Matrix2D

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

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_ 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_Gr 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 mat2D_sub_row_to_row(), mat2D_SVD_full(), mat2D_SVD_thin(), mat2D_swap_rows(), mat2D_transpose(), mat2D_upper_triangulate(), 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_triangulate(), 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:

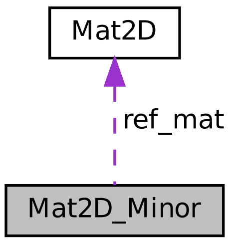
- Matrix2D.h

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:

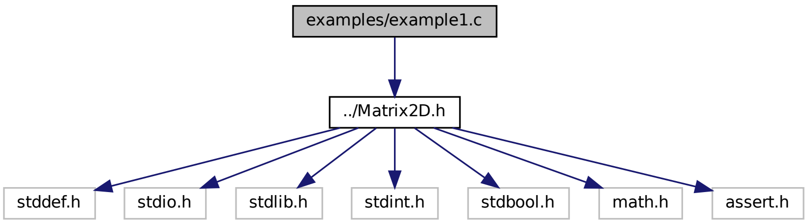
- Matrix2D.h

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

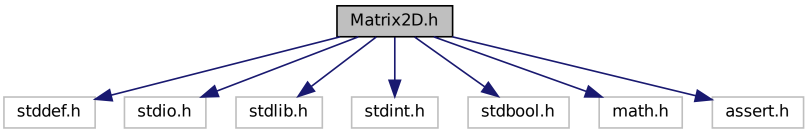
```
00001 #define MATRIX2D_IMPLEMENTATION
00002 #include "../Matrix2D.h"
00003
00004 int main(void)
00005 {
00006     int n = 4;
00007     int m = 5;
00008
00009     Mat2D A = mat2D_alloc(n, m);
00010     Mat2D U = mat2D_alloc(n, n);
00011     Mat2D S = mat2D_alloc(n, m);
00012     Mat2D V = mat2D_alloc(m, 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

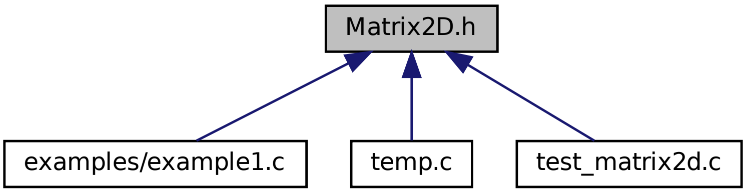
Lightweight 2D matrix helpers (double / uint32_t).

```
#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_triangulate_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(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: dst = a x b for 3x1 vectors.
- void mat2D_dot (Mat2D dst, Mat2D a, Mat2D b)

  Matrix product: 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: 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∗A = L∗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 ∗= factor.
- void mat2D_mult_row (Mat2D m, size_t r, double factor)

  In-place row scaling: row(r) ∗= 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: des = v ∗ 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] += shift.
- void mat2D_solve_linear_sys_LUP_decomposition (Mat2D A, Mat2D x, Mat2D B)

  Solve the linear system A x = B using an LUP-based approach.
- void mat2D_sub (Mat2D dst, Mat2D a)

  In-place subtraction: 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: row(des_r) -= factor ∗ row(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: des = 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/L1TbWe8b↩VOc?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

dst	Destination matrix to be incremented.
a	Summand of same shape as dst.

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

des	Destination matrix (same row count as src).
des_col	Column index in destination.
src	Source matrix.
src_col	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: row(des_r) += factor ∗ row(src_r).

Parameters

m	Matrix.
des_r	Destination row index.
src_r	Source row index.
factor	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: des[des_row, :] += src[src_row, :]

Parameters

des	Destination matrix (same number of columns as `src`).
des_row	Row index in destination.
src	Source matrix.
src_row	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

rows	Number of rows. Must be > 0.
cols	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_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_sw 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

rows	Number of rows. Must be $>$ 0.
cols	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

m	Matrix.
c	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

m	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

m	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

m	Matrix.
digit	Value to compare.
c	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

des	Destination matrix.
src	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

des	Destination matrix (same row count as src).
des_col	Column index in destination.
src	Source matrix.
src_col	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

des	Destination matrix (same number of columns as src).
des_row	Row index in destination.
src	Source matrix.
src_row	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

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

des	Destination matrix. Must have size (ie - is + 1) x (je - js + 1).
src	Source matrix.
is	Start row index in src (inclusive).
js	Start column index in src (inclusive).
ie	End row index in src (inclusive).
je	End column index in src (inclusive).

Precondition

0 <= is <= ie < src.rows, 0 <= js <= 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 x 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

src	Source matrix.
c	Column index in `src`.

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 x b for 3x1 vectors.

Parameters

dst	3x1 destination vector.
a	3x1 input vector.
b	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

m	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

m	Matrix (must be 2x2).

Returns

det(m) = m00∗m11 - m01∗m10.

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

mm	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

dst	Destination matrix (size a.rows x b.cols).
a	Left matrix (size a.rows x a.cols).
b	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_ide 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

a	Vector (shape n x 1 or 1 x n).
b	Vector (same shape as a).

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 \Lambda).

Parameters

A	Square matrix (N x N).
eigenvalues	Diagonal matrix (\Lambda) (N x N).
eigenvectors	Matrix of eigenvectors (V) (N x N), typically with eigenvectors stored as columns.
res	Destination matrix (N x N) receiving the residual.

Postcondition

`res` is overwritten with (A V - V \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

	A	Input square matrix (N x N).
out	*eigenvalues*	Destination (N x N) written as a diagonal matrix.
out	*eigenvectors*	Destination (N x N) whose columns are the estimated eigenvectors.
	init_vector	Initial guess (N x 1), must have non-zero norm.
	norm_inf_vectors	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

m	Matrix to fill.
x	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

m	Matrix to fill.
start	First value in the sequence.
step	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

m	Matrix to fill.
x	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

	m	Matrix to search.
	r	Row index to search (0-based).
out	*non_zero_col*	On success, receives the column index of the first element in row `r` 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

m	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_sw 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

m	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 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

v	Vector (shape n x 1 or 1 x n).

Returns

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

Precondition

v.cols == 1 || 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

des	Destination matrix (same shape as src).
src	Source square matrix.

Precondition

src is square.

des is allocated as the same shape as src.

Forms an augmented matrix [src | 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∗A = L∗U with L unit diagonal.

Parameters

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

des	Destination matrix.
A	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^TA 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

des	Destination matrix (overwritten).
A	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

m	Matrix.
digit	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

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

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

ref_mm	Reference minor.
i	Excluded row index in the minor.
j	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

mm	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

mm	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

mm	Minor to print.
name	Label to print.
padding	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

m	Matrix.
factor	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: row(r) $*=$ factor.

Parameters

m	Matrix.
r	Row index.
factor	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_triangulate(), 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

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

Returns

The linear offset i * stride_r + j.

Precondition

0 <= i < m.rows and 0 <= j < 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

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

Returns

The linear offset i * stride_r + j.

Precondition

0 <= i < m.rows and 0 <= j < 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: des = v * v^T.

Parameters

des	Destination square matrix (n x n).
v	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

	A	Square matrix (N x N).
`in,out`	*v*	Initial guess vector (N x 1). Overwritten with the estimated dominant eigenvector of the shifted matrix.
`out`	*lambda*	Optional output for the eigenvalue estimate (may be NULL).
	shift	Diagonal shift applied as described above.
	norm_inf↩_v	If true, normalize `v` 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

m	Matrix to print.
name	Label to print.
padding	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

m	Matrix to print (flattened in row-major).
name	Label to print.
padding	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

m	Matrix to fill.
low	Lower bound (inclusive).
high	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

m	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 | 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

m	Matrix.
digit	Value to compare.
r	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

DCM	3x3 destination matrix.
yaw_deg	Rotation about Z in degrees.
pitch_deg	Rotation about Y in degrees.
roll_deg	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

m	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

m	3x3 destination matrix.
angle_deg	Angle in degrees.

The matrix written is: [1, 0 , 0] [0, cos(a), sin(a)] [0,-sin(a), cos(a)]

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

m	3x3 destination matrix.
angle_deg	Angle in degrees.

The matrix written is: [cos(a), 0,-sin(a)] [0 , 1, 0] [sin(a), 0, cos(a)]

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

m	3x3 destination matrix.
angle_deg	Angle in degrees.

The matrix written is: [cos(a), sin(a), 0] [-sin(a), cos(a), 0] [0 , 0 , 1]

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] += shift.

Parameters

m	Square matrix modified in-place.
shift	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 A x = B using an LUP-based approach.

Parameters

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

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

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

dst	Destination matrix to be decremented.
a	Subtrahend of same shape as dst.

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

des	Destination matrix (same row count as src).
des_col	Column index in destination.
src	Source matrix.
src_col	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

m	Matrix.
des↩_r	Destination row index.
src↩_r	Source row index.
factor	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

des	Destination matrix (same number of columns as src).
des_row	Row index in destination.
src	Source matrix.
src_row	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

A	Input matrix (n x m).
U	Output U (n x n).
S	Output S (n x m) with singular values on its diagonal.
V	Output V (m x m) or V^T depending on `return_v_transpose`.
init_vec_u	Initial vector for eigen/power iteration when using (A A^T) (n x 1).
init_vec_v	Initial vector for eigen/power iteration when using (A^T A) (m x 1).
return_v_transpose	If true, writes V^T into `V` (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 <= m: compute eigenpairs of (A A^T) (n x n), fill U directly, then compute (v_i = A^T u_i / \sigma_i).
- Else: compute eigenpairs of (A^T A) (m x 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

A	Input matrix (n x m).
U	Output matrix (n x n). Only the first min(n,m) columns corresponding to non-zero singular values are meaningfully populated by this step.
S	Output matrix (n x m). Singular values written on the diagonal.
V	Output matrix (m x m) (or V^T if `return_v_transpose` is true).
init_vec_u	Initial vector for eigen iteration in the (A A^T) path (n x 1).
init_vec_v	Initial vector for eigen iteration in the (A^T A) path (m x 1).
return_v_transpose	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

m	Matrix.
r1	First row index.
r2	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

des	Destination matrix (shape src.cols x src.rows).
src	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

m	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_trian

4.4 Matrix2D.h

```
#ifndef MATRIX2D_H_
#define MATRIX2D_H_

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

#ifndef MAT2D_MALLOC
#define MAT2D_MALLOC malloc
#endif //MAT2D_MALLOC

#ifndef MAT2D_FREE
#define MAT2D_FREE free
#endif //MAT2D_FREE

#ifndef MAT2D_ASSERT
#include <assert.h>
#define MAT2D_ASSERT assert
#endif //MAT2D_ASSERT

typedef struct {
    size_t rows;
    size_t cols;
    size_t stride_r; /* elements to traverse to reach the next row */
    double *elements;
} Mat2D;

typedef struct {
    size_t rows;
    size_t cols;
    size_t stride_r; /* elements to traverse to reach the next row */
    uint32_t *elements;
} Mat2D_uint32;

typedef struct {
    size_t rows;
    size_t cols;
    size_t stride_r; /* logical stride for the minor shape (not used for access) */
    size_t *rows_list;
    size_t *cols_list;
    Mat2D ref_mat;
} Mat2D_Minor;

#if 1
#define MAT2D_AT(m, i, j) (m).elements[mat2D_offset2d((m), (i), (j))]
#define MAT2D_AT_UINT32(m, i, j) (m).elements[mat2D_offset2d_uint32((m), (i), (j))]
#else /* use this macro for batter performance but no assertion */
#define MAT2D_AT(m, i, j) (m).elements[(i) * (m).stride_r + (j)]
#define MAT2D_AT_UINT32(m, i, j) (m).elements[(i) * (m).stride_r + (j)]
#endif

#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)

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

```

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

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

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

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

#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)

enum mat2D_upper_triangulate_flag{
    MAT2D_ONES_ON_DIAG = 1 « 0,
    MAT2D_ROW_SWAPPING = 1 « 1,
};

void            mat2D_add(Mat2D dst, Mat2D a);
void            mat2D_add_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
void            mat2D_add_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
void            mat2D_add_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor);
Mat2D           mat2D_alloc(size_t rows, size_t cols);
Mat2D_uint32    mat2D_alloc_uint32(size_t rows, size_t cols);

double          mat2D_calc_col_norma(Mat2D m, size_t c);
double          mat2D_calc_norma(Mat2D m);
double          mat2D_calc_norma_inf(Mat2D m);
bool            mat2D_col_is_all_digit(Mat2D m, double digit, size_t c);
void            mat2D_copy(Mat2D des, Mat2D src);
void            mat2D_copy_col_from_src_to_des(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
void            mat2D_copy_row_from_src_to_des(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
void            mat2D_copy_src_to_des_window(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie,
   size_t je);
void            mat2D_copy_src_window_to_des(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie,
   size_t je);
Mat2D           mat2D_create_col_ref(Mat2D src, size_t c);
void            mat2D_cross(Mat2D dst, Mat2D v1, Mat2D v2);

void            mat2D_dot(Mat2D dst, Mat2D a, Mat2D b);
double          mat2D_dot_product(Mat2D v1, Mat2D v2);
double          mat2D_det(Mat2D m);
double          mat2D_det_2x2_mat(Mat2D m);
double          mat2D_det_2x2_mat_minor(Mat2D_Minor mm);

void            mat2D_eig_check(Mat2D A, Mat2D eigenvalues, Mat2D eigenvectors, Mat2D res);
void            mat2D_eig_power_iteration(Mat2D A, Mat2D eigenvalues, Mat2D eigenvectors, Mat2D
   init_vector, bool norm_inf_vectors);

void            mat2D_fill(Mat2D m, double x);
void            mat2D_fill_sequence(Mat2D m, double start, double step);
void            mat2D_fill_uint32(Mat2D_uint32 m, uint32_t x);
bool            mat2D_find_first_non_zero_value(Mat2D m, size_t r, size_t *non_zero_col);
void            mat2D_free(Mat2D m);
void            mat2D_free_uint32(Mat2D_uint32 m);

double          mat2D_inner_product(Mat2D v);
void            mat2D_invert(Mat2D des, Mat2D src);

