h](#).

Referenced by [mat2D_alloc_uint32\(\)](#), and [mat2D_free_uint32\(\)](#).

3.3.2.3 rows

```
size_t Mat2D_uint32::rows
```

Definition at line 131 of file [Matrix2D.h](#).

Referenced by [mat2D_alloc_uint32\(\)](#), [mat2D_fill_uint32\(\)](#), and [mat2D_offset2d_uint32\(\)](#).

3.3.2.4 stride_r

```
size_t Mat2D_uint32::stride_r
```

Definition at line 133 of file [Matrix2D.h](#).

Referenced by [mat2D_alloc_uint32\(\)](#), and [mat2D_offset2d_uint32\(\)](#).

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

- [Matrix2D.h](#)

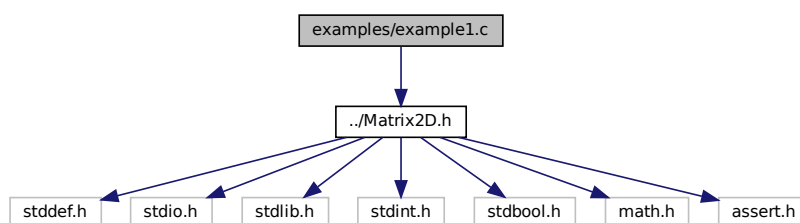
Chapter 4

File Documentation

4.1 examples/example1.c File Reference

```
#include "../Matrix2D.h"
```

Include dependency graph for example1.c:



Macros

- `#define` [MATRIX2D_IMPLEMENTATION](#)

Functions

- `int` [main](#) (void)

4.1.1 Macro Definition Documentation

4.1.1.1 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

Definition at line [1](#) of file [example1.c](#).

4.1.2 Function Documentation

4.1.2.1 main()

```
int main (
    void )
```

Definition at line 4 of file [example1.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_AT](#), [mat2D_dot\(\)](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), [MAT2D_PRINT](#), [mat2D_rand\(\)](#), [mat2D_SVD_thin\(\)](#), [mat2D_transpose\(\)](#), and [Mat2D::rows](#).

4.2 example1.c

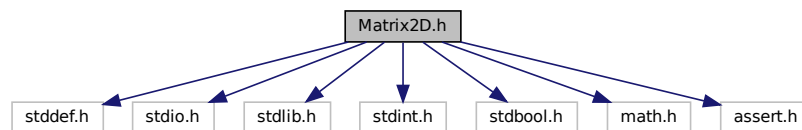
```
00001 #define MATRIX2D_IMPLEMENTATION
00002 #include "../Matrix2D.h"
00003
00004 int main(void)
00005 {
00006     int n = 4;
00007     int m = 5;
00008
00009     Mat2D A = mat2D_alloc(n, m);
00010     Mat2D U = mat2D_alloc(n, n);
00011     Mat2D S = mat2D_alloc(n, m);
00012     Mat2D V = mat2D_alloc(m, m);
00013     Mat2D VT = mat2D_alloc(m, m);
00014     Mat2D SV = mat2D_alloc(n, m);
00015     Mat2D USVT = mat2D_alloc(n, m);
00016     Mat2D init_vec_u = mat2D_alloc(U.rows, 1);
00017     Mat2D init_vec_v = mat2D_alloc(V.rows, 1);
00018
00019     mat2D_rand(init_vec_u, 0, 1);
00020     mat2D_rand(init_vec_v, 0, 1);
00021
00022     mat2D_fill(A, 0);
00023
00024     MAT2D_AT(A, 0, 0) = 1;
00025     MAT2D_AT(A, 0, 4) = 2;
00026     MAT2D_AT(A, 1, 2) = 3;
00027     MAT2D_AT(A, 3, 1) = 2;
00028
00029     mat2D_SVD_thin(A, U, S, V, init_vec_u, init_vec_v, 0);
00030
00031     MAT2D_PRINT(A);
00032     MAT2D_PRINT(U);
00033     MAT2D_PRINT(S);
00034     MAT2D_PRINT(V);
00035
00036     mat2D_transpose(VT, V);
00037
00038     mat2D_dot(SV, S, VT);
00039     mat2D_dot(USVT, U, SV);
00040
00041     MAT2D_PRINT(USVT);
00042
00043
00044     mat2D_free(A);
00045     mat2D_free(U);
00046     mat2D_free(S);
00047     mat2D_free(V);
00048     mat2D_free(SV);
00049     mat2D_free(USVT);
00050     mat2D_free(init_vec_u);
00051     mat2D_free(init_vec_v);
00052
00053     return 0;
00054 }
```

4.3 Matrix2D.h File Reference

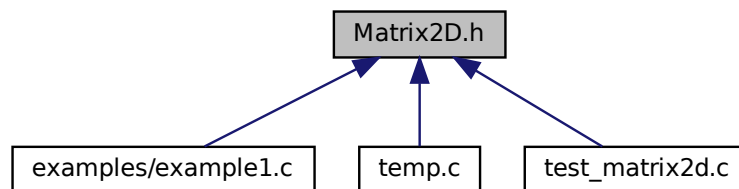
Lightweight 2D matrix helpers (double / uint32_t).

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <math.h>
#include <assert.h>
```

Include dependency graph for Matrix2D.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [Mat2D](#)
Dense row-major matrix of double.
- struct [Mat2D_uint32](#)
Dense row-major matrix of uint32_t.
- struct [Mat2D_Minor](#)
A minor "view" into a reference matrix.

Macros

- `#define MAT2D_MALLOC` `malloc`
Allocation function used by this library.
- `#define MAT2D_FREE` `free`
Deallocation function used by this library.
- `#define MAT2D_ASSERT` `assert`
Assertion macro used by this library for parameter validation.
- `#define MAT2D_AT(m, i, j)` `(m).elements[mat2D_offset2d((m), (i), (j))]`
Access element (i, j) of a [Mat2D](#) (0-based).
- `#define MAT2D_AT_UINT32(m, i, j)` `(m).elements[mat2D_offset2d_uint32((m), (i), (j))]`
Access element (i, j) of a [Mat2D_uint32](#) (0-based).
- `#define MAT2D_PI` `3.14159265358979323846`
- `#define MAT2D_EPS` `1e-15`
- `#define MAT2D_MAX_POWER_ITERATION` `100`
- `#define MAT2D_IS_ZERO(x)` `(fabs(x) < MAT2D_EPS)`
Test whether a floating-point value is "near zero".
- `#define MAT2D_MINOR_AT(mm, i, j)` `MAT2D_AT((mm).ref_mat, (mm).rows_list[i], (mm).cols_list[j])`
Access element (i, j) of a [Mat2D_Minor](#) (0-based).
- `#define MAT2D_PRINT(m)` `mat2D_print(m, #m, 0)`
Convenience macro to print a matrix with its variable name.
- `#define MAT2D_PRINT_AS_COL(m)` `mat2D_print_as_col(m, #m, 0)`
Convenience macro to print a matrix as a single column with its name.
- `#define MAT2D_MINOR_PRINT(mm)` `mat2D_minor_print(mm, #mm, 0)`
Convenience macro to print a minor with its variable name.
- `#define mat2D_normalize(m)` `mat2D_mult((m), 1.0 / mat2D_calc_norma((m)))`
Normalize a matrix in-place to unit Frobenius norm.
- `#define mat2D_normalize_inf(m)` `mat2D_mult((m), 1.0 / mat2D_calc_norma_inf((m)))`
- `#define mat2D_dprintDOUBLE(expr)` `printf(#expr " = %g\n", expr)`
- `#define mat2D_dprintSIZE_T(expr)` `printf(#expr " = %zu\n", expr)`
- `#define mat2D_dprintINT(expr)` `printf(#expr " = %d\n", expr)`

Enumerations

- enum `mat2D_upper_triangularize_flag` { `MAT2D_ONES_ON_DIAG` = 1 << 0 , `MAT2D_ROW_SWAPPING` = 1 << 1 }

Functions

- void `mat2D_add` (`Mat2D` dst, `Mat2D` a)
In-place addition: dst += a.
- void `mat2D_add_col_to_col` (`Mat2D` des, `size_t` des_col, `Mat2D` src, `size_t` src_col)
Add a source column into a destination column.
- void `mat2D_add_row_to_row` (`Mat2D` des, `size_t` des_row, `Mat2D` src, `size_t` src_row)
Add a source row into a destination row.
- void `mat2D_add_row_time_factor_to_row` (`Mat2D` m, `size_t` des_r, `size_t` src_r, double factor)
*Row operation: row(des_r) += factor * row(src_r).*
- `Mat2D` `mat2D_alloc` (`size_t` rows, `size_t` cols)
Allocate a rows-by-cols matrix of double.
- `Mat2D_uint32` `mat2D_alloc_uint32` (`size_t` rows, `size_t` cols)

- Allocate a rows-by-cols matrix of uint32_t.*

 - double [mat2D_calc_col_norma](#) ([Mat2D](#) m, size_t c)

Compute the Euclidean (L2) norm of a matrix column.

 - double [mat2D_calc_norma](#) ([Mat2D](#) m)

Compute the Frobenius norm of a matrix, $\sqrt{\text{sum}(m_{ij}^2)}$.

 - double [mat2D_calc_norma_inf](#) ([Mat2D](#) m)

Compute the maximum absolute element value of a matrix.

 - bool [mat2D_col_is_all_digit](#) ([Mat2D](#) m, double digit, size_t c)

Check if all elements of a column equal a given digit.

 - void [mat2D_copy](#) ([Mat2D](#) des, [Mat2D](#) src)

Copy all elements from src to des.

 - void [mat2D_copy_col_from_src_to_des](#) ([Mat2D](#) des, size_t des_col, [Mat2D](#) src, size_t src_col)

Copy a column from src into a column of des.

 - void [mat2D_copy_row_from_src_to_des](#) ([Mat2D](#) des, size_t des_row, [Mat2D](#) src, size_t src_row)

Copy a row from src into a row of des.

 - void [mat2D_copy_src_to_des_window](#) ([Mat2D](#) des, [Mat2D](#) src, size_t is, size_t js, size_t ie, size_t je)

Copy src into a window of des.

 - void [mat2D_copy_src_window_to_des](#) ([Mat2D](#) des, [Mat2D](#) src, size_t is, size_t js, size_t ie, size_t je)

Copy a rectangular window from src into des.

 - [Mat2D](#) [mat2D_create_col_ref](#) ([Mat2D](#) src, size_t c)

Create a non-owning column "view" into an existing matrix.

 - void [mat2D_cross](#) ([Mat2D](#) dst, [Mat2D](#) v1, [Mat2D](#) v2)

3D cross product: $\text{dst} = a \times b$ for 3x1 vectors.

 - void [mat2D_dot](#) ([Mat2D](#) dst, [Mat2D](#) a, [Mat2D](#) b)

*Matrix product: $\text{dst} = a * b$.*

 - double [mat2D_dot_product](#) ([Mat2D](#) v1, [Mat2D](#) v2)

Dot product between two vectors.

 - double [mat2D_det](#) ([Mat2D](#) m)

Determinant of a square matrix via Gaussian elimination.

 - double [mat2D_det_2x2_mat](#) ([Mat2D](#) m)

Determinant of a 2x2 matrix.

 - double [mat2D_det_2x2_mat_minor](#) ([Mat2D_Minor](#) mm)

Determinant of a 2x2 minor.

 - void [mat2D_eig_check](#) ([Mat2D](#) A, [Mat2D](#) eigenvalues, [Mat2D](#) eigenvectors, [Mat2D](#) res)

Check an eigen-decomposition by forming the residual $(A V - V \Lambda)$.

 - void [mat2D_eig_power_iteration](#) ([Mat2D](#) A, [Mat2D](#) eigenvalues, [Mat2D](#) eigenvectors, [Mat2D](#) init_vector, bool norm_inf_vectors)

Estimate eigenvalues/eigenvectors using repeated power iteration with deflation.

 - void [mat2D_fill](#) ([Mat2D](#) m, double x)

Fill all elements of a matrix of doubles with a scalar value.

 - void [mat2D_fill_sequence](#) ([Mat2D](#) m, double start, double step)

Fill a matrix with an arithmetic sequence laid out in row-major order.

 - void [mat2D_fill_uint32](#) ([Mat2D_uint32](#) m, uint32_t x)

Fill all elements of a matrix of uint32_t with a scalar value.

 - bool [mat2D_find_first_non_zero_value](#) ([Mat2D](#) m, size_t r, size_t *non_zero_col)

Find the first non-zero (per `MAT2D_EPS`) element in a row.

 - void [mat2D_free](#) ([Mat2D](#) m)

Free the buffer owned by a `Mat2D`.

 - void [mat2D_free_uint32](#) ([Mat2D_uint32](#) m)

Free the buffer owned by a `Mat2D_uint32`.

 - double [mat2D_inner_product](#) ([Mat2D](#) v)

- Compute the inner product of a vector with itself: $\text{dot}(v, v)$.

 - void `mat2D_invert` (`Mat2D` des, `Mat2D` src)

Invert a square matrix using Gauss-Jordan elimination.
- void `mat2D_LUP_decomposition_with_swap` (`Mat2D` src, `Mat2D` l, `Mat2D` p, `Mat2D` u)

Compute LUP decomposition: $P \cdot A = L \cdot U$ with L unit diagonal.
- void `mat2D_make_orthogonal_Gaussian_elimination` (`Mat2D` des, `Mat2D` A)

Attempt to build an orthogonal(ized) matrix from A using Gaussian elimination.
- void `mat2D_make_orthogonal_modified_Gram_Schmidt` (`Mat2D` des, `Mat2D` A)

Build an orthonormal basis using modified Gram-Schmidt.
- bool `mat2D_mat_is_all_digit` (`Mat2D` m, double digit)

Check if all elements of a matrix equal a given digit.
- `Mat2D_Minor` `mat2D_minor_alloc_fill_from_mat` (`Mat2D` ref_mat, size_t i, size_t j)

Allocate a minor view by excluding row i and column j of ref_mat.
- `Mat2D_Minor` `mat2D_minor_alloc_fill_from_mat_minor` (`Mat2D_Minor` ref_mm, size_t i, size_t j)

Allocate a nested minor view from an existing minor by excluding row i and column j of the minor.
- double `mat2D_minor_det` (`Mat2D_Minor` mm)

Determinant of a minor via recursive expansion by minors.
- void `mat2D_minor_free` (`Mat2D_Minor` mm)

Free the index arrays owned by a minor.
- void `mat2D_minor_print` (`Mat2D_Minor` mm, const char *name, size_t padding)

Print a minor matrix to stdout with a name and indentation padding.
- void `mat2D_mult` (`Mat2D` m, double factor)

In-place scalar multiplication: $m \mathrel{*=} \text{factor}$.
- void `mat2D_mult_row` (`Mat2D` m, size_t r, double factor)

In-place row scaling: $\text{row}(r) \mathrel{*=} \text{factor}$.
- size_t `mat2D_offset2d` (`Mat2D` m, size_t i, size_t j)

Compute the linear offset of element (i, j) in a `Mat2D`.
- size_t `mat2D_offset2d_uint32` (`Mat2D_uint32` m, size_t i, size_t j)

Compute the linear offset of element (i, j) in a `Mat2D_uint32`.
- void `mat2D_outer_product` (`Mat2D` des, `Mat2D` v)

Compute the outer product of a vector with itself: $\text{des} = v \cdot v^T$.
- int `mat2D_power_iterate` (`Mat2D` A, `Mat2D` v, double *lambda, double shift, bool norm_inf_v)

Approximate an eigenpair using (shifted) power iteration.
- void `mat2D_print` (`Mat2D` m, const char *name, size_t padding)

Print a matrix to stdout with a name and indentation padding.
- void `mat2D_print_as_col` (`Mat2D` m, const char *name, size_t padding)

Print a matrix as a flattened column vector to stdout.
- void `mat2D_rand` (`Mat2D` m, double low, double high)

Fill a matrix with pseudo-random doubles in [low, high].
- double `mat2D_rand_double` (void)

Return a pseudo-random double in the range [0, 1].
- size_t `mat2D_reduce` (`Mat2D` m)

Reduce a matrix in-place to reduced row echelon form (RREF) and return its rank.
- bool `mat2D_row_is_all_digit` (`Mat2D` m, double digit, size_t r)

Check if all elements of a row equal a given digit.
- void `mat2D_set_DCM_zyx` (`Mat2D` DCM, float yaw_deg, float pitch_deg, float roll_deg)

Build a 3x3 direction cosine matrix (DCM) from Z-Y-X Euler angles.
- void `mat2D_set_identity` (`Mat2D` m)

Set a square matrix to the identity matrix.
- void `mat2D_set_rot_mat_x` (`Mat2D` m, float angle_deg)

Set a 3x3 rotation matrix for rotation about the X-axis.

- void `mat2D_set_rot_mat_y` (`Mat2D` m, float angle_deg)
Set a 3x3 rotation matrix for rotation about the Y-axis.
- void `mat2D_set_rot_mat_z` (`Mat2D` m, float angle_deg)
Set a 3x3 rotation matrix for rotation about the Z-axis.
- void `mat2D_shift` (`Mat2D` m, double shift)
Add a scalar shift to the diagonal: $m[i,i] += \text{shift}$.
- void `mat2D_solve_linear_sys_LUP_decomposition` (`Mat2D` A, `Mat2D` x, `Mat2D` B)
Solve the linear system $Ax = B$ using an LUP-based approach.
- void `mat2D_sub` (`Mat2D` dst, `Mat2D` a)
In-place subtraction: $\text{dst} -= a$.
- void `mat2D_sub_col_to_col` (`Mat2D` des, size_t des_col, `Mat2D` src, size_t src_col)
Subtract a source column from a destination column.
- void `mat2D_sub_row_to_row` (`Mat2D` des, size_t des_row, `Mat2D` src, size_t src_row)
Subtract a source row from a destination row.
- void `mat2D_sub_row_time_factor_to_row` (`Mat2D` m, size_t des_r, size_t src_r, double factor)
*Row operation: $\text{row}(\text{des_r}) -= \text{factor} * \text{row}(\text{src_r})$.*
- void `mat2D_SVD_full` (`Mat2D` A, `Mat2D` U, `Mat2D` S, `Mat2D` V, `Mat2D` init_vec_u, `Mat2D` init_vec_v, bool return_v_transpose)
Compute a "full" SVD by post-orthogonalizing the thin result.
- void `mat2D_SVD_thin` (`Mat2D` A, `Mat2D` U, `Mat2D` S, `Mat2D` V, `Mat2D` init_vec_u, `Mat2D` init_vec_v, bool return_v_transpose)
Compute an SVD using eigen-decomposition + power iteration (educational).
- void `mat2D_swap_rows` (`Mat2D` m, size_t r1, size_t r2)
Swap two rows of a matrix in-place.
- void `mat2D_transpose` (`Mat2D` des, `Mat2D` src)
Transpose a matrix: $\text{des} = \text{src}^T$.
- double `mat2D_upper_triangulate` (`Mat2D` m, uint8_t flags)
Transform a matrix to (row-echelon) upper triangular form by forward elimination.

4.3.1 Detailed Description

Lightweight 2D matrix helpers (double / uint32_t).

This single-header module provides small utilities for dense row-major matrices:

- Allocation/free for `Mat2D` (double) and `Mat2D_uint32`
- Basic arithmetic and row/column operations
- Matrix multiplication, transpose, dot and cross products
- Determinant and inversion (Gaussian / Gauss-Jordan style)
- A simple LUP decomposition helper and a linear system solver
- Rotation matrix helpers (X/Y/Z) and a Z-Y-X DCM builder (as implemented)
- "Minor" views (index lists into a reference matrix) for educational determinant-by-minors computation

Storage model

- Matrices are dense and row-major (C-style).

- Element at row *i* and column *j* (0-based) is: `elements[i * stride_r + j]`
- For matrices created by `mat2D_alloc()`, `stride_r == cols`.

Usage

- In exactly one translation unit, define `MATRIX2D_IMPLEMENTATION` before including this header to compile the implementation.
- In all other files, include the header without that macro to get declarations only.

Example: `#define MATRIX2D_IMPLEMENTATION #include "matrix2d.h"`

Notes and limitations

- This one-file library is heavily inspired by Tsoding's `nn.h` implementation of matrix creation and operations: <https://github.com/tsoding/nn.h> and the video: <https://youtu.be/L1TbWe8b4V0c?list=PLpM-Dvs8t0VZPZKggcql-MmjaBdZKeDMw>
- All APIs assume the caller provides correctly-sized destination matrices. Shape mismatches are checked with `MAT2D_ASSERT` in many routines.
- This library does not try to be numerically robust:
 - Pivoting is limited (only performed when a pivot is “near zero” per `MAT2D_EPS` in several routines).
 - Ill-conditioned matrices may produce inaccurate determinants/inverses.
- RNG uses `C rand()`; it is not cryptographically secure.

Warning

Numerical stability and correctness

- `mat2D_minor_det()` is factorial-time and is intended only for very small matrices (educational use).
- `mat2D_invert()` uses Gauss-Jordan elimination and may be unstable for ill-conditioned matrices. Consider a more robust decomposition for production use (full pivoting / QR / SVD).
- Several routines do not guard against aliasing (e.g. `dst == a`). Unless documented otherwise, assume inputs and outputs must not overlap.

Definition in file [Matrix2D.h](#).

4.3.2 Macro Definition Documentation

4.3.2.1 MAT2D_ASSERT

```
#define MAT2D_ASSERT assert
```

Assertion macro used by this library for parameter validation.

Defaults to `assert()`. Override by defining `MAT2D_ASSERT` before including this header to customize validation behavior.

Definition at line 101 of file [Matrix2D.h](#).

4.3.2.2 MAT2D_AT

```
#define MAT2D_AT(  
    m,  
    i,  
    j ) (m).elements[mat2D_offset2d((m), (i), (j))]
```

Access element (i, j) of a [Mat2D](#) (0-based).

Expands to row-major indexing using stride_r: (m).elements[(i) * (m).stride_r + (j)]

Warning

In the “fast” configuration this macro performs no bounds checking.

Definition at line 179 of file [Matrix2D.h](#).

4.3.2.3 MAT2D_AT_UINT32

```
#define MAT2D_AT_UINT32(  
    m,  
    i,  
    j ) (m).elements[mat2D_offset2d_uint32((m), (i), (j))]
```

Access element (i, j) of a [Mat2D_uint32](#) (0-based).

Warning

In the “fast” configuration this macro performs no bounds checking.

Definition at line 180 of file [Matrix2D.h](#).

4.3.2.4 mat2D_dprintDOUBLE

```
#define mat2D_dprintDOUBLE(  
    expr ) printf(#expr " = %g\n", expr)
```

Definition at line 243 of file [Matrix2D.h](#).

4.3.2.5 mat2D_dprintINT

```
#define mat2D_dprintINT(  
    expr ) printf(#expr " = %d\n", expr)
```

Definition at line 247 of file [Matrix2D.h](#).

4.3.2.6 mat2D_dprintSIZE_T

```
#define mat2D_dprintSIZE_T(  
    expr ) printf(#expr " = %zu\n", expr)
```

Definition at line 245 of file [Matrix2D.h](#).

4.3.2.7 MAT2D_EPS

```
#define MAT2D_EPS 1e-15
```

Definition at line 188 of file [Matrix2D.h](#).

4.3.2.8 MAT2D_FREE

```
#define MAT2D_FREE free
```

Deallocation function used by this library.

Defaults to free(). Override by defining MAT2D_FREE before including this header to match a custom allocator.

Definition at line 88 of file [Matrix2D.h](#).

4.3.2.9 MAT2D_IS_ZERO

```
#define MAT2D_IS_ZERO(  
    x ) (fabs(x) < MAT2D_EPS)
```

Test whether a floating-point value is “near zero”.

Uses `fabs(x) < MAT2D_EPS`.

Definition at line 200 of file [Matrix2D.h](#).

4.3.2.10 MAT2D_MALLOC

```
#define MAT2D_MALLOC malloc
```

Allocation function used by this library.

Defaults to malloc(). Override by defining MAT2D_MALLOC before including this header to use a custom allocator.

Definition at line 76 of file [Matrix2D.h](#).

4.3.2.11 MAT2D_MAX_POWER_ITERATION

```
#define MAT2D_MAX_POWER_ITERATION 100
```

Definition at line 190 of file [Matrix2D.h](#).

4.3.2.12 MAT2D_MINOR_AT

```
#define MAT2D_MINOR_AT(  
    mm,  
    i,  
    j ) MAT2D_AT((mm).ref_mat, (mm).rows_list[i], (mm).cols_list[j])
```

Access element (i, j) of a [Mat2D_Minor](#) (0-based).

Dereferences into the underlying reference matrix using rows_list/cols_list.

Definition at line 209 of file [Matrix2D.h](#).

4.3.2.13 MAT2D_MINOR_PRINT

```
#define MAT2D_MINOR_PRINT(  
    mm ) mat2D_minor_print(mm, #mm, 0)
```

Convenience macro to print a minor with its variable name.

Definition at line 227 of file [Matrix2D.h](#).

4.3.2.14 mat2D_normalize

```
#define mat2D_normalize(  
    m ) mat2D_mult((m), 1.0 / mat2D_calc_norma((m)))
```

Normalize a matrix in-place to unit Frobenius norm.

Equivalent to: `m *= 1.0 / mat2D_calc_norma(m)`

Warning

If the Frobenius norm is 0, this performs a division by zero.

Definition at line 239 of file [Matrix2D.h](#).

4.3.2.15 mat2D_normalize_inf

```
#define mat2D_normalize_inf(  
    m ) mat2D_mult ( (m), 1.0 / mat2D_calc_norma_inf ( (m) ) )
```

Definition at line 241 of file [Matrix2D.h](#).

4.3.2.16 MAT2D_PI

```
#define MAT2D_PI 3.14159265358979323846
```

Definition at line 186 of file [Matrix2D.h](#).

4.3.2.17 MAT2D_PRINT

```
#define MAT2D_PRINT(  
    m ) mat2D_print (m, #m, 0)
```

Convenience macro to print a matrix with its variable name.

Definition at line 215 of file [Matrix2D.h](#).

4.3.2.18 MAT2D_PRINT_AS_COL

```
#define MAT2D_PRINT_AS_COL(  
    m ) mat2D_print_as_col (m, #m, 0)
```

Convenience macro to print a matrix as a single column with its name.

Definition at line 221 of file [Matrix2D.h](#).

4.3.3 Enumeration Type Documentation

4.3.3.1 mat2D_upper_triangulate_flag

```
enum mat2D_upper_triangulate_flag
```

Enumerator

MAT2D_ONES_ON_DIAG	
MAT2D_ROW_SWAPPING	

Definition at line 249 of file [Matrix2D.h](#).

4.3.4 Function Documentation

4.3.4.1 `mat2D_add()`

```
void mat2D_add (
    Mat2D dst,
    Mat2D a )
```

In-place addition: `dst += a`.

Parameters

<i>dst</i>	Destination matrix to be incremented.
<i>a</i>	Summand of same shape as <i>dst</i> .

Precondition

`dst` and `a` have identical shape.

Definition at line 351 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_alloc_fill_copy_add_sub\(\)](#).

4.3.4.2 `mat2D_add_col_to_col()`

```
void mat2D_add_col_to_col (
    Mat2D des,
    size_t des_col,
    Mat2D src,
    size_t src_col )
```

Add a source column into a destination column.

Performs: `des[:, des_col] += src[:, src_col]`

Parameters

<i>des</i>	Destination matrix (same row count as <i>src</i>).
<i>des_col</i>	Column index in destination.
<i>src</i>	Source matrix.
<i>src_col</i>	Column index in source.

Definition at line 372 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_row_col_ops_and_scaling\(\)](#).

4.3.4.3 mat2D_add_row_time_factor_to_row()

```
void mat2D_add_row_time_factor_to_row (
    Mat2D m,
    size_t des_r,
    size_t src_r,
    double factor )
```

Row operation: $\text{row}(\text{des_r}) += \text{factor} * \text{row}(\text{src_r})$.

Parameters

<i>m</i>	Matrix.
<i>des↔ _r</i>	Destination row index.
<i>src↔ _r</i>	Source row index.
<i>factor</i>	Scalar multiplier.

Warning

Indices are not bounds-checked in this routine.

Definition at line 416 of file [Matrix2D.h](#).

References [Mat2D::cols](#), and [MAT2D_AT](#).

Referenced by [test_row_col_ops_and_scaling\(\)](#).

4.3.4.4 mat2D_add_row_to_row()

```
void mat2D_add_row_to_row (
    Mat2D des,
    size_t des_row,
    Mat2D src,
    size_t src_row )
```

Add a source row into a destination row.

Performs: $\text{des}[\text{des_row}, :] += \text{src}[\text{src_row}, :]$

Parameters

<i>des</i>	Destination matrix (same number of columns as <i>src</i>).
<i>des_row</i>	Row index in destination.
<i>src</i>	Source matrix.
<i>src_row</i>	Row index in source.

Precondition

`des.cols == src.cols`

Definition at line 396 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_row_col_ops_and_scaling\(\)](#).

4.3.4.5 `mat2D_alloc()`

```
Mat2D mat2D_alloc (
    size_t rows,
    size_t cols )
```

Allocate a rows-by-cols matrix of double.

Parameters

<i>rows</i>	Number of rows. Must be > 0 .
<i>cols</i>	Number of columns. Must be > 0 .

Returns

A [Mat2D](#) owning a contiguous buffer of `rows * cols` elements.

Postcondition

The returned matrix has `stride_r == cols`.

The returned matrix must be released with [mat2D_free\(\)](#).

Warning

This function asserts allocation success via [MAT2D_ASSERT](#).

Definition at line 435 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [Mat2D::elements](#), [MAT2D_ASSERT](#), [MAT2D_MALLOC](#), [Mat2D::rows](#), and [Mat2D::stride_r](#).

Referenced by [assert_inverse_identity_both_sides\(\)](#), [main\(\)](#), [mat2D_det\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_invert\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_power_iterate\(\)](#), [mat2D_set_DCM_zyx\(\)](#), [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), [mat2D_SVD_full\(\)](#), [mat2D_SVD_thin\(\)](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_copy_row_and_col_helpers\(\)](#), [test_copy_windows\(\)](#), [test_DCM_zyx_matches_product\(\)](#), [test_det_2x2_and_upper_triangular_sign\(\)](#), [test_det_and_minor_det_agree_3x3\(\)](#), [test_det_early_zero_row_and_zero_col_paths\(\)](#), [test_deterministic_fuzz_loop\(\)](#), [test_dot_product_and_vector_variants\(\)](#), [test_dot_product_matrix_multiply\(\)](#), [test_invert\(\)](#), [test_LUP_decomposition_identity_P_no_swap_case\(\)](#), [test_LUP_decomposition_swap_rows\(\)](#), [test_mat_is_all_digit\(\)](#), [test_minor_det_matches_gauss_4x4_known\(\)](#), [test_non_contiguous_stride_views\(\)](#), [test_norms_and_normalize\(\)](#), [test_offset2d_and_stride\(\)](#), [test_outer_product_and_cross\(\)](#), [test_outer_product_row_vector_path\(\)](#), [test_power_iterate_and_eig_helpers\(\)](#), [test_rand_range\(\)](#), [test_reduce_rank\(\)](#), [test_rotation_matrices_orthonormal\(\)](#), [test_row_col_ops_and_scaling\(\)](#), [test_shift_and_identity\(\)](#), [test_solve_linear_system_LUP\(\)](#), and [test_transpose\(\)](#).

4.3.4.6 mat2D_alloc_uint32()

