**MTX - 3.4**

**C ANSI/C++ BLAS-3 API**

Developed by Eng. Juan Camilo Gómez Cadavid MSc.

|  |  |  |
| --- | --- | --- |
| Macro/Function/Keyword | Description | MATLAB Eq. |
| Generation | | |
| matrix VAR=NULL  mtxdef(VAR) | Matrix MTX type definition Keyword.  matrix keyword, defines unallocated variables.  Always initialize matrix-type MTX variables to NULL  Use mtxdef to define an allocated empty MATRIX by default. | VAR=[]; |
| M=mtx\_new(n,m) | Returns a matrix *M* of *m*x*n* dimensions (initialized on 0) | M=zeros(n,m) |
| M=mtx\_init(n,m,initval) | Returns a matrix *M* of *m*x*n* dimensions initialized on *initval* | M=initval\*ones(n,m) |
| M=mtx\_rand(n,m) | Returns a matrix *M* of *m*x*n* dimensions containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1) | M=rand(n,m) |
| M=mtx\_1dtomtx(A) | Allocates a MTX matrix from the 1D array *A* |  |
| M=mtx\_2dtomtx(A) | Allocates a MTX matrix from the 2D array *A* |  |
| M=mtx\_eye(n,alpha) | Returns the *n*x*n* identity matrix *M = αI.*  *α* is scalar and *I* represents the identity matrix | M=alpha\*eye(n) |
| mtx\_del(M) | Releases the specified block of memory generated by M, back to the heap | clear M |
| M=mtx\_cpy(A) | Generates a copy of matrix *A* into *M*. | M=A |
| mtx\_memcpy(M,A) | Generates a copy of matrix *A* into *M*. (M is allocated, and has the same dimensions of *A*) | M=A |
| Indexing | | |
| x=A->pos[r][c] | Get the element at index *row-r* and *column-c* | x=A(r,c) |
| A->pos[r][c] = value | Set a single element at index *row-r* and *column-c* | A(r,c)=value |
| M=mtx\_getsubset (A,r1,r2,c1,c2) | Get submatrix *M🡨A* (*M* from *A*), from rows *r1* to *r2* and columns *c1* to *c2* | A(r1:r2,c1:c2) |
| M=mtx\_getrow(A,r) | Get the *r-row* from matrix *A* | M=A(r,:) |
| M=mtx\_getcol(A,c) | Get the *c-column* from matrix *A* | M=A(:,c) |
| M=mtx\_getrows(A,r1,r2) | Get rows from matrix *A*. From rows *r1* to *r2* | M=A(r1:r2,:) |
| M=mtx\_getcols(A,c1,c2) | Get columns from matrix *A*. From columns *c1* to c*2* | M=A(:,c1:c2) |
| mxt\_setsubset(M,A,r1,r2,c1,c2) | Put submatrix *A* into *M* from row *f1* to row *f2* and cols *c1* to col *c2* | M(r1:r2,c1:c2)=A |
| mtx\_setrow(M,A,r) | Set the *r-row* from matrix *M*, with the row vector *A* | M(r,:)=A |
| mtx\_setcol(M,A,c) | Set the *c-column* from matrix *M*, with the column vector *A* | M(:,c)=A |
| M=mtx\_vec(A) | Return the matrix A as single column vector | M=A(:) |
| Basic information | | |
| x=mtx\_isempty(M) | TRUE if *M* is an empty matrix | isempty(M) |
| x=mtx\_isrow(M) | TRUE if *M* is a row vector | isrow(M) |
| x=mtx\_iscolumn(M) | TRUE if *M* is a column vector | iscolumn(M) |
| x=mtx\_isvector(M) | TRUE if *M* is vector | isvector(M) |
| x=mtx\_numel(M) | Number of elements in matrix *M* | x=numel(M) |
| x=mtx\_length(M) | Largest matrix dimension of *M* | x=length(M) |
| x=mtx\_det(M) | Determinant of matrix *M* | x=det(M) |
| x=mtx\_trace(M) | Trace of matrix *M* | x=trace(M) |
| D=mtx\_diag(M) | Returns a column vector with all diagonal elements of *M* | D=diag(M) |
| M=mtx\_produ(A) | Product of matrix elements - by columns | M=prod(A) |
| x=mtx\_cumprod(A) | Total product of matrix elements | x=prod(A(:)) |
| M=mtx\_sum(A) | Sum of matrix elements - by columns | M=sum(A) |
| x=mtx\_cumsum(A) | Total sum of matrix elements | x=sum(A(:)) |
| M=mtx\_mean(A) | Mean of matrix elements – by columns | M=mean(A) |
| M=mtx\_max(A) | If *A* is a vector, returns the largest element in *A*.  If *A* is a matrix, treats the columns of *A* as vectors, returning a row vector containing the maximum element from each column. | M=max(A) |
| x=mtx\_cummax(A) | Largest element in matrix | x=max(A(:)) |
| M=mtx\_min(A) | If *A* is a vector, returns the smallest element in *A*.  If *A* is a matrix, treats the columns of *A* as vectors, returning a row vector containing the smallest element from each column. | M=min(A) |
| x=mtx\_cummin(A) | Smallest element in matrix | x=min(A(:)) |
