TMC Compiler (tiny M-to-C converter)

The TMC Compiler (tiny M-to-C converter)
Edition 1.10
January 2017

by Shmuel Safonov

This manual is for TMC Compiler, Tiny m-to-C , an open-source tool for conversion MALTAB[1] language-written projects into C-code for compiling it into a stand-alone executable or shared library.

version 1.10 of January 2017.

[1] MATLAB is a registered trademark of Mathworks Inc.

Table of Contents

TMC Compiler Copying Conditions

TMC Compiler consists of TMC Converter tool and TMC Run-time library.

TMC Converter is covered by GPL license:

TMC Converter is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

TMC Compiler is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

TMC Compiler Runtime library is covered by simplified BSD 2-Clause license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The following third-party software are used by TMC runtime library with their own license terms:

1. LAPACK (ver. 3.1)

   Additional copyrights may follow

   Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2. FDLIBM

   Copyright (C) 2004 by Sun Microsystems, Inc. All rights reserved.

   FDLIBM (Freely Distributable LIBM) is a C math library for supporting IEEE 754 -floating-point arithmetic.

3. FFTPACK

   FFTPACK is a package of Fortran subroutines for the fast Fourier transform. It includes complex, real, sine, cosine, and quarter-wave transforms. It was developed by Paul Swarztrauber of the National Center for Atmospheric Research. Public domain

4. F2C LIBS

   Copyright 1990 - 1997 by AT&T, Lucent Technologies and Bellcore.

   Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the names of AT&T, Bell Laboratories, Lucent or Bellcore or any of their entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

   AT&T, Lucent and Bellcore disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall AT&T, Lucent or Bellcore be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

Reporting Bugs

Any feedback may be send by the mailing list:

`https://sourceforge.net/p/tmc-m-files-to-c-compiler/mailman/search/?mail_list=tmc-m-files-to-c-compiler-tmccompiler`

or directly to athors's e-mail: `csafonov@gmail.com`

How to use this Manual

First read Chapter 2 [Installing TMC Compiler], page 5 in order to install the software. Then read Chapter 3 [Getting started with TMC Compiler], page 6 chapter that guides you through compiling your first project.

The chapter Chapter 6 [TMC run-time library], page 13 is recommended for reference of supported build-in functions.

A user is assumed to have a knowledge of MATLAB language.

Refer to Chapter 5 [Supported source language], page 10 for details which features of MATLAB are actually supported. The knowledge of C is not mandatory or needed very little.

The remainder of the manual can be used for later reference.

1 Introduction to TMC Compiler

TMC Compiler is a package for conversion MATLAB-written projects to C code. It consists of TMC Converter, a command-line utility, and TMC Library, a run-time support library .

This software may help to migrate from the prototype stage of the algorithm development to the final implementation. TMC Compiler provides the maximal portability and minimal dependencies on third-party software libraries. The generated C-code can be compiled together with slightly modified C code of existing MEX-files. User may extend or add it's own implementation to the set of build-in functions. The run-time support library may be linked statically or dynamically. Yet the syntax is a bit restricted the basics is provided: it supports cell arrays, structures, 2D matrixes.

TMC Library contains a small number of in-build functions that user can extend according to its own needs. The user functions may be written in C and linked with the application.

2 Installing TMC Compiler

TMC Compiler contains from TMC Converter tool '`tmcco`' and TMC Library, support run-time libraries built for different target platforms.

To build TMC, you need a C compiler; GCC version 4.8 or higher is recommended. And you need a standard Linux '`make`' program, plus some other standard Linux utility programs. For recompiling of the conversion tool '`tmcco`' you need '`BISON`' and '`FLEX`' programs installed.

Here are the steps needed to install the library on Linux systems:

1. '`unzip xzf tmc_src-1.10.zip`'
2. '`cd tmc_src-1.10\src\tmcwood`'
3. '`./make`'

   This compiles '`tmcco`' in the working directory. It should be copied to `tmc_src-1.10\bin`
4. '`cd tmc_src-1.10\src\tmcruntime`'
5. '`./make`'

   This compiles run-time library (static) in the working directory and put it in `tmc_src-1.10\lib`

   Note: you need write permissions on these directories.

Here are the steps needed to install the library on Windows systems (MINGW):

1. '`unzip xzf tmc_src-1.10.zip`'
2. '`cd tmc_src-1.10\src\tmcwood`'
3. '`tmcco_gcc.bat`'

   This compiles '`tmcco`' in the working directory and copies it to `tmc_src-1.10\bin`
4. '`cd tmc_src-1.10\src\tmcruntime`'
5. '`maketmcruntime_mingw.bat`'

   This compiles run-time library (static and shared) in the working directory and put it in `tmc_src-1.10\lib`
6. '`cd tmc_src-1.10\src\qdlapack_tmcruntime`'
7. '`make_qdlapack_mingw.bat`'

   This compiles 128bit-quad precision run-time library for '`roots_qd`' function (static and shared) in the working directory and put it in `tmc_src-1.10\lib`. This step is optional if you are not intended to use this feature.

   Note: you need write permissions on these directories.

3 Getting started with TMC Compiler

The installed package consists of the following directories:

`tmc_src-1.10\bin` -Converter tool binaries
`tmc_src-1.10\doc` - Manuals and documentation
`tmc_src-1.10\examples` -Usage examples
`tmc_src-1.10\include` - Include files
`tmc_src-1.10\lib` - Support run-time libraries (static and shared binaries)
`tmc_src-1.10\src` - Sources

The best Getting Started for a new user is to compiler the attached examples using batch file '`MakeEx1_minigw.bat`' or '`makefile`' that are present in the example directories.

TMC Compiler processes a single *root function* and generates C-code for it. All declarations needed to interface with the code generated by TMC Compiler are collected in the include file `tmc.h`. It is designed to work with C compiler. You should include that file in any program calling the generated code by adding the line

```
#include "tmc.h"
```

3.1 Types

Any variable or *matrix* in TMC Language is represented by a single C data type `tmsMatrix`. This type is implemented as a structure and may represent a real or complex matrix, cell array or a structure. The matrix values are double precision numbers. Any function parameter or result of operation has this type. Thus subsets of cell arrays or multi-dimension matrix are not supported. Integer and logical types are not suported also.

3.2 Initializing run-time library

Before calling any function the run-time library should be intialized:

```
#include "tmc.h"
// for MSVC: #define EXT_LINKAGE __declspec(dllimport) if calling DLL
extern EXT_LINKAGE const struct CInit_funcs_table Init_funcs_table ;
...
tmcInitLib(&Init_funcs_table); // initialization
...
// calling tmc functions
...
tmcFreeLib(); // finalization
```

3.3 Calling root function

The externally called *root function* as well as in-build and user-defined functions have the single prototype like:

`void tmcFunc` (*int* `nargout`, *int* `nargin`, *tmsMatrix* * `Output1`, ... *tmsMatrix* * `OutputM`, *tmsMatrix* * `Input1`, ... *tmsMatrix* * `InputN`) [Function]

The used input and output arguments passed to a function should be initialized to empty matrix. Unused arguments are passed by caller as NULLs.

Example: `Test0.m` contains

```
function y=Test0(x)
...
end
```

that is compiled to

`void tmcTest0` (*int* ***nargout***, *int* ***nargin***,*tmsMatrix * **y**,tmsMatrix * **x**); [Function]

The call sequence should be as

```
{
x = tmcNewMatrix();
y = tmcNewMatrix();
tmcScalar(x,100); // x=100
tmcTest0(1, 1,y,x);
d = y->m_rData[0]; // getting result d = y(1) to double variable
tmcFreeLocalVar(y);
tmcFreeLocalVar(x);
}
```

The following may be a more safe way of getting results:

```
{
 d = tmcMatrixValRe(y,m,n); // assign result element d = y(m,n) to double variable
}
```

If the returned variable *y* is a structure that has a field `P1` then the field can be assigned to a matrix *p* as following:

```
p = tmcNewMatrix();
tmcGetByFieldHcode(p,y,0x09e10000);/* P1 */
```

This is a bit advanced technique that requires from user to find the hash-code `0x09e10000` of the string '`P1`'. The hash-code is found in the file `Test0.hash_init.dat`:

```
...
"P1",0x09e10000,
...
```

provided that the field '`P1`' is explicitly reference somewhere in the processed source code.