```
Mat2D_uint32 mat2D_alloc_uint32 (
    size_t rows,
    size_t cols )
```

Allocate a rows-by-cols matrix of uint32_t.

Parameters

<i>rows</i>	Number of rows. Must be > 0.
<i>cols</i>	Number of columns. Must be > 0.

Returns

A [Mat2D_uint32](#) owning a contiguous buffer of rows * cols elements.

Postcondition

The returned matrix has stride_r == cols.

The returned matrix must be released with [mat2D_free_uint32\(\)](#).

Warning

This function asserts allocation success via MAT2D_ASSERT.

Definition at line 459 of file [Matrix2D.h](#).

References [Mat2D_uint32::cols](#), [Mat2D_uint32::elements](#), [MAT2D_ASSERT](#), [MAT2D_MALLOC](#), [Mat2D_uint32::rows](#), and [Mat2D_uint32::stride_r](#).

Referenced by [test_uint32_alloc_fill_and_at\(\)](#).

4.3.4.7 mat2D_calc_col_norma()

```
double mat2D_calc_col_norma (
    Mat2D m,
    size_t c )
```

Compute the Euclidean (L2) norm of a matrix column.

Parameters

<i>m</i>	Matrix.
<i>c</i>	Column index.

Returns

$(\sqrt{\sum_{i=0}^{m.rows-1} m_{i,c}^2})$.

Precondition

$c < m.cols$

Definition at line 480 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#).

4.3.4.8 mat2D_calc_norma()

```
double mat2D_calc_norma (
    Mat2D m )
```

Compute the Frobenius norm of a matrix, $\sqrt{\sum(m_{ij}^2)}$.

Parameters

<i>m</i>	Matrix.
----------	---------

Returns

Frobenius norm.

Definition at line 496 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_power_iterate\(\)](#), and [test_norms_and_normalize\(\)](#).

4.3.4.9 mat2D_calc_norma_inf()

```
double mat2D_calc_norma_inf (
    Mat2D m )
```

Compute the maximum absolute element value of a matrix.

Parameters

<i>m</i>	Matrix.
----------	---------

Returns

The element-wise maximum: $(\max_{i,j} |m_{ij}|)$.

Note

Despite the name, this is not the induced matrix infinity norm (maximum row sum). It is the max-absolute-entry metric.

Definition at line 517 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_eig_power_iteration\(\)](#), [mat2D_power_iterate\(\)](#), [test_dot_product_and_vector_variants\(\)](#), [test_norms_and_normalize\(\)](#), and [test_power_iterate_and_eig_helpers\(\)](#).

4.3.4.10 mat2D_col_is_all_digit()

```
bool mat2D_col_is_all_digit (
    Mat2D m,
    double digit,
    size_t c )
```

Check if all elements of a column equal a given digit.

Parameters

<i>m</i>	Matrix.
<i>digit</i>	Value to compare.
<i>c</i>	Column index.

Returns

true if every element equals digit, false otherwise.

Warning

Uses exact floating-point equality.

Definition at line 541 of file [Matrix2D.h](#).

References [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_det\(\)](#), and [test_det_early_zero_row_and_zero_col_paths\(\)](#).

4.3.4.11 mat2D_copy()

```
void mat2D_copy (
    Mat2D des,
    Mat2D src )
```

Copy all elements from src to des.

Parameters

<i>des</i>	Destination matrix.
<i>src</i>	Source matrix.

Precondition

Shapes match.

des and src have identical shape.

Definition at line 559 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_det\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_LUP_decomposition_with_swap\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_power_iterate\(\)](#), [mat2D_SVD_full\(\)](#), [mat2D_SVD_thin\(\)](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_det_2x2_and_upper_triangulate_sign\(\)](#), [test_deterministic_fuzz_loop\(\)](#), and [test_non_contiguous_stride_views\(\)](#).

4.3.4.12 mat2D_copy_col_from_src_to_des()

```
void mat2D_copy_col_from_src_to_des (
    Mat2D des,
    size_t des_col,
    Mat2D src,
    size_t src_col )
```

Copy a column from src into a column of des.

Parameters

<i>des</i>	Destination matrix (same row count as src).
<i>des_col</i>	Column index in destination.
<i>src</i>	Source matrix.
<i>src_col</i>	Column index in source.

Precondition

des.rows == src.rows

des_col < des.cols and src_col < src.cols

Definition at line 581 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_SVD_thin\(\)](#), and [test_copy_row_and_col_helpers\(\)](#).

4.3.4.13 mat2D_copy_row_from_src_to_des()

```
void mat2D_copy_row_from_src_to_des (
    Mat2D des,
    size_t des_row,
    Mat2D src,
    size_t src_row )
```

Copy a row from `src` into a row of `des`.

Parameters

<i>des</i>	Destination matrix (same number of columns as <code>src</code>).
<i>des_row</i>	Row index in destination.
<i>src</i>	Source matrix.
<i>src_row</i>	Row index in source.

Precondition

`des.cols == src.cols`

Definition at line 601 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_copy_row_and_col_helpers\(\)](#).

4.3.4.14 mat2D_copy_src_to_des_window()

```
void mat2D_copy_src_to_des_window (
    Mat2D des,
    Mat2D src,
    size_t is,
    size_t js,
    size_t ie,
    size_t je )
```

Copy `src` into a window of `des`.

Copies the entire `src` matrix into `des` at the rectangular region: rows `[is, ie]` and columns `[js, je]` (inclusive).

Parameters

<i>des</i>	Destination matrix.
<i>src</i>	Source matrix copied into the destination window.
<i>is</i>	Start row index in destination (inclusive).
<i>js</i>	Start column index in destination (inclusive).
<i>ie</i>	End row index in destination (inclusive).
<i>je</i>	End column index in destination (inclusive).

Precondition

$(je - js + 1) == src.cols$ and $(ie - is + 1) == src.rows$.

Definition at line 628 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_eig_power_iteration\(\)](#), [mat2D_invert\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), and [test_copy_windows\(\)](#).

4.3.4.15 mat2D_copy_src_window_to_des()

```
void mat2D_copy_src_window_to_des (
    Mat2D des,
    Mat2D src,
    size_t is,
    size_t js,
    size_t ie,
    size_t je )
```

Copy a rectangular window from src into des.

Parameters

<i>des</i>	Destination matrix. Must have size $(ie - is + 1) \times (je - js + 1)$.
<i>src</i>	Source matrix.
<i>is</i>	Start row index in src (inclusive).
<i>js</i>	Start column index in src (inclusive).
<i>ie</i>	End row index in src (inclusive).
<i>je</i>	End column index in src (inclusive).

Precondition

$0 \leq is \leq ie < src.rows$, $0 \leq js \leq je < src.cols$.

Definition at line 653 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_invert\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), and [test_copy_windows\(\)](#).

4.3.4.16 `mat2D_create_col_ref()`

```
Mat2D mat2D_create_col_ref (
    Mat2D src,
    size_t c )
```

Create a non-owning column "view" into an existing matrix.

Returns a [Mat2D](#) with shape $(src.rows \times 1)$ whose elements pointer refers to the first element of column `c` in `src`. The returned view preserves `src.stride_r` and therefore works for both contiguous and strided matrices.

Parameters

<code>src</code>	Source matrix.
<code>c</code>	Column index in <code>src</code> .

Returns

A shallow [Mat2D](#) view (does not allocate, does not own memory).

Precondition

`c < src.cols`

Warning

The returned matrix aliases `src`: modifying it modifies `src`.

The returned view is invalid after `src.elements` is freed/reallocated.

Definition at line 684 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [Mat2D::rows](#), and [Mat2D::stride_r](#).

Referenced by [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#).

4.3.4.17 `mat2D_cross()`

```
void mat2D_cross (
    Mat2D dst,
    Mat2D v1,
    Mat2D v2 )
```

3D cross product: $dst = a \times b$ for 3x1 vectors.

Parameters

<code>dst</code>	3x1 destination vector.
<code>a</code>	3x1 input vector.
<code>b</code>	3x1 input vector.

Precondition

All matrices have shape 3x1.

Definition at line 703 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_outer_product_and_cross\(\)](#).

4.3.4.18 mat2D_det()

```
double mat2D_det (
    Mat2D m )
```

Determinant of a square matrix via Gaussian elimination.

Parameters

<i>m</i>	Square matrix.
----------	----------------

Returns

det(m).

Copies *m* internally, transforms the copy to upper triangular form, and returns the product of diagonal elements adjusted by the row-swap factor.

Warning

The early “all-zero row/column” check uses exact comparisons to 0.
Limited pivoting may cause poor numerical results for some inputs.

Definition at line 792 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [Mat2D_Minor::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_col_is_all_digit\(\)](#), [mat2D_copy\(\)](#), [mat2D_det_2x2_mat_minor\(\)](#), [mat2D_free\(\)](#), [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_minor_det\(\)](#), [mat2D_minor_free\(\)](#), [mat2D_row_is_all_digit\(\)](#), [MAT2D_ROW_SWAPPING](#), [mat2D_upper_triangulate\(\)](#), [Mat2D::rows](#), and [Mat2D_Minor::rows](#).

Referenced by [assert_permutation_matrix\(\)](#), [test_det_2x2_and_upper_triangulate_sign\(\)](#), [test_det_and_minor_det_agree_3x3\(\)](#), [test_det_early_zero_row_and_zero_col_paths\(\)](#), [test_deterministic_fuzz_loop\(\)](#), [test_invert\(\)](#), [test_minor_det_matches_gauss_4x4_k](#), and [test_rotation_matrices_orthonormal\(\)](#).

4.3.4.19 mat2D_det_2x2_mat()

```
double mat2D_det_2x2_mat (
    Mat2D m )
```

Determinant of a 2x2 matrix.

Parameters

<i>m</i>	Matrix (must be 2x2).
----------	-----------------------

Returns

$\det(m) = m_{00} * m_{11} - m_{01} * m_{10}$.

Definition at line 844 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [det_by_minors_first_col\(\)](#), and [test_det_2x2_and_upper_triangulate_sign\(\)](#).

4.3.4.20 mat2D_det_2x2_mat_minor()

```
double mat2D_det_2x2_mat_minor (
    Mat2D\_Minor mm )
```

Determinant of a 2x2 minor.

Parameters

<i>mm</i>	Minor (must be 2x2).
-----------	----------------------

Returns

$\det(mm)$.

Definition at line 855 of file [Matrix2D.h](#).

References [Mat2D_Minor::cols](#), [MAT2D_ASSERT](#), [MAT2D_MINOR_AT](#), and [Mat2D_Minor::rows](#).

Referenced by [det_by_minors_first_col\(\)](#), [mat2D_det\(\)](#), and [mat2D_minor_det\(\)](#).

4.3.4.21 mat2D_dot()

```
void mat2D_dot (
    Mat2D dst,
    Mat2D a,
    Mat2D b )
```

Matrix product: $dst = a * b$.

Parameters

<i>dst</i>	Destination matrix (size a.rows x b.cols).
<i>a</i>	Left matrix (size a.rows x a.cols).
<i>b</i>	Right matrix (size a.cols x b.cols).

Precondition

`a.cols == b.rows`
`dst.rows == a.rows`
`dst.cols == b.cols`

Postcondition

`dst` is fully overwritten.

Warning

`dst` must not alias `a` or `b` (overlap is not handled).

Definition at line 729 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [assert_inverse_identity_both_sides\(\)](#), [main\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), [mat2D_power_iterate\(\)](#), [mat2D_set_DCM_zyx\(\)](#), [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), [mat2D_SVD_thin\(\)](#), [test_DCM_zyx_matches_product\(\)](#), [test_deterministic_fuzz_loop\(\)](#), [test_dot_product_matrix_multiply\(\)](#), [test_LUP_decomposition_identity\(\)](#), [test_LUP_decomposition_swap_required_case\(\)](#), [test_non_contiguous_stride_views\(\)](#), [test_rotation_matrices_orthonormal\(\)](#), and [test_solve_linear_system_LUP\(\)](#).

4.3.4.22 mat2D_dot_product()

```
double mat2D_dot_product (
    Mat2D v1,
    Mat2D v2 )
```

Dot product between two vectors.

Parameters

<i>a</i>	Vector (shape <code>n x 1</code> or <code>1 x n</code>).
<i>b</i>	Vector (same shape as <code>a</code>).

Returns

The scalar dot product sum.

Precondition

$a.rows == b.rows$ and $a.cols == b.cols$
 $(a.cols == 1 \ \&\& \ b.cols == 1) \ || \ (a.rows == 1 \ \&\& \ b.rows == 1)$

Definition at line 758 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_power_iterate\(\)](#), and [test_dot_product_and_vector_varia](#)

4.3.4.23 mat2D_eig_check()

```
void mat2D_eig_check (
    Mat2D A,
    Mat2D eigenvalues,
    Mat2D eigenvectors,
    Mat2D res )
```

Check an eigen-decomposition by forming the residual $(A V - V \backslash \text{Lambda})$.

Parameters

<i>A</i>	Square matrix (N x N).
<i>eigenvalues</i>	Diagonal matrix ($\backslash \text{Lambda}$) (N x N).
<i>eigenvectors</i>	Matrix of eigenvectors (V) (N x N), typically with eigenvectors stored as columns.
<i>res</i>	Destination matrix (N x N) receiving the residual.

Postcondition

res is overwritten with $(A V - V \backslash \text{Lambda})$.

Precondition

All inputs are N x N and shapes match.

Definition at line 874 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_copy\(\)](#), [mat2D_dot\(\)](#), [mat2D_free\(\)](#), [mat2D_mult\(\)](#), [mat2D_sub\(\)](#), [Mat2D::rows](#), and [Mat2D::stride_r](#).

Referenced by [test_power_iterate_and_eig_helpers\(\)](#).

4.3.4.24 mat2D_eig_power_iteration()

```
void mat2D_eig_power_iteration (
    Mat2D A,
    Mat2D eigenvalues,
    Mat2D eigenvectors,
    Mat2D init_vector,
    bool norm_inf_vectors )
```

Estimate eigenvalues/eigenvectors using repeated power iteration with deflation.

Repeatedly applies [mat2D_power_iterate\(\)](#) to estimate an eigenpair of the current matrix B, stores it into *eigenvalues* and *eigenvectors*, then deflates B by subtracting $(\lambda v v^T)$.

The vector *init_vector* is copied into each eigenvector slot as the initial guess before running iteration.

Parameters

	<i>A</i>	Input square matrix (N x N).
out	<i>eigenvalues</i>	Destination (N x N) written as a diagonal matrix.
out	<i>eigenvectors</i>	Destination (N x N) whose columns are the estimated eigenvectors.
	<i>init_vector</i>	Initial guess (N x 1), must have non-zero norm.
	<i>norm_inf_vectors</i>	If true, each output eigenvector column is normalized by mat2D_normalize_inf() .

Warning

This implementation is primarily educational and makes strong assumptions; it may fail or be inaccurate for matrices that do not satisfy the power-iteration convergence conditions.

Precondition

A is square; eigenvalues/eigenvectors are N x N; *init_vector* is N x 1.

Conditions:

- The eigenvectors must form an orthonormal basis
- The largest eigenvalue must be positive and unique

Definition at line 939 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_calc_norma_inf\(\)](#), [mat2D_copy\(\)](#), [mat2D_copy_src_to_des_window\(\)](#), [mat2D_free\(\)](#), [mat2D_mult\(\)](#), [mat2D_normalize_inf](#), [mat2D_outer_product\(\)](#), [mat2D_power_iterate\(\)](#), [mat2D_set_identity\(\)](#), [mat2D_sub\(\)](#), [Mat2D::rows](#), and [Mat2D::stride_r](#).

Referenced by [mat2D_SVD_thin\(\)](#), and [test_power_iterate_and_eig_helpers\(\)](#).

4.3.4.25 mat2D_fill()

```
void mat2D_fill (
    Mat2D m,
    double x )
```

Fill all elements of a matrix of doubles with a scalar value.

Parameters

<i>m</i>	Matrix to fill.
<i>x</i>	Value to assign to every element.

Definition at line 999 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#), [mat2D_LUP_decomposition_with_swap\(\)](#), [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), [mat2D_SVD_thin\(\)](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_copy_row_and_col_helpers\(\)](#), [test_copy_windows\(\)](#), [test_mat_is_all_digit\(\)](#), [test_outer_product_and_cross\(\)](#), and [test_power_iterate_and_eig_helpers\(\)](#).

4.3.4.26 mat2D_fill_sequence()

```
void mat2D_fill_sequence (
    Mat2D m,
    double start,
    double step )
```

Fill a matrix with an arithmetic sequence laid out in row-major order.

Parameters

<i>m</i>	Matrix to fill.
<i>start</i>	First value in the sequence.
<i>step</i>	Increment between consecutive elements.

Element at linear index *k* gets value $start + step * k$.

Definition at line 1015 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), [mat2D_offset2d\(\)](#), and [Mat2D::rows](#).

Referenced by [test_copy_row_and_col_helpers\(\)](#), [test_offset2d_and_stride\(\)](#), and [test_row_col_ops_and_scaling\(\)](#).

4.3.4.27 mat2D_fill_uint32()

```
void mat2D_fill_uint32 (
    Mat2D\_uint32 m,
    uint32_t x )
```

Fill all elements of a matrix of `uint32_t` with a scalar value.

Parameters

<i>m</i>	Matrix to fill.
<i>x</i>	Value to assign to every element.

Definition at line 1028 of file [Matrix2D.h](#).

References [Mat2D_uint32::cols](#), [MAT2D_AT_UINT32](#), and [Mat2D_uint32::rows](#).

Referenced by [test_uint32_alloc_fill_and_at\(\)](#).

4.3.4.28 mat2D_find_first_non_zero_value()

```
bool mat2D_find_first_non_zero_value (
    Mat2D m,
    size_t r,
    size_t * non_zero_col )
```

Find the first non-zero (per MAT2D_EPS) element in a row.

Parameters

	<i>m</i>	Matrix to search.
	<i>r</i>	Row index to search (0-based).
out	<i>non_zero_col</i>	On success, receives the column index of the first element in row <i>r</i> such that !MAT2D_IS_ZERO(value).

Returns

true if a non-zero element was found, false if the row is all zeros (within MAT2D_EPS).

Note

Scans columns from 0 to m.cols-1 (left to right).

Definition at line 1049 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [MAT2D_IS_ZERO](#).

Referenced by [mat2D_reduce\(\)](#).

4.3.4.29 mat2D_free()

```
void mat2D_free (
    Mat2D m )
```

Free the buffer owned by a [Mat2D](#).

Parameters

<i>m</i>	Matrix whose elements were allocated via MAT2D_MALLOC.
----------	--

Note

This does not modify `m` (it is passed by value).

It is safe to call with `m.elements == NULL`.

Definition at line 1068 of file [Matrix2D.h](#).

References [Mat2D::elements](#), and [MAT2D_FREE](#).

Referenced by [assert_inverse_identity_both_sides\(\)](#), [main\(\)](#), [mat2D_det\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_invert\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_power_iterate\(\)](#), [mat2D_set_DCM_zyx\(\)](#), [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), [mat2D_SVD_full\(\)](#), [mat2D_SVD_thin\(\)](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_copy_row_and_col_helpers\(\)](#), [test_copy_windows\(\)](#), [test_DCM_zyx_matches_product\(\)](#), [test_det_2x2_and_upper_triangulate_sign\(\)](#), [test_det_and_minor_det_agree_3x3\(\)](#), [test_det_early_zero_row_and_zero_col_paths\(\)](#), [test_deterministic_fuzz_loop\(\)](#), [test_dot_product_and_vector_variants\(\)](#), [test_dot_product_matrix_multiply\(\)](#), [test_invert\(\)](#), [test_LUP_decomposition_identity_P_no_swap_case\(\)](#), [test_LUP_decomposition_swap_rows\(\)](#), [test_mat_is_all_digit\(\)](#), [test_minor_det_matches_gauss_4x4_known\(\)](#), [test_non_contiguous_stride_views\(\)](#), [test_norms_and_normalize\(\)](#), [test_offset2d_and_stride\(\)](#), [test_outer_product_and_cross\(\)](#), [test_outer_product_row_vector_path\(\)](#), [test_power_iterate_and_eig_helpers\(\)](#), [test_rand_range\(\)](#), [test_reduce_rank\(\)](#), [test_rotation_matrices_orthonormal\(\)](#), [test_row_col_ops_and_scaling\(\)](#), [test_shift_and_identity\(\)](#), [test_solve_linear_system_LUP\(\)](#), and [test_transpose\(\)](#).

4.3.4.30 mat2D_free_uint32()

```
void mat2D_free_uint32 (
    Mat2D_uint32 m )
```

Free the buffer owned by a [Mat2D_uint32](#).

Parameters

<code>m</code>	Matrix whose elements were allocated via <code>MAT2D_MALLOC</code> .
----------------	--

Note

This does not modify `m` (it is passed by value).

It is safe to call with `m.elements == NULL`.

Definition at line 1081 of file [Matrix2D.h](#).

References [Mat2D_uint32::elements](#), and [MAT2D_FREE](#).

Referenced by [test_uint32_alloc_fill_and_at\(\)](#).

4.3.4.31 mat2D_inner_product()

```
double mat2D_inner_product (
    Mat2D v )
```

Compute the inner product of a vector with itself: `dot(v, v)`.

Parameters

<i>v</i>	Vector (shape $n \times 1$ or $1 \times n$).
----------	---

Returns

$(\sum_k v_k^2)$ (the squared Euclidean norm).

Precondition

$v.cols == 1 \parallel v.rows == 1$

Definition at line 1094 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [test_dot_product_and_vector_variants\(\)](#), and [test_norms_and_normalize\(\)](#).

4.3.4.32 mat2D_invert()

```
void mat2D_invert (
    Mat2D des,
    Mat2D src )
```

Invert a square matrix using Gauss-Jordan elimination.

Parameters

<i>des</i>	Destination matrix (same shape as src).
<i>src</i>	Source square matrix.

Precondition

src is square.

des is allocated as the same shape as src.

Forms an augmented matrix $[src \mid I]$, performs Gauss-Jordan style reduction in-place (via [mat2D_reduce\(\)](#)), and then copies the right half into *des*.

Warning

This routine does not explicitly detect singular matrices. If *src* is singular (or nearly singular), [mat2D_reduce\(\)](#) may assert on a near-zero pivot or produce unstable/undefined results.

Definition at line 1130 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_copy_src_to_des_window\(\)](#), [mat2D_copy_src_window_to_des\(\)](#), [mat2D_free\(\)](#), [mat2D_reduce\(\)](#), [mat2D_set_identity\(\)](#), and [Mat2D::rows](#).

Referenced by [assert_inverse_identity_both_sides\(\)](#), [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), and [test_deterministic_fuzz_loop\(\)](#).

4.3.4.33 mat2D_LUP_decomposition_with_swap()

```
void mat2D_LUP_decomposition_with_swap (
    Mat2D src,
    Mat2D l,
    Mat2D p,
    Mat2D u )
```

Compute LUP decomposition: $P \cdot A = L \cdot U$ with L unit diagonal.

Parameters

<i>src</i>	Input matrix A (not modified by this function).
<i>l</i>	Output lower-triangular-like matrix (intended to have unit diagonal).
<i>p</i>	Output permutation matrix.
<i>u</i>	Output upper-triangular-like matrix.

Precondition

src is square.

l, *p*, *u* are allocated with the same shape as *src*.

Warning

Pivoting is limited: a row swap is performed only when the pivot is “near zero” ([MAT2D_IS_ZERO\(\)](#)).

This routine swaps rows of *L* during decomposition; for a standard LUP implementation, care is required when swapping partially-built *L*.

Definition at line 1164 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), [mat2D_copy\(\)](#), [mat2D_fill\(\)](#), [MAT2D_IS_ZERO](#), [mat2D_set_identity\(\)](#), [mat2D_sub_row_time_factor_to_row\(\)](#), [mat2D_swap_rows\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_solve_linear_sys_LUP_decomposition\(\)](#), [test_LUP_decomposition_identity_P_no_swap_case\(\)](#), and [test_LUP_decomposition_swap_required_case\(\)](#).

4.3.4.34 mat2D_make_orthogonal_Gaussian_elimination()

```
void mat2D_make_orthogonal_Gaussian_elimination (
    Mat2D des,
    Mat2D A )
```

Attempt to build an orthogonal(ized) matrix from A using Gaussian elimination.

This routine follows the idea sketched in the in-body notes: it forms $(A^T A)$ and $(A^T A)$, augments $([A^T A \mid A^T])$, performs elimination, then transposes the resulting right block into *des*.

Mathematical condition (from the function's internal comment):

- $(A^T A)$ must be full rank (invertible). Equivalently, the columns of (A) must be linearly independent and non-zero.

Parameters

<i>des</i>	Destination matrix.
<i>A</i>	Input matrix.

Precondition

des.rows == *A*.rows and *des*.cols == *A*.cols

Warning

This is an educational routine; it is not a standard QR/GS implementation and may be numerically unstable.
Prints debug output via [MAT2D_PRINT\(temp\)](#).

$A^T A$ must be fully ranked, i.e. columns must be linearly independent and non zero.

Definition at line 1221 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_copy_src_to_des_window\(\)](#), [mat2D_copy_src_window_to_des\(\)](#), [mat2D_dot\(\)](#), [mat2D_free\(\)](#), [MAT2D_ONES_ON_DIAG](#), [MAT2D_PRINT](#), [mat2D_transpose\(\)](#), [mat2D_upper_triangulate\(\)](#), and [Mat2D::rows](#).

4.3.4.35 mat2D_make_orthogonal_modified_Gram_Schmidt()

```
void mat2D_make_orthogonal_modified_Gram_Schmidt (
    Mat2D des,
    Mat2D A )
```

Build an orthonormal basis using modified Gram-Schmidt.

Uses a modified Gram-Schmidt process on the columns of *des*. The implementation copies the leading non-zero columns of *A* into *des*, and initializes the remaining columns of *des* with random values before orthogonalization/normalization, attempting to complete a full basis.

Mathematical conditions:

- Gram-Schmidt requires non-zero vectors. This code stops copying columns from *A* once it encounters a column with (near) zero norm.
- For stable/meaningful results, the set of input columns you expect to preserve should be linearly independent; otherwise a vector can become (near) zero during orthogonalization and normalization may divide by zero.

Parameters

<i>des</i>	Destination matrix (overwritten).
<i>A</i>	Input matrix providing initial columns.

Precondition

`des.rows == A.rows`
`des.cols == des.rows` (destination is square)

Warning

Uses `rand()` via [mat2D_rand\(\)](#) for the extra columns.

Definition at line 1283 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_calc_col_norma\(\)](#), [mat2D_calc_norma\(\)](#), [mat2D_copy\(\)](#), [mat2D_copy_col_from_src_to_des\(\)](#), [mat2D_create_col_ref\(\)](#), [mat2D_dot_product\(\)](#), [mat2D_dprintSIZE_T](#), [mat2D_free\(\)](#), [mat2D_inner_product\(\)](#), [MAT2D_IS_ZERO](#), [mat2D_mult\(\)](#), [mat2D_normalize](#), [mat2D_rand\(\)](#), [mat2D_sub\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_SVD_full\(\)](#).

4.3.4.36 mat2D_mat_is_all_digit()

```
bool mat2D_mat_is_all_digit (
    Mat2D m,
    double digit )
```

Check if all elements of a matrix equal a given digit.

Parameters

<i>m</i>	Matrix.
<i>digit</i>	Value to compare.

Returns

true if every element equals digit, false otherwise.

Warning

Uses exact floating-point equality.

Definition at line 1336 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_mat_is_all_digit\(\)](#).

4.3.4.37 mat2D_minor_alloc_fill_from_mat()

```
Mat2D_Minor mat2D_minor_alloc_fill_from_mat (
    Mat2D ref_mat,
    size_t i,
    size_t j )
```

Allocate a minor view by excluding row *i* and column *j* of *ref_mat*.

Parameters

<i>ref_mat</i>	Reference square matrix.
<i>i</i>	Excluded row index in <i>ref_mat</i> .
<i>j</i>	Excluded column index in <i>ref_mat</i> .

Returns

A [Mat2D_Minor](#) that references *ref_mat*.

Note

The returned minor owns *rows_list* and *cols_list* and must be released with [mat2D_minor_free\(\)](#).

The returned minor does not own *ref_mat.elements*.

Definition at line 1359 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [Mat2D_Minor::cols](#), [Mat2D_Minor::cols_list](#), [MAT2D_ASSERT](#), [MAT2D_MALLOC](#), [Mat2D_Minor::ref_mat](#), [Mat2D::rows](#), [Mat2D_Minor::rows](#), [Mat2D_Minor::rows_list](#), and [Mat2D_Minor::stride_r](#).

Referenced by [det_by_minors_first_col\(\)](#), and [mat2D_det\(\)](#).

4.3.4.38 mat2D_minor_alloc_fill_from_mat_minor()

```
Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor (
    Mat2D_Minor ref_mm,
    size_t i,
    size_t j )
```

Allocate a nested minor view from an existing minor by excluding row *i* and column *j* of the minor.

Parameters

<i>ref_mm</i>	Reference minor.
<i>i</i>	Excluded row index in the minor.
<i>j</i>	Excluded column index in the minor.

Returns

A new [Mat2D_Minor](#) that references the same underlying matrix.

Note

The returned minor owns `rows_list` and `cols_list` and must be released with [mat2D_minor_free\(\)](#).

The returned minor does not own the underlying reference matrix data.

Definition at line [1401](#) of file [Matrix2D.h](#).

References [Mat2D_Minor::cols](#), [Mat2D_Minor::cols_list](#), [MAT2D_ASSERT](#), [MAT2D_MALLOC](#), [Mat2D_Minor::ref_mat](#), [Mat2D_Minor::rows](#), [Mat2D_Minor::rows_list](#), and [Mat2D_Minor::stride_r](#).

Referenced by [mat2D_minor_det\(\)](#).

4.3.4.39 mat2D_minor_det()