| BLAS-Operations | | |
| M=mtx\_t(A) | Returns the transpose of *A* | M=A' |
| M=mtx\_gadd(alpha,A,beta,B) | Returns *M= αA +* β*B*  *α* and β are scalars, and *A* and *B* matrices. | M=alpha\*A+beta\*B |
| M=mtx\_prod(alpha,A,B) | Returns matrix Multiplication *M=αAB*  *α* is scalar, *A* and *B* matrices. | M=alpha\*A\*B |
| ret=mtx\_dgemm(alpha,A,TA,B,TB,beta,C) | Computes *C = αAB +* β*C* and related operations.  α and β are scalars, and A, B and C are matrices.  *TA* = ‘T’ or ‘t’ 🡪 *A* is considered transposed (*AT*).  *TB* = ‘T’ or ‘t’ 🡪 *B* is considered transposed (*BT*).  On success, returns (0) and matrix *C* is overwritten, otherwise returns (-1). | C = alpha\*A\*B + beta\*C  C = alpha\*A'\*B + beta\*C  C = alpha\*A\*B' + beta\*C  C = alpha\*A'\*B' + beta\*C |
| M=mtx\_kron(A,B) | Returns the Kronecker tensor product of matrices *A* and *B*, *M=AB*.  If *A* is an m-by-n matrix and *B* is a p-by-q matrix, then mtx\_kron(A,B) is an m\*p-by-n\*q matrix formed by taking all possible products between the elements of *A* and the matrix *B*. | M=kron(A,B) |
| M=mtx\_powui(A,n) | Returns *An* (Matrix power - n must be an unsigned int and *A* an square matrix) | M=A^n |
| M=mtx\_ptpprod(A,B) | Returns *A.B* (point by point product) ( *A* and *B* must have the same dimensions) | M=A.\*B |
| M=mtx\_koper(A,Op,beta) | Returns standard scalar matrix operations.  *C = A(Op)β*  Related operations (Op) = ‘+’ , ‘-’ , ’\*’, ‘/’ and ‘^’ (power)  Example:  M=mtx\_koper(A,’+’,beta); | M=A+beta  M=A-beta  M=A\*beta  M=A/beta  M=A^beta |
| M=mtx\_inv(A) | Returns A.1 (Matrix inverse) | M=inv(A) |
| M=mtx\_linsolve(A,B) | Returns *M= A-1B* | M=inv(A)\*B  M=A\B |
| mtx\_lu(L,U,A) | Expresses the matrix *A*, as the product of two essentially triangular matrices, one of them, a permutation of a lower triangular matrix and the other an upper triangular matrix.  (*L* and *U*, must be allocated previously and must have the same dimensions of *A*) | [L,U] = lu(A) |
| mtx\_svd(X,U,W,V) | Given a matrix , this routine computes its singular value decomposition where , and | [U,W,V]=svd(X) |
| Q=mtx\_qr(A,R) | Orthogonal-triangular decomposition. (*R* is allocated previously and must have the same dimensions of *A*) | [Q,R]=qr(A) |
| X=mtx\_sylvester(A,B,C) | Solves the Sylvester equation *AX + XB = C*, where *A* is a m-by-m matrix, *B* is a n-by-n matrix, and *X* and *C* are m-by-n matrices.  The equation has a unique solution when the eigenvalues of A and -B are distinct. In terms of the Kronecker tensor product, , the equation is solved using:  *[ IA + BTI ] X(:) = C(:)*  where *I* is the identity matrix, and *X(:)* and *C(:)* denote the matrices *X* and *C* as single column vectors. | X=sylvester(A,B,C) |
| M=mtx\_rpinv(A,tol) | Returns the Moore-Penrose right pseudoinverse (tol = tolerance) |  |
| M=mtx\_lpinv(A,tol) | Returns the Moore-Penrose left pseudoinverse (tol = tolerance) |  |
| M=mtx\_expm(A, alpha) | Returns matrix exponential *eαA*. mtx\_expm uses the Padé approximation with scaling and squaring. | M=expm(alpha\*A) |
| M=mtx\_fxptp(A, fx) | Returns function *fx* evaluated for each element of matrix *A*  Examples:  M=mtx\_fxptp(A, sin); 🡪 Element-wise sine  M=mtx\_fxptp(A, fabs); 🡪 Element-wise absolute value | M=fx(A) |
| Manipulation | | |
| M=mtx\_vcat(A,B) | Returns *[A ; B]* (Vertical concatenation - append by columns) | M=[A;B] |
| M=mtx\_hcat(A,B) | Returns *[A , B]* (Horizontal concatenation - append by rows) | M=[A,B] |
| M=mtx\_vcatn(A,B,C,…,X,Y,Z) | Return *[A ; B ; C; … , X ; Y ; Z]* (Vertical concatenation - append by columns) | M=[A;B;C;…;X;Y;Z] |
| M=mtx\_hcatn(A,B,C,…,X,Y,Z) | Return *[A , B , C, … , X , Y , Z]* (Horizontal concatenation - append by rows) | M=[A,B,C,…,X,Y,Z] |
| Visualization | | |
| mtx\_disp(A) | Display matrix *A* | disp(A) |
| mtx\_dispn(A,B,C,…,X,Y,Z) | Display n matrices | disp(A),disp(B),… |
| mtx\_show(A) | Display matrix with variable name |  |