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

void            mat2D_make_orthogonal_Gaussian_elimination(Mat2D des, Mat2D A);
void            mat2D_make_orthogonal_modified_Gram_Schmidt(Mat2D des, Mat2D A);
bool            mat2D_mat_is_all_digit(Mat2D m, double digit);
Mat2D_Minor     mat2D_minor_alloc_fill_from_mat(Mat2D ref_mat, size_t i, size_t j);
Mat2D_Minor     mat2D_minor_alloc_fill_from_mat_minor(Mat2D_Minor ref_mm, size_t i, size_t j);
double          mat2D_minor_det(Mat2D_Minor mm);
void            mat2D_minor_free(Mat2D_Minor mm);
void            mat2D_minor_print(Mat2D_Minor mm, const char *name, size_t padding);
void            mat2D_mult(Mat2D m, double factor);
void            mat2D_mult_row(Mat2D m, size_t r, double factor);

size_t          mat2D_offset2d(Mat2D m, size_t i, size_t j);
size_t          mat2D_offset2d_uint32(Mat2D_uint32 m, size_t i, size_t j);
void            mat2D_outer_product(Mat2D des, Mat2D v);

int             mat2D_power_iterate(Mat2D A, Mat2D v, double *lambda, double shift, bool norm_inf_v);
void            mat2D_print(Mat2D m, const char *name, size_t padding);
void            mat2D_print_as_col(Mat2D m, const char *name, size_t padding);

void            mat2D_rand(Mat2D m, double low, double high);
double          mat2D_rand_double(void);
size_t          mat2D_reduce(Mat2D m);
bool            mat2D_row_is_all_digit(Mat2D m, double digit, size_t r);
```

```
void            mat2D_set_DCM_zyx(Mat2D DCM, float yaw_deg, float pitch_deg, float roll_deg);
void            mat2D_set_identity(Mat2D m);
void            mat2D_set_rot_mat_x(Mat2D m, float angle_deg);
void            mat2D_set_rot_mat_y(Mat2D m, float angle_deg);
void            mat2D_set_rot_mat_z(Mat2D m, float angle_deg);
void            mat2D_shift(Mat2D m, double shift);
void            mat2D_solve_linear_sys_LUP_decomposition(Mat2D A, Mat2D x, Mat2D B);
void            mat2D_sub(Mat2D dst, Mat2D a);
void            mat2D_sub_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col);
void            mat2D_sub_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row);
void            mat2D_sub_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor);
void            mat2D_SVD_full(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v,
 bool return_v_transpose);
void            mat2D_SVD_thin(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v,
 bool return_v_transpose);
void            mat2D_swap_rows(Mat2D m, size_t r1, size_t r2);

void            mat2D_transpose(Mat2D des, Mat2D src);

double          mat2D_upper_triangulate(Mat2D m, uint8_t flags);

#endif // MATRIX2D_H_

#ifdef MATRIX2D_IMPLEMENTATION
#undef MATRIX2D_IMPLEMENTATION


void mat2D_add(Mat2D dst, Mat2D a)
{
    MAT2D_ASSERT(dst.rows == a.rows);
    MAT2D_ASSERT(dst.cols == a.cols);
    for (size_t i = 0; i < dst.rows; ++i) {
        for (size_t j = 0; j < dst.cols; ++j) {
            MAT2D_AT(dst, i, j) += MAT2D_AT(a, i, j);
        }
    }
}

void mat2D_add_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
{
    MAT2D_ASSERT(src_col < src.cols);
    MAT2D_ASSERT(des.rows == src.rows);
    MAT2D_ASSERT(des_col < des.cols);

    for (size_t i = 0; i < des.rows; i++) {
        MAT2D_AT(des, i, des_col) += MAT2D_AT(src, i, src_col);
    }
}

void mat2D_add_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
{
    MAT2D_ASSERT(src_row < src.rows);
    MAT2D_ASSERT(des.cols == src.cols);
    MAT2D_ASSERT(des_row < des.rows);

    for (size_t j = 0; j < des.cols; j++) {
        MAT2D_AT(des, des_row, j) += MAT2D_AT(src, src_row, j);
    }
}

void mat2D_add_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor)
{
    for (size_t j = 0; j < m.cols; ++j) {
        MAT2D_AT(m, des_r, j) += factor * MAT2D_AT(m, src_r, j);
    }
}

Mat2D mat2D_alloc(size_t rows, size_t cols)
{
    Mat2D m;
    m.rows = rows;
    m.cols = cols;
    m.stride_r = cols;
    m.elements = (double*)MAT2D_MALLOC(sizeof(double)*rows*cols);
    MAT2D_ASSERT(m.elements != NULL);

    return m;
}

Mat2D_uint32 mat2D_alloc_uint32(size_t rows, size_t cols)
{
    Mat2D_uint32 m;
    m.rows = rows;
    m.cols = cols;
    m.stride_r = cols;
    m.elements = (uint32_t*)MAT2D_MALLOC(sizeof(uint32_t)*rows*cols);
```

```
    MAT2D_ASSERT(m.elements != NULL);

    return m;
}

double mat2D_calc_col_norma(Mat2D m, size_t c)
{
    MAT2D_ASSERT(c < m.cols);

    double sum = 0;
    for (size_t i = 0; i < m.rows; ++i) {
        sum += MAT2D_AT(m, i, c) * MAT2D_AT(m, i, c);
    }
    return sqrt(sum);
}

double mat2D_calc_norma(Mat2D m)
{
    double sum = 0;

    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            sum += MAT2D_AT(m, i, j) * MAT2D_AT(m, i, j);
        }
    }
    return sqrt(sum);
}

double mat2D_calc_norma_inf(Mat2D m)
{
    double max = 0;
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            double current = fabs(MAT2D_AT(m, i, j));
            if (current > max) {
                max = current;
            }
        }
    }

    return max;
}

bool mat2D_col_is_all_digit(Mat2D m, double digit, size_t c)
{
    for (size_t i = 0; i < m.rows; ++i) {
        if (MAT2D_AT(m, i, c) != digit) {
            return false;
        }
    }
    return true;
}

void mat2D_copy(Mat2D des, Mat2D src)
{
    MAT2D_ASSERT(des.cols == src.cols);
    MAT2D_ASSERT(des.rows == src.rows);

    for (size_t i = 0; i < des.rows; ++i) {
        for (size_t j = 0; j < des.cols; ++j) {
            MAT2D_AT(des, i, j) = MAT2D_AT(src, i, j);
        }
    }
}

void mat2D_copy_col_from_src_to_des(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
{
    MAT2D_ASSERT(src_col < src.cols);
    MAT2D_ASSERT(des.rows == src.rows);
    MAT2D_ASSERT(des_col < des.cols);

    for (size_t i = 0; i < des.rows; i++) {
        MAT2D_AT(des, i, des_col) = MAT2D_AT(src, i, src_col);
    }
}

void mat2D_copy_row_from_src_to_des(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
{
    MAT2D_ASSERT(src_row < src.rows);
    MAT2D_ASSERT(des.cols == src.cols);
    MAT2D_ASSERT(des_row < des.rows);

    for (size_t j = 0; j < des.cols; j++) {
        MAT2D_AT(des, des_row, j) = MAT2D_AT(src, src_row, j);
    }
}
```

```c
void mat2D_copy_src_to_des_window(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie, size_t je)
{
    MAT2D_ASSERT(je >= js && ie >= is);
    MAT2D_ASSERT(je-js+1 == src.cols);
    MAT2D_ASSERT(ie-is+1 == src.rows);
    MAT2D_ASSERT(je-js+1 <= des.cols);
    MAT2D_ASSERT(ie-is+1 <= des.rows);

    for (size_t index = 0; index < src.rows; ++index) {
        for (size_t jndex = 0; jndex < src.cols; ++jndex) {
            MAT2D_AT(des, is+index, js+jndex) = MAT2D_AT(src, index, jndex);
        }
    }
}

void mat2D_copy_src_window_to_des(Mat2D des, Mat2D src, size_t is, size_t js, size_t ie, size_t je)
{
    MAT2D_ASSERT(je >= js && ie >= is);
    MAT2D_ASSERT(je-js+1 == des.cols);
    MAT2D_ASSERT(ie-is+1 == des.rows);
    MAT2D_ASSERT(je-js+1 <= src.cols);
    MAT2D_ASSERT(ie-is+1 <= src.rows);

    for (size_t index = 0; index < des.rows; ++index) {
        for (size_t jndex = 0; jndex < des.cols; ++jndex) {
            MAT2D_AT(des, index, jndex) = MAT2D_AT(src, is+index, js+jndex);
        }
    }
}

Mat2D mat2D_create_col_ref(Mat2D src, size_t c)
{
    MAT2D_ASSERT(c < src.cols);

    Mat2D col = {.cols = 1,
                 .rows = src.rows,
                 .stride_r = src.stride_r,
                 .elements = &(MAT2D_AT(src, 0, c))};

    return col;
}

void mat2D_cross(Mat2D dst, Mat2D v1, Mat2D v2)
{
    MAT2D_ASSERT(3 == dst.rows && 1 == dst.cols);
    MAT2D_ASSERT(3 == v1.rows && 1 == v1.cols);
    MAT2D_ASSERT(3 == v2.rows && 1 == v2.cols);

    MAT2D_AT(dst, 0, 0) = MAT2D_AT(v1, 1, 0) * MAT2D_AT(v2, 2, 0) - MAT2D_AT(v1, 2, 0) * MAT2D_AT(v2, 1, 0);
    MAT2D_AT(dst, 1, 0) = MAT2D_AT(v1, 2, 0) * MAT2D_AT(v2, 0, 0) - MAT2D_AT(v1, 0, 0) * MAT2D_AT(v2, 2, 0);
    MAT2D_AT(dst, 2, 0) = MAT2D_AT(v1, 0, 0) * MAT2D_AT(v2, 1, 0) - MAT2D_AT(v1, 1, 0) * MAT2D_AT(v2, 0, 0);
}

void mat2D_dot(Mat2D dst, Mat2D a, Mat2D b)
{
    MAT2D_ASSERT(a.cols == b.rows);
    MAT2D_ASSERT(a.rows == dst.rows);
    MAT2D_ASSERT(b.cols == dst.cols);

    size_t i, j, k;

    for (i = 0; i < dst.rows; i++) {
        for (j = 0; j < dst.cols; j++) {
            MAT2D_AT(dst, i, j) = 0;
            for (k = 0; k < a.cols; k++) {
                MAT2D_AT(dst, i, j) += MAT2D_AT(a, i, k)*MAT2D_AT(b, k, j);
            }
        }
    }

}

double mat2D_dot_product(Mat2D v1, Mat2D v2)
{
    MAT2D_ASSERT(v1.rows == v2.rows);
    MAT2D_ASSERT(v1.cols == v2.cols);
    MAT2D_ASSERT((1 == v1.cols && 1 == v2.cols) || (1 == v1.rows && 1 == v2.rows));

    double dot_product = 0;

    if (1 == v1.cols) {
        for (size_t i = 0; i < v1.rows; i++) {
            dot_product += MAT2D_AT(v1, i, 0) * MAT2D_AT(v2, i, 0);
        }
```

```
    } else {
        for (size_t j = 0; j < v1.cols; j++) {
            dot_product += MAT2D_AT(v1, 0, j) * MAT2D_AT(v2, 0, j);
        }
    }

    return dot_product;
}

double mat2D_det(Mat2D m)
{
    MAT2D_ASSERT(m.cols == m.rows && "should be a square matrix");

    /* checking if there is a row or column with all zeros */
    /* checking rows */
    for (size_t i = 0; i < m.rows; i++) {
        if (mat2D_row_is_all_digit(m, 0, i)) {
            return 0;
        }
    }
    /* checking cols */
    for (size_t j = 0; j < m.rows; j++) {
        if (mat2D_col_is_all_digit(m, 0, j)) {
            return 0;
        }
    }

    #if 0/* This is an implementation of naive determinant calculation using minors. This is too slow
  */
    double det = 0;
    /* TODO: finding beast row or col? */
    for (size_t i = 0, j = 0; i < m.rows; i++) { /* first column */
        if (MAT2D_AT(m, i, j) < 1e-10) continue;
        Mat2D_Minor sub_mm = mat2D_minor_alloc_fill_from_mat(m, i, j);
        int factor = (i+j)%2 ? -1 : 1;
        if (sub_mm.cols != 2) {
            MAT2D_ASSERT(sub_mm.cols == sub_mm.rows && "should be a square matrix");
            det += MAT2D_AT(m, i, j) * (factor) * mat2D_minor_det(sub_mm);
        } else if (sub_mm.cols == 2 && sub_mm.rows == 2) {
            det += MAT2D_AT(m, i, j) * (factor) * mat2D_det_2x2_mat_minor(sub_mm);;
        }
        mat2D_minor_free(sub_mm);
    }
    #endif