```
double mat2D_minor_det (
    Mat2D\_Minor mm )
```

Determinant of a minor via recursive expansion by minors.

Parameters

<i>mm</i>	Square minor.
-----------	---------------

Returns

`det(mm)`.

Warning

Exponential complexity (factorial). Intended for educational or very small matrices only.

Definition at line [1438](#) of file [Matrix2D.h](#).

References [Mat2D_Minor::cols](#), [MAT2D_ASSERT](#), [mat2D_det_2x2_mat_minor\(\)](#), [mat2D_minor_alloc_fill_from_mat_minor\(\)](#), [MAT2D_MINOR_AT](#), [mat2D_minor_free\(\)](#), and [Mat2D_Minor::rows](#).

Referenced by [det_by_minors_first_col\(\)](#), and [mat2D_det\(\)](#).

4.3.4.40 mat2D_minor_free()

```
void mat2D_minor_free (
    Mat2D\_Minor mm )
```

Free the index arrays owned by a minor.

Parameters

<i>mm</i>	Minor to free.
-----------	----------------

Note

After this call, `mm.rows_list` and `mm.cols_list` are invalid.

Definition at line 1464 of file [Matrix2D.h](#).

References [Mat2D_Minor::cols_list](#), [MAT2D_FREE](#), and [Mat2D_Minor::rows_list](#).

Referenced by [det_by_minors_first_col\(\)](#), [mat2D_det\(\)](#), and [mat2D_minor_det\(\)](#).

4.3.4.41 mat2D_minor_print()

```
void mat2D_minor_print (
    Mat2D_Minor mm,
    const char * name,
    size_t padding )
```

Print a minor matrix to stdout with a name and indentation padding.

Parameters

<i>mm</i>	Minor to print.
<i>name</i>	Label to print.
<i>padding</i>	Left padding in spaces.

Definition at line 1476 of file [Matrix2D.h](#).

References [Mat2D_Minor::cols](#), [MAT2D_MINOR_AT](#), and [Mat2D_Minor::rows](#).

4.3.4.42 mat2D_mult()

```
void mat2D_mult (
    Mat2D m,
    double factor )
```

In-place scalar multiplication: `m *= factor`.

Parameters

<i>m</i>	Matrix.
<i>factor</i>	Scalar multiplier.

Definition at line 1494 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_power_iterate\(\)](#), and [mat2D_SVD_thin\(\)](#).

4.3.4.43 mat2D_mult_row()

```
void mat2D_mult_row (
    Mat2D m,
    size_t r,
    double factor )
```

In-place row scaling: $\text{row}(r) \mathrel{*=} \text{factor}$.

Parameters

<i>m</i>	Matrix.
<i>r</i>	Row index.
<i>factor</i>	Scalar multiplier.

Warning

Indices are not bounds-checked in this routine.

Definition at line 1511 of file [Matrix2D.h](#).

References [Mat2D::cols](#), and [MAT2D_AT](#).

Referenced by [mat2D_upper_triangularize\(\)](#), and [test_row_col_ops_and_scaling\(\)](#).

4.3.4.44 mat2D_offset2d()

```
size_t mat2D_offset2d (
    Mat2D m,
    size_t i,
    size_t j )
```

Compute the linear offset of element (i, j) in a [Mat2D](#).

Parameters

<i>m</i>	Matrix.
<i>i</i>	Row index (0-based).
<i>j</i>	Column index (0-based).

Returns

The linear offset $i * \text{stride_r} + j$.

Precondition

$0 \leq i < \text{m.rows}$ and $0 \leq j < \text{m.cols}$ (checked by MAT2D_ASSERT).

Definition at line 1528 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [Mat2D::rows](#), and [Mat2D::stride_r](#).

Referenced by [mat2D_fill_sequence\(\)](#), [test_non_contiguous_stride_views\(\)](#), and [test_offset2d_and_stride\(\)](#).

4.3.4.45 mat2D_offset2d_uint32()

```
size_t mat2D_offset2d_uint32 (
    Mat2D_uint32 m,
    size_t i,
    size_t j )
```

Compute the linear offset of element (i, j) in a [Mat2D_uint32](#).

Parameters

<i>m</i>	Matrix.
<i>i</i>	Row index (0-based).
<i>j</i>	Column index (0-based).

Returns

The linear offset $i * \text{stride_r} + j$.

Precondition

$0 \leq i < \text{m.rows}$ and $0 \leq j < \text{m.cols}$ (checked by MAT2D_ASSERT).

Definition at line 1544 of file [Matrix2D.h](#).

References [Mat2D_uint32::cols](#), [MAT2D_ASSERT](#), [Mat2D_uint32::rows](#), and [Mat2D_uint32::stride_r](#).

4.3.4.46 mat2D_outer_product()

```
void mat2D_outer_product (
    Mat2D des,
    Mat2D v )
```

Compute the outer product of a vector with itself: $\text{des} = v * v^T$.

Parameters

<i>des</i>	Destination square matrix (n x n).
<i>v</i>	Vector (shape n x 1 or 1 x n).

Postcondition

des is fully overwritten with $(v v^T)$.

Precondition

des.rows == *des*.cols
 $(v.cols == 1 \ \&\& \ des.rows == v.rows) \ || \ (v.rows == 1 \ \&\& \ des.cols == v.cols)$

Definition at line 1561 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_eig_power_iteration\(\)](#), [test_outer_product_and_cross\(\)](#), and [test_outer_product_row_vector_path\(\)](#).

4.3.4.47 mat2D_power_iterate()

```
int mat2D_power_iterate (
    Mat2D A,
    Mat2D v,
    double * lambda,
    double shift,
    bool norm_inf_v )
```

Approximate an eigenpair using (shifted) power iteration.

Runs power iteration on the shifted matrix ($B = A - \text{shift} \cdot I$). The input/output vector *v* is iteratively updated (in-place) and normalized. An eigenvalue estimate is written to *lambda* (if non-NULL) as: $(\lambda \approx \lambda(B) + \text{shift})$.

Parameters

	<i>A</i>	Square matrix (N x N).
<i>in, out</i>	<i>v</i>	Initial guess vector (N x 1). Overwritten with the estimated dominant eigenvector of the shifted matrix.
<i>out</i>	<i>lambda</i>	Optional output for the eigenvalue estimate (may be NULL).
	<i>shift</i>	Diagonal shift applied as described above.
	<i>norm_inf_v</i>	If true, normalize <i>v</i> at the end by mat2D_normalize_inf() .

Return values

0	Converged (difference below MAT2D_EPS within MAT2D_MAX_POWER_ITERATION).
1	Did not converge (often corresponds to sign-flip/alternation behavior).

Precondition

A is square and v has shape (A.rows x 1).

Conditions:

- The eigenvectors must form a basis
- The largest eigenvalue must be positive and unique

Definition at line 1604 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_calc_norma\(\)](#), [mat2D_calc_norma_inf\(\)](#), [mat2D_copy\(\)](#), [mat2D_dot\(\)](#), [mat2D_dot_product\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), [MAT2D_MAX_POWER_ITERATION](#), [mat2D_mult\(\)](#), [mat2D_normalize](#), [mat2D_normalize_inf](#), [mat2D_shift\(\)](#), [mat2D_sub\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_eig_power_iteration\(\)](#), and [test_power_iterate_and_eig_helpers\(\)](#).

4.3.4.48 mat2D_print()

```
void mat2D_print (
    Mat2D m,
    const char * name,
    size_t padding )
```

Print a matrix to stdout with a name and indentation padding.

Parameters

<i>m</i>	Matrix to print.
<i>name</i>	Label to print.
<i>padding</i>	Left padding in spaces.

Definition at line 1667 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [Mat2D::rows](#).

4.3.4.49 mat2D_print_as_col()

```
void mat2D_print_as_col (
    Mat2D m,
    const char * name,
    size_t padding )
```

Print a matrix as a flattened column vector to stdout.

Parameters

<i>m</i>	Matrix to print (flattened in row-major).
<i>name</i>	Label to print.
<i>padding</i>	Left padding in spaces.

Definition at line 1686 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [Mat2D::elements](#), and [Mat2D::rows](#).

4.3.4.50 mat2D_rand()

```
void mat2D_rand (
    Mat2D m,
    double low,
    double high )
```

Fill a matrix with pseudo-random doubles in [low, high].

Parameters

<i>m</i>	Matrix to fill.
<i>low</i>	Lower bound (inclusive).
<i>high</i>	Upper bound (inclusive).

Precondition

$high > low$ (not checked here; caller responsibility).

Note

Uses [mat2D_rand_double\(\)](#) ([rand\(\)](#)).

Definition at line 1706 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), [mat2D_rand_double\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), and [test_rand_range\(\)](#).

4.3.4.51 mat2D_rand_double()

```
double mat2D_rand_double (
    void )
```

Return a pseudo-random double in the range [0, 1].

Uses [rand\(\)](#) / [RAND_MAX](#) from the C standard library.

Note

This RNG is not cryptographically secure and may have weak statistical properties depending on the platform.

Definition at line 1724 of file [Matrix2D.h](#).

Referenced by [mat2D_rand\(\)](#).

4.3.4.52 mat2D_reduce()

```
size_t mat2D_reduce (
    Mat2D m )
```

Reduce a matrix in-place to reduced row echelon form (RREF) and return its rank.

Parameters

<i>m</i>	Matrix modified in-place.
----------	---------------------------

Returns

The computed rank (number of pivot rows found).

Internally calls [mat2D_upper_triangulate\(\)](#) and then performs backward elimination and row scaling to produce a reduced row echelon form.

Note

When used on an augmented matrix (e.g. $[A \mid I]$), this can be used as part of Gauss-Jordan inversion when A is nonsingular.

Definition at line 1742 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_AT](#), [mat2D_find_first_non_zero_value\(\)](#), [MAT2D_ONES_ON_DIAG](#), [MAT2D_ROW_SWAPPING](#), [mat2D_sub_row_time_factor_to_row\(\)](#), [mat2D_upper_triangulate\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_invert\(\)](#), [test_deterministic_fuzz_loop\(\)](#), and [test_reduce_rank\(\)](#).

4.3.4.53 mat2D_row_is_all_digit()

```
bool mat2D_row_is_all_digit (
    Mat2D m,
    double digit,
    size_t r )
```

Check if all elements of a row equal a given digit.

Parameters

<i>m</i>	Matrix.
<i>digit</i>	Value to compare.
<i>r</i>	Row index.

Returns

true if every element equals digit, false otherwise.

Warning

Uses exact floating-point equality.

Definition at line 1775 of file [Matrix2D.h](#).

References [Mat2D::cols](#), and [MAT2D_AT](#).

Referenced by [mat2D_det\(\)](#), and [test_det_early_zero_row_and_zero_col_paths\(\)](#).

4.3.4.54 mat2D_set_DCM_zyx()

```
void mat2D_set_DCM_zyx (
    Mat2D DCM,
    float yaw_deg,
    float pitch_deg,
    float roll_deg )
```

Build a 3x3 direction cosine matrix (DCM) from Z-Y-X Euler angles.

Parameters

<i>DCM</i>	3x3 destination matrix.
<i>yaw_deg</i>	Rotation about Z in degrees.
<i>pitch_deg</i>	Rotation about Y in degrees.
<i>roll_deg</i>	Rotation about X in degrees.

Computes $DCM = R_x(roll) * R_y(pitch) * R_z(yaw)$.

Note

This routine allocates temporary 3x3 matrices internally.

Definition at line 1796 of file [Matrix2D.h](#).

References [mat2D_alloc\(\)](#), [mat2D_dot\(\)](#), [mat2D_free\(\)](#), [mat2D_set_rot_mat_x\(\)](#), [mat2D_set_rot_mat_y\(\)](#), and [mat2D_set_rot_mat_z\(\)](#).

Referenced by [test_DCM_zyx_matches_product\(\)](#).

4.3.4.55 mat2D_set_identity()

```
void mat2D_set_identity (
    Mat2D m )
```

Set a square matrix to the identity matrix.

Parameters

<i>m</i>	Matrix (must be square).
----------	--------------------------

Precondition

`m.rows == m.cols` (checked by `MAT2D_ASSERT`).

Definition at line 1821 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_eig_power_iteration\(\)](#), [mat2D_invert\(\)](#), [mat2D_LUP_decomposition_with_swap\(\)](#), [mat2D_set_rot_mat_x\(\)](#), [mat2D_set_rot_mat_y\(\)](#), [mat2D_set_rot_mat_z\(\)](#), [test_non_contiguous_stride_views\(\)](#), and [test_shift_and_identity\(\)](#).

4.3.4.56 mat2D_set_rot_mat_x()

```
void mat2D_set_rot_mat_x (
    Mat2D m,
    float angle_deg )
```

Set a 3x3 rotation matrix for rotation about the X-axis.

Parameters

<i>m</i>	3x3 destination matrix.
<i>angle_deg</i>	Angle in degrees.

The matrix written is: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(a) & \sin(a) \\ 0 & -\sin(a) & \cos(a) \end{bmatrix}$

Definition at line 1849 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [MAT2D_PI](#), [mat2D_set_identity\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_set_DCM_zyx\(\)](#), [test_DCM_zyx_matches_product\(\)](#), and [test_rotation_matrices_orthonormal\(\)](#).

4.3.4.57 mat2D_set_rot_mat_y()

```
void mat2D_set_rot_mat_y (
    Mat2D m,
    float angle_deg )
```

Set a 3x3 rotation matrix for rotation about the Y-axis.

Parameters

<i>m</i>	3x3 destination matrix.
<i>angle_deg</i>	Angle in degrees.

The matrix written is: $\begin{bmatrix} \cos(a) & 0 & -\sin(a) \\ 0 & 1 & 0 \\ \sin(a) & 0 & \cos(a) \end{bmatrix}$

Definition at line 1873 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [MAT2D_PI](#), [mat2D_set_identity\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_set_DCM_zyx\(\)](#), [test_DCM_zyx_matches_product\(\)](#), and [test_rotation_matrices_orthonormal\(\)](#).

4.3.4.58 mat2D_set_rot_mat_z()

```
void mat2D_set_rot_mat_z (
    Mat2D m,
    float angle_deg )
```

Set a 3x3 rotation matrix for rotation about the Z-axis.

Parameters

<i>m</i>	3x3 destination matrix.
<i>angle_deg</i>	Angle in degrees.

The matrix written is: $\begin{bmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Definition at line 1897 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [MAT2D_PI](#), [mat2D_set_identity\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_set_DCM_zyx\(\)](#), [test_DCM_zyx_matches_product\(\)](#), and [test_rotation_matrices_orthonormal\(\)](#).

4.3.4.59 mat2D_shift()

```
void mat2D_shift (
    Mat2D m,
    double shift )
```

Add a scalar shift to the diagonal: $m[i,i] += \text{shift}$.

Parameters

<i>m</i>	Square matrix modified in-place.
<i>shift</i>	Value added to each diagonal element.

Precondition

`m.rows == m.cols`

Definition at line 1917 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_power_iterate\(\)](#), and [test_shift_and_identity\(\)](#).

4.3.4.60 mat2D_solve_linear_sys_LUP_decomposition()

```
void mat2D_solve_linear_sys_LUP_decomposition (
    Mat2D A,
    Mat2D x,
    Mat2D B )
```

Solve the linear system $Ax = B$ using an LUP-based approach.

Parameters

<i>A</i>	Coefficient matrix (N x N).
<i>x</i>	Solution vector (N x 1). Written on success.
<i>B</i>	Right-hand side vector (N x 1).

This routine computes an LUP decomposition and then forms explicit inverses of L and U ($\text{inv}(L)$, $\text{inv}(U)$) to compute: $x = \text{inv}(U) * \text{inv}(L) * (P * B)$

Warning

Explicitly inverting L and U is typically less stable and slower than forward/back substitution. Prefer substitution for production-quality solvers.

Definition at line 1941 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_dot\(\)](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), [mat2D_invert\(\)](#), [mat2D_LUP_decomposition_with_swap\(\)](#), and [Mat2D::rows](#).

Referenced by [test_solve_linear_system_LUP\(\)](#).

4.3.4.61 mat2D_sub()

```
void mat2D_sub (
    Mat2D dst,
    Mat2D a )
```

In-place subtraction: `dst -= a`.

Parameters

<i>dst</i>	Destination matrix to be decremented.
<i>a</i>	Subtrahend of same shape as <i>dst</i> .

Precondition

dst and *a* have identical shape.

Definition at line 1983 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_power_iterate\(\)](#), and [test_alloc_fill_copy_add_sub\(\)](#).

4.3.4.62 mat2D_sub_col_to_col()

```
void mat2D_sub_col_to_col (
    Mat2D des,
    size_t des_col,
    Mat2D src,
    size_t src_col )
```

Subtract a source column from a destination column.

Performs: `des[:, des_col] -= src[:, src_col]`

Parameters

<i>des</i>	Destination matrix (same row count as <i>src</i>).
<i>des_col</i>	Column index in destination.
<i>src</i>	Source matrix.
<i>src_col</i>	Column index in source.

Definition at line 2004 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_row_col_ops_and_scaling\(\)](#).

4.3.4.63 mat2D_sub_row_time_factor_to_row()

```
void mat2D_sub_row_time_factor_to_row (
    Mat2D m,
    size_t des_r,
    size_t src_r,
    double factor )
```

Row operation: `row(des_r) -= factor * row(src_r)`.

Parameters

<i>m</i>	Matrix.
<i>des</i> _{<i>r</i>}	Destination row index.
<i>src</i> _{<i>r</i>}	Source row index.
<i>factor</i>	Scalar multiplier.

Warning

Indices are not bounds-checked in this routine.

Definition at line 2045 of file [Matrix2D.h](#).

References [Mat2D::cols](#), and [MAT2D_AT](#).

Referenced by [mat2D_LUP_decomposition_with_swap\(\)](#), [mat2D_reduce\(\)](#), and [mat2D_upper_triangulate\(\)](#).

4.3.4.64 mat2D_sub_row_to_row()

```
void mat2D_sub_row_to_row (
    Mat2D des,
    size_t des_row,
    Mat2D src,
    size_t src_row )
```

Subtract a source row from a destination row.

Performs: `des[des_row, :] -= src[src_row, :]`

Parameters

<i>des</i>	Destination matrix (same number of columns as src).
<i>des_row</i>	Row index in destination.
<i>src</i>	Source matrix.
<i>src_row</i>	Row index in source.

Definition at line 2025 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_row_col_ops_and_scaling\(\)](#).

4.3.4.65 mat2D_SVD_full()

```
void mat2D_SVD_full (
    Mat2D A,
    Mat2D U,
    Mat2D S,
    Mat2D V,
    Mat2D init_vec_u,
    Mat2D init_vec_v,
    bool return_v_transpose )
```

Compute a "full" SVD by post-orthogonalizing the thin result.

Calls [mat2D_SVD_thin\(\)](#) first, then applies [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#) to expand/orthogonalize U and V into full orthonormal bases.

Mathematical/algorithmic conditions (inherited from internals):

- [mat2D_SVD_thin\(\)](#) uses repeated power iteration on $(A A^T)$ or $(A^T A)$, so convergence depends on the power-iteration conditions documented in [mat2D_power_iterate\(\)/mat2D_eig_power_iteration](#) (dominant eigenvalue separation, suitable initial vectors, etc.).
- The Gram-Schmidt completion step assumes it can produce non-zero independent vectors; if columns become (near) dependent, normalization can be unstable.

Parameters

<i>A</i>	Input matrix (n x m).
<i>U</i>	Output U (n x n).
<i>S</i>	Output S (n x m) with singular values on its diagonal.
<i>V</i>	Output V (m x m) or V^T depending on <code>return_v_transpose</code> .
<i>init_vec_u</i>	Initial vector for eigen/power iteration when using $(A A^T)$ (n x 1).
<i>init_vec_v</i>	Initial vector for eigen/power iteration when using $(A^T A)$ (m x 1).
<i>return_v_transpose</i>	If true, writes V^T into <i>V</i> (in-place transpose at the end).

Precondition

`U.rows==U.cols==A.rows`

`V.rows==V.cols==A.cols`

`S.rows==A.rows` and `S.cols==A.cols`

Definition at line 2080 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [mat2D_copy\(\)](#), [mat2D_free\(\)](#), [mat2D_make_orthogonal_modified_Gram_Schmidt\(\)](#), [mat2D_SVD_thin\(\)](#), [mat2D_transpose\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#).

4.3.4.66 mat2D_SVD_thin()

```
void mat2D_SVD_thin (
    Mat2D A,
    Mat2D U,
    Mat2D S,
    Mat2D V,
    Mat2D init_vec_u,
    Mat2D init_vec_v,
    bool return_v_transpose )
```

Compute an SVD using eigen-decomposition + power iteration (educational).

Implements the standard identities:

- Left singular vectors: eigenvectors of $(A A^T)$
- Right singular vectors: eigenvectors of $(A^T A)$
- Singular values: $(\sigma_i = \sqrt{\lambda_i})$ of the chosen PSD matrix

The function chooses:

- If $n \leq m$: compute eigenpairs of $(A A^T)$ ($n \times n$), fill U directly, then compute $(v_i = A^T u_i / \sigma_i)$.
- Else: compute eigenpairs of $(A^T A)$ ($m \times m$), fill V directly, then compute $(u_i = A v_i / \sigma_i)$.

Mathematical conditions:

- Power iteration assumes the eigenvectors form a basis for the matrix being iterated and that the dominant eigenvalue is positive and unique, which improves convergence.
- The initial vectors ($init_vec_u / init_vec_v$) must be non-zero and should not be (nearly) orthogonal to the dominant eigenvector(s), or convergence can be slow/fail.

Notes on numerical behavior:

- $(A A^T)$ and $(A^T A)$ are symmetric positive semidefinite in exact arithmetic, so eigenvalues should be non-negative. Due to floating-point error, small negative values may appear; this implementation clamps those to 0 by setting the corresponding singular value to 0.

Parameters

<i>A</i>	Input matrix ($n \times m$).
<i>U</i>	Output matrix ($n \times n$). Only the first $\min(n,m)$ columns corresponding to non-zero singular values are meaningfully populated by this step.
<i>S</i>	Output matrix ($n \times m$). Singular values written on the diagonal.
<i>V</i>	Output matrix ($m \times m$) (or V^T if <i>return_v_transpose</i> is true).
<i>init_vec_u</i>	Initial vector for eigen iteration in the $(A A^T)$ path ($n \times 1$).
<i>init_vec_v</i>	Initial vector for eigen iteration in the $(A^T A)$ path ($m \times 1$).
<i>return_v_transpose</i>	If true, the function transposes V before returning.

Precondition

```

U.rows==U.cols==A.rows
V.rows==V.cols==A.cols
S.rows==A.rows and S.cols==A.cols
init_vec_u.rows==A.rows && init_vec_u.cols==1
init_vec_v.rows==A.cols && init_vec_v.cols==1

```

Warning

This is not a production-quality SVD (no QR/SVD bidiagonalization). It is sensitive to convergence issues of power iteration and to deflation error accumulation.

fill U with the eigenvectors of AAT that have non zero singular value and fill V with the corresponding eigenvector according to: $v_i = A^T u_i / \sigma_i$

fill V with the eigenvectors of ATA that have non zero singular value and fill U with the corresponding eigenvector according to: $u_i = A v_i / \sigma_i$

Definition at line 2149 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_copy\(\)](#), [mat2D_copy_col_from_src_to_des\(\)](#), [mat2D_dot\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), [MAT2D_IS_ZERO](#), [mat2D_mult\(\)](#), [mat2D_transpose\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#), and [mat2D_SVD_full\(\)](#).

4.3.4.67 mat2D_swap_rows()

```

void mat2D_swap_rows (
    Mat2D m,
    size_t r1,
    size_t r2 )

```

Swap two rows of a matrix in-place.

Parameters

<i>m</i>	Matrix.
<i>r1</i>	First row index.
<i>r2</i>	Second row index.

Warning

Row indices are not bounds-checked in this routine.

Definition at line 2264 of file [Matrix2D.h](#).

References [Mat2D::cols](#), and [MAT2D_AT](#).

Referenced by [mat2D_LUP_decomposition_with_swap\(\)](#), and [mat2D_upper_triangulate\(\)](#).

4.3.4.68 mat2D_transpose()

```
void mat2D_transpose (
    Mat2D des,
    Mat2D src )
```

Transpose a matrix: $des = src^T$.

Parameters

<i>des</i>	Destination matrix (shape src.cols x src.rows).
<i>src</i>	Source matrix.

Warning

If *des* aliases *src*, results are undefined (no in-place transpose).

Definition at line 2280 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), [mat2D_SVD_full\(\)](#), [mat2D_SVD_thin\(\)](#), [test_deterministic_fuzz_loop\(\)](#), [test_non_contiguous_stride_views\(\)](#), [test_rotation_matrices_orthonormal\(\)](#), and [test_transpose\(\)](#).

4.3.4.69 mat2D_upper_triangulate()

```
double mat2D_upper_triangulate (
    Mat2D m,
    uint8_t flags )
```

Transform a matrix to (row-echelon) upper triangular form by forward elimination.

Parameters

<i>m</i>	Matrix transformed in-place.
----------	------------------------------

Returns

A determinant sign factor caused by row swaps: +1.0 or -1.0.

This routine performs Gaussian elimination using row operations of the form: $row_j = row_j - (m[j,i] / m[i,i]) * row_i$ which do not change the determinant. Row swaps flip the determinant sign and are tracked by the returned factor. Performs partial pivoting by selecting the row with the largest absolute pivot candidate in each column. Uses elimination operations that (in exact arithmetic) do not change the determinant. Each row swap flips the determinant sign; the cumulative sign is returned.

Warning

Not robust for linearly dependent rows or very small pivots.

Definition at line 2310 of file [Matrix2D.h](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [MAT2D_IS_ZERO](#), [mat2D_mult_row\(\)](#), [MAT2D_ONES_ON_DIAG](#), [MAT2D_ROW_SWAPPING](#), [mat2D_sub_row_time_factor_to_row\(\)](#), [mat2D_swap_rows\(\)](#), and [Mat2D::rows](#).

Referenced by [mat2D_det\(\)](#), [mat2D_make_orthogonal_Gaussian_elimination\(\)](#), [mat2D_reduce\(\)](#), and [test_det_2x2_and_upper_triangular](#).

4.4 Matrix2D.h

```

00001
00057 #ifndef MATRIX2D_H_
00058 #define MATRIX2D_H_
00059
00060 #include <stddef.h>
00061 #include <stdio.h>
00062 #include <stdlib.h>
00063 #include <stdint.h>
00064 #include <stdbool.h>
00065 #include <math.h>
00066
00075 #ifndef MAT2D_MALLOC
00076 #define MAT2D_MALLOC malloc
00077 #endif //MAT2D_MALLOC
00078
00087 #ifndef MAT2D_FREE
00088 #define MAT2D_FREE free
00089 #endif //MAT2D_FREE
00090
00099 #ifndef MAT2D_ASSERT
00100 #include <assert.h>
00101 #define MAT2D_ASSERT assert
00102 #endif //MAT2D_ASSERT
00103
00117 typedef struct {
00118     size_t rows;
00119     size_t cols;
00120     size_t stride_r; /* elements to traverse to reach the next row */
00121     double *elements;
00122 } Mat2D;
00123
00130 typedef struct {
00131     size_t rows;
00132     size_t cols;
00133     size_t stride_r; /* elements to traverse to reach the next row */
00134     uint32_t *elements;
00135 } Mat2D_uint32;
00136
00152 typedef struct {
00153     size_t rows;
00154     size_t cols;
00155     size_t stride_r; /* logical stride for the minor shape (not used for access) */
00156     size_t *rows_list;
00157     size_t *cols_list;
00158     Mat2D ref_mat;
00159 } Mat2D_Minor;
00160
00178 #if 1
00179 #define MAT2D_AT(m, i, j) (m).elements[mat2D_offset2d((m), (i), (j))]
00180 #define MAT2D_AT_UINT32(m, i, j) (m).elements[mat2D_offset2d_uint32((m), (i), (j))]
00181 #else /* use this macro for better performance but no assertion */
00182 #define MAT2D_AT(m, i, j) (m).elements[(i) * (m).stride_r + (j)]
00183 #define MAT2D_AT_UINT32(m, i, j) (m).elements[(i) * (m).stride_r + (j)]
00184 #endif
00185
00186 #define MAT2D_PI 3.14159265358979323846
00187
00188 #define MAT2D_EPS 1e-15
00189
00190 #define MAT2D_MAX_POWER_ITERATION 100
00191
00192
00200 #define MAT2D_IS_ZERO(x) (fabs(x) < MAT2D_EPS)
00201
00209 #define MAT2D_MINOR_AT(mm, i, j) MAT2D_AT((mm).ref_mat, (mm).rows_list[i], (mm).cols_list[j])
00210

```