3.4 User-defined C-functions

The set of in-build functions may be extended by user. To perfom it, the function implementation should be written with prototype described in Section 3.3 [Calling root function], page 6 and it must be available to converted by putting the function *signature* to *symbol definition file*. Each row of the file should have format as

```
name,maximal_number_of_inputs,maximal_number_of_outputs,x;
```

The function names in C-code must begin with `tmc`.

4 TMC Converter

TMC Converter is a command-line tool that should be run in two-passes:

1. Preparing actual source files list and common project symbol file
2. Generartion C-code from the source files using the symbol file

The pass 1 is needed to buid the symbol file that contains the signature definitions of all the functions that are referenced in the project. The compiler parses the referenced files starting from the root file, using the project specific `external_fnc.sym.dat` file together with the TMC library `buildin_fnc.sym.dat` file and generates the output common project symbol file.

Before calling TMC Compiler `external_fnc.sym.dat` file should be copied from the source to output directory:

```
type .\Stubs\external_fnc.sym.dat > .\OutL\external_fnc.sym.dat (Windows)
```

or

```
cp ./Stubs/external_fnc.sym.dat  ./OutL/external_fnc.sym.dat (Linux)
```

4.1 TMC Converter Command-line switches

Mode switches:

`-L` Parse m-files, generate symbol table (1st pass)

`-g2` Generate C-code from m-files using the symbol table (2nd pass)

Input source switches:

`-r` *name* Defines the file *name* with root function definition.

`-i` *path* Include the directory *path* in the input files search. Multiple `-i` switches may be given. The switch may be put in a response file.

`-s` *path* Include the directory *path* in the sources files search for source-comment generation. Multiple `-i` switches may be given. The switch may be put in a response file.

`-@` *response file*
Response file with `-i` and `-s` options

`-h` *path* Path to TMC library includes directory that conatains `buildin_fnc.sym.dat` file

Output destination switches:

`-w` *name* Workspace name

`-o` *path* Output directory name

Code generation control switches:

`-c` Generate C-code

`-S` Generate x86 assembler code

`-A` Parse all the files (ignore dependencies)

`-P` Generate lsp-output (1st pass)

Debugging switches:

`-d` Include debugging support code.

`-C`	Include m-code as comments in generated C-code
`-q`	Quite mode, minimal display
`-f` *name*	Generate C - code for a single file
`-l` *name*	Listing filename

Invoking examples:

1. Symbol table generation from m-files (1st pass)

```
mkdir .\OutL
mkdir .\OutC
type  .\Stubs\external_fnc.sym.dat > .\OutL\external_fnc.sym.dat

tmcco  -L -w TestO  -r ./Stubs/TestO.m -h ..\..\include/ -l ./OutL/testO.lsp.txt
 -@ woo1_rsp.txt  -o ./OutL/ > out.txt
```

`woo1_rsp.txt`:

```
-i ./MatSRC/
-i ./Utils/
```

2. C-code generation from m-files and the symbol table (2nd pass):

```
copy  .\OutL\TestO.sym.dat .\Stubs\
tmcco -c -C   -d -q  -g2  -w TestO  -r .\Stubs\TestO.m -l .\OutC\testO2.err.log
 -@ woo2_rsp2.txt  -o .\OutC\ > outC2.txt
```

`woo2_rsp2.txt`:

```
-i ./MatSRC/
-s ./MatSRC/
-s ./Stubs/
```

3. Symbol table and lsp-files generation from m-fies (1st pass):

```
type  .\Stubs\external_fnc.sym.dat > .\OutL\external_fnc.sym.dat
tmcco -L -P -w TestO  -r ./Stubs/TestO.m -h ..\..\include/  -l ./OutL/testO.lsp.txt
 -@ woo1_rsp.txt -o ./OutL
```

4. C-code generation from lsp-files and the symbol table (2nd pass):

```
copy  .\OutL\TestO.sym.dat .\Stubs\
tmcco  -c -C -d -g2 -w TestO -r ./Stubs/TestO.lsp -l -l ./OutL/testO.lsp.txt
-@ woo2_rsp3 -o .\OutC\
```

`woo2_rsp3.txt`:

```
-s ./MatSRC/
-s ./Stubs/
```

5 Supported source language

This chapter describes the MATLAB language subset supported by TMC Compiler. The following list summarizes some differences from MATLAB(TM) language :

- No graphics and GUI support. The generated code is suggested to be a library used by an application that provides its own GUI. However for debugging purpose code that may call user-provided graphics may be generated.
- Only functions are compiled, not scripts; each file may contain a single function only with the same name as the fie itself, case-sensitive.
- The usage of `end` keyword has some restrictions. The simple matrixes are supported but not expressions like `strct.fld(end)`
- The expression of `if` command should be separated from instructions body by ',' or newline separator.
- Only 2D matrixes are supported; the 3D matrixes may be supported in future.
- Structure arrays are not supported. User is recommended to use cell arrays to replace them.
- Cell arrays may be indexed only by simple scalar index: e.g S{n} where n is a number, not array.
- Logical variables are not supported. E.g.:

  ```
  x =[1,2,3]
  ```

  Expression

  ```
  x(x>1)
  ```

  that in MATLAB results in [2,3] in TMC language is equal to x([0,1,1]) and produces an error. User should replace the code by

  ```
  x(find(x>1)).
  ```
- No classes support
- `try/catch` are supported by the host platform and C compiler: this works with MS VC, Borland C Builder but doesn't work with Android/Linux GNU C.
- Symbols detected as functions (in-build functions e.g.) and reserved symbols `for j=i=sqrt(-1)` may not be re-assigned to become variables. Thus `j` and `i` may not be used as variables!

5.1 Data initialization

Matrix and arrays are initialized like

```
X =  [value1, value2, ...  ]
```

where separator '`,`' can be ommited but is recommended.

String array is initialized as

```
s = 'character string'
```

Cell arrays are initialized like

```
C =  {value1, value2, ...  }
```

or by an assignment like

```
C{k} = value1
```

Complex numbers are assigned as

```
x = 1 + j
```

or

```
x = 1 + i
```

Note, that imaginal unit symbols symbols `i` and `j` are always reserved and **may not be overridden** and used as variables or function names !!!

Structures are initialized like

```
S =  struct('fn1', value1,  ...)
```

or by an assignment like

```
S.fn1= value1
```

Function handles are initialized and used like

```
FH = @sin   % assign function handle of sunction sin(x) to variable FH
Y=FH(X) % call sin(X) through the handle FH
```

Only handles to functions with a single argument are supported by TMC Library.

A global scope variable is declared as

```
global X
```

5.2 `end` symbol for last index

Special symbol `end` is supported for last index. The value of `end` when it states in k-th dimension index position is size(X,k). This syntax is implemented in expressions like

```
X(1:end-4)
```

5.3 Control Flow constructions

5.3.1 Loop `for`

For loop is defined like

```
for var=expression
<instruction block>
end
```

5.3.2 Loop `while`

`while` loop is defined like

```
while expression
<instruction block>
end
```

5.3.3 Block `if`

Block `if` is defined like

```
if expression1
<instruction block1>
elseif expression2
<instruction block2>
...
else
<instruction block >
end
```

A separator like '`,`' or newline must present between condition a expression and the corresponding instruction block.

5.3.4 Block `switch`

Block `switch` is defined like

```
switch expression1
case value-list1
<instruction block1>
case value-list2
<instruction block2>
...
otherwise
<instruction block >
end
```

5.3.5 Block `try/catch`

Block `try/catch` is defined like

```
try
  <instruction block1>
catch
<instruction block2 >
end
```

The `try/catch` functionaly is supported only if the C-compiler supports SEH exception handling (e.g. MSVC, C-Builder). If `try/catch` is not supported then the generated C-code should be compiled with the command options that define the corrsponding symbols as folowing:

```
-DTMC_NO_SEH=";"  -DTRY=";" -DCATCH=";" -DENDCATCH=";" -DFINALLY=";"  -DENDFINALLY=";"
```

5.3.6 Special commands

- `break` Branch out of `for` or `while` loop.
- `continue` Branch to the next iteration of `for` or `while` loop.
- `return` Return from the function

6 TMC run-time library

The TMC run-time library provides:

- External API: Functions for the code initialization, passing data to compiled code and retrieving results.
- Internal support for basic operations
- Build-in functions implementation

6.1 Initialization functions

The code calling the TMC-generated function should be initialized and finally un-initialized. Functions `tmcInitLib` and `tmcFreeLib` are used for this purpose:

short tmcInitLib(const struct CInit_funcs_table *pInit_funcs_table); short tmcFreeLib(void);

`short tmcInitLib` (*const struct CInit_funcs_table * **pInit_funcs_table***) [Function]
Initialize TMC run-time. Parameter `pInit_funcs_table` should pass to the initialization function the pointer `&Init_funcs_table` to the global table that is generated by TMC compiler.