    Mat2D temp_m = mat2D_alloc(m.rows, m.cols);
    mat2D_copy(temp_m, m);
    double factor = mat2D_upper_triangulate(temp_m, MAT2D_ROW_SWAPPING);
    double diag_mul = 1;
    for (size_t i = 0; i < temp_m.rows; i++) {
        diag_mul *= MAT2D_AT(temp_m, i, i);
    }
    mat2D_free(temp_m);

    return diag_mul / factor;
}

double mat2D_det_2x2_mat(Mat2D m)
{
    MAT2D_ASSERT(2 == m.cols && 2 == m.rows && "Not a 2x2 matrix");
    return MAT2D_AT(m, 0, 0) * MAT2D_AT(m, 1, 1) - MAT2D_AT(m, 0, 1) * MAT2D_AT(m, 1, 0);
}

double mat2D_det_2x2_mat_minor(Mat2D_Minor mm)
{
    MAT2D_ASSERT(2 == mm.cols && 2 == mm.rows && "Not a 2x2 matrix");
    return MAT2D_MINOR_AT(mm, 0, 0) * MAT2D_MINOR_AT(mm, 1, 1) - MAT2D_MINOR_AT(mm, 0, 1) *
  MAT2D_MINOR_AT(mm, 1, 0);
}

void mat2D_eig_check(Mat2D A, Mat2D eigenvalues, Mat2D eigenvectors, Mat2D res)
{
    MAT2D_ASSERT(A.cols == A.rows);
    MAT2D_ASSERT(eigenvalues.cols == A.cols);
    MAT2D_ASSERT(eigenvalues.rows == A.rows);
    MAT2D_ASSERT(eigenvectors.cols == A.cols);
    MAT2D_ASSERT(eigenvectors.rows == A.rows);
    MAT2D_ASSERT(res.cols == A.cols);
    MAT2D_ASSERT(res.rows == A.rows);

    #if 1
    mat2D_dot(res, A, eigenvectors);
    Mat2D VL = mat2D_alloc(A.rows, A.cols);
    mat2D_dot(VL, eigenvectors, eigenvalues);

    mat2D_sub(res, VL);

```

```
    mat2D_free(VL);
    #else
    Mat2D temp_v = mat2D_alloc(A.rows, 1);
    for (size_t i = 0; i < A.rows; i++) {
        Mat2D eig_vector = {.cols = 1,
                            .elements = &MAT2D_AT(eigenvectors, 0, i),
                            .rows = A.rows,
                            .stride_r = eigenvectors.stride_r};
        Mat2D v = {.cols = 1,
                   .elements = &MAT2D_AT(res, 0, i),
                   .rows = A.rows,
                   .stride_r = res.stride_r};

        mat2D_dot(temp_v, A, eig_vector);
        mat2D_copy(v, eig_vector);
        mat2D_mult(v, MAT2D_AT(eigenvalues, i, i ));

        mat2D_sub(v, temp_v);
    }
    mat2D_free(temp_v);
    #endif
}

void mat2D_eig_power_iteration(Mat2D A, Mat2D eigenvalues, Mat2D eigenvectors, Mat2D init_vector, bool norm_inf_vectors)
{
    /* https://www.youtube.com/watch?v=c8DIOzuZqBs */

    MAT2D_ASSERT(A.cols == A.rows);
    MAT2D_ASSERT(eigenvalues.cols == A.cols);
    MAT2D_ASSERT(eigenvalues.rows == A.rows);
    MAT2D_ASSERT(eigenvectors.cols == A.cols);
    MAT2D_ASSERT(eigenvectors.rows == A.rows);
    MAT2D_ASSERT(init_vector.cols == 1);
    MAT2D_ASSERT(init_vector.rows == A.rows);
    MAT2D_ASSERT(mat2D_calc_norma_inf(init_vector) > 0);

    mat2D_set_identity(eigenvalues);
    Mat2D B = mat2D_alloc(A.rows, A.cols);
    Mat2D temp_mat = mat2D_alloc(A.rows, A.cols);
    mat2D_copy(B, A);

    for (int i = 0, shift_value = 0; i < (int)A.rows; i++) {
        mat2D_copy_src_to_des_window(eigenvectors, init_vector, 0, i, init_vector.rows-1, i);
        Mat2D v = {.cols = init_vector.cols,
                   .elements = &MAT2D_AT(eigenvectors, 0, i),
                   .rows = init_vector.rows,
                   .stride_r = eigenvectors.stride_r};
        if (mat2D_power_iterate(B, v, &MAT2D_AT(eigenvalues, i, i), shift_value, 0)) { /* norm_inf_v must be zero*/
            shift_value++;
            i--;
            continue;
        } else {
            shift_value = 0;
        }
        mat2D_outer_product(temp_mat, v);
        mat2D_mult(temp_mat, MAT2D_AT(eigenvalues, i, i));
        mat2D_sub(B, temp_mat);
    }

    if (norm_inf_vectors) {
        for (size_t c = 0; c < eigenvectors.cols; c++) {
            Mat2D v = {.cols = init_vector.cols,
                       .elements = &MAT2D_AT(eigenvectors, 0, c),
                       .rows = init_vector.rows,
                       .stride_r = eigenvectors.stride_r};
            mat2D_normalize_inf(v);
        }
    }

    mat2D_free(B);
    mat2D_free(temp_mat);
}

void mat2D_fill(Mat2D m, double x)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) = x;
        }
    }
}

void mat2D_fill_sequence(Mat2D m, double start, double step) {
    for (size_t i = 0; i < m.rows; i++) {
        for (size_t j = 0; j < m.cols; j++) {
```

```c
            MAT2D_AT(m, i, j) = start + step * mat2D_offset2d(m, i, j);
        }
    }
}

void mat2D_fill_uint32(Mat2D_uint32 m, uint32_t x)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT_UINT32(m, i, j) = x;
        }
    }
}

bool mat2D_find_first_non_zero_value(Mat2D m, size_t r, size_t *non_zero_col)
{
    for (size_t c = 0; c < m.cols; ++c) {
        if (!MAT2D_IS_ZERO(MAT2D_AT(m, r, c))) {
            *non_zero_col = c;
            return true;
        }
    }
    return false;
}

void mat2D_free(Mat2D m)
{
    MAT2D_FREE(m.elements);
}

void mat2D_free_uint32(Mat2D_uint32 m)
{
    MAT2D_FREE(m.elements);
}

double mat2D_inner_product(Mat2D v)
{
    MAT2D_ASSERT((1 == v.cols) || (1 == v.rows));

    double dot_product = 0;

    if (1 == v.cols) {
        for (size_t i = 0; i < v.rows; i++) {
            dot_product += MAT2D_AT(v, i, 0) * MAT2D_AT(v, i, 0);
        }
    } else {
        for (size_t j = 0; j < v.cols; j++) {
            dot_product += MAT2D_AT(v, 0, j) * MAT2D_AT(v, 0, j);
        }
    }

    return dot_product;
}

void mat2D_invert(Mat2D des, Mat2D src)
{
    MAT2D_ASSERT(src.cols == src.rows && "Must be an NxN matrix");
    MAT2D_ASSERT(des.cols == src.cols && des.rows == des.cols);

    Mat2D m = mat2D_alloc(src.rows, src.cols * 2);
    mat2D_copy_src_to_des_window(m, src, 0, 0, src.rows-1, src.cols-1);

    mat2D_set_identity(des);
    mat2D_copy_src_to_des_window(m, des, 0, src.cols, des.rows-1, 2 * des.cols-1);

    mat2D_reduce(m);

    mat2D_copy_src_window_to_des(des, m, 0, src.cols, des.rows-1, 2 * des.cols-1);

    mat2D_free(m);
}

void mat2D_LUP_decomposition_with_swap(Mat2D src, Mat2D l, Mat2D p, Mat2D u)
{
    /* performing LU decomposition Following the Wikipedia page:
  https://en.wikipedia.org/wiki/LU_decomposition */

    mat2D_copy(u, src);
    mat2D_set_identity(p);
    mat2D_fill(l, 0);

    for (size_t i = 0; i < (size_t)fmin(u.rows-1, u.cols); i++) {
        if (MAT2D_IS_ZERO(MAT2D_AT(u, i, i))) {    /* swapping only if it is zero */
            /* finding biggest first number (absolute value) */
            size_t biggest_r = i;
            for (size_t index = i; index < u.rows; index++) {
                if (fabs(MAT2D_AT(u, index, i)) > fabs(MAT2D_AT(u, biggest_r, i))) {
```

```
                    biggest_r = index;
                }
            }
            if (i != biggest_r) {
                mat2D_swap_rows(u, i, biggest_r);
                mat2D_swap_rows(p, i, biggest_r);
                mat2D_swap_rows(l, i, biggest_r);
            }
        }
        for (size_t j = i+1; j < u.cols; j++) {
            double factor = 1 / MAT2D_AT(u, i, i);
            if (!isfinite(factor)) {
                printf("%s:%d:\n%s:\n[Error] unable to transfrom into uper triangular matrix. Probably some of the rows are not independent.\n", __FILE__, __LINE__, __func__);
            }
            double mat_value = MAT2D_AT(u, j, i);
            mat2D_sub_row_time_factor_to_row(u, j, i, mat_value * factor);
            MAT2D_AT(l, j, i) = mat_value * factor;
        }
        MAT2D_AT(l, i, i) = 1;
    }
    MAT2D_AT(l, l.rows-1, l.cols-1) = 1;
}

void mat2D_make_orthogonal_Gaussian_elimination(Mat2D des, Mat2D A)
{
    /* https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process */
    MAT2D_ASSERT(des.cols == A.cols);
    MAT2D_ASSERT(des.rows == A.rows);

    Mat2D AT = mat2D_alloc(A.cols, A.rows);
    Mat2D ATA = mat2D_alloc(A.cols, A.cols);
    Mat2D temp = mat2D_alloc(ATA.rows, ATA.cols + A.cols);
    Mat2D temp_des = mat2D_alloc(des.cols, des.rows);

    mat2D_transpose(AT, A);
    mat2D_dot(ATA, AT, A);
    mat2D_copy_src_to_des_window(temp, ATA, 0, 0, ATA.rows-1, ATA.cols-1);
    mat2D_copy_src_to_des_window(temp, AT, 0, ATA.cols, AT.rows-1, ATA.cols + AT.cols-1);

    MAT2D_PRINT(temp);

    mat2D_upper_triangulate(temp, MAT2D_ONES_ON_DIAG);

    mat2D_copy_src_window_to_des(temp_des, temp, 0, ATA.cols, AT.rows-1, ATA.cols + AT.cols-1);

    mat2D_transpose(des, temp_des);

    MAT2D_PRINT(temp);

    mat2D_free(AT);
    mat2D_free(ATA);
    mat2D_free(temp);
    mat2D_free(temp_des);
}

void mat2D_make_orthogonal_modified_Gram_Schmidt(Mat2D des, Mat2D A)
{
    /* https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process */
    MAT2D_ASSERT(des.rows == A.rows);
    MAT2D_ASSERT(des.cols == des.rows);

    size_t num_non_zero_vec = 0;
    for (size_t c = 0; c < A.cols; c++) {
        if (MAT2D_IS_ZERO(mat2D_calc_col_norma(A, c))) {
            break;
        }
        num_non_zero_vec++;
    }

    mat2D_dprintSIZE_T(num_non_zero_vec);

    mat2D_rand(des, 1, 2);

    Mat2D temp_col = mat2D_alloc(des.rows, 1);
    for (size_t c = 0; c < num_non_zero_vec; c++) {
        mat2D_copy_col_from_src_to_des(des, c, A, c);
    }
    for (size_t c = 0; c < des.cols-1; c++) {
        Mat2D vc = mat2D_create_col_ref(des, c);
        double vc_vc = mat2D_inner_product(vc);
        for (size_t k = c+1; k < des.cols; k++) {
            mat2D_copy(temp_col, vc);
            Mat2D vk = mat2D_create_col_ref(des, k);
            double vk_vc = mat2D_dot_product(vc, vk);
            mat2D_mult(temp_col, vk_vc / vc_vc);
            mat2D_sub(vk, temp_col);
```

```
        }
        if (!MAT2D_IS_ZERO(mat2D_calc_norma(vc))) {
            mat2D_normalize(vc);
        }
    }
    Mat2D vc = mat2D_create_col_ref(des, des.cols-1);
    if (!MAT2D_IS_ZERO(mat2D_calc_norma(vc))) {
        mat2D_normalize(vc);
    }


    mat2D_free(temp_col);
}

bool mat2D_mat_is_all_digit(Mat2D m, double digit)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            if (MAT2D_AT(m, i, j) != digit) {
                return false;
            }
        }
    }
    return true;
}

Mat2D_Minor mat2D_minor_alloc_fill_from_mat(Mat2D ref_mat, size_t i, size_t j)
{
    MAT2D_ASSERT(ref_mat.cols == ref_mat.rows && "minor is defined only for square matrix");

    Mat2D_Minor mm;
    mm.cols = ref_mat.cols-1;
    mm.rows = ref_mat.rows-1;
    mm.stride_r = ref_mat.cols-1;
    mm.cols_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mat.cols-1));
    mm.rows_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mat.rows-1));
    mm.ref_mat = ref_mat;

    MAT2D_ASSERT(mm.cols_list != NULL && mm.rows_list != NULL);

    for (size_t index = 0, temp_index = 0; index < ref_mat.rows; index++) {
        if (index != i) {
            mm.rows_list[temp_index] = index;
            temp_index++;
        }
    }
    for (size_t jndex = 0, temp_jndex = 0; jndex < ref_mat.cols; jndex++) {
        if (jndex != j) {
            mm.cols_list[temp_jndex] = jndex;
            temp_jndex++;
        }
    }

    return mm;
}

Mat2D_Minor mat2D_minor_alloc_fill_from_mat_minor(Mat2D_Minor ref_mm, size_t i, size_t j)
{
    MAT2D_ASSERT(ref_mm.cols == ref_mm.rows && "minor is defined only for square matrix");