```

00215 #define MAT2D_PRINT(m) mat2D_print(m, #m, 0)
00216
00221 #define MAT2D_PRINT_AS_COL(m) mat2D_print_as_col(m, #m, 0)
00222
00227 #define MAT2D_MINOR_PRINT(mm) mat2D_minor_print(mm, #mm, 0)
00228
00239 #define mat2D_normalize(m) mat2D_mult((m), 1.0 / mat2D_calc_norma((m)))
00240
00241 #define mat2D_normalize_inf(m) mat2D_mult((m), 1.0 / mat2D_calc_norma_inf((m)))
00242
00243 #define mat2D_dprintDOUBLE(expr) printf(#expr " = %g\n", expr)
00244
00245 #define mat2D_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)
00246
00247 #define mat2D_dprintINT(expr) printf(#expr " = %d\n", expr)
00248
00249 enum mat2D_upper_triangular_flag{
00250     MAT2D_ONES_ON_DIAG = 1 << 0,
00251     MAT2D_ROW_SWAPPING = 1 << 1,
00252 };
00253
00254 void mat2D_add(Mat2D dst, Mat2D a);
00255 void mat2D_add_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
00256 void mat2D_add_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
00257 void mat2D_add_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor);
00258 Mat2D mat2D_alloc(size_t rows, size_t cols);
00259 Mat2D_uint32 mat2D_alloc_uint32(size_t rows, size_t cols);
00260
00261 double mat2D_calc_col_norma(Mat2D m, size_t c);
00262 double mat2D_calc_norma(Mat2D m);
00263 double mat2D_calc_norma_inf(Mat2D m);
00264 bool mat2D_col_is_all_digit(Mat2D m, double digit, size_t c);
00265 void mat2D_copy(Mat2D des, Mat2D src);
00266 void mat2D_copy_col_from_src_to_des(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
00267 void mat2D_copy_row_from_src_to_des(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
00268 void mat2D_copy_src_to_des_window(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie,
    size_t je);
00269 void mat2D_copy_src_window_to_des(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie,
    size_t je);
00270 Mat2D mat2D_create_col_ref(Mat2D src, size_t c);
00271 void mat2D_cross(Mat2D dst, Mat2D v1, Mat2D v2);
00272
00273 void mat2D_dot(Mat2D dst, Mat2D a, Mat2D b);
00274 double mat2D_dot_product(Mat2D v1, Mat2D v2);
00275 double mat2D_det(Mat2D m);
00276 double mat2D_det_2x2_mat(Mat2D m);
00277 double mat2D_det_2x2_mat_minor(Mat2D_Minor mm);
00278
00279 void mat2D_eig_check(Mat2D A, Mat2D eigenvalues, Mat2D eigenvectors, Mat2D res);
00280 void mat2D_eig_power_iteration(Mat2D A, Mat2D eigenvalues, Mat2D eigenvectors, Mat2D
    init_vector, bool norm_inf_vectors);
00281
00282 void mat2D_fill(Mat2D m, double x);
00283 void mat2D_fill_sequence(Mat2D m, double start, double step);
00284 void mat2D_fill_uint32(Mat2D_uint32 m, uint32_t x);
00285 bool mat2D_find_first_non_zero_value(Mat2D m, size_t r, size_t *non_zero_col);
00286 void mat2D_free(Mat2D m);
00287 void mat2D_free_uint32(Mat2D_uint32 m);
00288
00289 double mat2D_inner_product(Mat2D v);
00290 void mat2D_invert(Mat2D des, Mat2D src);
00291
00292 void mat2D_LUP_decomposition_with_swap(Mat2D src, Mat2D l, Mat2D p, Mat2D u);
00293
00294 void mat2D_make_orthogonal_Gaussian_elimination(Mat2D des, Mat2D A);
00295 void mat2D_make_orthogonal_modified_Gram_Schmidt(Mat2D des, Mat2D A);
00296 bool mat2D_mat_is_all_digit(Mat2D m, double digit);
00297 Mat2D_Minor mat2D_minor_alloc_fill_from_mat(Mat2D ref_mat, size_t i, size_t j);
00298 Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor(Mat2D_Minor ref_mm, size_t i, size_t j);
00299 double mat2D_minor_det(Mat2D_Minor mm);
00300 void mat2D_minor_free(Mat2D_Minor mm);
00301 void mat2D_minor_print(Mat2D_Minor mm, const char *name, size_t padding);
00302 void mat2D_mult(Mat2D m, double factor);
00303 void mat2D_mult_row(Mat2D m, size_t r, double factor);
00304
00305 size_t mat2D_offset2d(Mat2D m, size_t i, size_t j);
00306 size_t mat2D_offset2d_uint32(Mat2D_uint32 m, size_t i, size_t j);
00307 void mat2D_outer_product(Mat2D des, Mat2D v);
00308
00309 int mat2D_power_iterate(Mat2D A, Mat2D v, double *lambda, double shift, bool norm_inf_v);
00310 void mat2D_print(Mat2D m, const char *name, size_t padding);
00311 void mat2D_print_as_col(Mat2D m, const char *name, size_t padding);
00312
00313 void mat2D_rand(Mat2D m, double low, double high);
00314 double mat2D_rand_double(void);
00315 size_t mat2D_reduce(Mat2D m);
00316 bool mat2D_row_is_all_digit(Mat2D m, double digit, size_t r);

```

```

00317
00318 void          mat2D_set_DCM_zyx(Mat2D DCM, float yaw_deg, float pitch_deg, float roll_deg);
00319 void          mat2D_set_identity(Mat2D m);
00320 void          mat2D_set_rot_mat_x(Mat2D m, float angle_deg);
00321 void          mat2D_set_rot_mat_y(Mat2D m, float angle_deg);
00322 void          mat2D_set_rot_mat_z(Mat2D m, float angle_deg);
00323 void          mat2D_shift(Mat2D m, double shift);
00324 void          mat2D_solve_linear_sys_LUP_decomposition(Mat2D A, Mat2D x, Mat2D B);
00325 void          mat2D_sub(Mat2D dst, Mat2D a);
00326 void          mat2D_sub_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
00327 void          mat2D_sub_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
00328 void          mat2D_sub_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor);
00329 void          mat2D_SVD_full(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v,
    bool return_v_transpose);
00330 void          mat2D_SVD_thin(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v,
    bool return_v_transpose);
00331 void          mat2D_swap_rows(Mat2D m, size_t r1, size_t r2);
00332
00333 void          mat2D_transpose(Mat2D des, Mat2D src);
00334
00335 double        mat2D_upper_triangularize(Mat2D m, uint8_t flags);
00336
00337 #endif // MATRIX2D_H_
00338
00339 #ifdef MATRIX2D_IMPLEMENTATION
00340 #undef MATRIX2D_IMPLEMENTATION
00341
00342
00351 void mat2D_add(Mat2D dst, Mat2D a)
00352 {
00353     MAT2D_ASSERT(dst.rows == a.rows);
00354     MAT2D_ASSERT(dst.cols == a.cols);
00355     for (size_t i = 0; i < dst.rows; ++i) {
00356         for (size_t j = 0; j < dst.cols; ++j) {
00357             MAT2D_AT(dst, i, j) += MAT2D_AT(a, i, j);
00358         }
00359     }
00360 }
00361
00372 void mat2D_add_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
00373 {
00374     MAT2D_ASSERT(src_col < src.cols);
00375     MAT2D_ASSERT(des.rows == src.rows);
00376     MAT2D_ASSERT(des_col < des.cols);
00377
00378     for (size_t i = 0; i < des.rows; ++i) {
00379         MAT2D_AT(des, i, des_col) += MAT2D_AT(src, i, src_col);
00380     }
00381 }
00382
00396 void mat2D_add_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
00397 {
00398     MAT2D_ASSERT(src_row < src.rows);
00399     MAT2D_ASSERT(des.cols == src.cols);
00400     MAT2D_ASSERT(des_row < des.rows);
00401
00402     for (size_t j = 0; j < des.cols; ++j) {
00403         MAT2D_AT(des, des_row, j) += MAT2D_AT(src, src_row, j);
00404     }
00405 }
00406
00416 void mat2D_add_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor)
00417 {
00418     for (size_t j = 0; j < m.cols; ++j) {
00419         MAT2D_AT(m, des_r, j) += factor * MAT2D_AT(m, src_r, j);
00420     }
00421 }
00422
00435 Mat2D mat2D_alloc(size_t rows, size_t cols)
00436 {
00437     Mat2D m;
00438     m.rows = rows;
00439     m.cols = cols;
00440     m.stride_r = cols;
00441     m.elements = (double*)MAT2D_MALLOC(sizeof(double)*rows*cols);
00442     MAT2D_ASSERT(m.elements != NULL);
00443
00444     return m;
00445 }
00446
00459 Mat2D_uint32 mat2D_alloc_uint32(size_t rows, size_t cols)
00460 {
00461     Mat2D_uint32 m;
00462     m.rows = rows;
00463     m.cols = cols;
00464     m.stride_r = cols;
00465     m.elements = (uint32_t*)MAT2D_MALLOC(sizeof(uint32_t)*rows*cols);

```

```

00466     MAT2D_ASSERT(m.elements != NULL);
00467
00468     return m;
00469 }
00470
00480 double mat2D_calc_col_norma(Mat2D m, size_t c)
00481 {
00482     MAT2D_ASSERT(c < m.cols);
00483
00484     double sum = 0;
00485     for (size_t i = 0; i < m.rows; ++i) {
00486         sum += MAT2D_AT(m, i, c) * MAT2D_AT(m, i, c);
00487     }
00488     return sqrt(sum);
00489 }
00490
00496 double mat2D_calc_norma(Mat2D m)
00497 {
00498     double sum = 0;
00499
00500     for (size_t i = 0; i < m.rows; ++i) {
00501         for (size_t j = 0; j < m.cols; ++j) {
00502             sum += MAT2D_AT(m, i, j) * MAT2D_AT(m, i, j);
00503         }
00504     }
00505     return sqrt(sum);
00506 }
00507
00517 double mat2D_calc_norma_inf(Mat2D m)
00518 {
00519     double max = 0;
00520     for (size_t i = 0; i < m.rows; ++i) {
00521         for (size_t j = 0; j < m.cols; ++j) {
00522             double current = fabs(MAT2D_AT(m, i, j));
00523             if (current > max) {
00524                 max = current;
00525             }
00526         }
00527     }
00528
00529     return max;
00530 }
00531
00541 bool mat2D_col_is_all_digit(Mat2D m, double digit, size_t c)
00542 {
00543     for (size_t i = 0; i < m.rows; ++i) {
00544         if (MAT2D_AT(m, i, c) != digit) {
00545             return false;
00546         }
00547     }
00548     return true;
00549 }
00550
00559 void mat2D_copy(Mat2D des, Mat2D src)
00560 {
00561     MAT2D_ASSERT(des.cols == src.cols);
00562     MAT2D_ASSERT(des.rows == src.rows);
00563
00564     for (size_t i = 0; i < des.rows; ++i) {
00565         for (size_t j = 0; j < des.cols; ++j) {
00566             MAT2D_AT(des, i, j) = MAT2D_AT(src, i, j);
00567         }
00568     }
00569 }
00570
00581 void mat2D_copy_col_from_src_to_des(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
00582 {
00583     MAT2D_ASSERT(src_col < src.cols);
00584     MAT2D_ASSERT(des.rows == src.rows);
00585     MAT2D_ASSERT(des_col < des.cols);
00586
00587     for (size_t i = 0; i < des.rows; ++i) {
00588         MAT2D_AT(des, i, des_col) = MAT2D_AT(src, i, src_col);
00589     }
00590 }
00591
00601 void mat2D_copy_row_from_src_to_des(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
00602 {
00603     MAT2D_ASSERT(src_row < src.rows);
00604     MAT2D_ASSERT(des.cols == src.cols);
00605     MAT2D_ASSERT(des_row < des.rows);
00606
00607     for (size_t j = 0; j < des.cols; ++j) {
00608         MAT2D_AT(des, des_row, j) = MAT2D_AT(src, src_row, j);
00609     }
00610 }
00611

```

```

00628 void mat2D_copy_src_to_des_window(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie, size_t je)
00629 {
00630     MAT2D_ASSERT(je >= js && ie >= is);
00631     MAT2D_ASSERT(je-js+1 == src.cols);
00632     MAT2D_ASSERT(ie-is+1 == src.rows);
00633     MAT2D_ASSERT(je-js+1 <= des.cols);
00634     MAT2D_ASSERT(ie-is+1 <= des.rows);
00635
00636     for (size_t index = 0; index < src.rows; ++index) {
00637         for (size_t jindex = 0; jindex < src.cols; ++jindex) {
00638             MAT2D_AT(des, is+index, js+jindex) = MAT2D_AT(src, index, jindex);
00639         }
00640     }
00641 }
00642
00653 void mat2D_copy_src_window_to_des(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie, size_t je)
00654 {
00655     MAT2D_ASSERT(je >= js && ie >= is);
00656     MAT2D_ASSERT(je-js+1 == des.cols);
00657     MAT2D_ASSERT(ie-is+1 == des.rows);
00658     MAT2D_ASSERT(je-js+1 <= src.cols);
00659     MAT2D_ASSERT(ie-is+1 <= src.rows);
00660
00661     for (size_t index = 0; index < des.rows; ++index) {
00662         for (size_t jindex = 0; jindex < des.cols; ++jindex) {
00663             MAT2D_AT(des, index, jindex) = MAT2D_AT(src, is+index, js+jindex);
00664         }
00665     }
00666 }
00667
00684 Mat2D mat2D_create_col_ref(Mat2D src, size_t c)
00685 {
00686     MAT2D_ASSERT(c < src.cols);
00687
00688     Mat2D col = {.cols = 1,
00689                 .rows = src.rows,
00690                 .stride_r = src.stride_r,
00691                 .elements = &(MAT2D_AT(src, 0, c))};
00692
00693     return col;
00694 }
00695
00703 void mat2D_cross(Mat2D dst, Mat2D v1, Mat2D v2)
00704 {
00705     MAT2D_ASSERT(3 == dst.rows && 1 == dst.cols);
00706     MAT2D_ASSERT(3 == v1.rows && 1 == v1.cols);
00707     MAT2D_ASSERT(3 == v2.rows && 1 == v2.cols);
00708
00709     MAT2D_AT(dst, 0, 0) = MAT2D_AT(v1, 1, 0) * MAT2D_AT(v2, 2, 0) - MAT2D_AT(v1, 2, 0) * MAT2D_AT(v2,
00710 1, 0);
00711     MAT2D_AT(dst, 1, 0) = MAT2D_AT(v1, 2, 0) * MAT2D_AT(v2, 0, 0) - MAT2D_AT(v1, 0, 0) * MAT2D_AT(v2,
00712 2, 0);
00713     MAT2D_AT(dst, 2, 0) = MAT2D_AT(v1, 0, 0) * MAT2D_AT(v2, 1, 0) - MAT2D_AT(v1, 1, 0) * MAT2D_AT(v2,
00714 0, 0);
00715 }
00716
00729 void mat2D_dot(Mat2D dst, Mat2D a, Mat2D b)
00730 {
00731     MAT2D_ASSERT(a.cols == b.rows);
00732     MAT2D_ASSERT(a.rows == dst.rows);
00733     MAT2D_ASSERT(b.cols == dst.cols);
00734
00735     size_t i, j, k;
00736
00737     for (i = 0; i < dst.rows; i++) {
00738         for (j = 0; j < dst.cols; j++) {
00739             MAT2D_AT(dst, i, j) = 0;
00740             for (k = 0; k < a.cols; k++) {
00741                 MAT2D_AT(dst, i, j) += MAT2D_AT(a, i, k) * MAT2D_AT(b, k, j);
00742             }
00743         }
00744     }
00745 }
00746
00758 double mat2D_dot_product(Mat2D v1, Mat2D v2)
00759 {
00760     MAT2D_ASSERT(v1.rows == v2.rows);
00761     MAT2D_ASSERT(v1.cols == v2.cols);
00762     MAT2D_ASSERT((1 == v1.cols && 1 == v2.cols) || (1 == v1.rows && 1 == v2.rows));
00763
00764     double dot_product = 0;
00765
00766     if (1 == v1.cols) {
00767         for (size_t i = 0; i < v1.rows; i++) {
00768             dot_product += MAT2D_AT(v1, i, 0) * MAT2D_AT(v2, i, 0);
00769         }
00770     }

```

```

00770     } else {
00771         for (size_t j = 0; j < vl.cols; j++) {
00772             dot_product += MAT2D_AT(v1, 0, j) * MAT2D_AT(v2, 0, j);
00773         }
00774     }
00775
00776     return dot_product;
00777 }
00778
00792 double mat2D_det(Mat2D m)
00793 {
00794     MAT2D_ASSERT(m.cols == m.rows && "should be a square matrix");
00795
00796     /* checking if there is a row or column with all zeros */
00797     /* checking rows */
00798     for (size_t i = 0; i < m.rows; i++) {
00799         if (mat2D_row_is_all_digit(m, 0, i)) {
00800             return 0;
00801         }
00802     }
00803     /* checking cols */
00804     for (size_t j = 0; j < m.rows; j++) {
00805         if (mat2D_col_is_all_digit(m, 0, j)) {
00806             return 0;
00807         }
00808     }
00809
00810     #if 0/* This is an implementation of naive determinant calculation using minors. This is too slow
00811     */
00812     double det = 0;
00813     /* TODO: finding beast row or col? */
00814     for (size_t i = 0, j = 0; i < m.rows; i++) { /* first column */
00815         if (MAT2D_AT(m, i, j) < 1e-10) continue;
00816         Mat2D_Minor sub_mm = mat2D_minor_alloc_fill_from_mat(m, i, j);
00817         int factor = (i+j)%2 ? -1 : 1;
00818         if (sub_mm.cols != 2) {
00819             MAT2D_ASSERT(sub_mm.cols == sub_mm.rows && "should be a square matrix");
00820             det += MAT2D_AT(m, i, j) * (factor) * mat2D_minor_det(sub_mm);
00821         } else if (sub_mm.cols == 2 && sub_mm.rows == 2) {
00822             det += MAT2D_AT(m, i, j) * (factor) * mat2D_det_2x2_mat_minor(sub_mm);
00823         }
00824         mat2D_minor_free(sub_mm);
00825     }
00826     #endif
00827
00828     Mat2D temp_m = mat2D_alloc(m.rows, m.cols);
00829     mat2D_copy(temp_m, m);
00830     double factor = mat2D_upper_triangulate(temp_m, MAT2D_ROW_SWAPPING);
00831     double diag_mul = 1;
00832     for (size_t i = 0; i < temp_m.rows; i++) {
00833         diag_mul *= MAT2D_AT(temp_m, i, i);
00834     }
00835     mat2D_free(temp_m);
00836
00837     return diag_mul / factor;
00838 }
00844 double mat2D_det_2x2_mat(Mat2D m)
00845 {
00846     MAT2D_ASSERT(2 == m.cols && 2 == m.rows && "Not a 2x2 matrix");
00847     return MAT2D_AT(m, 0, 0) * MAT2D_AT(m, 1, 1) - MAT2D_AT(m, 0, 1) * MAT2D_AT(m, 1, 0);
00848 }
00849
00855 double mat2D_det_2x2_mat_minor(Mat2D_Minor mm)
00856 {
00857     MAT2D_ASSERT(2 == mm.cols && 2 == mm.rows && "Not a 2x2 matrix");
00858     return MAT2D_MINOR_AT(mm, 0, 0) * MAT2D_MINOR_AT(mm, 1, 1) - MAT2D_MINOR_AT(mm, 0, 1) *
00859     MAT2D_MINOR_AT(mm, 1, 0);
00860 }
00874 void mat2D_eig_check(Mat2D A, Mat2D eigenvalues, Mat2D eigenvectors, Mat2D res)
00875 {
00876     MAT2D_ASSERT(A.cols == A.rows);
00877     MAT2D_ASSERT(eigenvalues.cols == A.cols);
00878     MAT2D_ASSERT(eigenvalues.rows == A.rows);
00879     MAT2D_ASSERT(eigenvectors.cols == A.cols);
00880     MAT2D_ASSERT(eigenvectors.rows == A.rows);
00881     MAT2D_ASSERT(res.cols == A.cols);
00882     MAT2D_ASSERT(res.rows == A.rows);
00883
00884     #if 1
00885     mat2D_dot(res, A, eigenvectors);
00886     Mat2D VL = mat2D_alloc(A.rows, A.cols);
00887     mat2D_dot(VL, eigenvectors, eigenvalues);
00888
00889     mat2D_sub(res, VL);
00890

```

```

00891     mat2D_free(VL);
00892 #else
00893 Mat2D temp_v = mat2D_alloc(A.rows, 1);
00894 for (size_t i = 0; i < A.rows; i++) {
00895     Mat2D eig_vector = {.cols = 1,
00896                         .elements = &MAT2D_AT(eigenvalues, 0, i),
00897                         .rows = A.rows,
00898                         .stride_r = eig_vectors.stride_r};
00899     Mat2D v = {.cols = 1,
00900               .elements = &MAT2D_AT(res, 0, i),
00901               .rows = A.rows,
00902               .stride_r = res.stride_r};
00903
00904     mat2D_dot(temp_v, A, eig_vector);
00905     mat2D_copy(v, eig_vector);
00906     mat2D_mult(v, MAT2D_AT(eigenvalues, i, i));
00907
00908     mat2D_sub(v, temp_v);
00909 }
00910 mat2D_free(temp_v);
00911 #endif
00912 }
00913
00939 void mat2D_eig_power_iteration(Mat2D A, Mat2D eigenvalues, Mat2D eig_vectors, Mat2D init_vector, bool
    norm_inf_vectors)
00940 {
00941     /* https://www.youtube.com/watch?v=c8DIOzuZqBs */
00942
00943     MAT2D_ASSERT(A.cols == A.rows);
00944     MAT2D_ASSERT(eigenvalues.cols == A.cols);
00945     MAT2D_ASSERT(eigenvalues.rows == A.rows);
00946     MAT2D_ASSERT(eig_vectors.cols == A.cols);
00947     MAT2D_ASSERT(eig_vectors.rows == A.rows);
00948     MAT2D_ASSERT(init_vector.cols == 1);
00949     MAT2D_ASSERT(init_vector.rows == A.rows);
00950     MAT2D_ASSERT(mat2D_calc_norm_inf(init_vector) > 0);
00951
00952     mat2D_set_identity(eigenvalues);
00953     Mat2D B = mat2D_alloc(A.rows, A.cols);
00954     Mat2D temp_mat = mat2D_alloc(A.rows, A.cols);
00955     mat2D_copy(B, A);
00956
00957     for (int i = 0, shift_value = 0; i < (int)A.rows; i++) {
00958         mat2D_copy_src_to_des_window(eig_vectors, init_vector, 0, i, init_vector.rows-1, i);
00959         Mat2D v = {.cols = init_vector.cols,
00960                   .elements = &MAT2D_AT(eig_vectors, 0, i),
00961                   .rows = init_vector.rows,
00962                   .stride_r = eig_vectors.stride_r};
00963         if (mat2D_power_iterate(B, v, &MAT2D_AT(eigenvalues, i, i), shift_value, 0)) { /* norm_inf_v
00964             must be zero*/
00965             shift_value++;
00966             i--;
00967             continue;
00968         } else {
00969             shift_value = 0;
00970         }
00971         mat2D_outer_product(temp_mat, v);
00972         mat2D_mult(temp_mat, MAT2D_AT(eigenvalues, i, i));
00973         mat2D_sub(B, temp_mat);
00974     }
00975
00976     if (norm_inf_vectors) {
00977         for (size_t c = 0; c < eig_vectors.cols; c++) {
00978             Mat2D v = {.cols = init_vector.cols,
00979                       .elements = &MAT2D_AT(eig_vectors, 0, c),
00980                       .rows = init_vector.rows,
00981                       .stride_r = eig_vectors.stride_r};
00982             mat2D_normalize_inf(v);
00983         }
00984     }
00985
00986     mat2D_free(B);
00987     mat2D_free(temp_mat);
00988 }
00989
00999 void mat2D_fill(Mat2D m, double x)
01000 {
01001     for (size_t i = 0; i < m.rows; ++i) {
01002         for (size_t j = 0; j < m.cols; ++j) {
01003             MAT2D_AT(m, i, j) = x;
01004         }
01005     }
01006 }
01007
01015 void mat2D_fill_sequence(Mat2D m, double start, double step) {
01016     for (size_t i = 0; i < m.rows; i++) {
01017         for (size_t j = 0; j < m.cols; j++) {

```

```

01018         MAT2D_AT(m, i, j) = start + step * mat2D_offset2d(m, i, j);
01019     }
01020 }
01021 }
01022
01028 void mat2D_fill_uint32(Mat2D_uint32 m, uint32_t x)
01029 {
01030     for (size_t i = 0; i < m.rows; ++i) {
01031         for (size_t j = 0; j < m.cols; ++j) {
01032             MAT2D_AT_UINT32(m, i, j) = x;
01033         }
01034     }
01035 }
01036
01049 bool mat2D_find_first_non_zero_value(Mat2D m, size_t r, size_t *non_zero_col)
01050 {
01051     for (size_t c = 0; c < m.cols; ++c) {
01052         if (!MAT2D_IS_ZERO(MAT2D_AT(m, r, c))) {
01053             *non_zero_col = c;
01054             return true;
01055         }
01056     }
01057     return false;
01058 }
01059
01068 void mat2D_free(Mat2D m)
01069 {
01070     MAT2D_FREE(m.elements);
01071 }
01072
01081 void mat2D_free_uint32(Mat2D_uint32 m)
01082 {
01083     MAT2D_FREE(m.elements);
01084 }
01085
01094 double mat2D_inner_product(Mat2D v)
01095 {
01096     MAT2D_ASSERT((1 == v.cols) || (1 == v.rows));
01097
01098     double dot_product = 0;
01099
01100     if (1 == v.cols) {
01101         for (size_t i = 0; i < v.rows; i++) {
01102             dot_product += MAT2D_AT(v, i, 0) * MAT2D_AT(v, i, 0);
01103         }
01104     } else {
01105         for (size_t j = 0; j < v.cols; j++) {
01106             dot_product += MAT2D_AT(v, 0, j) * MAT2D_AT(v, 0, j);
01107         }
01108     }
01109
01110     return dot_product;
01111 }
01112
01130 void mat2D_invert(Mat2D des, Mat2D src)
01131 {
01132     MAT2D_ASSERT(src.cols == src.rows && "Must be an NxN matrix");
01133     MAT2D_ASSERT(des.cols == src.cols && des.rows == des.cols);
01134
01135     Mat2D m = mat2D_alloc(src.rows, src.cols * 2);
01136     mat2D_copy_src_to_des_window(m, src, 0, 0, src.rows-1, src.cols-1);
01137
01138     mat2D_set_identity(des);
01139     mat2D_copy_src_to_des_window(m, des, 0, src.cols, des.rows-1, 2 * des.cols-1);
01140
01141     mat2D_reduce(m);
01142
01143     mat2D_copy_src_window_to_des(des, m, 0, src.cols, des.rows-1, 2 * des.cols-1);
01144
01145     mat2D_free(m);
01146 }
01147
01164 void mat2D_LUP_decomposition_with_swap(Mat2D src, Mat2D l, Mat2D p, Mat2D u)
01165 {
01166     /* performing LU decomposition Following the Wikipedia page:
01167     https://en.wikipedia.org/wiki/LU_decomposition */
01167
01168     mat2D_copy(u, src);
01169     mat2D_set_identity(p);
01170     mat2D_fill(l, 0);
01171
01172     for (size_t i = 0; i < (size_t)fmin(u.rows-1, u.cols); i++) {
01173         if (MAT2D_IS_ZERO(MAT2D_AT(u, i, i))) { /* swapping only if it is zero */
01174             /* finding biggest first number (absolute value) */
01175             size_t biggest_r = i;
01176             for (size_t index = i; index < u.rows; index++) {
01177                 if (fabs(MAT2D_AT(u, index, i)) > fabs(MAT2D_AT(u, biggest_r, i))) {

```

```

01178         biggest_r = index;
01179     }
01180 }
01181 if (i != biggest_r) {
01182     mat2D_swap_rows(u, i, biggest_r);
01183     mat2D_swap_rows(p, i, biggest_r);
01184     mat2D_swap_rows(l, i, biggest_r);
01185 }
01186 }
01187 for (size_t j = i+1; j < u.cols; j++) {
01188     double factor = 1 / MAT2D_AT(u, i, i);
01189     if (!isfinite(factor)) {
01190         printf("%s:%d:\n%s:\n[Error] unable to transform into upper triangular matrix. Probably
some of the rows are not independent.\n", __FILE__, __LINE__, __func__);
01191     }
01192     double mat_value = MAT2D_AT(u, j, i);
01193     mat2D_sub_row_time_factor_to_row(u, j, i, mat_value * factor);
01194     MAT2D_AT(l, j, i) = mat_value * factor;
01195 }
01196     MAT2D_AT(l, i, i) = 1;
01197 }
01198     MAT2D_AT(l, l.rows-1, l.cols-1) = 1;
01199 }
01200
01221 void mat2D_make_orthogonal_Gaussian_elimination(Mat2D des, Mat2D A)
01222 {
01223     /* https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process */
01224     MAT2D_ASSERT(des.cols == A.cols);
01225     MAT2D_ASSERT(des.rows == A.rows);
01226
01227     Mat2D AT = mat2D_alloc(A.cols, A.rows);
01228     Mat2D ATA = mat2D_alloc(A.cols, A.cols);
01229     Mat2D temp = mat2D_alloc(ATA.rows, ATA.cols + A.cols);
01230     Mat2D temp_des = mat2D_alloc(des.cols, des.rows);
01231
01232     mat2D_transpose(AT, A);
01233     mat2D_dot(ATA, AT, A);
01234     mat2D_copy_src_to_des_window(temp, ATA, 0, 0, ATA.rows-1, ATA.cols-1);
01235     mat2D_copy_src_to_des_window(temp, AT, 0, ATA.cols, AT.rows-1, ATA.cols + AT.cols-1);
01236
01237     MAT2D_PRINT(temp);
01238
01239     mat2D_upper_triangulate(temp, MAT2D_ONES_ON_DIAG);
01240
01241     mat2D_copy_src_window_to_des(temp_des, temp, 0, ATA.cols, AT.rows-1, ATA.cols + AT.cols-1);
01242
01243     mat2D_transpose(des, temp_des);
01244
01245     MAT2D_PRINT(temp);
01246
01247     mat2D_free(AT);
01248     mat2D_free(ATA);
01249     mat2D_free(temp);
01250     mat2D_free(temp_des);
01251 }
01252
01283 void mat2D_make_orthogonal_modified_Gram_Schmidt(Mat2D des, Mat2D A)
01284 {
01285     /* https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process */
01286     MAT2D_ASSERT(des.rows == A.rows);
01287     MAT2D_ASSERT(des.cols == des.rows);
01288
01289     size_t num_non_zero_vec = 0;
01290     for (size_t c = 0; c < A.cols; c++) {
01291         if (MAT2D_IS_ZERO(mat2D_calc_col_norma(A, c))) {
01292             break;
01293         }
01294         num_non_zero_vec++;
01295     }
01296
01297     mat2D_dprintSIZE_T(num_non_zero_vec);
01298
01299     mat2D_rand(des, 1, 2);
01300
01301     Mat2D temp_col = mat2D_alloc(des.rows, 1);
01302     for (size_t c = 0; c < num_non_zero_vec; c++) {
01303         mat2D_copy_col_from_src_to_des(des, c, A, c);
01304     }
01305     for (size_t c = 0; c < des.cols-1; c++) {
01306         Mat2D vc = mat2D_create_col_ref(des, c);
01307         double vc_vc = mat2D_inner_product(vc);
01308         for (size_t k = c+1; k < des.cols; k++) {
01309             mat2D_copy(temp_col, vc);
01310             Mat2D vk = mat2D_create_col_ref(des, k);
01311             double vk_vc = mat2D_dot_product(vc, vk);
01312             mat2D_mult(temp_col, vk_vc / vc_vc);
01313             mat2D_sub(vk, temp_col);

```