`short tmcFreeLib` (*void*) [Function]
Free the resources occupied by the runtime.

Example:

```
#include "tmc.h"
int main()
{
// for MSVC: #define EXT_LINKAGE __declspec(dllimport) if calling DLL
extern EXT_LINKAGE const struct CInit_funcs_table Init_funcs_table ;
...
tmcInitLib(&Init_funcs_table); // initialization
...
// calling tmc functions
...
tmcFreeLib(); // finalization
}
```

6.2 Data Assigning functions

These functions initialize and assign new values to matrix.

`tmsMatrix tmcNewMatrix` (*void*) [Function]
Create new matrix. The matrix is initialized by empty value.

Example:

```
tmsMatrix *x = tmcNewMatrix(); // x = []
```

`void tmcFreeLocalVar` (*tmsMatrix * **src***) [Function]
Free local variable *src* at exit from a function.

Example:

```
tmsMatrix *x = tmcNewMatrix(); // x = []
...
tmcFreeLocalVar(x);
```

`void tmcScalar` (*tmsMatrix * **dest**, double x*) [Function]
Assign real value *dest*=x

`void tmcComplexScalar` (*tmsMatrix * **dest** ,double **xr** ,double **xi**)* [Function]
Assign complex value *dest*=xr + j*xi

6.3 Build-in MATLAB functions support

The section contains the list of supported build-in functions. Each function is listed with its C-code function prototype from TMC Library. As noted, the following naming convention takes place: each function call `foo` is compiled to `tmcfoo`. The parameters are passed in the following sequence: `actual number of outputs, actual number of inputs, output parameters, input parameters`

`Y=abs( X)`

`void tmcabs` (*long **nout**, long **ninput** , tmsMatrix *y,tmsMatrix *x*) [Function]
Returns matrix Y composed of the absolute values of X.

`Y=acos( X)`

`void tmcacos` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns matrix Y=acos(X).
For complex argument `acos(z)=-i * log(z+i * sqrt(1-z^2))`

`Y=all( X)`

`void tmcall` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x,tmsMatrix *dim_a*) [Function]
For vectors, all(X) is 1 if all X are non-zero
For matrices, all(X) is a row vector with all() over each column.

`Y=angle( Z)`

`void tmcangle` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns matrix `Y=atan2(imag(Z), real(Z))`, that is argument *phi* of `Z=|Z| * (cos (phi) + j * sin (phi))` in radians

`Y=any( X)`

`void tmcany` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x,tmsMatrix *dim_a*) [Function]

For vectors, any(X) is 1 if any X are non-zero
For matrices, any(X) is a row vector with any() over each column.

`Y=asin( X)`

`void tmcasin` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]

Returns matrix Y=asin(X).
For complex argument `asin(z)=-i * log(i*z + sqrt(1-z^2))`

`Y=atan( X)`

`void tmcatan` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]

Returns matrix Y=atan(X).
For complex argument `atan(z)=(i/2) * log((i+z)/(i-z) )`

`Y=atan2( Y,X)`

`void tmcatan2` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *xs,tmsMatrix *xc*) [Function]

Returns matrix four-quadrant inverse tangent of the real parts of Y and X. Imaginary parts are ignored.

`Y=axis( Y,...)`

`void tmcaxis` (*long nout,long ninput,tmsMatrix *hand,...*) [Function]

N/A, reserved.

`Y=bitand( A,B)`

`void tmcbitand` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *a,tmsMatrix *b*) [Function]

Returns matrix `Y = A & B` (bit-wise AND). Defined only for real arguments.

`Y=bitor( A,B)`

`void tmcbitor` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *a,tmsMatrix *b*) [Function]

Returns matrix `Y = A | B` (bit-wise OR). Defined only for real arguments.

`Y=bitshift( A,K [,M])`

`void tmcbitshift` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *a,tmsMatrix *k,tmsMatrix *n*) [Function]

Returns `Y=A<<K, K>0` or `Y=A>>(-K), K<0`. Only N bits are treated (default N=53).
A must be real, *K*,*N* must be scalars.

`Y=ceil( X)`

`void tmcceil` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]

Returns a value representing the smallest integer that is greater than or equal to *X*.
For complex *X* applied separately to real and imagine parts.

`Y=cell( N,[,M])`

`void tmccell` (*long nout,long ninput,tmsMatrix *Y, tmsMatrix *in1,. . .*); [Function]

Returns empty cell array. Only one-dimention and two dimension cell arrays are supported.

`Y=cell2mat( X)`

`void tmccell2mat` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *x*) [Function]

N/A, reserved.

`Y=char( X)`

`void tmcchar` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]

Convert real double array X to 16bit character string.

`Y=close( H)`

`void tmcclose` (*long nout,long ninput,tmsMatrix *stat,tmsMatrix *hand*) [Function]

N/A, reserved.

`Y=cond( X,[,p])`

`void tmccond` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *A,tmsMatrix *p*) [Function]

Calculate conditional number for matrix:
`cond(X,2)=max(sigma (X))/min(sigma (X))` where `sigma=svd(X)` is vector of singalar values of *X*.
`cond(X )=cond(X,2)`
`cond(X,p)= norm(X,p) * norm(inv(X),p)` where

`norm(A,1)` is largest column sum of A, `max(sum(abs(A))`
`norm(A,Inf)` infinity norm, or largest row sum of A, `max(sum(abs(A')))`
`norm(A,'fro')` Frobenius-norm of matrix A, `sqrt(sum(diag(A'*A)))`

The implementation uses LAPACK's `ZGESVD/DGESVD` functions.

`Y=conj( X)`

`void tmcconj` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]

Calculate complex conjugate for matrix:
`conj(X)=real(X)-j .* imag(X)`

`C=conv( A,B)`

`void tmcconv` (*long nout,long ninput, tmsMatrix *C, tmsMatrix *A, tmsMatrix *B*) [Function]

Calculate convolution of two vectors:
`C=(c(1), ..., c(n_a+n_b-1) )` of `A=(a(1),...,a(n_a))` and `B=(b(1),..b(n_b))` :
`c(k) = sum (for j=max(1,k+1-n_b) to min(k,n_a) of a(j)* b(k+1-j) )`
The coefficients of two polynomials are the convolution of their coefficients.

`Y=cos( X)`

`void tmccos` (*long nout,long ninput,tmsMatrix *matres,tmsMatrix *x*) [Function]

Calculate cosine of X. For complex argument z
`cos(z)=(e ^a +e^( -a)) cos(b)/2 + i* ( e^ a -e ^(-a)) sin(b)/2`
where `a=-imag(z)` , `b= real(z)`.

`Y=cumprod( X [,dim])`

`void tmccumprod` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x,tmsMatrix *dim*) [Function]

Calculate cumulative product of elements of *X*.

`Y=deal( X [,...])`

`void tmcdeal` (*long nout,long ninput,tmsMatrix *y,...*) [Function]

`Y=dec2hex( X [,N])`

`void tmcdec2hex` (*int nargout, int nargin,tmsMatrix *y,tmsMatrix *x,tmsMatrix *n*) [Function]

Convert decimal positive integer *X* to hexadecimal string *Y* with *N* hexadecimal digits.

`[Q,R]=deconv( V ,U)`

`void tmcdeconv` (*long nout,long ninput, tmsMatrix *q, tmsMatrix *r, tmsMatrix *v,tmsMatrix *u*) [Function]

Deconvolution and polynomial division V(z)/U(z): `V=Q*U+R`

Assumed *V* ,*U* are vectors and `U(1)<>0`

`Y=det( A)`

`void tmcdet` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *A*) [Function]

Returns the determinant *Y* of the square matrix *A*

Uses LAPACK's `ZGETRF/DGETRF` functions.