    Mat2D_Minor mm;
    mm.cols = ref_mm.cols-1;
    mm.rows = ref_mm.rows-1;
    mm.stride_r = ref_mm.cols-1;
    mm.cols_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mm.cols-1));
    mm.rows_list = (size_t*)MAT2D_MALLOC(sizeof(size_t)*(ref_mm.rows-1));
    mm.ref_mat = ref_mm.ref_mat;

    MAT2D_ASSERT(mm.cols_list != NULL && mm.rows_list != NULL);

    for (size_t index = 0, temp_index = 0; index < ref_mm.rows; index++) {
        if (index != i) {
            mm.rows_list[temp_index] = ref_mm.rows_list[index];
            temp_index++;
        }
    }
    for (size_t jndex = 0, temp_jndex = 0; jndex < ref_mm.cols; jndex++) {
        if (jndex != j) {
            mm.cols_list[temp_jndex] = ref_mm.cols_list[jndex];
            temp_jndex++;
        }
    }

    return mm;
}

double mat2D_minor_det(Mat2D_Minor mm)
```

```
{
    MAT2D_ASSERT(mm.cols == mm.rows && "should be a square matrix");

    double det = 0;
    /* TODO: finding beast row or col? */
    for (size_t i = 0, j = 0; i < mm.rows; i++) { /* first column */
        if (fabs(MAT2D_MINOR_AT(mm, i, j)) < 1e-10) continue;
        Mat2D_Minor sub_mm = mat2D_minor_alloc_fill_from_mat_minor(mm, i, j);
        int factor = (i+j)%2 ? -1 : 1;
        if (sub_mm.cols != 2) {
            MAT2D_ASSERT(sub_mm.cols == sub_mm.rows && "should be a square matrix");
            det += MAT2D_MINOR_AT(mm, i, j) * (factor) * mat2D_minor_det(sub_mm);
        } else if (sub_mm.cols == 2 && sub_mm.rows == 2) {
            det += MAT2D_MINOR_AT(mm, i, j) * (factor) * mat2D_det_2x2_mat_minor(sub_mm);;
        }
        mat2D_minor_free(sub_mm);
    }
    return det;
}

void mat2D_minor_free(Mat2D_Minor mm)
{
    MAT2D_FREE(mm.cols_list);
    MAT2D_FREE(mm.rows_list);
}

void mat2D_minor_print(Mat2D_Minor mm, const char *name, size_t padding)
{
    printf("%*s%s = [\n", (int) padding, "", name);
    for (size_t i = 0; i < mm.rows; ++i) {
        printf("%*s    ", (int) padding, "");
        for (size_t j = 0; j < mm.cols; ++j) {
            printf("%f ", MAT2D_MINOR_AT(mm, i, j));
        }
        printf("\n");
    }
    printf("%*s]\n", (int) padding, "");
}

void mat2D_mult(Mat2D m, double factor)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) *= factor;
        }
    }
}

void mat2D_mult_row(Mat2D m, size_t r, double factor)
{
    for (size_t j = 0; j < m.cols; ++j) {
        MAT2D_AT(m, r, j) *= factor;
    }
}

size_t mat2D_offset2d(Mat2D m, size_t i, size_t j)
{
    MAT2D_ASSERT(i < m.rows && j < m.cols);
    return i * m.stride_r + j;
}

size_t mat2D_offset2d_uint32(Mat2D_uint32 m, size_t i, size_t j)
{
    MAT2D_ASSERT(i < m.rows && j < m.cols);
    return i * m.stride_r + j;
}

void mat2D_outer_product(Mat2D des, Mat2D v)
{
    MAT2D_ASSERT(des.cols == des.rows);
    MAT2D_ASSERT((1 == v.cols && des.rows == v.rows) || (1 == v.rows && des.cols == v.cols));

    // mat2D_fill(des, 0);

    if (1 == v.cols) {
        for (size_t i = 0; i < des.rows; i++) {
            for (size_t j = 0; j < des.cols; j++) {
                MAT2D_AT(des, i, j) = MAT2D_AT(v, i, 0) * MAT2D_AT(v, j, 0);
            }
        }
    } else {
        for (size_t i = 0; i < des.rows; i++) {
            for (size_t j = 0; j < des.cols; j++) {
                MAT2D_AT(des, i, j) = MAT2D_AT(v, 0, i) * MAT2D_AT(v, 0, j);
            }
        }
    }
```

```
}

int mat2D_power_iterate(Mat2D A, Mat2D v, double *lambda, double shift, bool norm_inf_v)
{
    /* https://www.youtube.com/watch?v=SkPusgctgpI */

    MAT2D_ASSERT(A.cols == A.rows);
    MAT2D_ASSERT(v.cols == 1);
    MAT2D_ASSERT(v.rows == A.rows);
    MAT2D_ASSERT(mat2D_calc_norma_inf(v) > 0);

    Mat2D current_v = mat2D_alloc(v.rows, v.cols);
    Mat2D temp_v = mat2D_alloc(v.rows, v.cols);
    Mat2D B = mat2D_alloc(A.rows, A.cols);
    mat2D_copy(B, A);
    mat2D_shift(B, shift * -1.0);

    double temp_lambda = 0;
    double diff = 0;

    /* Rayleigh quotient */
    mat2D_dot(temp_v, B, v);
    temp_lambda = mat2D_dot_product(temp_v, v) / (mat2D_calc_norma(v) * mat2D_calc_norma(v));
    int i = 0;
    for (i = 0; i < MAT2D_MAX_POWER_ITERATION; i++) {
        mat2D_copy(current_v, v);
        mat2D_dot(v, B, current_v);
        mat2D_normalize(v);
        mat2D_mult(v, temp_lambda > 0 ? 1 : -1);
        // mat2D_mult(v, fabs(lambda) / lambda);
        mat2D_dot(temp_v, B, v);
        temp_lambda = mat2D_dot_product(temp_v, v);

        mat2D_sub(current_v, v);
        diff = mat2D_calc_norma_inf(current_v);
        if (diff < MAT2D_EPS) {
            break;
        }
    }

    mat2D_free(current_v);
    mat2D_free(temp_v);
    mat2D_free(B);

    if (norm_inf_v) mat2D_normalize_inf(v);
    if (lambda) *lambda = temp_lambda + shift;

    if (diff > MAT2D_EPS) {
        return 1; /* eigenvector alternating between two options */
    } else {
        return 0;
    }
}

void mat2D_print(Mat2D m, const char *name, size_t padding)
{
    printf("%*s%s = [\n", (int) padding, "", name);
    for (size_t i = 0; i < m.rows; ++i) {
        printf("%*s    ", (int) padding, "");
        for (size_t j = 0; j < m.cols; ++j) {
            printf("%9.6f ", MAT2D_AT(m, i, j));
        }
        printf("\n");
    }
    printf("%*s]\n", (int) padding, "");
}

void mat2D_print_as_col(Mat2D m, const char *name, size_t padding)
{
    printf("%*s%s = [\n", (int) padding, "", name);
    for (size_t i = 0; i < m.rows*m.cols; ++i) {
            printf("%*s    ", (int) padding, "");
            printf("%f\n", m.elements[i]);
    }
    printf("%*s]\n", (int) padding, "");
}

void mat2D_rand(Mat2D m, double low, double high)
{
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) = mat2D_rand_double()*(high - low) + low;
        }
    }
}

double mat2D_rand_double(void)
```

```
{
    return (double) rand() / (double) RAND_MAX;
}

size_t mat2D_reduce(Mat2D m)
{
    /* preforming Gauss-Jordan reduction to Reduced Row Echelon Form (RREF) */
    /* Gauss elimination: https://en.wikipedia.org/wiki/Gaussian_elimination */

    mat2D_upper_triangulate(m, MAT2D_ONES_ON_DIAG | MAT2D_ROW_SWAPPING);

    size_t rank = 0;

    for (int r = m.rows-1; r >= 0; r--) {
        size_t c = m.cols-1;
        if (!mat2D_find_first_non_zero_value(m, r, &c)) {
            continue; /* row of zeros */
        }
        for (int i = 0; i < r; i++) {
            double factor = MAT2D_AT(m, i, c);
            mat2D_sub_row_time_factor_to_row(m, i, r, factor);
        }
        rank++;
    }

    return rank;
}

bool mat2D_row_is_all_digit(Mat2D m, double digit, size_t r)
{
    for (size_t j = 0; j < m.cols; ++j) {
        if (MAT2D_AT(m, r, j) != digit) {
            return false;
        }
    }
    return true;
}

void mat2D_set_DCM_zyx(Mat2D DCM, float yaw_deg, float pitch_deg, float roll_deg)
{
    Mat2D RotZ = mat2D_alloc(3,3);
    mat2D_set_rot_mat_z(RotZ, yaw_deg);
    Mat2D RotY = mat2D_alloc(3,3);
    mat2D_set_rot_mat_y(RotY, pitch_deg);
    Mat2D RotX = mat2D_alloc(3,3);
    mat2D_set_rot_mat_x(RotX, roll_deg);
    Mat2D temp = mat2D_alloc(3,3);

    mat2D_dot(temp, RotY, RotZ);
    mat2D_dot(DCM, RotX, temp); /* I have a DCM */

    mat2D_free(RotZ);
    mat2D_free(RotY);
    mat2D_free(RotX);
    mat2D_free(temp);
}

void mat2D_set_identity(Mat2D m)
{
    MAT2D_ASSERT(m.cols == m.rows);
    for (size_t i = 0; i < m.rows; ++i) {
        for (size_t j = 0; j < m.cols; ++j) {
            MAT2D_AT(m, i, j) = i == j ? 1 : 0;
            // if (i == j) {
            //     MAT2D_AT(m, i, j) = 1;
            // }
            // else {
            //     MAT2D_AT(m, i, j) = 0;
            // }
        }
    }
}

void mat2D_set_rot_mat_x(Mat2D m, float angle_deg)
{
    MAT2D_ASSERT(3 == m.cols && 3 == m.rows);

    float angle_rad = angle_deg * MAT2D_PI / 180;
    mat2D_set_identity(m);
    MAT2D_AT(m, 1, 1) =  cos(angle_rad);
    MAT2D_AT(m, 1, 2) =  sin(angle_rad);
    MAT2D_AT(m, 2, 1) = -sin(angle_rad);
    MAT2D_AT(m, 2, 2) =  cos(angle_rad);
}

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

```c
    MAT2D_ASSERT(3 == m.cols && 3 == m.rows);

    float angle_rad = angle_deg * MAT2D_PI / 180;
    mat2D_set_identity(m);
    MAT2D_AT(m, 0, 0) =  cos(angle_rad);
    MAT2D_AT(m, 0, 2) = -sin(angle_rad);
    MAT2D_AT(m, 2, 0) =  sin(angle_rad);
    MAT2D_AT(m, 2, 2) =  cos(angle_rad);
}

void mat2D_set_rot_mat_z(Mat2D m, float angle_deg)
{
    MAT2D_ASSERT(3 == m.cols && 3 == m.rows);

    float angle_rad = angle_deg * MAT2D_PI / 180;
    mat2D_set_identity(m);
    MAT2D_AT(m, 0, 0) =  cos(angle_rad);
    MAT2D_AT(m, 0, 1) =  sin(angle_rad);
    MAT2D_AT(m, 1, 0) = -sin(angle_rad);
    MAT2D_AT(m, 1, 1) =  cos(angle_rad);
}

void mat2D_shift(Mat2D m, double shift)
{
    MAT2D_ASSERT(m.cols == m.rows);
    for (size_t i = 0; i < m.rows; i++) {
        MAT2D_AT(m, i, i) += shift;
    }
}

void mat2D_solve_linear_sys_LUP_decomposition(Mat2D A, Mat2D x, Mat2D B)
{
    MAT2D_ASSERT(A.cols == x.rows);
    MAT2D_ASSERT(1 == x.cols);
    MAT2D_ASSERT(A.rows == B.rows);
    MAT2D_ASSERT(1 == B.cols);

    Mat2D y     = mat2D_alloc(x.rows, x.cols);
    Mat2D l     = mat2D_alloc(A.rows, A.cols);
    Mat2D p     = mat2D_alloc(A.rows, A.cols);
    Mat2D u     = mat2D_alloc(A.rows, A.cols);
    Mat2D inv_l = mat2D_alloc(l.rows, l.cols);
    Mat2D inv_u = mat2D_alloc(u.rows, u.cols);

    mat2D_LUP_decomposition_with_swap(A, l, p, u);

    mat2D_invert(inv_l, l);
    mat2D_invert(inv_u, u);

    mat2D_fill(x, 0);   /* x here is only a temp mat*/
    mat2D_fill(y, 0);
    mat2D_dot(x, p, B);
    mat2D_dot(y, inv_l, x);

    mat2D_fill(x, 0);
    mat2D_dot(x, inv_u, y);

    mat2D_free(y);
    mat2D_free(l);
    mat2D_free(p);
    mat2D_free(u);
    mat2D_free(inv_l);
    mat2D_free(inv_u);
}

void mat2D_sub(Mat2D dst, Mat2D a)
{
    MAT2D_ASSERT(dst.rows == a.rows);
    MAT2D_ASSERT(dst.cols == a.cols);
    for (size_t i = 0; i < dst.rows; ++i) {
        for (size_t j = 0; j < dst.cols; ++j) {
            MAT2D_AT(dst, i, j) -= MAT2D_AT(a, i, j);
        }
    }
}

void mat2D_sub_col_to_col(Mat2D des, size_t des_col, Mat2D src, size_t src_col)
{
    MAT2D_ASSERT(src_col < src.cols);
    MAT2D_ASSERT(des.rows == src.rows);
    MAT2D_ASSERT(des_col < des.cols);

    for (size_t i = 0; i < des.rows; i++) {
        MAT2D_AT(des, i, des_col) -= MAT2D_AT(src, i, src_col);
    }
}