```

01314     }
01315     if (!MAT2D_IS_ZERO(mat2D_calc_norma(vc))) {
01316         mat2D_normalize(vc);
01317     }
01318 }
01319 Mat2D vc = mat2D_create_col_ref(des, des.cols-1);
01320 if (!MAT2D_IS_ZERO(mat2D_calc_norma(vc))) {
01321     mat2D_normalize(vc);
01322 }
01323
01324
01325     mat2D_free(temp_col);
01326 }
01327
01336 bool mat2D_mat_is_all_digit(Mat2D m, double digit)
01337 {
01338     for (size_t i = 0; i < m.rows; ++i) {
01339         for (size_t j = 0; j < m.cols; ++j) {
01340             if (MAT2D_AT(m, i, j) != digit) {
01341                 return false;
01342             }
01343         }
01344     }
01345     return true;
01346 }
01347
01359 Mat2D_Minor mat2D_minor_alloc_fill_from_mat(Mat2D ref_mat, size_t i, size_t j)
01360 {
01361     MAT2D_ASSERT(ref_mat.cols == ref_mat.rows && "minor is defined only for square matrix");
01362
01363     Mat2D_Minor mm;
01364     mm.cols = ref_mat.cols-1;
01365     mm.rows = ref_mat.rows-1;
01366     mm.stride_r = ref_mat.cols-1;
01367     mm.cols_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mat.cols-1));
01368     mm.rows_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mat.rows-1));
01369     mm.ref_mat = ref_mat;
01370
01371     MAT2D_ASSERT(mm.cols_list != NULL && mm.rows_list != NULL);
01372
01373     for (size_t index = 0, temp_index = 0; index < ref_mat.rows; index++) {
01374         if (index != i) {
01375             mm.rows_list[temp_index] = index;
01376             temp_index++;
01377         }
01378     }
01379     for (size_t jindex = 0, temp_jindex = 0; jindex < ref_mat.cols; jindex++) {
01380         if (jindex != j) {
01381             mm.cols_list[temp_jindex] = jindex;
01382             temp_jindex++;
01383         }
01384     }
01385
01386     return mm;
01387 }
01388
01401 Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor(Mat2D_Minor ref_mm, size_t i, size_t j)
01402 {
01403     MAT2D_ASSERT(ref_mm.cols == ref_mm.rows && "minor is defined only for square matrix");
01404
01405     Mat2D_Minor mm;
01406     mm.cols = ref_mm.cols-1;
01407     mm.rows = ref_mm.rows-1;
01408     mm.stride_r = ref_mm.cols-1;
01409     mm.cols_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mm.cols-1));
01410     mm.rows_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mm.rows-1));
01411     mm.ref_mat = ref_mm.ref_mat;
01412
01413     MAT2D_ASSERT(mm.cols_list != NULL && mm.rows_list != NULL);
01414
01415     for (size_t index = 0, temp_index = 0; index < ref_mm.rows; index++) {
01416         if (index != i) {
01417             mm.rows_list[temp_index] = ref_mm.rows_list[index];
01418             temp_index++;
01419         }
01420     }
01421     for (size_t jindex = 0, temp_jindex = 0; jindex < ref_mm.cols; jindex++) {
01422         if (jindex != j) {
01423             mm.cols_list[temp_jindex] = ref_mm.cols_list[jindex];
01424             temp_jindex++;
01425         }
01426     }
01427
01428     return mm;
01429 }
01430
01438 double mat2D_minor_det(Mat2D_Minor mm)

```

```

01439 {
01440     MAT2D_ASSERT(mm.cols == mm.rows && "should be a square matrix");
01441
01442     double det = 0;
01443     /* TODO: finding beast row or col? */
01444     for (size_t i = 0, j = 0; i < mm.rows; i++) { /* first column */
01445         if (fabs(MAT2D_MINOR_AT(mm, i, j)) < 1e-10) continue;
01446         Mat2D_Minor sub_mm = mat2D_minor_alloc_fill_from_mat_minor(mm, i, j);
01447         int factor = (i+j)%2 ? -1 : 1;
01448         if (sub_mm.cols != 2) {
01449             MAT2D_ASSERT(sub_mm.cols == sub_mm.rows && "should be a square matrix");
01450             det += MAT2D_MINOR_AT(mm, i, j) * (factor) * mat2D_minor_det(sub_mm);
01451         } else if (sub_mm.cols == 2 && sub_mm.rows == 2) {
01452             det += MAT2D_MINOR_AT(mm, i, j) * (factor) * mat2D_det_2x2_mat_minor(sub_mm);
01453         }
01454         mat2D_minor_free(sub_mm);
01455     }
01456     return det;
01457 }
01458
01464 void mat2D_minor_free(Mat2D_Minor mm)
01465 {
01466     MAT2D_FREE(mm.cols_list);
01467     MAT2D_FREE(mm.rows_list);
01468 }
01469
01476 void mat2D_minor_print(Mat2D_Minor mm, const char *name, size_t padding)
01477 {
01478     printf("%s%s = [\n", (int) padding, "", name);
01479     for (size_t i = 0; i < mm.rows; ++i) {
01480         printf("%s", (int) padding, "");
01481         for (size_t j = 0; j < mm.cols; ++j) {
01482             printf("%f ", MAT2D_MINOR_AT(mm, i, j));
01483         }
01484         printf("\n");
01485     }
01486     printf("%s]\n", (int) padding, "");
01487 }
01488
01494 void mat2D_mult(Mat2D m, double factor)
01495 {
01496     for (size_t i = 0; i < m.rows; ++i) {
01497         for (size_t j = 0; j < m.cols; ++j) {
01498             MAT2D_AT(m, i, j) *= factor;
01499         }
01500     }
01501 }
01502
01511 void mat2D_mult_row(Mat2D m, size_t r, double factor)
01512 {
01513     for (size_t j = 0; j < m.cols; ++j) {
01514         MAT2D_AT(m, r, j) *= factor;
01515     }
01516 }
01517
01528 size_t mat2D_offset2d(Mat2D m, size_t i, size_t j)
01529 {
01530     MAT2D_ASSERT(i < m.rows && j < m.cols);
01531     return i * m.stride_r + j;
01532 }
01533
01544 size_t mat2D_offset2d_uint32(Mat2D_uint32 m, size_t i, size_t j)
01545 {
01546     MAT2D_ASSERT(i < m.rows && j < m.cols);
01547     return i * m.stride_r + j;
01548 }
01549
01561 void mat2D_outer_product(Mat2D des, Mat2D v)
01562 {
01563     MAT2D_ASSERT(des.cols == des.rows);
01564     MAT2D_ASSERT((1 == v.cols && des.rows == v.rows) || (1 == v.rows && des.cols == v.cols));
01565
01566     // mat2D_fill(des, 0);
01567
01568     if (1 == v.cols) {
01569         for (size_t i = 0; i < des.rows; i++) {
01570             for (size_t j = 0; j < des.cols; j++) {
01571                 MAT2D_AT(des, i, j) = MAT2D_AT(v, i, 0) * MAT2D_AT(v, j, 0);
01572             }
01573         }
01574     } else {
01575         for (size_t i = 0; i < des.rows; i++) {
01576             for (size_t j = 0; j < des.cols; j++) {
01577                 MAT2D_AT(des, i, j) = MAT2D_AT(v, 0, i) * MAT2D_AT(v, 0, j);
01578             }
01579         }
01580     }

```

```

01581 }
01582
01604 int mat2D_power_iterate(Mat2D A, Mat2D v, double *lambda, double shift, bool norm_inf_v)
01605 {
01606     /* https://www.youtube.com/watch?v=SkPusgctgpl */
01607
01613     MAT2D_ASSERT(A.cols == A.rows);
01614     MAT2D_ASSERT(v.cols == 1);
01615     MAT2D_ASSERT(v.rows == A.rows);
01616     MAT2D_ASSERT(mat2D_calc_norma_inf(v) > 0);
01617
01618     Mat2D current_v = mat2D_alloc(v.rows, v.cols);
01619     Mat2D temp_v = mat2D_alloc(v.rows, v.cols);
01620     Mat2D B = mat2D_alloc(A.rows, A.cols);
01621     mat2D_copy(B, A);
01622     mat2D_shift(B, shift * -1.0);
01623
01624     double temp_lambda = 0;
01625     double diff = 0;
01626
01627     /* Rayleigh quotient */
01628     mat2D_dot(temp_v, B, v);
01629     temp_lambda = mat2D_dot_product(temp_v, v) / (mat2D_calc_norma(v) * mat2D_calc_norma(v));
01630     int i = 0;
01631     for (i = 0; i < MAT2D_MAX_POWER_ITERATION; i++) {
01632         mat2D_copy(current_v, v);
01633         mat2D_dot(v, B, current_v);
01634         mat2D_normalize(v);
01635         mat2D_mult(v, temp_lambda > 0 ? 1 : -1);
01636         // mat2D_mult(v, fabs(lambda) / lambda);
01637         mat2D_dot(temp_v, B, v);
01638         temp_lambda = mat2D_dot_product(temp_v, v);
01639
01640         mat2D_sub(current_v, v);
01641         diff = mat2D_calc_norma_inf(current_v);
01642         if (diff < MAT2D_EPS) {
01643             break;
01644         }
01645     }
01646
01647     mat2D_free(current_v);
01648     mat2D_free(temp_v);
01649     mat2D_free(B);
01650
01651     if (norm_inf_v) mat2D_normalize_inf(v);
01652     if (lambda) *lambda = temp_lambda + shift;
01653
01654     if (diff > MAT2D_EPS) {
01655         return 1; /* eigenvector alternating between two options */
01656     } else {
01657         return 0;
01658     }
01659 }
01660
01667 void mat2D_print(Mat2D m, const char *name, size_t padding)
01668 {
01669     printf("%s%s = [\n", (int) padding, "", name);
01670     for (size_t i = 0; i < m.rows; ++i) {
01671         printf("%s", "", (int) padding, "");
01672         for (size_t j = 0; j < m.cols; ++j) {
01673             printf("%9.6f ", MAT2D_AT(m, i, j));
01674         }
01675         printf("\n");
01676     }
01677     printf("%s]\n", (int) padding, "");
01678 }
01679
01686 void mat2D_print_as_col(Mat2D m, const char *name, size_t padding)
01687 {
01688     printf("%s%s = [\n", (int) padding, "", name);
01689     for (size_t i = 0; i < m.rows*m.cols; ++i) {
01690         printf("%s", "", (int) padding, "");
01691         printf("%f\n", m.elements[i]);
01692     }
01693     printf("%s]\n", (int) padding, "");
01694 }
01695
01706 void mat2D_rand(Mat2D m, double low, double high)
01707 {
01708     for (size_t i = 0; i < m.rows; ++i) {
01709         for (size_t j = 0; j < m.cols; ++j) {
01710             MAT2D_AT(m, i, j) = mat2D_rand_double()*(high - low) + low;
01711         }
01712     }
01713 }
01714
01724 double mat2D_rand_double(void)

```

```

01725 {
01726     return (double) rand() / (double) RAND_MAX;
01727 }
01728
01742 size_t mat2D_reduce(Mat2D m)
01743 {
01744     /* preforming Gauss-Jordan reduction to Reduced Row Echelon Form (RREF) */
01745     /* Gauss elimination: https://en.wikipedia.org/wiki/Gaussian\_elimination */
01746
01747     mat2D_upper_triangularize(m, MAT2D_ONES_ON_DIAG | MAT2D_ROW_SWAPPING);
01748
01749     size_t rank = 0;
01750
01751     for (int r = m.rows-1; r >= 0; r--) {
01752         size_t c = m.cols-1;
01753         if (!mat2D_find_first_non_zero_value(m, r, &c)) {
01754             continue; /* row of zeros */
01755         }
01756         for (int i = 0; i < r; i++) {
01757             double factor = MAT2D_AT(m, i, c);
01758             mat2D_sub_row_time_factor_to_row(m, i, r, factor);
01759         }
01760         rank++;
01761     }
01762
01763     return rank;
01764 }
01765
01775 bool mat2D_row_is_all_digit(Mat2D m, double digit, size_t r)
01776 {
01777     for (size_t j = 0; j < m.cols; ++j) {
01778         if (MAT2D_AT(m, r, j) != digit) {
01779             return false;
01780         }
01781     }
01782     return true;
01783 }
01784
01796 void mat2D_set_DCM_zyx(Mat2D DCM, float yaw_deg, float pitch_deg, float roll_deg)
01797 {
01798     Mat2D RotZ = mat2D_alloc(3,3);
01799     mat2D_set_rot_mat_z(RotZ, yaw_deg);
01800     Mat2D RotY = mat2D_alloc(3,3);
01801     mat2D_set_rot_mat_y(RotY, pitch_deg);
01802     Mat2D RotX = mat2D_alloc(3,3);
01803     mat2D_set_rot_mat_x(RotX, roll_deg);
01804     Mat2D temp = mat2D_alloc(3,3);
01805
01806     mat2D_dot(temp, RotY, RotZ);
01807     mat2D_dot(DCM, RotX, temp); /* I have a DCM */
01808
01809     mat2D_free(RotZ);
01810     mat2D_free(RotY);
01811     mat2D_free(RotX);
01812     mat2D_free(temp);
01813 }
01814
01821 void mat2D_set_identity(Mat2D m)
01822 {
01823     MAT2D_ASSERT(m.cols == m.rows);
01824     for (size_t i = 0; i < m.rows; ++i) {
01825         for (size_t j = 0; j < m.cols; ++j) {
01826             MAT2D_AT(m, i, j) = i == j ? 1 : 0;
01827             // if (i == j) {
01828             //     MAT2D_AT(m, i, j) = 1;
01829             // }
01830             // else {
01831             //     MAT2D_AT(m, i, j) = 0;
01832             // }
01833         }
01834     }
01835 }
01836
01849 void mat2D_set_rot_mat_x(Mat2D m, float angle_deg)
01850 {
01851     MAT2D_ASSERT(3 == m.cols && 3 == m.rows);
01852
01853     float angle_rad = angle_deg * MAT2D_PI / 180;
01854     mat2D_set_identity(m);
01855     MAT2D_AT(m, 1, 1) = cos(angle_rad);
01856     MAT2D_AT(m, 1, 2) = sin(angle_rad);
01857     MAT2D_AT(m, 2, 1) = -sin(angle_rad);
01858     MAT2D_AT(m, 2, 2) = cos(angle_rad);
01859 }
01860
01873 void mat2D_set_rot_mat_y(Mat2D m, float angle_deg)
01874 {

```

```

01875     MAT2D_ASSERT(3 == m.cols && 3 == m.rows);
01876
01877     float angle_rad = angle_deg * MAT2D_PI / 180;
01878     mat2D_set_identity(m);
01879     MAT2D_AT(m, 0, 0) = cos(angle_rad);
01880     MAT2D_AT(m, 0, 2) = -sin(angle_rad);
01881     MAT2D_AT(m, 2, 0) = sin(angle_rad);
01882     MAT2D_AT(m, 2, 2) = cos(angle_rad);
01883 }
01884
01897 void mat2D_set_rot_mat_z(Mat2D m, float angle_deg)
01898 {
01899     MAT2D_ASSERT(3 == m.cols && 3 == m.rows);
01900
01901     float angle_rad = angle_deg * MAT2D_PI / 180;
01902     mat2D_set_identity(m);
01903     MAT2D_AT(m, 0, 0) = cos(angle_rad);
01904     MAT2D_AT(m, 0, 1) = sin(angle_rad);
01905     MAT2D_AT(m, 1, 0) = -sin(angle_rad);
01906     MAT2D_AT(m, 1, 1) = cos(angle_rad);
01907 }
01908
01917 void mat2D_shift(Mat2D m, double shift)
01918 {
01919     MAT2D_ASSERT(m.cols == m.rows);
01920     for (size_t i = 0; i < m.rows; i++) {
01921         MAT2D_AT(m, i, i) += shift;
01922     }
01923 }
01924
01941 void mat2D_solve_linear_sys_LUP_decomposition(Mat2D A, Mat2D x, Mat2D B)
01942 {
01943     MAT2D_ASSERT(A.cols == x.rows);
01944     MAT2D_ASSERT(1 == x.cols);
01945     MAT2D_ASSERT(A.rows == B.rows);
01946     MAT2D_ASSERT(1 == B.cols);
01947
01948     Mat2D y = mat2D_alloc(x.rows, x.cols);
01949     Mat2D l = mat2D_alloc(A.rows, A.cols);
01950     Mat2D p = mat2D_alloc(A.rows, A.cols);
01951     Mat2D u = mat2D_alloc(A.rows, A.cols);
01952     Mat2D inv_l = mat2D_alloc(l.rows, l.cols);
01953     Mat2D inv_u = mat2D_alloc(u.rows, u.cols);
01954
01955     mat2D_LUP_decomposition_with_swap(A, l, p, u);
01956
01957     mat2D_invert(inv_l, l);
01958     mat2D_invert(inv_u, u);
01959
01960     mat2D_fill(x, 0); /* x here is only a temp mat*/
01961     mat2D_fill(y, 0);
01962     mat2D_dot(x, p, B);
01963     mat2D_dot(y, inv_l, x);
01964
01965     mat2D_fill(x, 0);
01966     mat2D_dot(x, inv_u, y);
01967
01968     mat2D_free(y);
01969     mat2D_free(l);
01970     mat2D_free(p);
01971     mat2D_free(u);
01972     mat2D_free(inv_l);
01973     mat2D_free(inv_u);
01974 }
01975
01983 void mat2D_sub(Mat2D dst, Mat2D a)
01984 {
01985     MAT2D_ASSERT(dst.rows == a.rows);
01986     MAT2D_ASSERT(dst.cols == a.cols);
01987     for (size_t i = 0; i < dst.rows; ++i) {
01988         for (size_t j = 0; j < dst.cols; ++j) {
01989             MAT2D_AT(dst, i, j) -= MAT2D_AT(a, i, j);
01990         }
01991     }
01992 }
01993
02004 void mat2D_sub_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
02005 {
02006     MAT2D_ASSERT(src_col < src.cols);
02007     MAT2D_ASSERT(des.rows == src.rows);
02008     MAT2D_ASSERT(des_col < des.cols);
02009
02010     for (size_t i = 0; i < des.rows; i++) {
02011         MAT2D_AT(des, i, des_col) -= MAT2D_AT(src, i, src_col);
02012     }
02013 }
02014

```

```

02025 void mat2D_sub_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
02026 {
02027     MAT2D_ASSERT(src_row < src.rows);
02028     MAT2D_ASSERT(des.cols == src.cols);
02029     MAT2D_ASSERT(des_row < des.rows);
02030
02031     for (size_t j = 0; j < des.cols; j++) {
02032         MAT2D_AT(des, des_row, j) -= MAT2D_AT(src, src_row, j);
02033     }
02034 }
02035
02045 void mat2D_sub_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor)
02046 {
02047     for (size_t j = 0; j < m.cols; ++j) {
02048         MAT2D_AT(m, des_r, j) -= factor * MAT2D_AT(m, src_r, j);
02049     }
02050 }
02051
02080 void mat2D_SVD_full(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v, bool
    return_v_transpose)
02081 {
02082     mat2D_SVD_thin(A, U, S, V, init_vec_u, init_vec_v, false);
02083
02084     Mat2D U_full = mat2D_alloc(U.rows, U.cols);
02085     Mat2D V_full = mat2D_alloc(V.rows, V.cols);
02086
02087     mat2D_make_orthogonal_modified_Gram_Schmidt(U_full, U);
02088     mat2D_make_orthogonal_modified_Gram_Schmidt(V_full, V);
02089
02090     mat2D_copy(U, U_full);
02091     if (return_v_transpose) {
02092         mat2D_transpose(V, V_full);
02093     } else {
02094         mat2D_copy(V, V_full);
02095     }
02096
02097     mat2D_free(U_full);
02098     mat2D_free(V_full);
02099 }
02100
02149 void mat2D_SVD_thin(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v, bool
    return_v_transpose)
02150 {
02151     /* https://www.youtube.com/watch?v=nbBvuunVfco */
02152     /* https://en.wikipedia.org/wiki/Singular_value_decomposition */
02153     size_t n = A.rows;
02154     size_t m = A.cols;
02155     MAT2D_ASSERT(U.rows == n);
02156     MAT2D_ASSERT(U.cols == n);
02157     MAT2D_ASSERT(S.rows == n);
02158     MAT2D_ASSERT(S.cols == m);
02159     MAT2D_ASSERT(V.rows == m);
02160     MAT2D_ASSERT(V.cols == m);
02161     MAT2D_ASSERT(init_vec_u.rows == n);
02162     MAT2D_ASSERT(init_vec_u.cols == 1);
02163     MAT2D_ASSERT(init_vec_v.rows == m);
02164     MAT2D_ASSERT(init_vec_v.cols == 1);
02165
02166     mat2D_fill(U, 0);
02167     mat2D_fill(S, 0);
02168     mat2D_fill(V, 0);
02169
02170     Mat2D AT = mat2D_alloc(m, n);
02171     mat2D_transpose(AT, A);
02172
02173     if (n <= m) {
02174         Mat2D AAT = mat2D_alloc(n, n);
02175         Mat2D left_eigenvalues = mat2D_alloc(n, n);
02176         Mat2D left_eigenvectors = mat2D_alloc(n, n);
02177         Mat2D temp_u_vec = mat2D_alloc(n, 1);
02178         Mat2D temp_v_vec = mat2D_alloc(m, 1);
02179         mat2D_dot(AAT, A, AT);
02180         mat2D_eig_power_iteration(AAT, left_eigenvalues, left_eigenvectors, init_vec_u, 0);
02181         /* fill matrix sigma (S) */
02182         size_t non_zero_n = 0;
02183         for (size_t i = 0; i < n; i++) {
02184             if (MAT2D_IS_ZERO(MAT2D_AT(left_eigenvalues, i, i)) || MAT2D_AT(left_eigenvalues, i, i) <
02185                 0) {
02186                 MAT2D_AT(S, i, i) = 0; /* AAT is positive definit */
02187             } else {
02188                 MAT2D_AT(S, i, i) = sqrt(MAT2D_AT(left_eigenvalues, i, i));
02189                 non_zero_n++;
02190             }
02191         }
02192         for (size_t c = 0; c < non_zero_n; c++) {
02193             mat2D_copy_col_from_src_to_des(U, c, left_eigenvectors, c);
02194             mat2D_copy_col_from_src_to_des(temp_u_vec, 0, left_eigenvectors, c);
02195         }
02196     }

```

```

02199         mat2D_dot(temp_v_vec, AT, temp_u_vec);
02200         mat2D_mult(temp_v_vec, 1.0 / MAT2D_AT(S, c, c));
02201         mat2D_copy_col_from_src_to_des(V, c, temp_v_vec, 0);
02202     }
02203     mat2D_free(AAT);
02204     mat2D_free(left_eigenvalues);
02205     mat2D_free(left_eigenvectors);
02206     mat2D_free(temp_u_vec);
02207     mat2D_free(temp_v_vec);
02208 } else {
02209     Mat2D ATA = mat2D_alloc(m, m);
02210     Mat2D right_eigenvalues = mat2D_alloc(m, m);
02211     Mat2D right_eigenvectors = mat2D_alloc(m, m);
02212     Mat2D temp_u_vec = mat2D_alloc(n, 1);
02213     Mat2D temp_v_vec = mat2D_alloc(m, 1);
02214     mat2D_dot(ATA, AT, A);
02215     mat2D_eig_power_iteration(ATA, right_eigenvalues, right_eigenvectors, init_vec_v, 0);
02216     /* fill matrix sigma (S) */
02217     size_t non_zero_m = 0;
02218     for (size_t i = 0; i < m; i++) {
02219         if (MAT2D_IS_ZERO(MAT2D_AT(right_eigenvalues, i, i)) || MAT2D_AT(right_eigenvalues, i, i)
02220 < 0) {
02221             MAT2D_AT(S, i, i) = 0; /* ATA is positive definit */
02222         } else {
02223             MAT2D_AT(S, i, i) = sqrt(MAT2D_AT(right_eigenvalues, i, i));
02224             non_zero_m++;
02225         }
02226     }
02227     for (size_t c = 0; c < non_zero_m; c++) {
02228         mat2D_copy_col_from_src_to_des(V, c, right_eigenvectors, c);
02229         mat2D_copy_col_from_src_to_des(temp_v_vec, 0, right_eigenvectors, c);
02230         mat2D_dot(temp_u_vec, A, temp_v_vec);
02231         mat2D_mult(temp_u_vec, 1.0 / MAT2D_AT(S, c, c));
02232         mat2D_copy_col_from_src_to_des(U, c, temp_u_vec, 0);
02233     }
02234     mat2D_free(ATA);
02235     mat2D_free(right_eigenvalues);
02236     mat2D_free(right_eigenvectors);
02237     mat2D_free(temp_u_vec);
02238     mat2D_free(temp_v_vec);
02239 }
02240
02241 if (return_v_transpose) {
02242     Mat2D v_trans = mat2D_alloc(V.cols, V.rows);
02243     mat2D_transpose(v_trans, V);
02244     mat2D_copy(V, v_trans);
02245     mat2D_free(v_trans);
02246 }
02247
02248 mat2D_free(AT);
02249 }
02250
02251 void mat2D_swap_rows(Mat2D m, size_t r1, size_t r2)
02252 {
02253     for (size_t j = 0; j < m.cols; j++) {
02254         double temp = MAT2D_AT(m, r1, j);
02255         MAT2D_AT(m, r1, j) = MAT2D_AT(m, r2, j);
02256         MAT2D_AT(m, r2, j) = temp;
02257     }
02258 }
02259
02260 void mat2D_transpose(Mat2D des, Mat2D src)
02261 {
02262     MAT2D_ASSERT(des.cols == src.rows);
02263     MAT2D_ASSERT(des.rows == src.cols);
02264
02265     for (size_t index = 0; index < des.rows; ++index) {
02266         for (size_t jindex = 0; jindex < des.cols; ++jindex) {
02267             MAT2D_AT(des, index, jindex) = MAT2D_AT(src, jindex, index);
02268         }
02269     }
02270 }
02271
02272 double mat2D_upper_triangularize(Mat2D m, uint8_t flags)
02273 {
02274     /* preforming Gauss elimination: https://en.wikipedia.org/wiki/Gaussian_elimination */
02275     /* returns the factor multiplying the determinant */
02276
02277     double factor_to_return = 1;
02278
02279     size_t r = 0;
02280     for (size_t c = 0; c < m.cols && r < m.rows; c++) {
02281         if (flags & MAT2D_ROW_SWAPPING) {
02282             /* finding biggest first number (absolute value); partial pivoting */
02283             size_t piv = r;
02284             double best = fabs(MAT2D_AT(m, r, c));

```

```

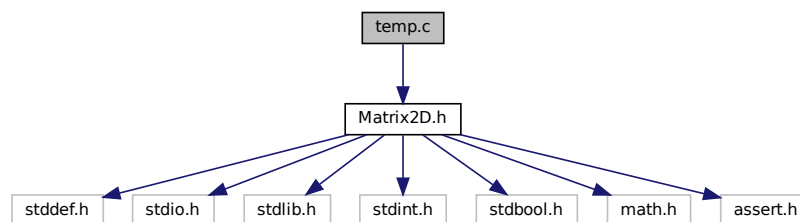
02323         for (size_t i = r + 1; i < m.rows; i++) {
02324             double v = fabs(MAT2D_AT(m, i, c));
02325             if (v > best) {
02326                 best = v;
02327                 piv = i;
02328             }
02329         }
02330         if (MAT2D_IS_ZERO(best)) {
02331             continue; /* move to next column, same pivot row r */
02332         }
02333         if (piv != r) {
02334             mat2D_swap_rows(m, piv, r);
02335             factor_to_return *= -1.0;
02336         }
02337     }
02338
02339     double pivot = MAT2D_AT(m, r, c);
02340     MAT2D_ASSERT(!MAT2D_IS_ZERO(pivot));
02341
02342     if (flags & MAT2D_ONES_ON_DIAG) {
02343         mat2D_mult_row(m, r, 1.0 / pivot);
02344         factor_to_return *= pivot;
02345         pivot = 1.0;
02346     }
02347
02348     /* Eliminate entries below pivot in column c */
02349     for (size_t i = r + 1; i < m.rows; i++) {
02350         double f = MAT2D_AT(m, i, c) / pivot;
02351         mat2D_sub_row_time_factor_to_row(m, i, r, f);
02352     }
02353     r++;
02354 }
02355 return factor_to_return;
02356 }
02357
02358 #endif // MATRIX2D_IMPLEMENTATION

```

4.5 temp.c File Reference

```
#include "Matrix2D.h"
```

Include dependency graph for temp.c:



Macros

- `#define` [MATRIX2D_IMPLEMENTATION](#)

Functions

- `int` [main](#) (void)

4.5.1 Macro Definition Documentation

4.5.1.1 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

Definition at line 1 of file [temp.c](#).

4.5.2 Function Documentation

4.5.2.1 main()

```
int main (
    void )
```

Definition at line 4 of file [temp.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_AT](#), [mat2D_dot\(\)](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), [MAT2D_PRINT](#), [mat2D_rand\(\)](#), [mat2D_SVD_full\(\)](#), and [Mat2D::rows](#).

4.6 temp.c