`Y=diag( d)`

`void tmcdiag` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *d*) [Function]

Returns diagonal matrix with diagonal *d*

`Y=diff( X)`

`void tmcdiff` (*long nout,long ninput, tmsMatrix *dx,tmsMatrix *x*) [Function]

Returns difference:

For a vector *X*, is [X(2)-X(1), X(3)-X(2), . . . , X(n)-X(n-1)].

For a 2D matrix *X*, is the matrix of row differences: [X(2:n,:) - X(1:n-1,:)]

`Y=disp( X)`

`void tmcdisp` (*long nout,long ninput,tmsMatrix *ydummy,tmsMatrix *x*) [Function]
Display matrix *X*.
Returns []

`Y=double( X)`

`void tmcdouble` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]
Convert matrix or string *X* to doube numeric. Defined only for matrix or string (not cells or structures).

`Y=eig( A)`

`void tmceig` (*long nout,long ninput,tmsMatrix *out1,tmsMatrix *out2,tmsMatrix *in1,tmsMatrix *in2,tmsMatrix *in3*) [Function]
Returns the determinant *Y* of the square matrix *A*
Uses LAPACK's `ZGEEVX/DGEEVX` functions .

`Y=eps( [R])`

`void tmceps` (*long nout,long ninput,tmsMatrix *dest,tmsMatrix *R*) [Function]
Floating point relative accuracy. `Y=eps` : Returns distance from 1.0 to the next largest double-point number, actually 2^(-52)
`Y=eps(R)` : if R<DBL_MIN=2.2250738585072014e-308, returns 2^(-1074)
else returns 2^(E-52), where E=floor(log(x)/ log(2))

`error( S )`

`void tmcerror` (*long nout,long ninput,tmsMatrix *ydummy, tmsMatrix *msg_string*) [Function]
Raise error with message given by string *S* . If the error is not catched the program execution terminates.

`Y=eval ( S )`

`void tmceval` (*long nout,long ninput,tmsMatrix *ydummy, tmsMatrix *str*) [Function]
NA, not implemented, raises error.
See also: `feval`.

`Y=exist ( S )`

`void tmcexist` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x,tmsMatrix *mtype*) [Function]
NA, not implemented, raises warning message.

`Y=exp ( X )`

`void tmcexp` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]
Return exponent of elements of matrix or vector *X*.

`Y=eye ( M [,N] )`

`void tmceye` (*long nout,long ninput,tmsMatrix *Y, tmsMatrix *in1,tmsMatrix *in2*) [Function]
Return 2D Identity matrix *Y* with dimensions given by M,N. Diagonal elements are 1, others are 0.
`Y=eye(m)` has dimensions (m,m)
`Y=eye([m,n])=eye(m,n)` have dimensions (m,n)

`Y=fclose ( h )`

`void tmcfclose` (*long nout,long ninput,tmsMatrix *ydummy,tmsMatrix *h*) [Function]
Close the file with handle *h* that was open by `fopen` function.
If *h* is invalid handle raises ae error.

`Y=feof ( h )`

`void tmcfeof` (*long nout,long ninput,tmsMatrix *mIsEof,tmsMatrix *h*) [Function]
Return 1 if end-of-file condition is found for the file with handle *h* opened by `fopen`.
Otherwise returns 0.
If *h* is invalid handle raises ae error.

`Y=feval ( fnch ,x1 )`

`void tmcfeval` (*long nout,long ninput,tmsMatrix *y, tmsMatrix *fnc_handle, tmsMatrix *x1, ...*) [Function]

Calls function specified by the function handle *fnch* with argument *x1* .
Example:

```
FH = @sin   % assign function handle of sunction sin(x) to variable FH
Y=FH(X) % call sin(X) through the handle FH
```

Only single argument *x1* is supported.

`Y=tmcfft ( X )`

`void tmcfft` (*long nout,long ninput,tmsMatrix *out,tmsMatrix *in*) [Function]

Calculate FFT for vector X:
`Y(k) = sum for (n=1,..N): x(n)*exp(-j*2*pi*(k-1)*(n-1)/N), 1 <= k <= N`
Only one-dimensional vector X is supported.

`Y=fgetl ( h )`

`void tmcfgetl` (*long nout,long ninput,tmsMatrix *str, tmsMatrix *h*) [Function]

Read a line from the file with handle *h* that was open by `fopen`
The line terminator is NOT included. If an end-of-file is encountered then return -1.

`Y=fieldnames ( S )`

`void tmcfieldnames` (*long nout,long ninput,tmsMatrix *flist,tmsMatrix *S*) [Function]

Return cell array Y with field names of structure S

`Y=fields ( S )`

`void tmcfields` (*long nout,long ninput,tmsMatrix *flist,tmsMatrix *S*) [Function]

Obsolete. The same as fieldnames. Return cell array Y with field names of structure S

`Y=figure ( n )`

`void tmcfigure` (*long nout,long ninput,tmsMatrix *fhand,tmsMatrix *fnum*) [Function]

Not suppored.

`Y=fill ( n )`

`void tmcfill` (*long nout,long ninput,tmsMatrix *hand, tmsMatrix *x, tmsMatrix *y, tmsMatrix *c*) [Function]

NA.

`[I,J,V]=find ( X , [Opt,sOpt] )`

`void tmcfind` (*long nout,long ninput, tmsMatrix *I,tmsMatrix *J,tmsMatrix *V, tmsMatrix *x,tmsMatrix *Opt,tmsMatrix *sOpt*) [Function]

Find indexes if non-zero values of matrix X

`[I,J,V]=find ( X)` return all rows I, columns J indexes and the values.

`I=find ( X,n ) = find ( X,n,'first' )` return first n indexes I of non-zero elements of vector X.

`I=find ( X,n ) = find ( X,n,'last' )` return last n indexes I of non-zero elements of vector X.

Example:

```
x =[1,20,30]
I = find(x>1) % return I = [2,3]
Y = x(I) % return Y = [20,30]
```

Note, that expression `I = (x>1)` produces result `[0,1,1]` and expression

```
x(x>1)
```

that in MATLAB produces `[2,3]`, in TMC is equivalent to `x([0,1,1])` and raises an error! In TMC user should always replace

```
x(x>1)
```

by

```
x(find(x>1)).
```

See also: `isfinite, isnan`

`K=findstr ( S1 , S2 )`

`void tmcfindstr` (*long nout,long ninput,tmsMatrix *K,tmsMatrix *S1,tmsMatrix *S2*) [Function]

Returns the starting indices of any occurrences of the shorter of the two strings *S1* , *S2* in the longer.

See also: *strfind*

`Y=fix ( X )`

`void tmcfix` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Rounds the values of X to the nearest integers towards zero.

`Y=fliplr ( X )`

`void tmcfliplr` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns X with columns flipped in the left/right direction and rows preserved.

`Y=floor ( X )`

`void tmcfloor` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Rounds the values of X to the nearest integers towards minus infinity.

`Y=fopen ( fname, permission )`

`void tmcfopen` (*long nout,long ninput,tmsMatrix *h,tmsMatrix* [Function]
**fname,tmsMatrix *perm*)
Opens a file for formatted I/O with mode specified by string *permission*.
Actually calls standart C-library function `fopen`.
Modes are:

`'r'` Opens a file for reading. The file must exist.

`'w'` write (create if necessary)
Creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.

`'a'` append (create if necessary)
Writing operations, append data at the end of the file. The file is created if it does not exist.

`'r+'` read and write (do not create)
Opens a file to update both reading and writing. The file must exist.

`'w+'` truncate or create for read and write
Creates an empty file for both reading and writing.

`'a+'` read and append (create if necessary)
Opens a file for reading and appending.

`'W'` write without automatic flushing - not available

`'A'` append without automatic flushing - not available

`fprintf ( fm,... )`

`void tmcfprintf` (*long nout,long ninput, tmsMatrix *fm,. . .*) [Function]
Print formatted string to the file opend by `fopen` function. Behavours like C-library `fprintf` but has some limitations.