```

```
void mat2D_sub_row_to_row(Mat2D des, size_t des_row, Mat2D src, size_t src_row)
{
    MAT2D_ASSERT(src_row < src.rows);
    MAT2D_ASSERT(des.cols == src.cols);
    MAT2D_ASSERT(des_row < des.rows);

    for (size_t j = 0; j < des.cols; j++) {
        MAT2D_AT(des, des_row, j) -= MAT2D_AT(src, src_row, j);
    }
}

void mat2D_sub_row_time_factor_to_row(Mat2D m, size_t des_r, size_t src_r, double factor)
{
    for (size_t j = 0; j < m.cols; ++j) {
        MAT2D_AT(m, des_r, j) -= factor * MAT2D_AT(m, src_r, j);
    }
}

void mat2D_SVD_full(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v, bool return_v_transpose)
{
    mat2D_SVD_thin(A, U, S, V, init_vec_u, init_vec_v, false);

    Mat2D U_full = mat2D_alloc(U.rows, U.cols);
    Mat2D V_full = mat2D_alloc(V.rows, V.cols);

    mat2D_make_orthogonal_modified_Gram_Schmidt(U_full, U);
    mat2D_make_orthogonal_modified_Gram_Schmidt(V_full, V);

    mat2D_copy(U, U_full);
    if (return_v_transpose) {
        mat2D_transpose(V, V_full);
    } else {
        mat2D_copy(V, V_full);
    }

    mat2D_free(U_full);
    mat2D_free(V_full);
}

void mat2D_SVD_thin(Mat2D A, Mat2D U, Mat2D S, Mat2D V, Mat2D init_vec_u, Mat2D init_vec_v, bool return_v_transpose)
{
    /* https://www.youtube.com/watch?v=nbBvuuNVfco */
    /* https://en.wikipedia.org/wiki/Singular_value_decomposition */
    size_t n = A.rows;
    size_t m = A.cols;
    MAT2D_ASSERT(U.rows == n);
    MAT2D_ASSERT(U.cols == n);
    MAT2D_ASSERT(S.rows == n);
    MAT2D_ASSERT(S.cols == m);
    MAT2D_ASSERT(V.rows == m);
    MAT2D_ASSERT(V.cols == m);
    MAT2D_ASSERT(init_vec_u.rows == n);
    MAT2D_ASSERT(init_vec_u.cols == 1);
    MAT2D_ASSERT(init_vec_v.rows == m);
    MAT2D_ASSERT(init_vec_v.cols == 1);

    mat2D_fill(U, 0);
    mat2D_fill(S, 0);
    mat2D_fill(V, 0);

    Mat2D AT = mat2D_alloc(m, n);
    mat2D_transpose(AT, A);

    if (n <= m) {
        Mat2D AAT = mat2D_alloc(n, n);
        Mat2D left_eigenvalues = mat2D_alloc(n, n);
        Mat2D left_eigenvectors = mat2D_alloc(n, n);
        Mat2D temp_u_vec = mat2D_alloc(n, 1);
        Mat2D temp_v_vec = mat2D_alloc(m, 1);
        mat2D_dot(AAT, A, AT);
        mat2D_eig_power_iteration(AAT, left_eigenvalues, left_eigenvectors, init_vec_u, 0);
        /* fill matrix sigma (S) */
        size_t non_zero_n = 0;
        for (size_t i = 0; i < n; i++) {
            if (MAT2D_IS_ZERO(MAT2D_AT(left_eigenvalues, i, i)) || MAT2D_AT(left_eigenvalues, i, i) < 0) {
                MAT2D_AT(S, i, i) = 0; /* AAT is positive definet */
            } else {
                MAT2D_AT(S, i, i) = sqrt(MAT2D_AT(left_eigenvalues, i, i));
                non_zero_n++;
            }
        }
        for (size_t c = 0; c < non_zero_n; c++) {
            mat2D_copy_col_from_src_to_des(U, c, left_eigenvectors, c);
            mat2D_copy_col_from_src_to_des(temp_u_vec, 0, left_eigenvectors, c);
```

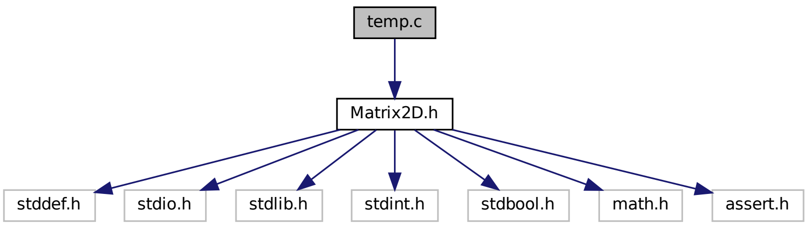
```
            mat2D_dot(temp_v_vec, AT, temp_u_vec);
            mat2D_mult(temp_v_vec, 1.0 / MAT2D_AT(S, c, c));
            mat2D_copy_col_from_src_to_des(V, c, temp_v_vec, 0);
        }
        mat2D_free(AAT);
        mat2D_free(left_eigenvalues);
        mat2D_free(left_eigenvectors);
        mat2D_free(temp_u_vec);
        mat2D_free(temp_v_vec);
    } else {
        Mat2D ATA = mat2D_alloc(m, m);
        Mat2D right_eigenvalues = mat2D_alloc(m, m);
        Mat2D right_eigenvectors = mat2D_alloc(m, m);
        Mat2D temp_u_vec = mat2D_alloc(n, 1);
        Mat2D temp_v_vec = mat2D_alloc(m, 1);
        mat2D_dot(ATA, AT, A);
        mat2D_eig_power_iteration(ATA, right_eigenvalues, right_eigenvectors, init_vec_v, 0);
        /* fill matrix sigma (S) */
        size_t non_zero_m = 0;
        for (size_t i = 0; i < m; i++) {
            if (MAT2D_IS_ZERO(MAT2D_AT(right_eigenvalues, i, i)) || MAT2D_AT(right_eigenvalues, i, i)
    < 0) {
                MAT2D_AT(S, i, i) = 0; /* ATA is positive definet */
            } else {
                MAT2D_AT(S, i, i) = sqrt(MAT2D_AT(right_eigenvalues, i, i));
                non_zero_m++;
            }
        }
        for (size_t c = 0; c < non_zero_m; c++) {
            mat2D_copy_col_from_src_to_des(V, c, right_eigenvectors, c);
            mat2D_copy_col_from_src_to_des(temp_v_vec, 0, right_eigenvectors, c);
            mat2D_dot(temp_u_vec, A, temp_v_vec);
            mat2D_mult(temp_u_vec, 1.0 / MAT2D_AT(S, c, c));
            mat2D_copy_col_from_src_to_des(U, c, temp_u_vec, 0);
        }
        mat2D_free(ATA);
        mat2D_free(right_eigenvalues);
        mat2D_free(right_eigenvectors);
        mat2D_free(temp_u_vec);
        mat2D_free(temp_v_vec);
    }

    if (return_v_transpose) {
        Mat2D v_trans = mat2D_alloc(V.cols, V.rows);
        mat2D_transpose(v_trans, V);
        mat2D_copy(V, v_trans);

        mat2D_free(v_trans);
    }

    mat2D_free(AT);
}

void mat2D_swap_rows(Mat2D m, size_t r1, size_t r2)
{
    for (size_t j = 0; j < m.cols; j++) {
        double temp = MAT2D_AT(m, r1, j);
        MAT2D_AT(m, r1, j) = MAT2D_AT(m, r2, j);
        MAT2D_AT(m, r2, j) = temp;
    }
}

void mat2D_transpose(Mat2D des, Mat2D src)
{
    MAT2D_ASSERT(des.cols == src.rows);
    MAT2D_ASSERT(des.rows == src.cols);

    for (size_t index = 0; index < des.rows; ++index) {
        for (size_t jndex = 0; jndex < des.cols; ++jndex) {
            MAT2D_AT(des, index, jndex) = MAT2D_AT(src, jndex, index);
        }
    }
}

double mat2D_upper_triangulate(Mat2D m, uint8_t flags)
{
    /* preforming Gauss elimination: https://en.wikipedia.org/wiki/Gaussian_elimination */
    /* returns the factor multiplying the determinant */

    double factor_to_return = 1;

    size_t r = 0;
    for (size_t c = 0; c < m.cols && r < m.rows; c++) {
        if (flags & MAT2D_ROW_SWAPPING) {
            /* finding biggest first number (absolute value); partial pivoting */
            size_t piv = r;
            double best = fabs(MAT2D_AT(m, r, c));
```

```
            for (size_t i = r + 1; i < m.rows; i++) {
                double v = fabs(MAT2D_AT(m, i, c));
                if (v > best) {
                    best = v;
                    piv = i;
                }
            }
            if (MAT2D_IS_ZERO(best)) {
                continue; /* move to next column, same pivot row r */
            }
            if (piv != r) {
                mat2D_swap_rows(m, piv, r);
                factor_to_return *= -1.0;
            }
        }

        double pivot = MAT2D_AT(m, r, c);
        MAT2D_ASSERT(!MAT2D_IS_ZERO(pivot));

        if (flags & MAT2D_ONES_ON_DIAG) {
            mat2D_mult_row(m, r, 1.0 / pivot);
            factor_to_return *= pivot;
            pivot = 1.0;
        }

        /* Eliminate entries below pivot in column c */
        for (size_t i = r + 1; i < m.rows; i++) {
            double f = MAT2D_AT(m, i, c) / pivot;
            mat2D_sub_row_time_factor_to_row(m, i, r, f);
        }
        r++;
    }
    return factor_to_return;
}

#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

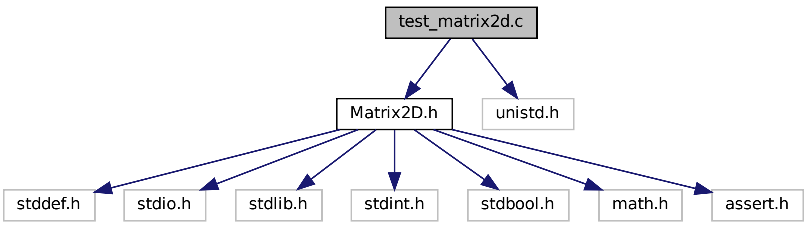
```
00001 #define MATRIX2D_IMPLEMENTATION
00002 #include "Matrix2D.h"
00003
00004 int main(void)
00005 {
00006     int n = 4;
00007     int m = 5;
00008
00009     Mat2D A = mat2D_alloc(n, m);
00010     Mat2D U = mat2D_alloc(n, n);
00011     Mat2D S = mat2D_alloc(n, m);
00012     Mat2D VT = mat2D_alloc(m, 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
```

```c
    MAT2D_PRINT(USVT);

    mat2D_free(A);
    mat2D_free(U);
    mat2D_free(S);
    mat2D_free(VT);
    mat2D_free(SV);
    mat2D_free(USVT);
    mat2D_free(init_vec_u);
    mat2D_free(init_vec_v);