```
00001 #define MATRIX2D_IMPLEMENTATION
00002 #include "Matrix2D.h"
00003
00004 int main(void)
00005 {
00006     int n = 4;
00007     int m = 5;
00008
00009     Mat2D A = mat2D_alloc(n, m);
00010     Mat2D U = mat2D_alloc(n, n);
00011     Mat2D S = mat2D_alloc(n, m);
00012     Mat2D VT = mat2D_alloc(m, m);
00013     Mat2D SV = mat2D_alloc(n, m);
00014     Mat2D USVT = mat2D_alloc(n, m);
00015     Mat2D init_vec_u = mat2D_alloc(U.rows, 1);
00016     Mat2D init_vec_v = mat2D_alloc(VT.rows, 1);
00017
00018     mat2D_rand(init_vec_u, 0, 1);
00019     mat2D_rand(init_vec_v, 0, 1);
00020
00021     mat2D_fill(A, 0);
00022
00023     MAT2D_AT(A, 0, 0) = 1;
00024     MAT2D_AT(A, 0, 4) = 2;
00025     MAT2D_AT(A, 1, 2) = 3;
00026     MAT2D_AT(A, 3, 1) = 2;
00027
00028     mat2D_SVD_full(A, U, S, VT, init_vec_u, init_vec_v, 1);
00029
00030     MAT2D_PRINT(A);
00031     MAT2D_PRINT(U);
00032     MAT2D_PRINT(S);
00033     MAT2D_PRINT(VT);
00034
00035     mat2D_dot(SV, S, VT);
00036     mat2D_dot(USVT, U, SV);
00037
```

```

00038     MAT2D_PRINT (USVT);
00039
00040     mat2D_free(A);
00041     mat2D_free(U);
00042     mat2D_free(S);
00043     mat2D_free(VT);
00044     mat2D_free(SV);
00045     mat2D_free(USVT);
00046     mat2D_free(init_vec_u);
00047     mat2D_free(init_vec_v);
00048
00049     return 0;
00050 }

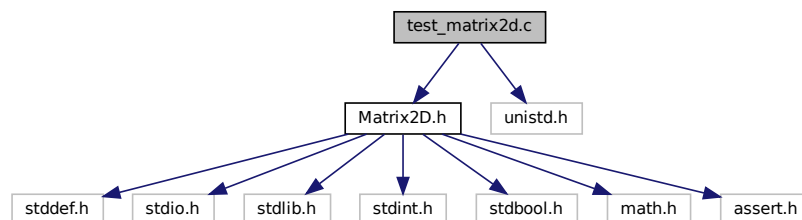
```

4.7 test_matrix2d.c File Reference

```
#include "Matrix2D.h"
```

```
#include <unistd.h>
```

Include dependency graph for test_matrix2d.c:



Macros

- `#define` [MATRIX2D_IMPLEMENTATION](#)
- `#define` [RUN_TEST](#)(fn)

Functions

- static uint64_t [xorshift64star](#) (uint64_t *state)
- static double [rng_unit01](#) (uint64_t *state)
- static double [rng_range](#) (uint64_t *state, double low, double high)
- static int [close_rel_abs](#) (double a, double b, double abs_eps, double rel_eps)
- static void [fill_strictly_diag_dominant](#) (Mat2D a, uint64_t *rng)
- static double [det_by_minors_first_col](#) (Mat2D a)
- static int [nearly_equal](#) (double a, double b, double eps)
- static void [assert_mat_close](#) (Mat2D a, Mat2D b, double eps)
- static void [assert_identity_close](#) (Mat2D m, double eps)
- static void [fill_mat_from_array](#) (Mat2D m, const double *data)
- static void [assert_permutation_matrix](#) (Mat2D p)
- static void [assert_inverse_identity_both_sides](#) (Mat2D a, double eps)
- static void [test_alloc_fill_copy_add_sub](#) (void)
- static void [test_transpose](#) (void)
- static void [test_dot_product_matrix_multiply](#) (void)
- static void [test_det_and_minor_det_agree_3x3](#) (void)

- static void [test_invert](#) (void)
- static void [test_LUP_decomposition_identity_P_no_swap_case](#) (void)
- static void [test_LUP_decomposition_swap_required_case](#) (void)
- static void [test_uint32_alloc_fill_and_at](#) (void)
- static void [test_offset2d_and_stride](#) (void)
- static void [test_copy_windows](#) (void)
- static void [test_non_contiguous_stride_views](#) (void)
- static void [test_row_col_ops_and_scaling](#) (void)
- static void [test_shift_and_identity](#) (void)
- static void [test_norms_and_normalize](#) (void)
- static void [test_outer_product_and_cross](#) (void)
- static void [test_det_2x2_and_upper_triangularize_sign](#) (void)
- static void [test_minor_det_matches_gauss_4x4_known](#) (void)
- static void [test_reduce_rank](#) (void)
- static void [test_rotation_matrices_orthonormal](#) (void)
- static void [test_DCM_zyx_matches_product](#) (void)
- static void [test_solve_linear_system_LUP](#) (void)
- static void [test_rand_range](#) (void)
- static void [test_copy_row_and_col_helpers](#) (void)
- static void [test_dot_product_and_vector_variants](#) (void)
- static void [test_outer_product_row_vector_path](#) (void)
- static void [test_det_early_zero_row_and_zero_col_paths](#) (void)
- static void [test_mat_is_all_digit](#) (void)
- static void [test_power_iterate_and_eig_helpers](#) (void)
- static void [test_deterministic_fuzz_loop](#) (void)
- int [main](#) (void)

4.7.1 Macro Definition Documentation

4.7.1.1 MATRIX2D_IMPLEMENTATION

```
#define MATRIX2D_IMPLEMENTATION
```

tests/test_matrix2d.c

written by AI

Definition at line 7 of file [test_matrix2d.c](#).

4.7.1.2 RUN_TEST

```
#define RUN_TEST(  
    fn )
```

Value:

```
do {  
    fn();  
    printf("[INFO] passed | %s\n", \br/>        #fn);  
    fflush(stdout);  
} while (0)
```

4.7.2 Function Documentation

4.7.2.1 `assert_identity_close()`

```
static void assert_identity_close (
    Mat2D m,
    double eps ) [static]
```

Definition at line 94 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [nearly_equal\(\)](#), and [Mat2D::rows](#).

Referenced by [assert_inverse_identity_both_sides\(\)](#), [test_deterministic_fuzz_loop\(\)](#), and [test_rotation_matrices_orthonormal\(\)](#).

4.7.2.2 `assert_inverse_identity_both_sides()`

```
static void assert_inverse_identity_both_sides (
    Mat2D a,
    double eps ) [static]
```

Definition at line 141 of file [test_matrix2d.c](#).

References [assert_identity_close\(\)](#), [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_dot\(\)](#), [mat2D_free\(\)](#), [mat2D_invert\(\)](#), and [Mat2D::rows](#).

Referenced by [test_invert\(\)](#).

4.7.2.3 `assert_mat_close()`

```
static void assert_mat_close (
    Mat2D a,
    Mat2D b,
    double eps ) [static]
```

Definition at line 72 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [nearly_equal\(\)](#), and [Mat2D::rows](#).

Referenced by [test_alloc_fill_copy_add_sub\(\)](#), [test_copy_windows\(\)](#), [test_DCM_zyx_matches_product\(\)](#), [test_LUP_decomposition_identity_P_no_swap_case\(\)](#), [test_LUP_decomposition_swap_required_case\(\)](#), and [test_non_contiguous_stride_views\(\)](#).

4.7.2.4 assert_permutation_matrix()

```
static void assert_permutation_matrix (
    Mat2D p ) [static]
```

Definition at line 114 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_det\(\)](#), [MAT2D_EPS](#), [nearly_equal\(\)](#), and [Mat2D::rows](#).

Referenced by [test_LUP_decomposition_swap_required_case\(\)](#).

4.7.2.5 close_rel_abs()

```
static int close_rel_abs (
    double a,
    double b,
    double abs_eps,
    double rel_eps ) [static]
```

Definition at line 39 of file [test_matrix2d.c](#).

Referenced by [test_deterministic_fuzz_loop\(\)](#), and [test_power_iterate_and_eig_helpers\(\)](#).

4.7.2.6 det_by_minors_first_col()

```
static double det_by_minors_first_col (
    Mat2D a ) [static]
```

Definition at line 659 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [Mat2D_Minor::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_det_2x2_mat\(\)](#), [mat2D_det_2x2_mat_minor\(\)](#), [mat2D_minor_alloc_fill_from_mat\(\)](#), [mat2D_minor_det\(\)](#), [mat2D_minor_free\(\)](#), [Mat2D::rows](#), and [Mat2D_Minor::rows](#).

Referenced by [test_det_and_minor_det_agree_3x3\(\)](#), and [test_minor_det_matches_gauss_4x4_known\(\)](#).

4.7.2.7 fill_mat_from_array()

```
static void fill_mat_from_array (
    Mat2D m,
    const double * data ) [static]
```

Definition at line 105 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [MAT2D_AT](#), and [Mat2D::rows](#).

Referenced by [test_invert\(\)](#), and [test_LUP_decomposition_swap_required_case\(\)](#).

4.7.2.8 fill_strictly_diag_dominant()

```
static void fill_strictly_diag_dominant (
    Mat2D a,
    uint64_t * rng ) [static]
```

Definition at line 46 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [rng_range\(\)](#), and [Mat2D::rows](#).

Referenced by [test_deterministic_fuzz_loop\(\)](#).

4.7.2.9 main()

```
int main (
    void )
```

Definition at line 1163 of file [test_matrix2d.c](#).

References [RUN_TEST](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_copy_row_and_col_helpers\(\)](#), [test_copy_windows\(\)](#), [test_DCM_zyx_matches_product\(\)](#), [test_det_2x2_and_upper_triangulate_sign\(\)](#), [test_det_and_minor_det_agree_3x3\(\)](#), [test_det_early_zero_row_and_zero_col_paths\(\)](#), [test_deterministic_fuzz_loop\(\)](#), [test_dot_product_and_vector_variants\(\)](#), [test_dot_product_matrix_multiply\(\)](#), [test_invert\(\)](#), [test_LUP_decomposition_identity_P_no_swap_case\(\)](#), [test_LUP_decomposition_swap_rows\(\)](#), [test_mat_is_all_digit\(\)](#), [test_minor_det_matches_gauss_4x4_known\(\)](#), [test_non_contiguous_stride_views\(\)](#), [test_norms_and_normalize\(\)](#), [test_offset2d_and_stride\(\)](#), [test_outer_product_and_cross\(\)](#), [test_outer_product_row_vector_path\(\)](#), [test_power_iterate_and_eig_helpers\(\)](#), [test_rand_range\(\)](#), [test_reduce_rank\(\)](#), [test_rotation_matrices_orthonormal\(\)](#), [test_row_col_ops_and_scaling\(\)](#), [test_shift_and_identity\(\)](#), [test_solve_linear_system_LUP\(\)](#), [test_transpose\(\)](#), and [test_uint32_alloc_fill_and_at\(\)](#).

4.7.2.10 nearly_equal()

```
static int nearly_equal (
    double a,
    double b,
    double eps ) [static]
```

Definition at line 67 of file [test_matrix2d.c](#).

Referenced by [assert_identity_close\(\)](#), [assert_mat_close\(\)](#), [assert_permutation_matrix\(\)](#), [test_alloc_fill_copy_add_sub\(\)](#), [test_copy_row_and_col_helpers\(\)](#), [test_copy_windows\(\)](#), [test_det_2x2_and_upper_triangulate_sign\(\)](#), [test_det_and_minor_det_agree_3x3\(\)](#), [test_det_early_zero_row_and_zero_col_paths\(\)](#), [test_dot_product_and_vector_variants\(\)](#), [test_dot_product_matrix_multiply\(\)](#), [test_minor_det_matches_gauss_4x4_known\(\)](#), [test_non_contiguous_stride_views\(\)](#), [test_norms_and_normalize\(\)](#), [test_offset2d_and_stride\(\)](#), [test_outer_product_and_cross\(\)](#), [test_outer_product_row_vector_path\(\)](#), [test_rotation_matrices_orthonormal\(\)](#), [test_row_col_ops_and_scaling\(\)](#), [test_shift_and_identity\(\)](#), [test_solve_linear_system_LUP\(\)](#), and [test_transpose\(\)](#).

4.7.2.11 rng_range()

```
static double rng_range (
    uint64_t * state,
    double low,
    double high ) [static]
```

Definition at line 34 of file [test_matrix2d.c](#).

References [rng_unit01\(\)](#).

Referenced by [fill_strictly_diag_dominant\(\)](#).

4.7.2.12 rng_unit01()

```
static double rng_unit01 (
    uint64_t * state ) [static]
```

Definition at line 26 of file [test_matrix2d.c](#).

References [xorshift64star\(\)](#).

Referenced by [rng_range\(\)](#).

4.7.2.13 test_alloc_fill_copy_add_sub()

```
static void test_alloc_fill_copy_add_sub (
    void ) [static]
```

Definition at line 164 of file [test_matrix2d.c](#).

References [assert_mat_close\(\)](#), [Mat2D::cols](#), [mat2D_add\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_copy\(\)](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), [mat2D_sub\(\)](#), [nearly_equal\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#).

4.7.2.14 test_copy_row_and_col_helpers()

```
static void test_copy_row_and_col_helpers (
    void ) [static]
```

Definition at line 884 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_copy_col_from_src_to_des\(\)](#), [mat2D_copy_row_from_src_to_des\(\)](#), [mat2D_fill\(\)](#), [mat2D_fill_sequence\(\)](#), [mat2D_free\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.15 test_copy_windows()

```
static void test_copy_windows (
    void ) [static]
```

Definition at line 435 of file [test_matrix2d.c](#).

References [assert_mat_close\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_copy_src_to_des_window\(\)](#), [mat2D_copy_src_window_to_des\(\)](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.16 test_DCM_zyx_matches_product()

```
static void test_DCM_zyx_matches_product (
    void ) [static]
```

Definition at line 793 of file [test_matrix2d.c](#).

References [assert_mat_close\(\)](#), [mat2D_alloc\(\)](#), [mat2D_dot\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), [mat2D_set_DCM_zyx\(\)](#), [mat2D_set_rot_mat_x\(\)](#), [mat2D_set_rot_mat_y\(\)](#), and [mat2D_set_rot_mat_z\(\)](#).

Referenced by [main\(\)](#).

4.7.2.17 test_det_2x2_and_upper_triangulate_sign()

```
static void test_det_2x2_and_upper_triangulate_sign (
    void ) [static]
```

Definition at line 637 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_copy\(\)](#), [mat2D_det\(\)](#), [mat2D_det_2x2_mat\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), [MAT2D_ROW_SWAPPING](#), [mat2D_upper_triangulate\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.18 test_det_and_minor_det_agree_3x3()

```
static void test_det_and_minor_det_agree_3x3 (
    void ) [static]
```

Definition at line 259 of file [test_matrix2d.c](#).

References [det_by_minors_first_col\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_det\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.19 test_det_early_zero_row_and_zero_col_paths()

```
static void test_det_early_zero_row_and_zero_col_paths (  
    void ) [static]
```

Definition at line 981 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_col_is_all_digit\(\)](#), [mat2D_det\(\)](#), [mat2D_free\(\)](#), [mat2D_row_is_all_digit\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.20 test_deterministic_fuzz_loop()

```
static void test_deterministic_fuzz_loop (  
    void ) [static]
```

Definition at line 1074 of file [test_matrix2d.c](#).

References [assert_identity_close\(\)](#), [close_rel_abs\(\)](#), [fill_strictly_diag_dominant\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_copy\(\)](#), [mat2D_det\(\)](#), [mat2D_dot\(\)](#), [mat2D_free\(\)](#), [mat2D_invert\(\)](#), [mat2D_reduce\(\)](#), [mat2D_transpose\(\)](#), and [xorshift64star\(\)](#).

Referenced by [main\(\)](#).

4.7.2.21 test_dot_product_and_vector_variants()

```
static void test_dot_product_and_vector_variants (  
    void ) [static]
```

Definition at line 914 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_calc_norma_inf\(\)](#), [mat2D_dot_product\(\)](#), [mat2D_free\(\)](#), [mat2D_inner_product\(\)](#), [mat2D_normalize_inf](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.22 test_dot_product_matrix_multiply()

```
static void test_dot_product_matrix_multiply (  
    void ) [static]
```

Definition at line 217 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_dot\(\)](#), [mat2D_free\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.23 test_invert()

```
static void test_invert (
    void ) [static]
```

Definition at line 287 of file [test_matrix2d.c](#).

References [assert_inverse_identity_both_sides\(\)](#), [fill_mat_from_array\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [mat2D_det\(\)](#), [MAT2D_EPS](#), and [mat2D_free\(\)](#).

Referenced by [main\(\)](#).

4.7.2.24 test_LUP_decomposition_identity_P_no_swap_case()

```
static void test_LUP_decomposition_identity_P_no_swap_case (
    void ) [static]
```

Definition at line 331 of file [test_matrix2d.c](#).

References [assert_mat_close\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_AT](#), [mat2D_dot\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), and [mat2D_LUP_decomposition_with_swap\(\)](#).

Referenced by [main\(\)](#).

4.7.2.25 test_LUP_decomposition_swap_required_case()

```
static void test_LUP_decomposition_swap_required_case (
    void ) [static]
```

Definition at line 369 of file [test_matrix2d.c](#).

References [assert_mat_close\(\)](#), [assert_permutation_matrix\(\)](#), [fill_mat_from_array\(\)](#), [mat2D_alloc\(\)](#), [mat2D_dot\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), and [mat2D_LUP_decomposition_with_swap\(\)](#).

Referenced by [main\(\)](#).

4.7.2.26 test_mat_is_all_digit()

```
static void test_mat_is_all_digit (
    void ) [static]
```

Definition at line 1007 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), and [mat2D_mat_is_all_digit\(\)](#).

Referenced by [main\(\)](#).

4.7.2.27 test_minor_det_matches_gauss_4x4_known()

```
static void test_minor_det_matches_gauss_4x4_known (
    void ) [static]
```

Definition at line 695 of file [test_matrix2d.c](#).

References [det_by_minors_first_col\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_det\(\)](#), [mat2D_free\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.28 test_non_contiguous_stride_views()

```
static void test_non_contiguous_stride_views (
    void ) [static]
```

Definition at line 467 of file [test_matrix2d.c](#).

References [assert_mat_close\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_copy\(\)](#), [mat2D_dot\(\)](#), [MAT2D_FREE](#), [mat2D_free\(\)](#), [MAT2D_MALLOC](#), [mat2D_offset2d\(\)](#), [mat2D_set_identity\(\)](#), [mat2D_transpose\(\)](#), [nearly_equal\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#).

4.7.2.29 test_norms_and_normalize()

```
static void test_norms_and_normalize (
    void ) [static]
```

Definition at line 577 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_calc_norma\(\)](#), [mat2D_calc_norma_inf\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), [mat2D_inner_product\(\)](#), [mat2D_normalize](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.30 test_offset2d_and_stride()

```
static void test_offset2d_and_stride (
    void ) [static]
```

Definition at line 420 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_fill_sequence\(\)](#), [mat2D_free\(\)](#), [mat2D_offset2d\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.31 test_outer_product_and_cross()

```
static void test_outer_product_and_cross (
    void ) [static]
```

Definition at line 595 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_cross\(\)](#), [MAT2D_EPS](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), [mat2D_outer_product\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.32 test_outer_product_row_vector_path()

```
static void test_outer_product_row_vector_path (
    void ) [static]
```

Definition at line 955 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_free\(\)](#), [mat2D_outer_product\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.33 test_power_iterate_and_eig_helpers()

```
static void test_power_iterate_and_eig_helpers (
    void ) [static]
```

Definition at line 1018 of file [test_matrix2d.c](#).

References [close_rel_abs\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_calc_norma_inf\(\)](#), [mat2D_eig_check\(\)](#), [mat2D_eig_power_iteration\(\)](#), [mat2D_fill\(\)](#), [mat2D_free\(\)](#), and [mat2D_power_iterate\(\)](#).

Referenced by [main\(\)](#).

4.7.2.34 test_rand_range()

```
static void test_rand_range (
    void ) [static]
```

Definition at line 867 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_free\(\)](#), [mat2D_rand\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#).

4.7.2.35 test_reduce_rank()

```
static void test_reduce_rank (
    void ) [static]
```

Definition at line 727 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_free\(\)](#), and [mat2D_reduce\(\)](#).

Referenced by [main\(\)](#).

4.7.2.36 test_rotation_matrices_orthonormal()

```
static void test_rotation_matrices_orthonormal (
    void ) [static]
```

Definition at line 750 of file [test_matrix2d.c](#).

References [assert_identity_close\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_det\(\)](#), [mat2D_dot\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), [mat2D_set_rot_mat_x\(\)](#), [mat2D_set_rot_mat_y\(\)](#), [mat2D_set_rot_mat_z\(\)](#), [mat2D_transpose\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.37 test_row_col_ops_and_scaling()

```
static void test_row_col_ops_and_scaling (
    void ) [static]
```

Definition at line 517 of file [test_matrix2d.c](#).

References [mat2D_add_col_to_col\(\)](#), [mat2D_add_row_time_factor_to_row\(\)](#), [mat2D_add_row_to_row\(\)](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_fill_sequence\(\)](#), [mat2D_free\(\)](#), [mat2D_mult_row\(\)](#), [mat2D_sub_col_to_col\(\)](#), [mat2D_sub_row_to_row\(\)](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.38 test_shift_and_identity()

```
static void test_shift_and_identity (
    void ) [static]
```

Definition at line 561 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_free\(\)](#), [mat2D_set_identity\(\)](#), [mat2D_shift\(\)](#), [nearly_equal\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#).

4.7.2.39 test_solve_linear_system_LUP()

```
static void test_solve_linear_system_LUP (  
    void ) [static]
```

Definition at line 826 of file [test_matrix2d.c](#).

References [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_dot\(\)](#), [MAT2D_EPS](#), [mat2D_free\(\)](#), [mat2D_solve_linear_sys_LUP](#), and [nearly_equal\(\)](#).

Referenced by [main\(\)](#).

4.7.2.40 test_transpose()

```
static void test_transpose (  
    void ) [static]
```

Definition at line 189 of file [test_matrix2d.c](#).

References [Mat2D::cols](#), [mat2D_alloc\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT](#), [mat2D_free\(\)](#), [mat2D_transpose\(\)](#), [nearly_equal\(\)](#), and [Mat2D::rows](#).

Referenced by [main\(\)](#).

4.7.2.41 test_uint32_alloc_fill_and_at()

```
static void test_uint32_alloc_fill_and_at (  
    void ) [static]
```

Definition at line 407 of file [test_matrix2d.c](#).

References [mat2D_alloc_uint32\(\)](#), [MAT2D_ASSERT](#), [MAT2D_AT_UINT32](#), [mat2D_fill_uint32\(\)](#), and [mat2D_free_uint32\(\)](#).

Referenced by [main\(\)](#).

4.7.2.42 xorshift64star()

```
static uint64_t xorshift64star (  
    uint64_t * state ) [static]
```

Definition at line 15 of file [test_matrix2d.c](#).

Referenced by [rng_unit01\(\)](#), and [test_deterministic_fuzz_loop\(\)](#).

4.8 test_matrix2d.c

```

00001
00007 #define MATRIX2D_IMPLEMENTATION
00008 #include "Matrix2D.h"
00009 #include <unistd.h> /* dup/dup2/close for silencing stdout in print-smoke tests */
00010
00011 /* ----- */
00012 /* Deterministic fuzz helpers (no libc rand(); stable across platforms) */
00013 /* ----- */
00014
00015 static uint64_t xorshift64star(uint64_t *state)
00016 {
00017     /* Marsaglia xorshift* variant */
00018     uint64_t x = *state;
00019     x ^= x >> 12;
00020     x ^= x << 25;
00021     x ^= x >> 27;
00022     *state = x;
00023     return x * 2685821657736338717ULL;
00024 }
00025
00026 static double rng_unit01(uint64_t *state)
00027 {
00028     /* Convert top 53 bits to a double in [0, 1). */
00029     uint64_t x = xorshift64star(state);
00030     uint64_t top53 = x >> 11;
00031     return (double)top53 * (1.0 / 9007199254740992.0); /* 2^53 */
00032 }
00033
00034 static double rng_range(uint64_t *state, double low, double high)
00035 {
00036     return low + (high - low) * rng_unit01(state);
00037 }
00038
00039 static int close_rel_abs(double a, double b, double abs_eps, double rel_eps)
00040 {
00041     double diff = fabs(a - b);
00042     if (diff <= abs_eps) return 1;
00043     return diff <= rel_eps * fmax(fabs(a), fabs(b));
00044 }
00045
00046 static void fill_strictly_diag_dominant(Mat2D a, uint64_t *rng)
00047 {
00048     /* Strict diagonal dominance => nonsingular (and usually stable to invert). */
00049     MAT2D_ASSERT(a.rows == a.cols);
00050     double mag = 2.0;
00051
00052     for (size_t i = 0; i < a.rows; i++) {
00053         double row_sum = 0.0;
00054         for (size_t j = 0; j < a.cols; j++) {
00055             if (i == j) continue;
00056             double v = rng_range(rng, -mag, mag);
00057             MAT2D_AT(a, i, j) = v;
00058             row_sum += fabs(v);
00059         }
00060         MAT2D_AT(a, i, i) = row_sum + 1.0;
00061     }
00062 }
00063
00064 static double det_by_minors_first_col(Mat2D a);
00065
00066 static int nearly_equal(double a, double b, double eps)
00067 {
00068     return fabs(a - b) <= eps;
00069 }
00070
00071 static void assert_mat_close(Mat2D a, Mat2D b, double eps)
00072 {
00073     MAT2D_ASSERT(a.rows == b.rows);
00074     MAT2D_ASSERT(a.cols == b.cols);
00075
00076     for (size_t i = 0; i < a.rows; i++) {
00077         for (size_t j = 0; j < a.cols; j++) {
00078             double va = MAT2D_AT(a, i, j);
00079             double vb = MAT2D_AT(b, i, j);
00080             if (!nearly_equal(va, vb, eps)) {
00081                 fprintf(stderr,
00082                     "Matrix mismatch at (%zu,%zu): a=%g b=%g\n",
00083                     i,
00084                     j,
00085                     va,
00086                     vb);
00087                 MAT2D_ASSERT(0);
00088             }
00089         }
00090     }

```

```

00091     }
00092 }
00093
00094 static void assert_identity_close(Mat2D m, double eps)
00095 {
00096     MAT2D_ASSERT(m.rows == m.cols);
00097     for (size_t i = 0; i < m.rows; i++) {
00098         for (size_t j = 0; j < m.cols; j++) {
00099             double expected = (i == j) ? 1.0 : 0.0;
00100             MAT2D_ASSERT(nearly_equal(MAT2D_AT(m, i, j), expected, eps));
00101         }
00102     }
00103 }
00104
00105 static void fill_mat_from_array(Mat2D m, const double *data)
00106 {
00107     for (size_t i = 0; i < m.rows; i++) {
00108         for (size_t j = 0; j < m.cols; j++) {
00109             MAT2D_AT(m, i, j) = data[i * m.cols + j];
00110         }
00111     }
00112 }
00113
00114 static void assert_permutation_matrix(Mat2D p)
00115 {
00116     MAT2D_ASSERT(p.rows == p.cols);
00117
00118     for (size_t i = 0; i < p.rows; i++) {
00119         double row_sum = 0.0;
00120         for (size_t j = 0; j < p.cols; j++) {
00121             double v = MAT2D_AT(p, i, j);
00122             MAT2D_ASSERT(v == 0.0 || v == 1.0);
00123             row_sum += v;
00124         }
00125         MAT2D_ASSERT(nearly_equal(row_sum, 1.0, 0.0));
00126     }
00127
00128     for (size_t j = 0; j < p.cols; j++) {
00129         double col_sum = 0.0;
00130         for (size_t i = 0; i < p.rows; i++) {
00131             col_sum += MAT2D_AT(p, i, j);
00132         }
00133         MAT2D_ASSERT(nearly_equal(col_sum, 1.0, 0.0));
00134     }
00135
00136     /* det(P) must be ±1 for a permutation matrix */
00137     double dp = mat2D_det(p);
00138     MAT2D_ASSERT(nearly_equal(fabs(dp), 1.0, MAT2D_EPS));
00139 }
00140
00141 static void assert_inverse_identity_both_sides(Mat2D a, double eps)
00142 {
00143     MAT2D_ASSERT(a.rows == a.cols);
00144
00145     Mat2D inv = mat2D_alloc(a.rows, a.cols);
00146     Mat2D prod1 = mat2D_alloc(a.rows, a.cols);
00147     Mat2D prod2 = mat2D_alloc(a.rows, a.cols);
00148
00149     mat2D_invert(inv, a);
00150
00151     /* A * inv(A) == I */
00152     mat2D_dot(prod1, a, inv);
00153     assert_identity_close(prod1, eps);
00154
00155     /* inv(A) * A == I (catches some subtle multiply/invert issues) */
00156     mat2D_dot(prod2, inv, a);
00157     assert_identity_close(prod2, eps);
00158
00159     mat2D_free(inv);
00160     mat2D_free(prod1);
00161     mat2D_free(prod2);
00162 }
00163
00164 static void test_alloc_fill_copy_add_sub(void)
00165 {
00166     Mat2D a = mat2D_alloc(2, 3);
00167     Mat2D b = mat2D_alloc(2, 3);
00168     Mat2D c = mat2D_alloc(2, 3);
00169
00170     mat2D_fill(a, 1.5);
00171     mat2D_fill(b, 2.0);
00172
00173     mat2D_copy(c, a);
00174     mat2D_add(c, b); // c = 3.5
00175     for (size_t i = 0; i < c.rows; i++) {
00176         for (size_t j = 0; j < c.cols; j++) {
00177             MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, i, j), 3.5, 0.0));

```