`Y=frd ( Resp,Freqs)`

`void tmcfrd` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *Resp,tmsMatrix *Freqs*) [Function]
NA, raises error

`[num,den,tsamp]=frdata ( sys,mopt)`

`void tmcfrdata` (*long nout,long ninput, tmsMatrix *num, tmsMatrix *den,tmsMatrix *tsamp, tmsMatrix *sys,tmsMatrix *mopt*) [Function]
NA, raises error

`y=freqresp ( sys,w)`

`void tmcfreqresp` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *sys,tmsMatrix *w*) [Function]
NA, raises an error

`hand=gca`

`void tmcgca` (*long nout,long ninput,tmsMatrix *hand*) [Function]
NA, produces a warning

`hand=gcf`

`void tmcgcf` (*long nout,long ninput , . . .*) [Function]
NA, produces a warning

`Y=getfield (S, m_fn)`

`void tmcgetfield` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *S, tmsMatrix *m_fn*) [Function]

Returns value of field with name *m_fn* from single element structure *S*. Available only when `length(S)=1`.

`grid (onoff)`

`void tmcgrid` (*long nout,long ninput,tmsMatrix *ydummy,tmsMatrix *onoff*) [Function]

NA, produces a warning

`Y = hex2dec (S)`

`void tmchex2dec` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *S*) [Function]

Calculate integer value from hex-decimal string presentation *S*.

`hold(onoff)`

`void tmchold` (*long nout,long ninput,tmsMatrix *ydummy,tmsMatrix *onoff*) [Function]

NA, produces a warning

`Y = imag(X)`

`void tmcimag` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]

Return imaginal part of complex number *X*.

`yi = interp1 (x, y, xi [,typeinter ,typeextr])`

`void tmcinterp1` (*long nout,long ninput, tmsMatrix *yi, tmsMatrix *x, tmsMatrix *y, tmsMatrix *xi,tmsMatrix *typeinter,tmsMatrix *typeextr*) [Function]

Calculate one-dimensional interpolation of function `y=y(x)` at the points *xi*.
Only linear interpolation is supported.
Only real data is supported.
Assumed that *x* is sorted.
Parameters *typeinter* (interpolation method), *typeextr* (extraplation mode) are ignored.

```
[AB,IA]=intersect (A,B)
```

`void tmcintersect` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *I,tmsMatrix *J,tmsMatrix *A,tmsMatrix *B*) [Function]

Intersect sets A and B: find the values AB of A that are present in B and their indexes IA .

```
Y=inv (X)
```

`void tmcinv` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]

Calculate inverse matrix for given square matrix X. Actually calculate such `Y=inv (X)=X\eye (n)` that solves the equation `X*Y=eye (n)`. Uses LAPACK's functions `ZGELS/DGELS`.

```
y= isa (x)
```

`void tmcisa` (*long nout,long ninput,tmsMatrix *y, tmsMatrix *x, tmsMatrix *str*) [Function]

NA, produces a warning

```
Y= iscell (X)
```

`void tmciscell` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]

Returns: 1 if X is a cell array, otherwise 0.

```
Y= ischar (X)
```

`void tmcischar` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]

Returns: 1 if X is a string array, otherwise 0.

```
Y= isempty (X)
```

`void tmcisempty` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]

Returns: 1 if X is an empty matrix or empty structure; otherwise 0.

```
Y= isequal (X1,X2 [,X3,...])
```

`void tmcisequal` (*long nout,long ninput,tmsMatrix *res,tmsMatrix *x1,...*) [Function]
Returns: 1 if all matrixes `X1,X2,...` are numerically equal; otherwise 0.
Only vectors, matrixes and strings are supported. NAN and Inf are not equal.

```
Y= isfield (S,fn)
```

`void tmcisfield` (*long nout,long ninput,tmsMatrix *y, tmsMatrix *S, tmsMatrix *m_fn*) [Function]
Returns: 1 if structure *S* has a field with names given by string *m_fn*; otherwise 0.

```
Y= isfinite (X)
```

`void tmcisfinite` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns: matrix of the same dimension as *X*, where 1 mark finite elements and 0 mark infinite values.
Note: since TMC does not support logicals, when indexing should decoreate `isinfinite (x)` by `find (isinfinite (x))`. See function `find`, an example.

```
Y= ishold (fhan)
```

`void tmcishold` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *fhan*) [Function]
NA, produces a warning

```
Y= ismember (Y,S)
```

`void tmcismember` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *A,tmsMatrix *S*) [Function]
Not implemented in this version

```
Y= isnan (X)
```

`void tmcisnan` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns: matrix of the same dimension as *X*, where 0 mark finite elements and 1 mark NaNs.
Note: since TMC does not support logicals, when indexing should decoreate `isnan (x)` by `find (isnan (x))`. See function `find`, an example.

`Y= isnumeric (X)`

`void tmcisnumeric` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns: 1 if X is a matrix, otherwise 0.

`Y= isreal (X)`

`void tmcisreal` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns: 1 if X is a matrix with numbers that has not imagine parts , otherwise 0.
For empty matrix also return 1.

`Y= isscalar (X)`

`void tmcisscalar` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns: 1 if X is a matrix with dimentions `1x1` , otherwise 0.
For empty matrix also return 1.

`Y= isstruct (X)`

`void tmcisstruct` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]
Returns: 1 if X is a struct

`Y= isvector (X)`

`void tmcisvector` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Returns: 1 if 2D matrix X is a vector or column, i.e. one of dimensions is 1.

`Y=lasterr ( X )`

`void tmclasterr` (*long nout,long ninput,tmsMatrix *msg_string_out,* [Function]
*tmsMatrix *msg_string*)
`Y=lasterr` returns the message string of the last error raised by `error` function.
`lasterr ([])` clears the message string of the last error raised by `error` function.
The function is useful when working with `try/catch` blocks if they are supported by C-compiler.

`Y=length ( X )`

`void tmclength` (*long nout,long ninput, tmsMatrix *len, tmsMatrix *X*) [Function]
Returns the length of vector *X* i.e. `max (size (X))`.

`Y=linspace ( X1, X2[, nP ] )`

`void tmclinspace` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *x1, tmsMatrix *x2,tmsMatrix *nP*) [Function]
Returns vector of *nP* lineary spaced points from *X1* to *X2*. If ommited, `nP=100`.

`Y=load ( S )`

`void tmcload` (*long nout,long ninput,tmsMatrix *W, tmsMatrix *fn, . . .*) [Function]
Loads MAT-file with name *S* into structure variable *Y*. Only MATLAB MAT-file formats V4 or V5 are supported and only such objects may be loaded that are upported by TMC.

`Y=log ( x)`

`void tmclog` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]
Calculate natural logarithm `ln (X)` of *X*. For complex argument x
`ln (x ) = ln |x| + i * arg (x)`

`Y=log10 ( x)`

`void tmclog10` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]
Calculate decimal logarithm of *X*. For complex argument x
`log10 (x ) = log10 |x| + i * arg (x)`

`Y=logspace ( X1, X2[, nP ] )`

`void tmclogspace` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *x1, tmsMatrix *x2,tmsMatrix *nP*) [Function]
Returns vector of *nP* decimal-logarithmically spaced points from `10^X1` to `10^X2`. If ommited, `nP=50`.

`Y=lower ( S )`

`void tmclower` (*long nout,long ninput,tmsMatrix *matres,tmsMatrix *src*) [Function]
Converts the string *S* characters to lower-case.

`[ Y,I]= max ( A [,B ])`

`void tmcmax` (*long nout,long ninput, tmsMatrix *y1,tmsMatrix *y2,tmsMatrix *x1,tmsMatrix *x2,tmsMatrix *d*) [Function]

Find maximum:
If *A* is matrix, `[Y,I]=max (A)` returns row of maximums of each column, and their indexes.
If *A* is vector, `[Y,I]=max (A)` returns maximal element and its index.
If *A*, *B* are matrixes of the same sizes or one of them scalar, `[Y,I]=max (A,B)` returns matrix of the same size with elements that are maximum between *A* and *B*.