    return 0;
}
```

4.7 test_matrix2d.c File Reference

```c
#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_triangulate_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", \
          #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_sw 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_agre 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_orthonor 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

```
#define MATRIX2D_IMPLEMENTATION
#include "Matrix2D.h"
#include <unistd.h> /* dup/dup2/close for silencing stdout in print-smoke tests */

/* ---------------------------------------------------------------------------- */
/* Deterministic fuzz helpers (no libc rand(); stable across platforms)         */
/* ---------------------------------------------------------------------------- */

static uint64_t xorshift64star(uint64_t *state)
{
    /* Marsaglia xorshift* variant */
    uint64_t x = *state;
    x ^= x » 12;
    x ^= x « 25;
    x ^= x » 27;
    *state = x;
    return x * 2685821657736338717ULL;
}

static double rng_unit01(uint64_t *state)
{
    /* Convert top 53 bits to a double in [0, 1). */
    uint64_t x = xorshift64star(state);
    uint64_t top53 = x » 11;
    return (double)top53 * (1.0 / 9007199254740992.0); /* 2^53 */
}

static double rng_range(uint64_t *state, double low, double high)
{
    return low + (high - low) * rng_unit01(state);
}

static int close_rel_abs(double a, double b, double abs_eps, double rel_eps)
{
    double diff = fabs(a - b);
    if (diff <= abs_eps) return 1;
    return diff <= rel_eps * fmax(fabs(a), fabs(b));
}

static void fill_strictly_diag_dominant(Mat2D a, uint64_t *rng)
{
    /* Strict diagonal dominance => nonsingular (and usually stable to invert). */
    MAT2D_ASSERT(a.rows == a.cols);
    double mag = 2.0;

    for (size_t i = 0; i < a.rows; i++) {
        double row_sum = 0.0;
        for (size_t j = 0; j < a.cols; j++) {
            if (i == j) continue;
            double v = rng_range(rng, -mag, mag);
            MAT2D_AT(a, i, j) = v;
            row_sum += fabs(v);
        }
        MAT2D_AT(a, i, i) = row_sum + 1.0;
    }
}


static double det_by_minors_first_col(Mat2D a);

static int nearly_equal(double a, double b, double eps)
{
    return fabs(a - b) <= eps;
}

static void assert_mat_close(Mat2D a, Mat2D b, double eps)
{
    MAT2D_ASSERT(a.rows == b.rows);
    MAT2D_ASSERT(a.cols == b.cols);

    for (size_t i = 0; i < a.rows; i++) {
        for (size_t j = 0; j < a.cols; j++) {
            double va = MAT2D_AT(a, i, j);
            double vb = MAT2D_AT(b, i, j);
            if (!nearly_equal(va, vb, eps)) {
                fprintf(stderr,
                        "Matrix mismatch at (%zu,%zu): a=%g b=%g\n",
                        i,
                        j,
                        va,
                        vb);
                MAT2D_ASSERT(0);
            }
        }
```

```
    }
}

static void assert_identity_close(Mat2D m, double eps)
{
    MAT2D_ASSERT(m.rows == m.cols);
    for (size_t i = 0; i < m.rows; i++) {
        for (size_t j = 0; j < m.cols; j++) {
            double expected = (i == j) ? 1.0 : 0.0;
            MAT2D_ASSERT(nearly_equal(MAT2D_AT(m, i, j), expected, eps));
        }
    }
}

static void fill_mat_from_array(Mat2D m, const double *data)
{
    for (size_t i = 0; i < m.rows; i++) {
        for (size_t j = 0; j < m.cols; j++) {
            MAT2D_AT(m, i, j) = data[i * m.cols + j];
        }
    }
}

static void assert_permutation_matrix(Mat2D p)
{
    MAT2D_ASSERT(p.rows == p.cols);

    for (size_t i = 0; i < p.rows; i++) {
        double row_sum = 0.0;
        for (size_t j = 0; j < p.cols; j++) {
            double v = MAT2D_AT(p, i, j);
            MAT2D_ASSERT(v == 0.0 || v == 1.0);
            row_sum += v;
        }
        MAT2D_ASSERT(nearly_equal(row_sum, 1.0, 0.0));
    }

    for (size_t j = 0; j < p.cols; j++) {
        double col_sum = 0.0;
        for (size_t i = 0; i < p.rows; i++) {
            col_sum += MAT2D_AT(p, i, j);
        }
        MAT2D_ASSERT(nearly_equal(col_sum, 1.0, 0.0));
    }

    /* det(P) must be ±1 for a permutation matrix */
    double dp = mat2D_det(p);
    MAT2D_ASSERT(nearly_equal(fabs(dp), 1.0, MAT2D_EPS));
}

static void assert_inverse_identity_both_sides(Mat2D a, double eps)
{
    MAT2D_ASSERT(a.rows == a.cols);

    Mat2D inv = mat2D_alloc(a.rows, a.cols);
    Mat2D prod1 = mat2D_alloc(a.rows, a.cols);
    Mat2D prod2 = mat2D_alloc(a.rows, a.cols);

    mat2D_invert(inv, a);

    /* A * inv(A) == I */
    mat2D_dot(prod1, a, inv);
    assert_identity_close(prod1, eps);

    /* inv(A) * A == I (catches some subtle multiply/invert issues) */
    mat2D_dot(prod2, inv, a);
    assert_identity_close(prod2, eps);

    mat2D_free(inv);
    mat2D_free(prod1);
    mat2D_free(prod2);
}

static void test_alloc_fill_copy_add_sub(void)
{
    Mat2D a = mat2D_alloc(2, 3);
    Mat2D b = mat2D_alloc(2, 3);
    Mat2D c = mat2D_alloc(2, 3);

    mat2D_fill(a, 1.5);
    mat2D_fill(b, 2.0);

    mat2D_copy(c, a);
    mat2D_add(c, b); // c = 3.5
    for (size_t i = 0; i < c.rows; i++) {
        for (size_t j = 0; j < c.cols; j++) {
            MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, i, j), 3.5, 0.0));
```

```
        }
    }

    mat2D_sub(c, b); // c back to 1.5
    assert_mat_close(c, a, 0.0);

    mat2D_free(a);
    mat2D_free(b);
    mat2D_free(c);
}

static void test_transpose(void)
{
    Mat2D a = mat2D_alloc(2, 3);
    Mat2D t = mat2D_alloc(3, 2);

    // a =
    // [1 2 3
    //  4 5 6]
    double v = 1.0;
    for (size_t i = 0; i < a.rows; i++) {
        for (size_t j = 0; j < a.cols; j++) {
            MAT2D_AT(a, i, j) = v++;
        }
    }

    mat2D_transpose(t, a);

    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 0, 0), 1.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 1, 0), 2.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 2, 0), 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 0, 1), 4.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 1, 1), 5.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 2, 1), 6.0, 0.0));

    mat2D_free(a);
    mat2D_free(t);
}

static void test_dot_product_matrix_multiply(void)
{
    Mat2D a = mat2D_alloc(2, 3);
    Mat2D b = mat2D_alloc(3, 2);
    Mat2D c = mat2D_alloc(2, 2);

    // a =
    // [1 2 3
    //  4 5 6]
    MAT2D_AT(a, 0, 0) = 1;
    MAT2D_AT(a, 0, 1) = 2;
    MAT2D_AT(a, 0, 2) = 3;
    MAT2D_AT(a, 1, 0) = 4;
    MAT2D_AT(a, 1, 1) = 5;
    MAT2D_AT(a, 1, 2) = 6;

    // b =
    // [ 7  8
    //   9 10
    //  11 12]
    MAT2D_AT(b, 0, 0) = 7;
    MAT2D_AT(b, 0, 1) = 8;
    MAT2D_AT(b, 1, 0) = 9;
    MAT2D_AT(b, 1, 1) = 10;
    MAT2D_AT(b, 2, 0) = 11;
    MAT2D_AT(b, 2, 1) = 12;

    mat2D_dot(c, a, b);

    // Expected:
    // [ 58  64
    //  139 154]
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 0, 0), 58.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 0, 1), 64.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 1, 0), 139.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 1, 1), 154.0, 0.0));

    mat2D_free(a);
    mat2D_free(b);
    mat2D_free(c);
}

static void test_det_and_minor_det_agree_3x3(void)
{
    Mat2D a = mat2D_alloc(3, 3);

    // Classic example det = -306
    // [  6   1   1
```

```
    //     4  -2   5
    //     2   8   7 ]
    MAT2D_AT(a, 0, 0) = 6;
    MAT2D_AT(a, 0, 1) = 1;
    MAT2D_AT(a, 0, 2) = 1;
    MAT2D_AT(a, 1, 0) = 4;
    MAT2D_AT(a, 1, 1) = -2;
    MAT2D_AT(a, 1, 2) = 5;
    MAT2D_AT(a, 2, 0) = 2;
    MAT2D_AT(a, 2, 1) = 8;
    MAT2D_AT(a, 2, 2) = 7;

    double det_gauss = mat2D_det(a);
    double det_minor = det_by_minors_first_col(a);

    MAT2D_ASSERT(nearly_equal(det_gauss, -306.0, MAT2D_EPS));
    MAT2D_ASSERT(nearly_equal(det_minor, -306.0, MAT2D_EPS));
    MAT2D_ASSERT(nearly_equal(det_minor, det_gauss, MAT2D_EPS));

    mat2D_free(a);
}

static void test_invert(void)
{
    /*
     * Adversarial inversion set:
     *  - a "normal" matrix
     *  - a matrix with a zero leading pivot (forces pivoting / row swaps)
     *  - an ill-conditioned-ish matrix (still invertible) to stress Gauss-Jordan
     */
    struct {
        const char *name;
        double eps;
        double data[9];
    } cases[] = {
        {
            "baseline det=3",
            MAT2D_EPS,
            {4, 7, 2, 3, 6, 1, 2, 5, 1},
        },
        {
            "forces row-swap pivoting (zero leading pivot), det=2",
            MAT2D_EPS,
            {0, 1, 1, 1, 0, 1, 1, 1, 0},
        },
        {
            "hilbert-3x3 (ill-conditioned-ish)",
            1e-10,
            {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 / 5.0},
        },
    };

    for (size_t k = 0; k < sizeof(cases) / sizeof(cases[0]); k++) {
        Mat2D a = mat2D_alloc(3, 3);
        fill_mat_from_array(a, cases[k].data);

        /* basic sanity: det should be finite and not ~0 for these cases */
        double d = mat2D_det(a);
        MAT2D_ASSERT(isfinite(d));
        MAT2D_ASSERT(fabs(d) > MAT2D_EPS);

        assert_inverse_identity_both_sides(a, cases[k].eps);
        mat2D_free(a);
    }
}

static void test_LUP_decomposition_identity_P_no_swap_case(void)
{
    const double eps = MAT2D_EPS;

    Mat2D a = mat2D_alloc(3, 3);
    Mat2D l = mat2D_alloc(3, 3);
    Mat2D p = mat2D_alloc(3, 3);
    Mat2D u = mat2D_alloc(3, 3);

    Mat2D pa = mat2D_alloc(3, 3);
    Mat2D lu = mat2D_alloc(3, 3);

    // Same matrix as invert test (should not need swaps in typical flow)
    MAT2D_AT(a, 0, 0) = 4;
    MAT2D_AT(a, 0, 1) = 7;
    MAT2D_AT(a, 0, 2) = 2;
    MAT2D_AT(a, 1, 0) = 3;
    MAT2D_AT(a, 1, 1) = 6;
    MAT2D_AT(a, 1, 2) = 1;
    MAT2D_AT(a, 2, 0) = 2;
```

```
    MAT2D_AT(a, 2, 1) = 5;
    MAT2D_AT(a, 2, 2) = 1;

    mat2D_LUP_decomposition_with_swap(a, l, p, u);

    mat2D_dot(pa, p, a);
    mat2D_dot(lu, l, u);

    assert_mat_close(pa, lu, eps);

    mat2D_free(a);
    mat2D_free(l);
    mat2D_free(p);
    mat2D_free(u);
    mat2D_free(pa);
    mat2D_free(lu);
}

static void test_LUP_decomposition_swap_required_case(void)
{
    const double eps = MAT2D_EPS;

    Mat2D a = mat2D_alloc(3, 3);
    Mat2D l = mat2D_alloc(3, 3);
    Mat2D p = mat2D_alloc(3, 3);
    Mat2D u = mat2D_alloc(3, 3);

    Mat2D pa = mat2D_alloc(3, 3);
    Mat2D lu = mat2D_alloc(3, 3);

    /*
     * This matrix is invertible but has a guaranteed zero pivot at (0,0),
     * so LUP must swap rows to proceed.
     * A =
     * [0 1 1
     *  1 0 1
     *  1 1 0]
     */
    double data[9] = {0, 1, 1, 1, 0, 1, 1, 1, 0};
    fill_mat_from_array(a, data);

    mat2D_LUP_decomposition_with_swap(a, l, p, u);
    assert_permutation_matrix(p);

    mat2D_dot(pa, p, a);
    mat2D_dot(lu, l, u);
    assert_mat_close(pa, lu, eps);

    mat2D_free(a);
    mat2D_free(l);
    mat2D_free(p);
    mat2D_free(u);
    mat2D_free(pa);
    mat2D_free(lu);
}

static void test_uint32_alloc_fill_and_at(void)
{
    Mat2D_uint32 m = mat2D_alloc_uint32(2, 3);
    mat2D_fill_uint32(m, 42);
    MAT2D_AT_UINT32(m, 0, 1) = 7;

    MAT2D_ASSERT(MAT2D_AT_UINT32(m, 0, 0) == 42);
    MAT2D_ASSERT(MAT2D_AT_UINT32(m, 0, 1) == 7);
    MAT2D_ASSERT(MAT2D_AT_UINT32(m, 1, 2) == 42);

    mat2D_free_uint32(m);
}

static void test_offset2d_and_stride(void)
{
    Mat2D a = mat2D_alloc(3, 4);
    mat2D_fill_sequence(a, 0.0, 1.0); // a[i,j] = offset(i,j)

    MAT2D_ASSERT(mat2D_offset2d(a, 0, 0) == 0);
    MAT2D_ASSERT(mat2D_offset2d(a, 0, 3) == 3);
    MAT2D_ASSERT(mat2D_offset2d(a, 1, 0) == 4);
    MAT2D_ASSERT(mat2D_offset2d(a, 2, 3) == 11);

    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 3), 11.0, 0.0));

    mat2D_free(a);
}

static void test_copy_windows(void)
{
    Mat2D des = mat2D_alloc(4, 5);
```

```
    Mat2D src = mat2D_alloc(2, 2);
    Mat2D got = mat2D_alloc(2, 2);

    mat2D_fill(des, -1.0);
    MAT2D_AT(src, 0, 0) = 1.0;
    MAT2D_AT(src, 0, 1) = 2.0;
    MAT2D_AT(src, 1, 0) = 3.0;
    MAT2D_AT(src, 1, 1) = 4.0;

    // Copy src into des window rows [1..2], cols [2..3]
    mat2D_copy_src_to_des_window(des, src, 1, 2, 2, 3);

    // Check placement and untouched area
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 0, 0), -1.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 2), 1.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 3), 2.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 2, 2), 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 2, 3), 4.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 3, 4), -1.0, 0.0));

    // Copy the same window back out into got and compare to src
    mat2D_copy_src_window_to_des(got, des, 1, 2, 2, 3);
    assert_mat_close(got, src, 0.0);

    mat2D_free(des);
    mat2D_free(src);
    mat2D_free(got);
}

static void test_non_contiguous_stride_views(void)
{
    /*
     * Many bugs only show up when stride_r != cols.
     * We build a padded (non-contiguous) view and make sure core ops use stride correctly.
     */
    const size_t rows = 2;
    const size_t cols = 2;
    const size_t stride = 7; /* intentional padding */

    double *buf = (double *)MAT2D_MALLOC(sizeof(double) * rows * stride);
    MAT2D_ASSERT(buf != NULL);

    /* poison padding so accidental reads are obvious */
    for (size_t i = 0; i < rows * stride; i++) {
        buf[i] = 1234567.0;
    }

    Mat2D a = {.rows = rows, .cols = cols, .stride_r = stride, .elements = buf};

    MAT2D_AT(a, 0, 0) = 1.0;
    MAT2D_AT(a, 0, 1) = 2.0;
    MAT2D_AT(a, 1, 0) = 3.0;
    MAT2D_AT(a, 1, 1) = 4.0;

    MAT2D_ASSERT(mat2D_offset2d(a, 1, 1) == 1 * stride + 1);

    Mat2D c = mat2D_alloc(2, 2);
    Mat2D t = mat2D_alloc(2, 2);
    Mat2D prod = mat2D_alloc(2, 2);
    Mat2D id = mat2D_alloc(2, 2);

    mat2D_copy(c, a);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(c, 1, 1), 4.0, 0.0));

    mat2D_transpose(t, a);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 0, 1), 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(t, 1, 0), 2.0, 0.0));

    mat2D_set_identity(id);
    mat2D_dot(prod, a, id);
    assert_mat_close(prod, c, 0.0);

    mat2D_free(c);
    mat2D_free(t);
    mat2D_free(prod);
    mat2D_free(id);
    MAT2D_FREE(buf);
}

static void test_row_col_ops_and_scaling(void)
{
    Mat2D a = mat2D_alloc(3, 3);
    mat2D_fill_sequence(a, 1.0, 1.0);
    // a =
    // [1 2 3
    //  4 5 6
    //  7 8 9]
```

```c

    // add_row_to_row: row0 += row1 => [5 7 9]
    mat2D_add_row_to_row(a, 0, a, 1);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 0, 0), 5.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 0, 1), 7.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 0, 2), 9.0, 0.0));

    // sub_row_to_row: row2 -= row1 => [3 3 3]
    mat2D_sub_row_to_row(a, 2, a, 1);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 0), 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 1), 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 2), 3.0, 0.0));

    // add_col_to_col: col1 += col2 (operate on current a)
    mat2D_add_col_to_col(a, 1, a, 2);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 1), 5.0 + 6.0, 0.0));

    // sub_col_to_col: col1 -= col2 => restore original col1 for that row
    mat2D_sub_col_to_col(a, 1, a, 2);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 1), 5.0, 0.0));

    // Row op: row1 += 2 * row2  (row2 currently [3 3 3])
    mat2D_add_row_time_factor_to_row(a, 1, 2, 2.0);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 0), 4.0 + 2.0 * 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 1), 5.0 + 2.0 * 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 1, 2), 6.0 + 2.0 * 3.0, 0.0));

    // Row scale
    mat2D_mult_row(a, 2, -1.0);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 0), -3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 1), -3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, 2, 2), -3.0, 0.0));

    mat2D_free(a);
}

static void test_shift_and_identity(void)
{
    Mat2D a = mat2D_alloc(4, 4);
    mat2D_set_identity(a);
    mat2D_shift(a, 2.0); // diag becomes 3

    for (size_t i = 0; i < a.rows; i++) {
        for (size_t j = 0; j < a.cols; j++) {
            double expected = (i == j) ? 3.0 : 0.0;
            MAT2D_ASSERT(nearly_equal(MAT2D_AT(a, i, j), expected, 0.0));
        }
    }

    mat2D_free(a);
}

static void test_norms_and_normalize(void)
{
    const double eps = MAT2D_EPS;

    Mat2D v = mat2D_alloc(2, 1);
    MAT2D_AT(v, 0, 0) = 3.0;
    MAT2D_AT(v, 1, 0) = 4.0;

    MAT2D_ASSERT(nearly_equal(mat2D_calc_norma(v), 5.0, eps));
    MAT2D_ASSERT(nearly_equal(mat2D_calc_norma_inf(v), 4.0, eps));
    MAT2D_ASSERT(nearly_equal(mat2D_inner_product(v), 25.0, eps));

    mat2D_normalize(v);
    MAT2D_ASSERT(nearly_equal(mat2D_calc_norma(v), 1.0, MAT2D_EPS));

    mat2D_free(v);
}

static void test_outer_product_and_cross(void)
{
    const double eps = MAT2D_EPS;

    Mat2D v = mat2D_alloc(3, 1);
    Mat2D out = mat2D_alloc(3, 3);

    MAT2D_AT(v, 0, 0) = 1.0;
    MAT2D_AT(v, 1, 0) = 2.0;
    MAT2D_AT(v, 2, 0) = 3.0;

    mat2D_outer_product(out, v);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 0), 1.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 1), 2.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 2), 3.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 0), 2.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 1), 4.0, eps));
```

```
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 2), 6.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 2, 2), 9.0, eps));

    // Cross: i x j = k
    Mat2D i = mat2D_alloc(3, 1);
    Mat2D j = mat2D_alloc(3, 1);
    Mat2D k = mat2D_alloc(3, 1);
    mat2D_fill(i, 0.0);
    mat2D_fill(j, 0.0);
    mat2D_fill(k, 0.0);
    MAT2D_AT(i, 0, 0) = 1.0;
    MAT2D_AT(j, 1, 0) = 1.0;

    mat2D_cross(k, i, j);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(k, 0, 0), 0.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(k, 1, 0), 0.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(k, 2, 0), 1.0, eps));

    mat2D_free(v);
    mat2D_free(out);
    mat2D_free(i);
    mat2D_free(j);
    mat2D_free(k);
}

static void test_det_2x2_and_upper_triangulate_sign(void)
{
    Mat2D a = mat2D_alloc(2, 2);
    MAT2D_AT(a, 0, 0) = 0.0;
    MAT2D_AT(a, 0, 1) = 1.0;
    MAT2D_AT(a, 1, 0) = 2.0;
    MAT2D_AT(a, 1, 1) = 3.0;

    // det should be -2
    MAT2D_ASSERT(nearly_equal(mat2D_det_2x2_mat(a), -2.0, 0.0));
    MAT2D_ASSERT(nearly_equal(mat2D_det(a), -2.0, MAT2D_EPS));

    // upper triangulation should have exactly one row swap => factor = -1
    Mat2D tmp = mat2D_alloc(2, 2);
    mat2D_copy(tmp, a);
    double f = mat2D_upper_triangulate(tmp, MAT2D_ROW_SWAPPING);
    MAT2D_ASSERT(nearly_equal(f, -1.0, 0.0));

    mat2D_free(a);
    mat2D_free(tmp);
}

static double det_by_minors_first_col(Mat2D a)
{
    MAT2D_ASSERT(a.rows == a.cols);
    size_t n = a.rows;

    if (n == 1) {
        return MAT2D_AT(a, 0, 0);
    }
    if (n == 2) {
        return mat2D_det_2x2_mat(a);
    }

    double det = 0.0;
    size_t j = 0;
    for (size_t i = 0; i < n; i++) {
        double aij = MAT2D_AT(a, i, j);
        if (aij == 0.0) {
            continue;
        }

        Mat2D_Minor mm = mat2D_minor_alloc_fill_from_mat(a, i, j);
        double minor_det = 0.0;
        if (mm.rows == 2 && mm.cols == 2) {
            minor_det = mat2D_det_2x2_mat_minor(mm);
        } else {
            minor_det = mat2D_minor_det(mm);
        }
        mat2D_minor_free(mm);

        double sign = ((i + j) % 2 == 0) ? 1.0 : -1.0;
        det += aij * sign * minor_det;
    }

    return det;
}

static void test_minor_det_matches_gauss_4x4_known(void)
{
    const double eps = 1e-9;