```

00178     }
00179 }
00180
00181     mat2D_sub(c, b); // c back to 1.5
00182     assert_mat_close(c, a, 0.0);
00183
00184     mat2D_free(a);
00185     mat2D_free(b);
00186     mat2D_free(c);
00187 }
00188
00189 static void test_transpose(void)
00190 {
00191     Mat2D a = mat2D_alloc(2, 3);
00192     Mat2D t = mat2D_alloc(3, 2);
00193
00194     // a =
00195     // [1 2 3
00196     //  4 5 6]
00197     double v = 1.0;
00198     for (size_t i = 0; i < a.rows; i++) {
00199         for (size_t j = 0; j < a.cols; j++) {
00200             MAT2D_AT(a, i, j) = v++;
00201         }
00202     }
00203
00204     mat2D_transpose(t, a);
00205
00206     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 0, 0), 1.0, 0.0));
00207     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 1, 0), 2.0, 0.0));
00208     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 2, 0), 3.0, 0.0));
00209     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 0, 1), 4.0, 0.0));
00210     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 1, 1), 5.0, 0.0));
00211     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 2, 1), 6.0, 0.0));
00212
00213     mat2D_free(a);
00214     mat2D_free(t);
00215 }
00216
00217 static void test_dot_product_matrix_multiply(void)
00218 {
00219     Mat2D a = mat2D_alloc(2, 3);
00220     Mat2D b = mat2D_alloc(3, 2);
00221     Mat2D c = mat2D_alloc(2, 2);
00222
00223     // a =
00224     // [1 2 3
00225     //  4 5 6]
00226     MAT2D_AT(a, 0, 0) = 1;
00227     MAT2D_AT(a, 0, 1) = 2;
00228     MAT2D_AT(a, 0, 2) = 3;
00229     MAT2D_AT(a, 1, 0) = 4;
00230     MAT2D_AT(a, 1, 1) = 5;
00231     MAT2D_AT(a, 1, 2) = 6;
00232
00233     // b =
00234     // [ 7  8
00235     //   9 10
00236     //  11 12]
00237     MAT2D_AT(b, 0, 0) = 7;
00238     MAT2D_AT(b, 0, 1) = 8;
00239     MAT2D_AT(b, 1, 0) = 9;
00240     MAT2D_AT(b, 1, 1) = 10;
00241     MAT2D_AT(b, 2, 0) = 11;
00242     MAT2D_AT(b, 2, 1) = 12;
00243
00244     mat2D_dot(c, a, b);
00245
00246     // Expected:
00247     // [ 58  64
00248     //  139 154]
00249     MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 0, 0), 58.0, 0.0));
00250     MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 0, 1), 64.0, 0.0));
00251     MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 1, 0), 139.0, 0.0));
00252     MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 1, 1), 154.0, 0.0));
00253
00254     mat2D_free(a);
00255     mat2D_free(b);
00256     mat2D_free(c);
00257 }
00258
00259 static void test_det_and_minor_det_agree_3x3(void)
00260 {
00261     Mat2D a = mat2D_alloc(3, 3);
00262
00263     // Classic example det = -306
00264     // [ 6  1  1

```

```

00265     //      4  -2  5
00266     //      2   8  7 ]
00267     MAT2D_AT(a, 0, 0) = 6;
00268     MAT2D_AT(a, 0, 1) = 1;
00269     MAT2D_AT(a, 0, 2) = 1;
00270     MAT2D_AT(a, 1, 0) = 4;
00271     MAT2D_AT(a, 1, 1) = -2;
00272     MAT2D_AT(a, 1, 2) = 5;
00273     MAT2D_AT(a, 2, 0) = 2;
00274     MAT2D_AT(a, 2, 1) = 8;
00275     MAT2D_AT(a, 2, 2) = 7;
00276
00277     double det_gauss = mat2D_det(a);
00278     double det_minor = det_by_minors_first_col(a);
00279
00280     MAT2D_ASSERT(nearly_equal(det_gauss, -306.0, MAT2D_EPS));
00281     MAT2D_ASSERT(nearly_equal(det_minor, -306.0, MAT2D_EPS));
00282     MAT2D_ASSERT(nearly_equal(det_minor, det_gauss, MAT2D_EPS));
00283
00284     mat2D_free(a);
00285 }
00286
00287 static void test_invert(void)
00288 {
00289     /*
00290     * Adversarial inversion set:
00291     * - a "normal" matrix
00292     * - a matrix with a zero leading pivot (forces pivoting / row swaps)
00293     * - an ill-conditioned-ish matrix (still invertible) to stress Gauss-Jordan
00294     */
00295     struct {
00296         const char *name;
00297         double eps;
00298         double data[9];
00299     } cases[] = {
00300     {
00301         "baseline det=3",
00302         MAT2D_EPS,
00303         {4, 7, 2, 3, 6, 1, 2, 5, 1},
00304     },
00305     {
00306         "forces row-swap pivoting (zero leading pivot), det=2",
00307         MAT2D_EPS,
00308         {0, 1, 1, 1, 0, 1, 1, 1, 0},
00309     },
00310     {
00311         "hilbert-3x3 (ill-conditioned-ish)",
00312         1e-10,
00313         {1.0, 1.0 / 2.0, 1.0 / 3.0, 1.0 / 2.0, 1.0 / 3.0, 1.0 / 4.0, 1.0 / 3.0, 1.0 / 4.0, 1.0 /
00314         5.0},
00315     },
00316     };
00317     for (size_t k = 0; k < sizeof(cases) / sizeof(cases[0]); k++) {
00318         Mat2D a = mat2D_alloc(3, 3);
00319         fill_mat_from_array(a, cases[k].data);
00320
00321         /* basic sanity: det should be finite and not ~0 for these cases */
00322         double d = mat2D_det(a);
00323         MAT2D_ASSERT(isfinite(d));
00324         MAT2D_ASSERT(fabs(d) > MAT2D_EPS);
00325
00326         assert_inverse_identity_both_sides(a, cases[k].eps);
00327         mat2D_free(a);
00328     }
00329 }
00330
00331 static void test_LUP_decomposition_identity_P_no_swap_case(void)
00332 {
00333     const double eps = MAT2D_EPS;
00334
00335     Mat2D a = mat2D_alloc(3, 3);
00336     Mat2D l = mat2D_alloc(3, 3);
00337     Mat2D p = mat2D_alloc(3, 3);
00338     Mat2D u = mat2D_alloc(3, 3);
00339
00340     Mat2D pa = mat2D_alloc(3, 3);
00341     Mat2D lu = mat2D_alloc(3, 3);
00342
00343     // Same matrix as invert test (should not need swaps in typical flow)
00344     MAT2D_AT(a, 0, 0) = 4;
00345     MAT2D_AT(a, 0, 1) = 7;
00346     MAT2D_AT(a, 0, 2) = 2;
00347     MAT2D_AT(a, 1, 0) = 3;
00348     MAT2D_AT(a, 1, 1) = 6;
00349     MAT2D_AT(a, 1, 2) = 1;
00350     MAT2D_AT(a, 2, 0) = 2;

```

```

00351     MAT2D_AT(a, 2, 1) = 5;
00352     MAT2D_AT(a, 2, 2) = 1;
00353
00354     mat2D_LUP_decomposition_with_swap(a, l, p, u);
00355
00356     mat2D_dot(pa, p, a);
00357     mat2D_dot(lu, l, u);
00358
00359     assert_mat_close(pa, lu, eps);
00360
00361     mat2D_free(a);
00362     mat2D_free(l);
00363     mat2D_free(p);
00364     mat2D_free(u);
00365     mat2D_free(pa);
00366     mat2D_free(lu);
00367 }
00368
00369 static void test_LUP_decomposition_swap_required_case(void)
00370 {
00371     const double eps = MAT2D_EPS;
00372
00373     Mat2D a = mat2D_alloc(3, 3);
00374     Mat2D l = mat2D_alloc(3, 3);
00375     Mat2D p = mat2D_alloc(3, 3);
00376     Mat2D u = mat2D_alloc(3, 3);
00377
00378     Mat2D pa = mat2D_alloc(3, 3);
00379     Mat2D lu = mat2D_alloc(3, 3);
00380
00381     /*
00382      * This matrix is invertible but has a guaranteed zero pivot at (0,0),
00383      * so LUP must swap rows to proceed.
00384      * A =
00385      * [0 1 1
00386      *  1 0 1
00387      *  1 1 0]
00388      */
00389     double data[9] = {0, 1, 1, 1, 0, 1, 1, 1, 0};
00390     fill_mat_from_array(a, data);
00391
00392     mat2D_LUP_decomposition_with_swap(a, l, p, u);
00393     assert_permutation_matrix(p);
00394
00395     mat2D_dot(pa, p, a);
00396     mat2D_dot(lu, l, u);
00397     assert_mat_close(pa, lu, eps);
00398
00399     mat2D_free(a);
00400     mat2D_free(l);
00401     mat2D_free(p);
00402     mat2D_free(u);
00403     mat2D_free(pa);
00404     mat2D_free(lu);
00405 }
00406
00407 static void test_uint32_alloc_fill_and_at(void)
00408 {
00409     Mat2D_uint32 m = mat2D_alloc_uint32(2, 3);
00410     mat2D_fill_uint32(m, 42);
00411     MAT2D_AT_UINT32(m, 0, 1) = 7;
00412
00413     MAT2D_ASSERT(MAT2D_AT_UINT32(m, 0, 0) == 42);
00414     MAT2D_ASSERT(MAT2D_AT_UINT32(m, 0, 1) == 7);
00415     MAT2D_ASSERT(MAT2D_AT_UINT32(m, 1, 2) == 42);
00416
00417     mat2D_free_uint32(m);
00418 }
00419
00420 static void test_offset2d_and_stride(void)
00421 {
00422     Mat2D a = mat2D_alloc(3, 4);
00423     mat2D_fill_sequence(a, 0.0, 1.0); // a[i,j] = offset(i,j)
00424
00425     MAT2D_ASSERT(mat2D_offset2d(a, 0, 0) == 0);
00426     MAT2D_ASSERT(mat2D_offset2d(a, 0, 3) == 3);
00427     MAT2D_ASSERT(mat2D_offset2d(a, 1, 0) == 4);
00428     MAT2D_ASSERT(mat2D_offset2d(a, 2, 3) == 11);
00429
00430     MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 3), 11.0, 0.0));
00431
00432     mat2D_free(a);
00433 }
00434
00435 static void test_copy_windows(void)
00436 {
00437     Mat2D des = mat2D_alloc(4, 5);

```

```

00438     Mat2D src = mat2D_alloc(2, 2);
00439     Mat2D got = mat2D_alloc(2, 2);
00440
00441     mat2D_fill(des, -1.0);
00442     MAT2D_AT(src, 0, 0) = 1.0;
00443     MAT2D_AT(src, 0, 1) = 2.0;
00444     MAT2D_AT(src, 1, 0) = 3.0;
00445     MAT2D_AT(src, 1, 1) = 4.0;
00446
00447     // Copy src into des window rows [1..2], cols [2..3]
00448     mat2D_copy_src_to_des_window(des, src, 1, 2, 2, 3);
00449
00450     // Check placement and untouched area
00451     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 0, 0), -1.0, 0.0));
00452     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 2), 1.0, 0.0));
00453     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 3), 2.0, 0.0));
00454     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 2, 2), 3.0, 0.0));
00455     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 2, 3), 4.0, 0.0));
00456     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 3, 4), -1.0, 0.0));
00457
00458     // Copy the same window back out into got and compare to src
00459     mat2D_copy_src_window_to_des(got, des, 1, 2, 2, 3);
00460     assert_mat_close(got, src, 0.0);
00461
00462     mat2D_free(des);
00463     mat2D_free(src);
00464     mat2D_free(got);
00465 }
00466
00467 static void test_non_contiguous_stride_views(void)
00468 {
00469     /*
00470      * Many bugs only show up when stride_r != cols.
00471      * We build a padded (non-contiguous) view and make sure core ops use stride correctly.
00472      */
00473     const size_t rows = 2;
00474     const size_t cols = 2;
00475     const size_t stride = 7; /* intentional padding */
00476
00477     double *buf = (double *)MAT2D_MALLOC(sizeof(double) * rows * stride);
00478     MAT2D_ASSERT(buf != NULL);
00479
00480     /* poison padding so accidental reads are obvious */
00481     for (size_t i = 0; i < rows * stride; i++) {
00482         buf[i] = 1234567.0;
00483     }
00484
00485     Mat2D a = {.rows = rows, .cols = cols, .stride_r = stride, .elements = buf};
00486
00487     MAT2D_AT(a, 0, 0) = 1.0;
00488     MAT2D_AT(a, 0, 1) = 2.0;
00489     MAT2D_AT(a, 1, 0) = 3.0;
00490     MAT2D_AT(a, 1, 1) = 4.0;
00491
00492     MAT2D_ASSERT(mat2D_offset2d(a, 1, 1) == 1 * stride + 1);
00493
00494     Mat2D c = mat2D_alloc(2, 2);
00495     Mat2D t = mat2D_alloc(2, 2);
00496     Mat2D prod = mat2D_alloc(2, 2);
00497     Mat2D id = mat2D_alloc(2, 2);
00498
00499     mat2D_copy(c, a);
00500     MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 1, 1), 4.0, 0.0));
00501
00502     mat2D_transpose(t, a);
00503     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 0, 1), 3.0, 0.0));
00504     MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 1, 0), 2.0, 0.0));
00505
00506     mat2D_set_identity(id);
00507     mat2D_dot(prod, a, id);
00508     assert_mat_close(prod, c, 0.0);
00509
00510     mat2D_free(c);
00511     mat2D_free(t);
00512     mat2D_free(prod);
00513     mat2D_free(id);
00514     MAT2D_FREE(buf);
00515 }
00516
00517 static void test_row_col_ops_and_scaling(void)
00518 {
00519     Mat2D a = mat2D_alloc(3, 3);
00520     mat2D_fill_sequence(a, 1.0, 1.0);
00521     // a =
00522     // [1 2 3
00523     //  4 5 6
00524     //  7 8 9]

```

```

00525
00526 // add_row_to_row: row0 += row1 => [5 7 9]
00527 mat2D_add_row_to_row(a, 0, a, 1);
00528 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 0, 0), 5.0, 0.0));
00529 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 0, 1), 7.0, 0.0));
00530 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 0, 2), 9.0, 0.0));
00531
00532 // sub_row_to_row: row2 -= row1 => [3 3 3]
00533 mat2D_sub_row_to_row(a, 2, a, 1);
00534 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 0), 3.0, 0.0));
00535 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 1), 3.0, 0.0));
00536 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 2), 3.0, 0.0));
00537
00538 // add_col_to_col: col1 += col2 (operate on current a)
00539 mat2D_add_col_to_col(a, 1, a, 2);
00540 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 1), 5.0 + 6.0, 0.0));
00541
00542 // sub_col_to_col: col1 -= col2 => restore original col1 for that row
00543 mat2D_sub_col_to_col(a, 1, a, 2);
00544 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 1), 5.0, 0.0));
00545
00546 // Row op: row1 += 2 * row2 (row2 currently [3 3 3])
00547 mat2D_add_row_time_factor_to_row(a, 1, 2, 2.0);
00548 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 0), 4.0 + 2.0 * 3.0, 0.0));
00549 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 1), 5.0 + 2.0 * 3.0, 0.0));
00550 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 2), 6.0 + 2.0 * 3.0, 0.0));
00551
00552 // Row scale
00553 mat2D_mult_row(a, 2, -1.0);
00554 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 0), -3.0, 0.0));
00555 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 1), -3.0, 0.0));
00556 MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 2), -3.0, 0.0));
00557
00558 mat2D_free(a);
00559 }
00560
00561 static void test_shift_and_identity(void)
00562 {
00563     Mat2D a = mat2D_alloc(4, 4);
00564     mat2D_set_identity(a);
00565     mat2D_shift(a, 2.0); // diag becomes 3
00566
00567     for (size_t i = 0; i < a.rows; i++) {
00568         for (size_t j = 0; j < a.cols; j++) {
00569             double expected = (i == j) ? 3.0 : 0.0;
00570             MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, i, j), expected, 0.0));
00571         }
00572     }
00573
00574     mat2D_free(a);
00575 }
00576
00577 static void test_norms_and_normalize(void)
00578 {
00579     const double eps = MAT2D_EPS;
00580
00581     Mat2D v = mat2D_alloc(2, 1);
00582     MAT2D_AT(v, 0, 0) = 3.0;
00583     MAT2D_AT(v, 1, 0) = 4.0;
00584
00585     MAT2D_ASSERT(nearly_equal(mat2D_calc_norma(v), 5.0, eps));
00586     MAT2D_ASSERT(nearly_equal(mat2D_calc_norma_inf(v), 4.0, eps));
00587     MAT2D_ASSERT(nearly_equal(mat2D_inner_product(v), 25.0, eps));
00588
00589     mat2D_normalize(v);
00590     MAT2D_ASSERT(nearly_equal(mat2D_calc_norma(v), 1.0, MAT2D_EPS));
00591
00592     mat2D_free(v);
00593 }
00594
00595 static void test_outer_product_and_cross(void)
00596 {
00597     const double eps = MAT2D_EPS;
00598
00599     Mat2D v = mat2D_alloc(3, 1);
00600     Mat2D out = mat2D_alloc(3, 3);
00601
00602     MAT2D_AT(v, 0, 0) = 1.0;
00603     MAT2D_AT(v, 1, 0) = 2.0;
00604     MAT2D_AT(v, 2, 0) = 3.0;
00605
00606     mat2D_outer_product(out, v);
00607     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 0), 1.0, eps));
00608     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 1), 2.0, eps));
00609     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 2), 3.0, eps));
00610     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 0), 2.0, eps));
00611     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 1), 4.0, eps));

```

```

00612     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 2), 6.0, eps));
00613     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 2, 2), 9.0, eps));
00614
00615     // Cross: i x j = k
00616     Mat2D i = mat2D_alloc(3, 1);
00617     Mat2D j = mat2D_alloc(3, 1);
00618     Mat2D k = mat2D_alloc(3, 1);
00619     mat2D_fill(i, 0.0);
00620     mat2D_fill(j, 0.0);
00621     mat2D_fill(k, 0.0);
00622     MAT2D_AT(i, 0, 0) = 1.0;
00623     MAT2D_AT(j, 1, 0) = 1.0;
00624
00625     mat2D_cross(k, i, j);
00626     MAT2D_ASSERT(nearly_equal(MAT2D_AT(k, 0, 0), 0.0, eps));
00627     MAT2D_ASSERT(nearly_equal(MAT2D_AT(k, 1, 0), 0.0, eps));
00628     MAT2D_ASSERT(nearly_equal(MAT2D_AT(k, 2, 0), 1.0, eps));
00629
00630     mat2D_free(v);
00631     mat2D_free(out);
00632     mat2D_free(i);
00633     mat2D_free(j);
00634     mat2D_free(k);
00635 }
00636
00637 static void test_det_2x2_and_upper_triangulate_sign(void)
00638 {
00639     Mat2D a = mat2D_alloc(2, 2);
00640     MAT2D_AT(a, 0, 0) = 0.0;
00641     MAT2D_AT(a, 0, 1) = 1.0;
00642     MAT2D_AT(a, 1, 0) = 2.0;
00643     MAT2D_AT(a, 1, 1) = 3.0;
00644
00645     // det should be -2
00646     MAT2D_ASSERT(nearly_equal(mat2D_det_2x2_mat(a), -2.0, 0.0));
00647     MAT2D_ASSERT(nearly_equal(mat2D_det(a), -2.0, MAT2D_EPS));
00648
00649     // upper triangulation should have exactly one row swap => factor = -1
00650     Mat2D tmp = mat2D_alloc(2, 2);
00651     mat2D_copy(tmp, a);
00652     double f = mat2D_upper_triangulate(tmp, MAT2D_ROW_SWAPPING);
00653     MAT2D_ASSERT(nearly_equal(f, -1.0, 0.0));
00654
00655     mat2D_free(a);
00656     mat2D_free(tmp);
00657 }
00658
00659 static double det_by_minors_first_col(Mat2D a)
00660 {
00661     MAT2D_ASSERT(a.rows == a.cols);
00662     size_t n = a.rows;
00663
00664     if (n == 1) {
00665         return MAT2D_AT(a, 0, 0);
00666     }
00667     if (n == 2) {
00668         return mat2D_det_2x2_mat(a);
00669     }
00670
00671     double det = 0.0;
00672     size_t j = 0;
00673     for (size_t i = 0; i < n; i++) {
00674         double aij = MAT2D_AT(a, i, j);
00675         if (aij == 0.0) {
00676             continue;
00677         }
00678
00679         Mat2D_Minor mm = mat2D_minor_alloc_fill_from_mat(a, i, j);
00680         double minor_det = 0.0;
00681         if (mm.rows == 2 && mm.cols == 2) {
00682             minor_det = mat2D_det_2x2_mat_minor(mm);
00683         } else {
00684             minor_det = mat2D_minor_det(mm);
00685         }
00686         mat2D_minor_free(mm);
00687
00688         double sign = ((i + j) % 2 == 0) ? 1.0 : -1.0;
00689         det += aij * sign * minor_det;
00690     }
00691
00692     return det;
00693 }
00694
00695 static void test_minor_det_matches_gauss_4x4_known(void)
00696 {
00697     const double eps = 1e-9;
00698

```

```

00699     Mat2D a = mat2D_alloc(4, 4);
00700     // det = 72
00701     // [1 2 3 4
00702     // 5 6 7 8
00703     // 2 6 4 8
00704     // 3 1 1 2]
00705     double data[16] = {
00706         1, 2, 3, 4, //
00707         5, 6, 7, 8, //
00708         2, 6, 4, 8, //
00709         3, 1, 1, 2 //
00710     };
00711     for (size_t i = 0; i < 4; i++) {
00712         for (size_t j = 0; j < 4; j++) {
00713             MAT2D_AT(a, i, j) = data[i * 4 + j];
00714         }
00715     }
00716
00717     double det_gauss = mat2D_det(a);
00718     double det_minor = det_by_minors_first_col(a);
00719
00720     MAT2D_ASSERT(nearly_equal(det_gauss, 72.0, eps));
00721     MAT2D_ASSERT(nearly_equal(det_minor, 72.0, eps));
00722     MAT2D_ASSERT(nearly_equal(det_minor, det_gauss, eps));
00723
00724     mat2D_free(a);
00725 }
00726
00727 static void test_reduce_rank(void)
00728 {
00729     Mat2D a = mat2D_alloc(3, 3);
00730     // rank 2:
00731     // [1 2 3
00732     // 2 4 6
00733     // 1 1 1]
00734     MAT2D_AT(a, 0, 0) = 1;
00735     MAT2D_AT(a, 0, 1) = 2;
00736     MAT2D_AT(a, 0, 2) = 3;
00737     MAT2D_AT(a, 1, 0) = 2;
00738     MAT2D_AT(a, 1, 1) = 4;
00739     MAT2D_AT(a, 1, 2) = 6;
00740     MAT2D_AT(a, 2, 0) = 1;
00741     MAT2D_AT(a, 2, 1) = 1;
00742     MAT2D_AT(a, 2, 2) = 1;
00743
00744     size_t r = mat2D_reduce(a);
00745     MAT2D_ASSERT(r == 2);
00746
00747     mat2D_free(a);
00748 }
00749
00750 static void test_rotation_matrices_orthonormal(void)
00751 {
00752     const double eps = 1e-7;
00753
00754     Mat2D r = mat2D_alloc(3, 3);
00755     Mat2D rt = mat2D_alloc(3, 3);
00756     Mat2D prod = mat2D_alloc(3, 3);
00757
00758     // Z rotation 90deg, per library's convention:
00759     // [ 0 1 0
00760     // -1 0 0
00761     //  0 0 1 ]
00762     mat2D_set_rot_mat_z(r, 90.0f);
00763     MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 0, 0), 0.0, eps));
00764     MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 0, 1), 1.0, eps));
00765     MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 1, 0), -1.0, eps));
00766     MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 1, 1), 0.0, eps));
00767     MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 2, 2), 1.0, eps));
00768
00769     mat2D_transpose(rt, r);
00770     mat2D_dot(prod, rt, r);
00771     assert_identity_close(prod, MAT2D_EPS);
00772     MAT2D_ASSERT(nearly_equal(mat2D_det(r), 1.0, MAT2D_EPS));
00773
00774     // X rotation 90deg should also be orthonormal with det ~ 1
00775     mat2D_set_rot_mat_x(r, 90.0f);
00776     mat2D_transpose(rt, r);
00777     mat2D_dot(prod, rt, r);
00778     assert_identity_close(prod, MAT2D_EPS);
00779     MAT2D_ASSERT(nearly_equal(mat2D_det(r), 1.0, MAT2D_EPS));
00780
00781     // Y rotation 90deg orthonormal with det ~ 1
00782     mat2D_set_rot_mat_y(r, 90.0f);
00783     mat2D_transpose(rt, r);
00784     mat2D_dot(prod, rt, r);
00785     assert_identity_close(prod, MAT2D_EPS);

```

```

00786     MAT2D_ASSERT(nearly_equal(mat2D_det(r), 1.0, MAT2D_EPS));
00787
00788     mat2D_free(r);
00789     mat2D_free(rt);
00790     mat2D_free(prod);
00791 }
00792
00793 static void test_DCM_zyx_matches_product(void)
00794 {
00795     const double eps = MAT2D_EPS;
00796
00797     Mat2D dcm = mat2D_alloc(3, 3);
00798     Mat2D rz = mat2D_alloc(3, 3);
00799     Mat2D ry = mat2D_alloc(3, 3);
00800     Mat2D rx = mat2D_alloc(3, 3);
00801     Mat2D tmp = mat2D_alloc(3, 3);
00802     Mat2D expected = mat2D_alloc(3, 3);
00803
00804     float yaw = 30.0f;
00805     float pitch = 20.0f;
00806     float roll = 10.0f;
00807
00808     mat2D_set_DCM_zyx(dcm, yaw, pitch, roll);
00809
00810     mat2D_set_rot_mat_z(rz, yaw);
00811     mat2D_set_rot_mat_y(ry, pitch);
00812     mat2D_set_rot_mat_x(rx, roll);
00813     mat2D_dot(tmp, ry, rz);
00814     mat2D_dot(expected, rx, tmp);
00815
00816     assert_mat_close(dcm, expected, eps);
00817
00818     mat2D_free(dcm);
00819     mat2D_free(rz);
00820     mat2D_free(ry);
00821     mat2D_free(rx);
00822     mat2D_free(tmp);
00823     mat2D_free(expected);
00824 }
00825
00826 static void test_solve_linear_system_LUP(void)
00827 {
00828     const double eps = MAT2D_EPS;
00829
00830     Mat2D a = mat2D_alloc(3, 3);
00831     Mat2D x = mat2D_alloc(3, 1);
00832     Mat2D b = mat2D_alloc(3, 1);
00833     Mat2D ax = mat2D_alloc(3, 1);
00834
00835     // A =
00836     // [3 0 2
00837     //  2 0 -2
00838     //  0 1 1]
00839     MAT2D_AT(a, 0, 0) = 3;
00840     MAT2D_AT(a, 0, 1) = 0;
00841     MAT2D_AT(a, 0, 2) = 2;
00842     MAT2D_AT(a, 1, 0) = 2;
00843     MAT2D_AT(a, 1, 1) = 0;
00844     MAT2D_AT(a, 1, 2) = -2;
00845     MAT2D_AT(a, 2, 0) = 0;
00846     MAT2D_AT(a, 2, 1) = 1;
00847     MAT2D_AT(a, 2, 2) = 1;
00848
00849     // Choose x_true = [1,2,3]^T, so b = A*x_true = [9, -4, 5]^T
00850     MAT2D_AT(b, 0, 0) = 9;
00851     MAT2D_AT(b, 1, 0) = -4;
00852     MAT2D_AT(b, 2, 0) = 5;
00853
00854     mat2D_solve_linear_sys_LUP_decomposition(a, x, b);
00855     mat2D_dot(ax, a, x);
00856
00857     MAT2D_ASSERT(nearly_equal(MAT2D_AT(ax, 0, 0), 9.0, eps));
00858     MAT2D_ASSERT(nearly_equal(MAT2D_AT(ax, 1, 0), -4.0, eps));
00859     MAT2D_ASSERT(nearly_equal(MAT2D_AT(ax, 2, 0), 5.0, eps));
00860
00861     mat2D_free(a);
00862     mat2D_free(x);
00863     mat2D_free(b);
00864     mat2D_free(ax);
00865 }
00866
00867 static void test_rand_range(void)
00868 {
00869     Mat2D a = mat2D_alloc(4, 4);
00870     srand(1);
00871     mat2D_rand(a, -2.0, 5.0);
00872

```



```

00873     for (size_t i = 0; i < a.rows; i++) {
00874         for (size_t j = 0; j < a.cols; j++) {
00875             double v = MAT2D_AT(a, i, j);
00876             MAT2D_ASSERT(v >= -2.0);
00877             MAT2D_ASSERT(v <= 5.0);
00878         }
00879     }
00880
00881     mat2D_free(a);
00882 }
00883
00884 static void test_copy_row_and_col_helpers(void)
00885 {
00886     Mat2D src = mat2D_alloc(3, 3);
00887     Mat2D des = mat2D_alloc(3, 3);
00888
00889     /*
00890      * src =
00891      * [1 2 3
00892      *  4 5 6
00893      *  7 8 9]
00894      */
00895     mat2D_fill_sequence(src, 1.0, 1.0);
00896     mat2D_fill(des, 0.0);
00897
00898     /* copy row 1 => des[1,:] = src[1,:] */
00899     mat2D_copy_row_from_src_to_des(des, 1, src, 1);
00900     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 0), 4.0, 0.0));
00901     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 1), 5.0, 0.0));
00902     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 2), 6.0, 0.0));
00903
00904     /* copy col 2 => des[:,2] = src[:,2] */
00905     mat2D_copy_col_from_src_to_des(des, 2, src, 2);
00906     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 0, 2), 3.0, 0.0));
00907     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 2), 6.0, 0.0));
00908     MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 2, 2), 9.0, 0.0));
00909
00910     mat2D_free(src);
00911     mat2D_free(des);
00912 }
00913
00914 static void test_dot_product_and_vector_variants(void)
00915 {
00916     const double eps = 1e-12;
00917
00918     /* Column vectors */
00919     Mat2D a = mat2D_alloc(3, 1);
00920     Mat2D b = mat2D_alloc(3, 1);
00921     MAT2D_AT(a, 0, 0) = 1.0;
00922     MAT2D_AT(a, 1, 0) = 2.0;
00923     MAT2D_AT(a, 2, 0) = 3.0;
00924     MAT2D_AT(b, 0, 0) = 4.0;
00925     MAT2D_AT(b, 1, 0) = 5.0;
00926     MAT2D_AT(b, 2, 0) = 6.0;
00927
00928     MAT2D_ASSERT(nearly_equal(mat2D_dot_product(a, b), 32.0, eps));
00929     MAT2D_ASSERT(nearly_equal(mat2D_inner_product(a), 14.0, eps));
00930
00931     /* Row vectors (exercise the other branch in dot_product/inner_product) */
00932     Mat2D ar = mat2D_alloc(1, 3);
00933     Mat2D br = mat2D_alloc(1, 3);
00934     MAT2D_AT(ar, 0, 0) = 1.0;
00935     MAT2D_AT(ar, 0, 1) = 2.0;
00936     MAT2D_AT(ar, 0, 2) = 3.0;
00937     MAT2D_AT(br, 0, 0) = 4.0;
00938     MAT2D_AT(br, 0, 1) = 5.0;
00939     MAT2D_AT(br, 0, 2) = 6.0;
00940
00941     MAT2D_ASSERT(nearly_equal(mat2D_dot_product(ar, br), 32.0, eps));
00942     MAT2D_ASSERT(nearly_equal(mat2D_inner_product(ar), 14.0, eps));
00943
00944     /* Normalize-by-inf macro (smoke) */
00945     MAT2D_ASSERT(nearly_equal(mat2D_calc_norma_inf(ar), 3.0, eps));
00946     mat2D_normalize_inf(ar);
00947     MAT2D_ASSERT(nearly_equal(mat2D_calc_norma_inf(ar), 1.0, 1e-12));
00948
00949     mat2D_free(a);
00950     mat2D_free(b);
00951     mat2D_free(ar);
00952     mat2D_free(br);
00953 }
00954
00955 static void test_outer_product_row_vector_path(void)
00956 {
00957     const double eps = 1e-12;
00958
00959     Mat2D v = mat2D_alloc(1, 3); /* row-vector form */

```