`[ Y,I]= min ( A [,B ])`

`void tmcmin` (*long nout,long ninput, tmsMatrix *y1,tmsMatrix *y2,tmsMatrix *x1,tmsMatrix *x2,tmsMatrix *d*) [Function]

Find minimum:
If *A* is matrix, `[Y,I]=max (A)` returns row of minimums of each column, and their indexes.
If *A* is vector, `[Y,I]=max (A)` returns minimum element and its index.
If *A*, *B* are matrixes of the same sizes or one of them scalar, `[Y,I]=min (A,B)` returns matrix of the same size with elements that are minimum between *A* and *B*.

`Y=mod ( A, B )`

`void tmcmod` (*long nout,long ninput,tmsMatrix *matres,tmsMatrix *src1,tmsMatrix *src2*) [Function]

Return modulus after division *A* by *B*.
Implemented as by-element `x-floor (x/y)*y`. By convension, mod (A,0)=A.
Result has the same size as B

`Y=nargchk ( low, high,n)`

`void tmcnargchk` (*long nout,long ninput, tmsMatrix *message, tmsMatrix *low, tmsMatrix *high,tmsMatrix *n*) [Function]

Validate that number of arguments *n* falls into the range of `low ... high`.

`Y=ndims ( M )`

`void tmcndims` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *M*) [Function]

Returns number of dimensions of matrix *M*. For vectors and 2D matrixes it is 2.

`Y=nichols ( Sys,mopt)`

`void tmcnichols` (*long nout,long ninput,tmsMatrix *sys,tmsMatrix *mopt*) [Function]
NA, produces a warning

`Y=norm ( X,[,p])`

`void tmcnorm` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *X, tmsMatrix *n*) [Function]
Calculate norm for matrix:

`norm (X,1)` is largest column sum of A, `max (sum (abs (X))`
`norm (X,2)` is is largest singular value of *X*
`norm (X,Inf)` infinity norm, or largest row sum of X, `max (sum (abs (X')))`
`norm (X,'fro')` Frobenius-norm of matrix X, `sqrt (sum (diag (X'*X)))`

Calculate norm for vector:
`norm (X,1)= sum (abs (X))`
`norm (X,2)=sum (abs (V).^2)^ (1/2)$`
`norm (X,Inf)= max (abs (V))`
`norm (X,-Inf)= min (abs (V))`

By default:
`Y=norm ( A ) = norm ( A,2 )`

The implementation uses LAPACK's `svd` function.

`S=num2str ( X [,fm])`

`void tmcnum2str` (*long nout,long ninput,tmsMatrix *sbuf, tmsMatrix *x, tmsMatrix *fm*) [Function]
Returns convert a number to string.
`S=num2str ( X ,'format' )` call standart C-function `sprintf (format,X)`.
`S=num2str ( X [ , digits] )` call `sprintf` using fixed format.
Length of the output string *S* is limited (`MAX_PRINTF_LEN= 5000`).

`S=numel ( X )`

`void tmcnumel` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *x*) [Function]
Returns number of elements in matrix or cell array X.

`Y=ones ( M ,N )`

`void tmcones` (*long nout,long ninput,tmsMatrix *Y, tmsMatrix *in1,tmsMatrix *in2*) [Function]

Return 2D matrix Y with dimensions given by M,N. All elements are 1.
`Y=ones (m)` has dimensions (m,m)
`Y=ones ([m,n])=ones (m,n)` have dimensions (m,n)

`[ne,pe]=orderfields ( S1 ,S2 )`

`void tmcorderfields` (*long nout,long ninput, tmsMatrix *ne, tmsMatrix *pe, tmsMatrix *S1,tmsMatrix *S2*) [Function]

NA, produces a warning

`pause ( secTimeout )`

`void tmcpause` (*long nout,long ninput,tmsMatrix *ydummy,tmsMatrix *d*) [Function]

Pauses the execution for *secTimeout* seconds (if supported by host operation system).

`Y=pi`

`void tmcpi` (*long nout,long ninput,tmsMatrix *dest*) [Function]

Return PI constant 3.14159265358979323846264338327 95 ....

`h=plot (x,y,c [, ...])`

`void tmcplot` (*long nout,long ninput, tmsMatrix *hand, tmsMatrix *x, tmsMatrix *y, tmsMatrix *c,...*) [Function]

NA, produces a warning

`Y=polyval (P,X)`

`void tmcpolyval` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *p,tmsMatrix *x*) [Function]

Evaluate a polynomial with coefficients P at argument X. `deg (P)=length (P)-1`.

`Y=prod (X)`

`void tmcprod` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Calculate elements product:
For vectors, `prod (X)` is the product of the elements of *X*.
For matrices, `prod (X)` is a row vector with the product of each column elements.

`[Q,R,E]=qr ( A)`

`void tmcqr` (*long nout,long ninput,tmsMatrix *Q,tmsMatrix *R,tmsMatrix *E,tmsMatrix *A,tmsMatrix *flg*) [Function]
computes a QR factorization of a matrix *A*
Uses LAPACK's `ZGEQRF` function .

`[Y]=real ( X)`

`void tmcreal` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Get real part of complex matrix *X*.

`Y=rem ( A, B )`

`void tmcrem` (*long nout,long ninput,tmsMatrix *matres,tmsMatrix *src1,tmsMatrix *src2*) [Function]
Return remainder after division *A* by *B*.
The floating-point remainder Y of A / B is such that `A = q * B + Y` , where `q` is an integer, *Y* has the same sign as *A*, and `abs (Y)< abs (B)`. Implemented as by-element `x-fix (x/y)*y` by C-function `fmod`.
By convension, `rem (A,0)=NaN`.
Result has the same sign as *A*

`[Y]=rmfield ( S, fn)`

`void tmcrmfield` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *S, tmsMatrix *m_fn*) [Function]
Remove field with name given by string *fn* from struct *S*. Returns the updated structure.

`[R]=roots ( P)`

`void tmcroots` (*long nout,long ninput,tmsMatrix *r,tmsMatrix *p*) [Function]
Compute roots of a the polynomial defined by its coefficients *p*
Uses LAPACK's `DGEEVX/ZGEEVX` functions.

`[Y]=round ( X)`

`void tmcround` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Round values of matrix X to nearest integer.

`Y=save ( fn,Mat1 [... ,MatN] )`

`void tmcsave` (*long nout,long ninput,tmsMatrix *filename,tmsMatrix* [Function]
**Mat1,char *varname1,...*)
Save variables *Mat1*, ... ,*MatN* into MAT-file with name *fn*. Only MATLAB MAT-file format V5 is supported.
In C-function should pass also the names of the variables as character strings. `ninput` should be number of saved matrices plus one.

`s=set (obj,pt,pv,...)`

`void tmcset` (*long nout,long ninput,tmsMatrix *stat,tmsMatrix* [Function]
**obj,tmsMatrix *pt,tmsMatrix *pv, ...*)
NA, produces a warning

`[Y,I]=setdiff (A,B [,rs])`

`void tmcsetdiff` (*long nout,long ninput, tmsMatrix *y, tmsMatrix* [Function]
**I,tmsMatrix *A,tmsMatrix *B,tmsMatrix *rs*)
Find elements in matrix A than are not present in matrix B. The result is sorted.

`[Y]=setfield ( S, fn,V)`

`void tmcsetfield` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *S,* [Function]
*tmsMatrix *m_fn,tmsMatrix *v*)
Assign for struct S to the field with name given by string *fn* new value V . Returns the updated structure.

`[Y]=sign ( X)`

`void tmcsign` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Return sign of values of struct X.

`Y=sin ( X)`

`void tmcsin` (*long nout,long ninput,tmsMatrix *matres,tmsMatrix *x*) [Function]
Calculate sine of X. For complex argument z
`sin (z)= (e ^a +e^ ( -a)) sin (b)/2 + i* ( e^ (- a) -e ^a) cos (b)/2`
where `a=-imag (z)` , `b= real (z)`.