```

```
    Mat2D a = mat2D_alloc(4, 4);
    // det = 72
    // [1 2 3 4
    //  5 6 7 8
    //  2 6 4 8
    //  3 1 1 2]
    double data[16] = {
        1, 2, 3, 4, //
        5, 6, 7, 8, //
        2, 6, 4, 8, //
        3, 1, 1, 2  //
    };
    for (size_t i = 0; i < 4; i++) {
        for (size_t j = 0; j < 4; j++) {
            MAT2D_AT(a, i, j) = data[i * 4 + j];
        }
    }

    double det_gauss = mat2D_det(a);
    double det_minor = det_by_minors_first_col(a);

    MAT2D_ASSERT(nearly_equal(det_gauss, 72.0, eps));
    MAT2D_ASSERT(nearly_equal(det_minor, 72.0, eps));
    MAT2D_ASSERT(nearly_equal(det_minor, det_gauss, eps));

    mat2D_free(a);
}

static void test_reduce_rank(void)
{
    Mat2D a = mat2D_alloc(3, 3);
    // rank 2:
    // [1 2 3
    //  2 4 6
    //  1 1 1]
    MAT2D_AT(a, 0, 0) = 1;
    MAT2D_AT(a, 0, 1) = 2;
    MAT2D_AT(a, 0, 2) = 3;
    MAT2D_AT(a, 1, 0) = 2;
    MAT2D_AT(a, 1, 1) = 4;
    MAT2D_AT(a, 1, 2) = 6;
    MAT2D_AT(a, 2, 0) = 1;
    MAT2D_AT(a, 2, 1) = 1;
    MAT2D_AT(a, 2, 2) = 1;

    size_t r = mat2D_reduce(a);
    MAT2D_ASSERT(r == 2);

    mat2D_free(a);
}

static void test_rotation_matrices_orthonormal(void)
{
    const double eps = 1e-7;

    Mat2D r = mat2D_alloc(3, 3);
    Mat2D rt = mat2D_alloc(3, 3);
    Mat2D prod = mat2D_alloc(3, 3);

    // Z rotation 90deg, per library's convention:
    // [ 0  1  0
    //  -1  0  0
    //   0  0  1 ]
    mat2D_set_rot_mat_z(r, 90.0f);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 0, 0), 0.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 0, 1), 1.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 1, 0), -1.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 1, 1), 0.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(r, 2, 2), 1.0, eps));

    mat2D_transpose(rt, r);
    mat2D_dot(prod, rt, r);
    assert_identity_close(prod, MAT2D_EPS);
    MAT2D_ASSERT(nearly_equal(mat2D_det(r), 1.0, MAT2D_EPS));

    // X rotation 90deg should also be orthonormal with det ~ 1
    mat2D_set_rot_mat_x(r, 90.0f);
    mat2D_transpose(rt, r);
    mat2D_dot(prod, rt, r);
    assert_identity_close(prod, MAT2D_EPS);
    MAT2D_ASSERT(nearly_equal(mat2D_det(r), 1.0, MAT2D_EPS));

    // Y rotation 90deg orthonormal with det ~ 1
    mat2D_set_rot_mat_y(r, 90.0f);
    mat2D_transpose(rt, r);
    mat2D_dot(prod, rt, r);
    assert_identity_close(prod, MAT2D_EPS);
```

```c
    MAT2D_ASSERT(nearly_equal(mat2D_det(r), 1.0, MAT2D_EPS));

    mat2D_free(r);
    mat2D_free(rt);
    mat2D_free(prod);
}

static void test_DCM_zyx_matches_product(void)
{
    const double eps = MAT2D_EPS;

    Mat2D dcm = mat2D_alloc(3, 3);
    Mat2D rz = mat2D_alloc(3, 3);
    Mat2D ry = mat2D_alloc(3, 3);
    Mat2D rx = mat2D_alloc(3, 3);
    Mat2D tmp = mat2D_alloc(3, 3);
    Mat2D expected = mat2D_alloc(3, 3);

    float yaw = 30.0f;
    float pitch = 20.0f;
    float roll = 10.0f;

    mat2D_set_DCM_zyx(dcm, yaw, pitch, roll);

    mat2D_set_rot_mat_z(rz, yaw);
    mat2D_set_rot_mat_y(ry, pitch);
    mat2D_set_rot_mat_x(rx, roll);
    mat2D_dot(tmp, ry, rz);
    mat2D_dot(expected, rx, tmp);

    assert_mat_close(dcm, expected, eps);

    mat2D_free(dcm);
    mat2D_free(rz);
    mat2D_free(ry);
    mat2D_free(rx);
    mat2D_free(tmp);
    mat2D_free(expected);
}

static void test_solve_linear_system_LUP(void)
{
    const double eps = MAT2D_EPS;

    Mat2D a = mat2D_alloc(3, 3);
    Mat2D x = mat2D_alloc(3, 1);
    Mat2D b = mat2D_alloc(3, 1);
    Mat2D ax = mat2D_alloc(3, 1);

    // A =
    // [3 0  2
    //  2 0 -2
    //  0 1  1]
    MAT2D_AT(a, 0, 0) = 3;
    MAT2D_AT(a, 0, 1) = 0;
    MAT2D_AT(a, 0, 2) = 2;
    MAT2D_AT(a, 1, 0) = 2;
    MAT2D_AT(a, 1, 1) = 0;
    MAT2D_AT(a, 1, 2) = -2;
    MAT2D_AT(a, 2, 0) = 0;
    MAT2D_AT(a, 2, 1) = 1;
    MAT2D_AT(a, 2, 2) = 1;

    // Choose x_true = [1,2,3]^T, so b = A*x_true = [9, -4, 5]^T
    MAT2D_AT(b, 0, 0) = 9;
    MAT2D_AT(b, 1, 0) = -4;
    MAT2D_AT(b, 2, 0) = 5;

    mat2D_solve_linear_sys_LUP_decomposition(a, x, b);
    mat2D_dot(ax, a, x);

    MAT2D_ASSERT(nearly_equal(MAT2D_AT(ax, 0, 0), 9.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(ax, 1, 0), -4.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(ax, 2, 0), 5.0, eps));

    mat2D_free(a);
    mat2D_free(x);
    mat2D_free(b);
    mat2D_free(ax);
}

static void test_rand_range(void)
{
    Mat2D a = mat2D_alloc(4, 4);
    srand(1);
    mat2D_rand(a, -2.0, 5.0);