```

00960     Mat2D out = mat2D_alloc(3, 3);
00961
00962     MAT2D_AT(v, 0, 0) = 1.0;
00963     MAT2D_AT(v, 0, 1) = 2.0;
00964     MAT2D_AT(v, 0, 2) = 3.0;
00965
00966     mat2D_outer_product(out, v);
00967
00968     /* out[i,j] = v[i]*v[j] (with row-vector indexing) */
00969     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 0), 1.0, eps));
00970     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 1), 2.0, eps));
00971     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 2), 3.0, eps));
00972     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 0), 2.0, eps));
00973     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 1), 4.0, eps));
00974     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 2), 6.0, eps));
00975     MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 2, 2), 9.0, eps));
00976
00977     mat2D_free(v);
00978     mat2D_free(out);
00979 }
00980
00981 static void test_det_early_zero_row_and_zero_col_paths(void)
00982 {
00983     /* Exercise mat2D_det() early return when it finds an all-zero row/col */
00984
00985     Mat2D zr = mat2D_alloc(2, 2);
00986     /* row0 all zeros => det 0 */
00987     MAT2D_AT(zr, 0, 0) = 0.0;
00988     MAT2D_AT(zr, 0, 1) = 0.0;
00989     MAT2D_AT(zr, 1, 0) = 1.0;
00990     MAT2D_AT(zr, 1, 1) = 2.0;
00991     MAT2D_ASSERT(mat2D_row_is_all_digit(zr, 0.0, 0));
00992     MAT2D_ASSERT(nearly_equal(mat2D_det(zr), 0.0, 0.0));
00993
00994     Mat2D zc = mat2D_alloc(2, 2);
00995     /* col0 all zeros but no all-zero row => det 0 */
00996     MAT2D_AT(zc, 0, 0) = 0.0;
00997     MAT2D_AT(zc, 0, 1) = 1.0;
00998     MAT2D_AT(zc, 1, 0) = 0.0;
00999     MAT2D_AT(zc, 1, 1) = 2.0;
01000     MAT2D_ASSERT(mat2D_col_is_all_digit(zc, 0.0, 0));
01001     MAT2D_ASSERT(nearly_equal(mat2D_det(zc), 0.0, 0.0));
01002
01003     mat2D_free(zr);
01004     mat2D_free(zc);
01005 }
01006
01007 static void test_mat_is_all_digit(void)
01008 {
01009     Mat2D m = mat2D_alloc(2, 3);
01010     mat2D_fill(m, 7.0);
01011     MAT2D_ASSERT(mat2D_mat_is_all_digit(m, 7.0));
01012     MAT2D_ASSERT(!mat2D_mat_is_all_digit(m, 8.0));
01013     MAT2D_AT(m, 1, 2) = 8.0;
01014     MAT2D_ASSERT(!mat2D_mat_is_all_digit(m, 7.0));
01015     mat2D_free(m);
01016 }
01017
01018 static void test_power_iterate_and_eig_helpers(void)
01019 {
01020     /*
01021      * Use a diagonal matrix with distinct positive eigenvalues:
01022      * A = diag(5,3,1)
01023      * Power iteration should find lambda=5 with eigenvector ~ e1.
01024      * eig_power_iteration should recover (5,3,1) and eig_check residual small.
01025      */
01026     const double eps = 1e-7;
01027
01028     Mat2D A = mat2D_alloc(3, 3);
01029     mat2D_fill(A, 0.0);
01030     MAT2D_AT(A, 0, 0) = 5.0;
01031     MAT2D_AT(A, 1, 1) = 3.0;
01032     MAT2D_AT(A, 2, 2) = 1.0;
01033
01034     /* mat2D_power_iterate */
01035     Mat2D v = mat2D_alloc(3, 1);
01036     MAT2D_AT(v, 0, 0) = 1.0;
01037     MAT2D_AT(v, 1, 0) = 1.0;
01038     MAT2D_AT(v, 2, 0) = 1.0;
01039     double lambda = 0.0;
01040     int rc = mat2D_power_iterate(A, v, &lambda, 0.0, true);
01041     MAT2D_ASSERT(rc == 0);
01042     MAT2D_ASSERT(close_rel_abs(lambda, 5.0, 1e-6, 1e-6));
01043     /* dominant component should be index 0 */
01044     MAT2D_ASSERT(fabs(MAT2D_AT(v, 0, 0)) > fabs(MAT2D_AT(v, 1, 0)));
01045     MAT2D_ASSERT(fabs(MAT2D_AT(v, 0, 0)) > fabs(MAT2D_AT(v, 2, 0)));
01046

```

```

01047     /* mat2D_eig_power_iteration + mat2D_eig_check */
01048     Mat2D evals = mat2D_alloc(3, 3);
01049     Mat2D evects = mat2D_alloc(3, 3);
01050     Mat2D res = mat2D_alloc(3, 3);
01051
01052     /* init vector must be non-zero */
01053     Mat2D init = mat2D_alloc(3, 1);
01054     MAT2D_AT(init, 0, 0) = 1.0;
01055     MAT2D_AT(init, 1, 0) = 1.0;
01056     MAT2D_AT(init, 2, 0) = 1.0;
01057
01058     mat2D_eig_power_iteration(A, evals, evects, init, true);
01059     MAT2D_ASSERT(close_rel_abs(MAT2D_AT(evals, 0, 0), 5.0, 1e-5, 1e-5));
01060     MAT2D_ASSERT(close_rel_abs(MAT2D_AT(evals, 1, 1), 3.0, 1e-5, 1e-5));
01061     MAT2D_ASSERT(close_rel_abs(MAT2D_AT(evals, 2, 2), 1.0, 1e-5, 1e-5));
01062
01063     mat2D_eig_check(A, evals, evects, res);
01064     MAT2D_ASSERT(mat2D_calc_norma_inf(res) < eps);
01065
01066     mat2D_free(A);
01067     mat2D_free(v);
01068     mat2D_free(evals);
01069     mat2D_free(evects);
01070     mat2D_free(res);
01071     mat2D_free(init);
01072 }
01073
01074 static void test_deterministic_fuzz_loop(void)
01075 {
01076     /*
01077      * Deterministic fuzz:
01078      * - generate many random (but well-conditioned) square matrices
01079      * - reject near-singular ones (fabs(det) too small / non-finite)
01080      * - assert invariants:
01081      *     inv(A) exists and A*inv(A)~I and inv(A)*A~I
01082      *     det(A^T)~det(A)
01083      *     det(A)*det(inv(A))~1
01084      *     rank(A)==N (via mat2D_reduce)
01085      */
01086
01087     const size_t iters = 500;
01088     const double inv_eps = 1e-7;
01089     const double det_min = 1e-8;
01090     const double det_abs_eps = 1e-6;
01091     const double det_rel_eps = 1e-6;
01092
01093     uint64_t rng = 0x9e3779b97f4a7c15ULL; /* fixed seed */
01094     size_t tested = 0;
01095     size_t skipped = 0;
01096
01097     for (size_t t = 0; t < iters; t++) {
01098         size_t n = 2 + (size_t)(xorshift64star(&rng) % 49); /* 2..50 */
01099
01100         Mat2D a = mat2D_alloc(n, n);
01101         Mat2D inv = mat2D_alloc(n, n);
01102         Mat2D prod1 = mat2D_alloc(n, n);
01103         Mat2D prod2 = mat2D_alloc(n, n);
01104         Mat2D at = mat2D_alloc(n, n);
01105         Mat2D tmp = mat2D_alloc(n, n);
01106
01107         fill_strictly_diag_dominant(a, &rng);
01108
01109         double det_a = mat2D_det(a);
01110         if (!isfinite(det_a) || fabs(det_a) < det_min) {
01111             skipped++;
01112             mat2D_free(a);
01113             mat2D_free(inv);
01114             mat2D_free(prod1);
01115             mat2D_free(prod2);
01116             mat2D_free(at);
01117             mat2D_free(tmp);
01118             continue;
01119         }
01120
01121         /* Inversion invariants */
01122         mat2D_invert(inv, a);
01123         mat2D_dot(prod1, a, inv);
01124         assert_identity_close(prod1, inv_eps);
01125         mat2D_dot(prod2, inv, a);
01126         assert_identity_close(prod2, inv_eps);
01127
01128         /* det(A^T) == det(A) */
01129         mat2D_transpose(at, a);
01130         double det_at = mat2D_det(at);
01131         MAT2D_ASSERT(
01132             close_rel_abs(det_at, det_a, det_abs_eps, det_rel_eps));
01133     }

```

```

01134     /* det(A) * det(inv(A)) == 1 */
01135     double det_inv = mat2D_det(inv);
01136     MAT2D_ASSERT(isfinite(det_inv));
01137     MAT2D_ASSERT(close_rel_abs(
01138         det_a * det_inv,
01139         1.0,
01140         1e-5, /* a bit looser: det() is numerically touchy */
01141         1e-5));
01142
01143     /* rank(A) == N */
01144     mat2D_copy(tmp, a);
01145     size_t r = mat2D_reduce(tmp);
01146     MAT2D_ASSERT(r == n);
01147
01148     tested++;
01149
01150     mat2D_free(a);
01151     mat2D_free(inv);
01152     mat2D_free(prod1);
01153     mat2D_free(prod2);
01154     mat2D_free(at);
01155     mat2D_free(tmp);
01156 }
01157
01158 /* Make sure the loop actually exercised enough cases. */
01159 MAT2D_ASSERT(tested >= iters / 2);
01160 printf("[INFO] fuzz summary: tested=%zu skipped=%zu\n", tested, skipped);
01161 }
01162
01163 int main(void)
01164 {
01165     #define RUN_TEST(fn) \
01166         do { \
01167             fn(); \
01168             printf("[INFO] passed | %s\n", \
01169                 #fn); \
01170             fflush(stdout); \
01171         } while (0)
01172
01173     RUN_TEST(test_uint32_alloc_fill_and_at);
01174     RUN_TEST(test_offset2d_and_stride);
01175     RUN_TEST(test_alloc_fill_copy_add_sub);
01176     RUN_TEST(test_transpose);
01177     RUN_TEST(test_copy_windows);
01178     RUN_TEST(test_dot_product_matrix_multiply);
01179     RUN_TEST(test_det_and_minor_det_agree_3x3);
01180     RUN_TEST(test_det_2x2_and_upper_triangular_sign);
01181     RUN_TEST(test_minor_det_matches_gauss_4x4_known);
01182     RUN_TEST(test_invert);
01183     RUN_TEST(test_LUP_decomposition_identity_P_no_swap_case);
01184     RUN_TEST(test_LUP_decomposition_swap_required_case);
01185     RUN_TEST(test_reduce_rank);
01186     RUN_TEST(test_norms_and_normalize);
01187     RUN_TEST(test_outer_product_and_cross);
01188     RUN_TEST(test_shift_and_identity);
01189     RUN_TEST(test_rotation_matrices_orthonormal);
01190     RUN_TEST(test_DCM_zyx_matches_product);
01191     RUN_TEST(test_solve_linear_system_LUP);
01192     RUN_TEST(test_non_contiguous_stride_views);
01193     RUN_TEST(test_row_col_ops_and_scaling);
01194     RUN_TEST(test_rand_range);
01195     RUN_TEST(test_copy_row_and_col_helpers);
01196     RUN_TEST(test_deterministic_fuzz_loop);
01197     RUN_TEST(test_dot_product_and_vector_variants);
01198     RUN_TEST(test_outer_product_row_vector_path);
01199     RUN_TEST(test_det_early_zero_row_and_zero_col_paths);
01200     RUN_TEST(test_mat_is_all_digit);
01201     RUN_TEST(test_power_iterate_and_eig_helpers);
01202
01203     #undef RUN_TEST
01204
01205     printf("\n [INFO] Matrix2D tests passed\n");
01206     return 0;
01207 }

```

Index

- assert_identity_close
 - test_matrix2d.c, [84](#)
- assert_inverse_identity_both_sides
 - test_matrix2d.c, [84](#)
- assert_mat_close
 - test_matrix2d.c, [84](#)
- assert_permutation_matrix
 - test_matrix2d.c, [84](#)
- close_rel_abs
 - test_matrix2d.c, [85](#)
- cols
 - Mat2D, [5](#)
 - Mat2D_Minor, [8](#)
 - Mat2D_uint32, [10](#)
- cols_list
 - Mat2D_Minor, [8](#)
- det_by_minors_first_col
 - test_matrix2d.c, [85](#)
- elements
 - Mat2D, [6](#)
 - Mat2D_uint32, [10](#)
- example1.c
 - main, [12](#)
 - MATRIX2D_IMPLEMENTATION, [11](#)
- examples/example1.c, [11](#), [12](#)
- fill_mat_from_array
 - test_matrix2d.c, [85](#)
- fill_strictly_diag_dominant
 - test_matrix2d.c, [85](#)
- main
 - example1.c, [12](#)
 - temp.c, [81](#)
 - test_matrix2d.c, [86](#)
- Mat2D, [5](#)
 - cols, [5](#)
 - elements, [6](#)
 - rows, [6](#)
 - stride_r, [6](#)
- mat2D_add
 - Matrix2D.h, [23](#)
- mat2D_add_col_to_col
 - Matrix2D.h, [23](#)
- mat2D_add_row_time_factor_to_row
 - Matrix2D.h, [24](#)
- mat2D_add_row_to_row
 - Matrix2D.h, [24](#)
- mat2D_alloc
 - Matrix2D.h, [25](#)
- mat2D_alloc_uint32
 - Matrix2D.h, [26](#)
- MAT2D_ASSERT
 - Matrix2D.h, [18](#)
- MAT2D_AT
 - Matrix2D.h, [18](#)
- MAT2D_AT_UINT32
 - Matrix2D.h, [19](#)
- mat2D_calc_col_norma
 - Matrix2D.h, [26](#)
- mat2D_calc_norma
 - Matrix2D.h, [27](#)
- mat2D_calc_norma_inf
 - Matrix2D.h, [27](#)
- mat2D_col_is_all_digit
 - Matrix2D.h, [28](#)
- mat2D_copy
 - Matrix2D.h, [28](#)
- mat2D_copy_col_from_src_to_des
 - Matrix2D.h, [29](#)
- mat2D_copy_row_from_src_to_des
 - Matrix2D.h, [30](#)
- mat2D_copy_src_to_des_window
 - Matrix2D.h, [30](#)
- mat2D_copy_src_window_to_des
 - Matrix2D.h, [31](#)
- mat2D_create_col_ref
 - Matrix2D.h, [31](#)
- mat2D_cross
 - Matrix2D.h, [32](#)
- mat2D_det
 - Matrix2D.h, [33](#)
- mat2D_det_2x2_mat
 - Matrix2D.h, [33](#)
- mat2D_det_2x2_mat_minor
 - Matrix2D.h, [34](#)
- mat2D_dot
 - Matrix2D.h, [34](#)
- mat2D_dot_product
 - Matrix2D.h, [35](#)
- mat2D_dprintDOUBLE
 - Matrix2D.h, [19](#)
- mat2D_dprintINT
 - Matrix2D.h, [19](#)
- mat2D_dprintSIZE_T
 - Matrix2D.h, [19](#)
- mat2D_eig_check

- Matrix2D.h, [36](#)
- mat2D_eig_power_iteration
 - Matrix2D.h, [36](#)
- MAT2D_EPS
 - Matrix2D.h, [20](#)
- mat2D_fill
 - Matrix2D.h, [37](#)
- mat2D_fill_sequence
 - Matrix2D.h, [38](#)
- mat2D_fill_uint32
 - Matrix2D.h, [38](#)
- mat2D_find_first_non_zero_value
 - Matrix2D.h, [39](#)
- MAT2D_FREE
 - Matrix2D.h, [20](#)
- mat2D_free
 - Matrix2D.h, [39](#)
- mat2D_free_uint32
 - Matrix2D.h, [40](#)
- mat2D_inner_product
 - Matrix2D.h, [40](#)
- mat2D_invert
 - Matrix2D.h, [41](#)
- MAT2D_IS_ZERO
 - Matrix2D.h, [20](#)
- mat2D_LUP_decomposition_with_swap
 - Matrix2D.h, [42](#)
- mat2D_make_orthogonal_Gaussian_elimination
 - Matrix2D.h, [42](#)
- mat2D_make_orthogonal_modified_Gram_Schmidt
 - Matrix2D.h, [43](#)
- MAT2D_MALLOC
 - Matrix2D.h, [20](#)
- mat2D_mat_is_all_digit
 - Matrix2D.h, [44](#)
- MAT2D_MAX_POWER_ITERATION
 - Matrix2D.h, [20](#)
- Mat2D_Minor, [7](#)
 - cols, [8](#)
 - cols_list, [8](#)
 - ref_mat, [8](#)
 - rows, [8](#)
 - rows_list, [9](#)
 - stride_r, [9](#)
- mat2D_minor_alloc_fill_from_mat
 - Matrix2D.h, [44](#)
- mat2D_minor_alloc_fill_from_mat_minor
 - Matrix2D.h, [45](#)
- MAT2D_MINOR_AT
 - Matrix2D.h, [21](#)
- mat2D_minor_det
 - Matrix2D.h, [46](#)
- mat2D_minor_free
 - Matrix2D.h, [46](#)
- MAT2D_MINOR_PRINT
 - Matrix2D.h, [21](#)
- mat2D_minor_print
 - Matrix2D.h, [47](#)
- mat2D_mult
 - Matrix2D.h, [47](#)
- mat2D_mult_row
 - Matrix2D.h, [48](#)
- mat2D_normalize
 - Matrix2D.h, [21](#)
- mat2D_normalize_inf
 - Matrix2D.h, [21](#)
- mat2D_offset2d
 - Matrix2D.h, [48](#)
- mat2D_offset2d_uint32
 - Matrix2D.h, [49](#)
- MAT2D_ONES_ON_DIAG
 - Matrix2D.h, [22](#)
- mat2D_outer_product
 - Matrix2D.h, [49](#)
- MAT2D_PI
 - Matrix2D.h, [22](#)
- mat2D_power_iterate
 - Matrix2D.h, [50](#)
- MAT2D_PRINT
 - Matrix2D.h, [22](#)
- mat2D_print
 - Matrix2D.h, [51](#)
- MAT2D_PRINT_AS_COL
 - Matrix2D.h, [22](#)
- mat2D_print_as_col
 - Matrix2D.h, [51](#)
- mat2D_rand
 - Matrix2D.h, [52](#)
- mat2D_rand_double
 - Matrix2D.h, [52](#)
- mat2D_reduce
 - Matrix2D.h, [53](#)
- mat2D_row_is_all_digit
 - Matrix2D.h, [53](#)
- MAT2D_ROW_SWAPPING
 - Matrix2D.h, [22](#)
- mat2D_set_DCM_zyx
 - Matrix2D.h, [54](#)
- mat2D_set_identity
 - Matrix2D.h, [54](#)
- mat2D_set_rot_mat_x
 - Matrix2D.h, [55](#)
- mat2D_set_rot_mat_y
 - Matrix2D.h, [55](#)
- mat2D_set_rot_mat_z
 - Matrix2D.h, [56](#)
- mat2D_shift
 - Matrix2D.h, [56](#)
- mat2D_solve_linear_sys_LUP_decomposition
 - Matrix2D.h, [57](#)
- mat2D_sub
 - Matrix2D.h, [57](#)
- mat2D_sub_col_to_col
 - Matrix2D.h, [58](#)
- mat2D_sub_row_time_factor_to_row
 - Matrix2D.h, [58](#)

- mat2D_sub_row_to_row
 - Matrix2D.h, [59](#)
- mat2D_SVD_full
 - Matrix2D.h, [59](#)
- mat2D_SVD_thin
 - Matrix2D.h, [60](#)
- mat2D_swap_rows
 - Matrix2D.h, [62](#)
- mat2D_transpose
 - Matrix2D.h, [62](#)
- Mat2D_uint32, [9](#)
 - cols, [10](#)
 - elements, [10](#)
 - rows, [10](#)
 - stride_r, [10](#)
- mat2D_upper_triangular
 - Matrix2D.h, [63](#)
- mat2D_upper_triangular_flag
 - Matrix2D.h, [22](#)
- Matrix2D.h, [13](#)
 - mat2D_add, [23](#)
 - mat2D_add_col_to_col, [23](#)
 - mat2D_add_row_time_factor_to_row, [24](#)
 - mat2D_add_row_to_row, [24](#)
 - mat2D_alloc, [25](#)
 - mat2D_alloc_uint32, [26](#)
 - MAT2D_ASSERT, [18](#)
 - MAT2D_AT, [18](#)
 - MAT2D_AT_UINT32, [19](#)
 - mat2D_calc_col_norma, [26](#)
 - mat2D_calc_norma, [27](#)
 - mat2D_calc_norma_inf, [27](#)
 - mat2D_col_is_all_digit, [28](#)
 - mat2D_copy, [28](#)
 - mat2D_copy_col_from_src_to_des, [29](#)
 - mat2D_copy_row_from_src_to_des, [30](#)
 - mat2D_copy_src_to_des_window, [30](#)
 - mat2D_copy_src_window_to_des, [31](#)
 - mat2D_create_col_ref, [31](#)
 - mat2D_cross, [32](#)
 - mat2D_det, [33](#)
 - mat2D_det_2x2_mat, [33](#)
 - mat2D_det_2x2_mat_minor, [34](#)
 - mat2D_dot, [34](#)
 - mat2D_dot_product, [35](#)
 - mat2D_dprintDOUBLE, [19](#)
 - mat2D_dprintINT, [19](#)
 - mat2D_dprintSIZE_T, [19](#)
 - mat2D_eig_check, [36](#)
 - mat2D_eig_power_iteration, [36](#)
 - MAT2D_EPS, [20](#)
 - mat2D_fill, [37](#)
 - mat2D_fill_sequence, [38](#)
 - mat2D_fill_uint32, [38](#)
 - mat2D_find_first_non_zero_value, [39](#)
 - MAT2D_FREE, [20](#)
 - mat2D_free, [39](#)
 - mat2D_free_uint32, [40](#)
 - mat2D_inner_product, [40](#)
 - mat2D_invert, [41](#)
 - MAT2D_IS_ZERO, [20](#)
 - mat2D_LUP_decomposition_with_swap, [42](#)
 - mat2D_make_orthogonal_Gaussian_elimination, [42](#)
 - mat2D_make_orthogonal_modified_Gram_Schmidt, [43](#)
 - MAT2D_MALLOC, [20](#)
 - mat2D_mat_is_all_digit, [44](#)
 - MAT2D_MAX_POWER_ITERATION, [20](#)
 - mat2D_minor_alloc_fill_from_mat, [44](#)
 - mat2D_minor_alloc_fill_from_mat_minor, [45](#)
 - MAT2D_MINOR_AT, [21](#)
 - mat2D_minor_det, [46](#)
 - mat2D_minor_free, [46](#)
 - MAT2D_MINOR_PRINT, [21](#)
 - mat2D_minor_print, [47](#)
 - mat2D_mult, [47](#)
 - mat2D_mult_row, [48](#)
 - mat2D_normalize, [21](#)
 - mat2D_normalize_inf, [21](#)
 - mat2D_offset2d, [48](#)
 - mat2D_offset2d_uint32, [49](#)
 - MAT2D_ONES_ON_DIAG, [22](#)
 - mat2D_outer_product, [49](#)
 - MAT2D_PI, [22](#)
 - mat2D_power_iterate, [50](#)
 - MAT2D_PRINT, [22](#)
 - mat2D_print, [51](#)
 - MAT2D_PRINT_AS_COL, [22](#)
 - mat2D_print_as_col, [51](#)
 - mat2D_rand, [52](#)
 - mat2D_rand_double, [52](#)
 - mat2D_reduce, [53](#)
 - mat2D_row_is_all_digit, [53](#)
 - MAT2D_ROW_SWAPPING, [22](#)
 - mat2D_set_DCM_zyx, [54](#)
 - mat2D_set_identity, [54](#)
 - mat2D_set_rot_mat_x, [55](#)
 - mat2D_set_rot_mat_y, [55](#)
 - mat2D_set_rot_mat_z, [56](#)
 - mat2D_shift, [56](#)
 - mat2D_solve_linear_sys_LUP_decomposition, [57](#)
 - mat2D_sub, [57](#)
 - mat2D_sub_col_to_col, [58](#)
 - mat2D_sub_row_time_factor_to_row, [58](#)
 - mat2D_sub_row_to_row, [59](#)
 - mat2D_SVD_full, [59](#)
 - mat2D_SVD_thin, [60](#)
 - mat2D_swap_rows, [62](#)
 - mat2D_transpose, [62](#)
 - mat2D_upper_triangular, [63](#)
 - mat2D_upper_triangular_flag, [22](#)
- MATRIX2D_IMPLEMENTATION
 - example1.c, [11](#)
 - temp.c, [81](#)
 - test_matrix2d.c, [83](#)

- nearly_equal
 - test_matrix2d.c, 86
- ref_mat
 - Mat2D_Minor, 8
- rng_range
 - test_matrix2d.c, 86
- rng_unit01
 - test_matrix2d.c, 87
- rows
 - Mat2D, 6
 - Mat2D_Minor, 8
 - Mat2D_uint32, 10
- rows_list
 - Mat2D_Minor, 9
- RUN_TEST
 - test_matrix2d.c, 83
- stride_r
 - Mat2D, 6
 - Mat2D_Minor, 9
 - Mat2D_uint32, 10
- temp.c, 80
 - main, 81
 - MATRIX2D_IMPLEMENTATION, 81
- test_alloc_fill_copy_add_sub
 - test_matrix2d.c, 87
- test_copy_row_and_col_helpers
 - test_matrix2d.c, 87
- test_copy_windows
 - test_matrix2d.c, 87
- test_DCM_zyx_matches_product
 - test_matrix2d.c, 88
- test_det_2x2_and_upper_triangular_sign
 - test_matrix2d.c, 88
- test_det_and_minor_det_agree_3x3
 - test_matrix2d.c, 88
- test_det_early_zero_row_and_zero_col_paths
 - test_matrix2d.c, 88
- test_deterministic_fuzz_loop
 - test_matrix2d.c, 89
- test_dot_product_and_vector_variants
 - test_matrix2d.c, 89
- test_dot_product_matrix_multiply
 - test_matrix2d.c, 89
- test_invert
 - test_matrix2d.c, 89
- test_LUP_decomposition_identity_P_no_swap_case
 - test_matrix2d.c, 90
- test_LUP_decomposition_swap_required_case
 - test_matrix2d.c, 90
- test_mat_is_all_digit
 - test_matrix2d.c, 90
- test_matrix2d.c, 82
 - assert_identity_close, 84
 - assert_inverse_identity_both_sides, 84
 - assert_mat_close, 84
 - assert_permutation_matrix, 84
 - close_rel_abs, 85
 - det_by_minors_first_col, 85
 - fill_mat_from_array, 85
 - fill_strictly_diag_dominant, 85
 - main, 86
 - MATRIX2D_IMPLEMENTATION, 83
 - nearly_equal, 86
 - rng_range, 86
 - rng_unit01, 87
 - RUN_TEST, 83
 - test_alloc_fill_copy_add_sub, 87
 - test_copy_row_and_col_helpers, 87
 - test_copy_windows, 87
 - test_DCM_zyx_matches_product, 88
 - test_det_2x2_and_upper_triangular_sign, 88
 - test_det_and_minor_det_agree_3x3, 88
 - test_det_early_zero_row_and_zero_col_paths, 88
 - test_deterministic_fuzz_loop, 89
 - test_dot_product_and_vector_variants, 89
 - test_dot_product_matrix_multiply, 89
 - test_invert, 89
 - test_LUP_decomposition_identity_P_no_swap_case, 90
 - test_LUP_decomposition_swap_required_case, 90
 - test_mat_is_all_digit, 90
 - test_minor_det_matches_gauss_4x4_known, 90
 - test_non_contiguous_stride_views, 91
 - test_norms_and_normalize, 91
 - test_offset2d_and_stride, 91
 - test_outer_product_and_cross, 91
 - test_outer_product_row_vector_path, 92
 - test_power_iterate_and_eig_helpers, 92
 - test_rand_range, 92
 - test_reduce_rank, 92
 - test_rotation_matrices_orthonormal, 93
 - test_row_col_ops_and_scaling, 93
 - test_shift_and_identity, 93
 - test_solve_linear_system_LUP, 93
 - test_transpose, 94
 - test_uint32_alloc_fill_and_at, 94
 - xorshift64star, 94
- test_minor_det_matches_gauss_4x4_known
 - test_matrix2d.c, 90
- test_non_contiguous_stride_views
 - test_matrix2d.c, 91
- test_norms_and_normalize
 - test_matrix2d.c, 91
- test_offset2d_and_stride
 - test_matrix2d.c, 91
- test_outer_product_and_cross
 - test_matrix2d.c, 91
- test_outer_product_row_vector_path
 - test_matrix2d.c, 92
- test_power_iterate_and_eig_helpers
 - test_matrix2d.c, 92
- test_rand_range
 - test_matrix2d.c, 92
- test_reduce_rank

- test_matrix2d.c, [92](#)
- test_rotation_matrices_orthonormal
 - test_matrix2d.c, [93](#)
- test_row_col_ops_and_scaling
 - test_matrix2d.c, [93](#)
- test_shift_and_identity
 - test_matrix2d.c, [93](#)
- test_solve_linear_system_LUP
 - test_matrix2d.c, [93](#)
- test_transpose
 - test_matrix2d.c, [94](#)
- test_uint32_alloc_fill_and_at
 - test_matrix2d.c, [94](#)
- xorshift64star
 - test_matrix2d.c, [94](#)