`[m,n,k,...]=size ( X[,dim])`

`void tmcsize` (*long nout,long ninput,tmsMatrix *out1,...*) [Function]
Return size of variable *X* dimensions:
`n=size (X)` returns vector of dimensions
`n=size (X,dim)` returns size of dimension *dim*
`[m,n]=size (X)` return sizes of 2D-matrix
`[m,n,k]=size (X)` return sizes of 3D-matrix

`[Y,I]=sort ( X[,c])`

`void tmcsort` (*long nout,long ninput, tmsMatrix *y1,tmsMatrix *y2,tmsMatrix *x,tmsMatrix *c*) [Function]
Sort vector *X*
`[Y,I]=sort ( X)` sorts ascending in order
`[Y,I]=sort ( X,'descent')` sorts in descending order
Imaginal part is ignored in the numeric comparison.

`S=sprintf ( fmt,... )`

`void tmcsprintf` (*long nout,long ninput, tmsMatrix *sbuf,tmsMatrix *fm,...*) [Function]
Print formatted string to string *S*. Behavours like C-library `sprintf` but has some limitations.

`[Y]=sqrt ( X)`

`void tmcsqrt` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Return square root of *X*.
For complex numbers: `y = sqrt (abs (x))* (cos (angle (x)*0.5)+j*sin (angle (x)*0.5))`

`[Y]=squeeze ( X)`

`void tmcsqueeze` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]
Remove external dimensions from multi-dimension marix if they are of length 1. 2-D arrays are unaffected.

`y=ss ( ... )`

`void tmcss` (*long nout,long ninput, ...*) [Function]
NA, raises an error

`y=ss2tf ( ... )`

`void tmcss2tf` (*long nout,long ninput,...*) [Function]
NA, raises an error

`y=ssdata ( ... )`

`void tmcssdata` (*long nout,long ninput,...*) [Function]
NA, raises an error

`Y= str2num ( X)`

`void tmcstr2num` (*long nout,long ninput,tmsMatrix *y,tmsMatrix *x*) [Function]
Parse string *X* and convert to numeric array.
Only real vectors are supported.

`Y= strcmp ( S1, S2)`

`void tmcstrcmp` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *s1,tmsMatrix *s2*) [Function]
Compare strings.
If *S1* = *S2* then return 1, otherwise return 0.
If *S1*, *S2* are not strings , return 0.

`K= strfind ( sTEXT, sPATTERN)`

`void tmcstrfind` (*long nout,long ninput,tmsMatrix *K,tmsMatrix *sTEXT, tmsMatrix *sPATTERN*) [Function]
Find starting indexes of any occurrences of the string *sPATTERN* in the string *sTEXT*.

`[Y]=struct ( [fn,V, ...])`

`void tmcstruct` (*long nout,long ninput,tmsMatrix *matres, ...*) [Function]
Create a structure.
No array of structure is created, even if V is not scalar.

```
h=subplot (mM,...)
```

`void tmcsubplot` (*long nout,long ninput, tmsMatrix *hand, tmsMatrix *mM, ...*) [Function]
NA, produces a warning

```
[ Y ]= sum ( X )
```

`void tmcsum` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x*) [Function]
Find sum:
If X is matrix, returns a row with the sums of each its columns.
If X is vector, returns sum of its elements.

```
[U,S,V]=svd ( X[,flag] )
```

`void tmcsvd` (*long nout,long ninput,tmsMatrix *U,tmsMatrix *S,tmsMatrix *V,tmsMatrix *X,tmsMatrix *flag*) [Function]
Computes a singular value decomposition (SVD) of matrix A
Uses LAPACK's `DGESVD/ZGESVD` functions .

```
Y=tan ( X)
```

`void tmctan` (*long nout,long ninput,tmsMatrix *matres,tmsMatrix *x*) [Function]
Calculate tan of X. For complex argument z
`tan (z)= sin (z)/cos (z)`

```
h=text (mM,x1,...)
```

`void tmctext` (*long nout,long ninput, tmsMatrix *hand, tmsMatrix *x1, ...*) [Function]
NA, produces a warning

```
y= tf ( ... )
```

`void tmctf` (*long nout,long ninput,...*) [Function]
NA, raises an error

`y=tmctf2ss ( ... )`

`void tmctf2ss` (*long nout,long ninput,...*) [Function]
NA, raises an error

`y=tmctfdata ( ... )`

`void tmctfdata` (*long nout,long ninput,...*) [Function]
NA, raises an error

`h=title (mM,str,...)`

`void tmctitle` (*long nout,long ninput, tmsMatrix *hand,tmsMatrix *str, ...*) [Function]
NA, produces a warning

`[Y ]=unique ( X )`

`void tmcunique` (*long nout,long ninput, tmsMatrix *y, tmsMatrix *I,tmsMatrix *J,tmsMatrix *mx*) [Function]
Return unique values of vector *X* sorted in ascending order.

`[Y ]=unwrap ( X, [mrange )`

`void tmcunwrap` (*long nout,long ninput, tmsMatrix *y,tmsMatrix *x,tmsMatrix *mrange*) [Function]
Unwrap vector *X* that was wrapped by modulo *mrange*. If *mrange* is ommitted, asumed `mrange=PI`
A matrix is considered as a column

`[h ]=waitbar ( frac, title, pt1, pv1, pt2, pv2 )`

`void tmcwaitbar` (*long nout,long ninput,tmsMatrix *hand,tmsMatrix *frac, tmsMatrix *title,tmsMatrix *pt1,tmsMatrix *pv1,tmsMatrix *pt2,tmsMatrix *pv2*) [Function]
Host platform-specific callback for displaying calculation status (if supported). *title* is a message title and *frac* is a progress fraction to be displayed.

`warning (srting1, string2)`

`void tmcwarning` (*long nout,long ninput, tmsMatrix *msg_string,tmsMatrix *msg_string2*) [Function]
NA, ignored

`h=xlabel (h,title)`

`void tmcxlabel` (*long nout,long ninput,tmsMatrix *hand, tmsMatrix *title*) [Function]
NA, produces a warning

`h=ylabel (h,title)`

`void tmcylabel` (*long nout,long ninput,tmsMatrix *hand, tmsMatrix *title*) [Function]
NA, produces a warning

`Y=zeros ( M ,N )`

`void tmczeros` (*long nout,long ninput,tmsMatrix *Y, tmsMatrix *in1,tmsMatrix *in2*) [Function]
Return 2D matrix Y with dimensions given by M,N. All elements are 0.
`Y=zeros (m)` has dimensions (m,m)
`Y=zeros ([m,n])=zeros (m,n)` have dimensions (m,n)

`y=tmctfdata ( ... )`

`void tmctfdata` (*long nout,long ninput,...*) [Function]
NA, raises an error

`y=zpk ( ... )`

`void tmczpk` (*long nout,long ninput,...*) [Function]
NA, raises an error

`y=zpkdata ( ... )`

`void tmczpkdata` (*long nout,long ninput,...*) [Function]
NA, raises an error

6.4 Internal functions

These functions are called by the code generated by TMC Compiler. User rare should call to this functions. They are listed here for reference for better understanding of the generated code.

Most of these functions assume that the matrix passed for output parameter is already initialized but it may contain a result of the previous operation. Thus the matrix is reallocated inside the function before it usage.

6.4.1 Operations

The section describes the functions that implements basic operations.

`operation (+)`

`void tmcAdd` (*tmsMatrix * sum, tmsMatrix * a , tmsMatrix * b*) [Function]

`operation (&&) (Short-circuit logical and, for scalars)`

`void tmcAndBoolean` (*tmsMatrix *res, tmsMatrix *a, tmsMatrix *b*) [Function]

`operation (&) (logical and)`

`void tmcAndScalar` (*tmsMatrix *res,tmsMatrix *a,tmsMatrix *b*) [Function]

`operation (=) (assignment)`

`void tmcAssign` (*tmsMatrix *dest,tmsMatrix *src*) [Function]

`operation (/) (right matrix divide)`

`void tmcDiv` (*tmsMatrix *X,tmsMatrix *A,tmsMatrix *B*) [Function]

Performs `X=A/B`. Uses LAPACK functions `DGELS/ZGELS`.