```

```
    for (size_t i = 0; i < a.rows; i++) {
        for (size_t j = 0; j < a.cols; j++) {
            double v = MAT2D_AT(a, i, j);
            MAT2D_ASSERT(v >= -2.0);
            MAT2D_ASSERT(v <= 5.0);
        }
    }

    mat2D_free(a);
}

static void test_copy_row_and_col_helpers(void)
{
    Mat2D src = mat2D_alloc(3, 3);
    Mat2D des = mat2D_alloc(3, 3);

    /*
     * src =
     * [1 2 3
     *  4 5 6
     *  7 8 9]
     */
    mat2D_fill_sequence(src, 1.0, 1.0);
    mat2D_fill(des, 0.0);

    /* copy row 1 => des[1,:] = src[1,:] */
    mat2D_copy_row_from_src_to_des(des, 1, src, 1);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 0), 4.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 1), 5.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 2), 6.0, 0.0));

    /* copy col 2 => des[:,2] = src[:,2] */
    mat2D_copy_col_from_src_to_des(des, 2, src, 2);
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 0, 2), 3.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 1, 2), 6.0, 0.0));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(des, 2, 2), 9.0, 0.0));

    mat2D_free(src);
    mat2D_free(des);
}

static void test_dot_product_and_vector_variants(void)
{
    const double eps = 1e-12;

    /* Column vectors */
    Mat2D a = mat2D_alloc(3, 1);
    Mat2D b = mat2D_alloc(3, 1);
    MAT2D_AT(a, 0, 0) = 1.0;
    MAT2D_AT(a, 1, 0) = 2.0;
    MAT2D_AT(a, 2, 0) = 3.0;
    MAT2D_AT(b, 0, 0) = 4.0;
    MAT2D_AT(b, 1, 0) = 5.0;
    MAT2D_AT(b, 2, 0) = 6.0;

    MAT2D_ASSERT(nearly_equal(mat2D_dot_product(a, b), 32.0, eps));
    MAT2D_ASSERT(nearly_equal(mat2D_inner_product(a), 14.0, eps));

    /* Row vectors (exercise the other branch in dot_product/inner_product) */
    Mat2D ar = mat2D_alloc(1, 3);
    Mat2D br = mat2D_alloc(1, 3);
    MAT2D_AT(ar, 0, 0) = 1.0;
    MAT2D_AT(ar, 0, 1) = 2.0;
    MAT2D_AT(ar, 0, 2) = 3.0;
    MAT2D_AT(br, 0, 0) = 4.0;
    MAT2D_AT(br, 0, 1) = 5.0;
    MAT2D_AT(br, 0, 2) = 6.0;

    MAT2D_ASSERT(nearly_equal(mat2D_dot_product(ar, br), 32.0, eps));
    MAT2D_ASSERT(nearly_equal(mat2D_inner_product(ar), 14.0, eps));

    /* Normalize-by-inf macro (smoke) */
    MAT2D_ASSERT(nearly_equal(mat2D_calc_norma_inf(ar), 3.0, eps));
    mat2D_normalize_inf(ar);
    MAT2D_ASSERT(nearly_equal(mat2D_calc_norma_inf(ar), 1.0, 1e-12));

    mat2D_free(a);
    mat2D_free(b);
    mat2D_free(ar);
    mat2D_free(br);
}

static void test_outer_product_row_vector_path(void)
{
    const double eps = 1e-12;

    Mat2D v = mat2D_alloc(1, 3);    /* row-vector form */
```

```
    Mat2D out = mat2D_alloc(3, 3);

    MAT2D_AT(v, 0, 0) = 1.0;
    MAT2D_AT(v, 0, 1) = 2.0;
    MAT2D_AT(v, 0, 2) = 3.0;

    mat2D_outer_product(out, v);

    /* out[i,j] = v[i]*v[j] (with row-vector indexing) */
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 0), 1.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 1), 2.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 0, 2), 3.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 0), 2.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 1), 4.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 1, 2), 6.0, eps));
    MAT2D_ASSERT(nearly_equal(MAT2D_AT(out, 2, 2), 9.0, eps));

    mat2D_free(v);
    mat2D_free(out);
}

static void test_det_early_zero_row_and_zero_col_paths(void)
{
    /* Exercise mat2D_det() early return when it finds an all-zero row/col */

    Mat2D zr = mat2D_alloc(2, 2);
    /* row0 all zeros => det 0 */
    MAT2D_AT(zr, 0, 0) = 0.0;
    MAT2D_AT(zr, 0, 1) = 0.0;
    MAT2D_AT(zr, 1, 0) = 1.0;
    MAT2D_AT(zr, 1, 1) = 2.0;
    MAT2D_ASSERT(mat2D_row_is_all_digit(zr, 0.0, 0));
    MAT2D_ASSERT(nearly_equal(mat2D_det(zr), 0.0, 0.0));

    Mat2D zc = mat2D_alloc(2, 2);
    /* col0 all zeros but no all-zero row => det 0 */
    MAT2D_AT(zc, 0, 0) = 0.0;
    MAT2D_AT(zc, 0, 1) = 1.0;
    MAT2D_AT(zc, 1, 0) = 0.0;
    MAT2D_AT(zc, 1, 1) = 2.0;
    MAT2D_ASSERT(mat2D_col_is_all_digit(zc, 0.0, 0));
    MAT2D_ASSERT(nearly_equal(mat2D_det(zc), 0.0, 0.0));

    mat2D_free(zr);
    mat2D_free(zc);
}

static void test_mat_is_all_digit(void)
{
    Mat2D m = mat2D_alloc(2, 3);
    mat2D_fill(m, 7.0);
    MAT2D_ASSERT(mat2D_mat_is_all_digit(m, 7.0));
    MAT2D_ASSERT(!mat2D_mat_is_all_digit(m, 8.0));
    MAT2D_AT(m, 1, 2) = 8.0;
    MAT2D_ASSERT(!mat2D_mat_is_all_digit(m, 7.0));
    mat2D_free(m);
}

static void test_power_iterate_and_eig_helpers(void)
{
    /*
     * Use a diagonal matrix with distinct positive eigenvalues:
     * A = diag(5,3,1)
     * Power iteration should find lambda=5 with eigenvector ~ e1.
     * eig_power_iteration should recover (5,3,1) and eig_check residual small.
     */
    const double eps = 1e-7;

    Mat2D A = mat2D_alloc(3, 3);
    mat2D_fill(A, 0.0);
    MAT2D_AT(A, 0, 0) = 5.0;
    MAT2D_AT(A, 1, 1) = 3.0;
    MAT2D_AT(A, 2, 2) = 1.0;

    /* mat2D_power_iterate */
    Mat2D v = mat2D_alloc(3, 1);
    MAT2D_AT(v, 0, 0) = 1.0;
    MAT2D_AT(v, 1, 0) = 1.0;
    MAT2D_AT(v, 2, 0) = 1.0;
    double lambda = 0.0;
    int rc = mat2D_power_iterate(A, v, &lambda, 0.0, true);
    MAT2D_ASSERT(rc == 0);
    MAT2D_ASSERT(close_rel_abs(lambda, 5.0, 1e-6, 1e-6));
    /* dominant component should be index 0 */
    MAT2D_ASSERT(fabs(MAT2D_AT(v, 0, 0)) > fabs(MAT2D_AT(v, 1, 0)));
    MAT2D_ASSERT(fabs(MAT2D_AT(v, 0, 0)) > fabs(MAT2D_AT(v, 2, 0)));

```

```
    /* mat2D_eig_power_iteration + mat2D_eig_check */
    Mat2D evals = mat2D_alloc(3, 3);
    Mat2D evecs = mat2D_alloc(3, 3);
    Mat2D res = mat2D_alloc(3, 3);

    /* init vector must be non-zero */
    Mat2D init = mat2D_alloc(3, 1);
    MAT2D_AT(init, 0, 0) = 1.0;
    MAT2D_AT(init, 1, 0) = 1.0;
    MAT2D_AT(init, 2, 0) = 1.0;

    mat2D_eig_power_iteration(A, evals, evecs, init, true);
    MAT2D_ASSERT(close_rel_abs(MAT2D_AT(evals, 0, 0), 5.0, 1e-5, 1e-5));
    MAT2D_ASSERT(close_rel_abs(MAT2D_AT(evals, 1, 1), 3.0, 1e-5, 1e-5));
    MAT2D_ASSERT(close_rel_abs(MAT2D_AT(evals, 2, 2), 1.0, 1e-5, 1e-5));

    mat2D_eig_check(A, evals, evecs, res);
    MAT2D_ASSERT(mat2D_calc_norma_inf(res) < eps);

    mat2D_free(A);
    mat2D_free(v);
    mat2D_free(evals);
    mat2D_free(evecs);
    mat2D_free(res);
    mat2D_free(init);
}

static void test_deterministic_fuzz_loop(void)
{
    /*
     * Deterministic fuzz:
     *  - generate many random (but well-conditioned) square matrices
     *  - reject near-singular ones (fabs(det) too small / non-finite)
     *  - assert invariants:
     *       inv(A) exists and A*inv(A)~I and inv(A)*A~I
     *       det(A^T)~det(A)
     *       det(A)*det(inv(A))~1
     *       rank(A)==N (via mat2D_reduce)
     */

    const size_t iters = 500;
    const double inv_eps = 1e-7;
    const double det_min = 1e-8;
    const double det_abs_eps = 1e-6;
    const double det_rel_eps = 1e-6;

    uint64_t rng = 0x9e3779b97f4a7c15ULL; /* fixed seed */
    size_t tested = 0;
    size_t skipped = 0;

    for (size_t t = 0; t < iters; t++) {
        size_t n = 2 + (size_t)(xorshift64star(&rng) % 49); /* 2..50 */

        Mat2D a = mat2D_alloc(n, n);
        Mat2D inv = mat2D_alloc(n, n);
        Mat2D prod1 = mat2D_alloc(n, n);
        Mat2D prod2 = mat2D_alloc(n, n);
        Mat2D at = mat2D_alloc(n, n);
        Mat2D tmp = mat2D_alloc(n, n);

        fill_strictly_diag_dominant(a, &rng);

        double det_a = mat2D_det(a);
        if (!isfinite(det_a) || fabs(det_a) < det_min) {
            skipped++;
            mat2D_free(a);
            mat2D_free(inv);
            mat2D_free(prod1);
            mat2D_free(prod2);
            mat2D_free(at);
            mat2D_free(tmp);
            continue;
        }

        /* Inversion invariants */
        mat2D_invert(inv, a);
        mat2D_dot(prod1, a, inv);
        assert_identity_close(prod1, inv_eps);
        mat2D_dot(prod2, inv, a);
        assert_identity_close(prod2, inv_eps);

        /* det(A^T) == det(A) */
        mat2D_transpose(at, a);
        double det_at = mat2D_det(at);
        MAT2D_ASSERT(
            close_rel_abs(det_at, det_a, det_abs_eps, det_rel_eps));

```

```
        /* det(A) * det(inv(A)) == 1 */
        double det_inv = mat2D_det(inv);
        MAT2D_ASSERT(isfinite(det_inv));
        MAT2D_ASSERT(close_rel_abs(
            det_a * det_inv,
            1.0,
            1e-5,   /* a bit looser: det() is numerically touchy */
            1e-5));

        /* rank(A) == N */
        mat2D_copy(tmp, a);
        size_t r = mat2D_reduce(tmp);
        MAT2D_ASSERT(r == n);

        tested++;

        mat2D_free(a);
        mat2D_free(inv);
        mat2D_free(prod1);
        mat2D_free(prod2);
        mat2D_free(at);
        mat2D_free(tmp);
    }

    /* Make sure the loop actually exercised enough cases. */
    MAT2D_ASSERT(tested >= iters / 2);
    printf("[INFO] fuzz summary: tested=%zu skipped=%zu\n", tested, skipped);
}

int main(void)
{
    #define RUN_TEST(fn)         \
        do {                    \
            fn();               \
            printf("[INFO] passed | %s\n", \
                   #fn);        \
            fflush(stdout);     \
        } while (0)

    RUN_TEST(test_uint32_alloc_fill_and_at);
    RUN_TEST(test_offset2d_and_stride);
    RUN_TEST(test_alloc_fill_copy_add_sub);
    RUN_TEST(test_transpose);
    RUN_TEST(test_copy_windows);
    RUN_TEST(test_dot_product_matrix_multiply);
    RUN_TEST(test_det_and_minor_det_agree_3x3);
    RUN_TEST(test_det_2x2_and_upper_triangulate_sign);
    RUN_TEST(test_minor_det_matches_gauss_4x4_known);
    RUN_TEST(test_invert);
    RUN_TEST(test_LUP_decomposition_identity_P_no_swap_case);
    RUN_TEST(test_LUP_decomposition_swap_required_case);
    RUN_TEST(test_reduce_rank);
    RUN_TEST(test_norms_and_normalize);
    RUN_TEST(test_outer_product_and_cross);
    RUN_TEST(test_shift_and_identity);
    RUN_TEST(test_rotation_matrices_orthonormal);
    RUN_TEST(test_DCM_zyx_matches_product);
    RUN_TEST(test_solve_linear_system_LUP);
    RUN_TEST(test_non_contiguous_stride_views);
    RUN_TEST(test_row_col_ops_and_scaling);
    RUN_TEST(test_rand_range);
    RUN_TEST(test_copy_row_and_col_helpers);
    RUN_TEST(test_deterministic_fuzz_loop);
    RUN_TEST(test_dot_product_and_vector_variants);
    RUN_TEST(test_outer_product_row_vector_path);
    RUN_TEST(test_det_early_zero_row_and_zero_col_paths);
    RUN_TEST(test_mat_is_all_digit);
    RUN_TEST(test_power_iterate_and_eig_helpers);

    #undef RUN_TEST

    printf("\n [INFO] Matrix2D tests passed\n");
    return 0;
}
```

Index