**MTX Examples**

1. Solving the Sylvester Equation AX+XB=C

/\* Solve Sylvester Equation A\*X + X\*B = C with 4-by-2 Output \*/

#include <stdio.h>

#include <stdlib.h>

#include "mtx.h"

int main(void){

/\*Create a 4-by-4 coefficient matrix, A, and 2-by-2 coefficient matrix, B.\*/

double a[4][4]= {

1, 0, 2, 3,

4, 1, 0, 2,

0, 5, 5, 6,

1, 7, 9, 0,

};

double b[2][2]= {

0, -1,

1, 0,

};

/\*Define C as a 4-by-2 matrix to match the corresponding sizes of A and B\*/

double c[4][2]={

1, 0,

2, 0,

0, 3,

1, 1,

};

/\*Create a mtx-type variables that points to the 2-dimensional arrays\*/

matrix A = mtx\_2dtomtx(a);

matrix B = mtx\_2dtomtx(b);

matrix C = mtx\_2dtomtx(c);

matrix X = NULL;

/\*Use the sylvester function to solve the Sylvester equation for these values of A, B, and C.\*/

X = mtx\_sylvester(A,B,C);

/\*Display the matrices A,B,C and the solution X\*/

mtx\_dispn(A,B,C,X);

/\*Release the heap\*/

mtx\_del(A);

mtx\_del(B);

mtx\_del(C);

mtx\_del(X);

return EXIT\_SUCCESS;

}

1. Solving the linear system AX=B

/\* Solving the linear equations A\*x=b using LAPACK \*/

#include <stdio.h>

#include <stdlib.h>

#include "mtx.h"

int main(void){

/\*Define the equations using the matrix notation\*/

double a[3][3]= {

3.1, 1.3, -5.7,

1.0, -6.9, 5.8,

3.4, 7.2, -8.8,

};

double b[3]= {

-1.3,

-0.1,

1.8,

};

/\*Create a mtx-type variables that points to the 2-dimensional arrays\*/

matrix A = mtx\_2dtomtx(a);

matrix B = mtx\_1dtomtx(b);

matrix X = NULL;

/\*find solution using MTX routine mtx\_linsolve\*/

X = mtx\_linsolve(A,B);

/\*Display the matrices A,B and the solution X\*/

mtx\_dispn(A,B,X);

/\*Release the heap\*/

mtx\_del(A);

mtx\_del(B);

mtx\_del(X);

return EXIT\_SUCCESS;

}

1. Evaluates a simple matrix expression

/\*

\* Evaluate the expression X= (A\*B)/(alpha + B'\*A\*B)

\* A is a 4-square random matrix and B is a 4-column random matrix

\* alpha is scalar

\*/

#include <stdio.h>

#include <stdlib.h>

#include "mtx.h"

int main(void){

/\*Define the using the matrix notation\*/

matrix A = mtx\_rand(4,4);

matrix B = mtx\_rand(4,1);

matrix X = NULL; //holds the expression

matrix AB = NULL; //holds the auxiliar A\*B result

matrix BtAB = mtx\_new(1,1); // holds the auxiliar B'\*A\*B result

double alpha = 0.8;

/\*Compute the operations\*/

AB = mtx\_prod(1.0, A,B); // AB = A\*B

mtx\_dgemm(1.0, B, 't', AB, '.', 0, BtAB); // compute B'\*A\*B using the generalized matrix product routine mtx\_dgemm

X = mtx\_koper(AB, '/', (alpha + BtAB->pos[0][0]) );

/\*Display the matrices\*/

mtx\_dispn(A,B,AB,BtAB,X);

/\*Release the heap\*/

mtx\_del(A);

mtx\_del(B);

mtx\_del(AB);

mtx\_del(BtAB);

mtx\_del(X);

return EXIT\_SUCCESS;

}