`operation (./) (right array divide)`

`void tmcDivScalar` (*tmsMatrix *X,tmsMatrix *A,tmsMatrix *B*) [Function]

`operation (\) (left matrix divide)`

`void tmcLeftDiv` (*tmsMatrix *X,tmsMatrix *A,tmsMatrix *B*) [Function]

`operation (*) (matrix multiply)`

`void tmcMul` (*tmsMatrix *prod,tmsMatrix *a,tmsMatrix *b*) [Function]

`operation (.*) (array multiply)`

`void tmcMulScalar` (*tmsMatrix *prod,tmsMatrix *a,tmsMatrix *b*) [Function]

`operation (-) (unary minus)`

`void tmcNeg` (*tmsMatrix *sum,tmsMatrix *x*) [Function]

`operation (~) (logical not)`

`void tmcNot` (*tmsMatrix *matres,tmsMatrix *src*) [Function]

`operation (|) (logical or)`

`void tmcOrScalar` (*tmsMatrix *res,tmsMatrix *a,tmsMatrix *b*) [Function]

`operation (||) (Short-circuit logical or, for scalars)`

`void tmcOrBoolean` (*tmsMatrix *res,tmsMatrix *a,tmsMatrix *b*) [Function]

`operation (^) (matrix power)`

`void tmcPower` (*tmsMatrix *matres,tmsMatrix *src1,tmsMatrix *src2*) [Function]

`operation (.^) (array power)`

`void tmcPowerScalar` (*tmsMatrix *matres,tmsMatrix *src1,tmsMatrix *src2*) [Function]

`operation (-)`

`void tmcSub` (*tmsMatrix *res,tmsMatrix *a,tmsMatrix *b*) [Function]

`operation (') (complex conjugate transpose)`

`void tmcTranspose` (*tmsMatrix *res,tmsMatrix *src*) [Function]
Performs transpose operation with complex conjugation: `res=src'`.

`operation (.') (complex conjugate transpose)`

`void tmcTransposeScalar` (*tmsMatrix *res,tmsMatrix *src*) [Function]
Performs non-conjugate transpose operation: `res=src.'`.

Comparisons

Evaluate a comparison: returns matrix *sum* of the same dimension as both *a* and *b* with `1` at the positions of true comparison and `0` at others.

`comparison (==) (equal)`

`void tmcEq` (*tmsMatrix *sum,tmsMatrix *a,tmsMatrix *b*) [Function]

`comparison (>=) (not equal)`

`void tmcGe` (*tmsMatrix *sum,tmsMatrix *a,tmsMatrix *b*) [Function]

`comparison (>) (not equal)`

`void tmcGt` (*tmsMatrix *sum,tmsMatrix *a,tmsMatrix *b*) [Function]

`comparison (<=) (not equal)`

`void tmcLe` (*tmsMatrix *sum,tmsMatrix *a,tmsMatrix *b*) [Function]

`comparison (<) (not equal)`

`void tmcLt` (*tmsMatrix *sum,tmsMatrix *a,tmsMatrix *b*) [Function]

`comparison (~=) (not equal)`

`void tmcNe` (*tmsMatrix *sum,tmsMatrix *a,tmsMatrix *b*) [Function]

Indexing Implements set/get for matrix elements given by indexes. Return a matrix. For cell array returns a single matrix given by single index selection.

`index in matrix (I1,...) (get)`

`void tmcGetByIndex` (*tmsMatrix *matres,tmsMatrix *src,long numdims,tmsMatrix *I1,...*) [Function]

`index in cell array {I1,... } (get)`

`void tmcGetByIndexCell` (*tmsMatrix *matres,tmsMatrix *src,long numdims,tmsMatrix *I1,...*) [Function]

`index in matrix (I1,...) (set)`

`void tmcGetRefByIndex` (*tmsMatrix *matres,tmsMatrix *src,long numdims,tmsMatrix *I1,...*) [Function]

`index in cell array I1,... (set)`

`void tmcGetRefByIndexCell` (*tmsMatrix *matres,tmsMatrix *src,long numdims,tmsMatrix *I1,...*) [Function]

`last index (end)`

`void tmcGetEnd` (*tmsMatrix *matres,tmsMatrix *src,long dim,long numdims*) [Function]

This function has so far some restrictions in its implementations. Only syntax like

```
array(m:n:end)
```

or

```
Structure.array(m:n:end)
```

is implemented.

Access struct fields

Implements set/get for structure fields.
The field name is accessed by its '`hash-code`' that is registered in the global strings hash table. The hash table is initialized at the TMC Library initialization by the values accepted during the m-code parsing. Thus for correct functioning all the accessed field names should be explicitly refered from the code. If an unknown field name is created dynamically or e.g. created by `tmcload` function, an error may occured. The issue of string hashing should be descussed in a separate section.

(*Structure.fieldname*) (get)

`void tmcGetByFieldHcode` (*tmsMatrix *matres, tmsMatrix *src, STRINGCODE hcode*) [Function]

(*Structure.fieldname*) (set)

`void tmcGetRefByFieldHcode` (*tmsMatrix *matres,tmsMatrix *src,STRINGCODE hcode*) [Function]

6.4.2 Internal utils

These functions are used only in the generated code.

`void tmcAssignBool` (*tmsMatrix *dest,tmsMatrix *src*) [Function]
Performs assignment to dest: 1 if all src elements are non-zero, otherwise 0

`void tmcCalcSwitchExpVal` (*tmsMatrix *exprcode,tmsMatrix *x*) [Function]
Switch-case operator support. Get numeric code for string or double expression of SWITCH.

`void _tmcClearRegister` (*tmsMatrix *x*) [Function]
Main tmsMatrix destructor. Clears *x* matrix before re-usage.

`void tmcCollectCellColumns` (*tmsMatrix *colres,long numcols,tmsMatrix *a,...*) [Function]
Collect a set of matrices into a cell array (row)
Arguments:
colres: result cell array
numcols: number of columns to be collected
a,...: matrices to be collected

`void tmcCollectCellRows` (*tmsMatrix *matres,long numrows,tmsMatrix *a,...*) [Function]

Collect some rows of matrices into a cell array.
Arguments:
matres: result cell array
numrows: number of rows to be collected
a,...: matrices to be collected. Must be cell arrays.

`void tmcCollectColumns` (*tmsMatrix *colres,long numcols,tmsMatrix *a,...*) [Function]

Collect a set of matrices into an array (row)
Arguments:
colres: result row array
numcols: number of columns to be collected
a,...: matrices to be collected

`void tmcCollectRows` (*tmsMatrix *matres,long numrows,tmsMatrix *a,...*) [Function]

Collect some rows of matrices into a matrix array.
Arguments:
matres: result matrix array
numrows: number of rows to be collected
a,...: matrices to be collected.

`void tmcComplexScalar` (*tmsMatrix *dest,double xr,double xi*) [Function]

Creates complex scalar matrix dest with real part xr and imagine part xi.
Arguments:
xr: real part
xi: imagine part dest: result

`void tmcCopyMat` (*tmsMatrix *des,tmsMatrix *src*) [Function]

Copy a matrix into another initialized matrix.
Arguments:
des: destination matrix
src: source matrix

`void _tmcCreateCellArray` (*tmsMatrix *res,long M,long N*) [Function]

Create a cell array matrix.
Arguments:
res: initialized destination matrix
M: number of rows
N: number of columns

`void tmcCreateCellEmpty` (*tmsMatrix *matres*) [Function]

Create an empty cell array matrix.
Arguments:
matres: initialized destination matrix

`void tmcCreateColonBaseIncLimit` (*tmsMatrix *matres,tmsMatrix *base,tmsMatrix *increment,tmsMatrix *limit*) [Function]

Create matrix `[base:increment:limit]`
Arguments:
matres: initialized destination matrix
base: base of matrix
increment : increment of matrix
limit : limit of matrix

`void tmcCreateColonBaseLimit` (*tmsMatrix *matres,tmsMatrix *base,tmsMatrix *limit*) [Function]

Create matrix `[base:limit]`
Arguments:
base: base of matrix
limit : limit of matrix

`void tmcCreateMagicColon` (*tmsMatrix *magcolM*) [Function]

Create matrix for internal presentation of colon operation `(:)`
Arguments:
magcolM: destination matrix

`void _tmcCreateMatrix` (*tmsMatrix *res,long M,long N,short bHasImagine*) [Function]

Create a numeric matrix array.
Arguments:
res: initialized destination matrix
M: number of rows
N: number of columns
bHasImagine: presence of imagine part flag `(tmcCOMPLEX=1 or tmcREAL=0)`

`void tmcCreateMatrixEmpty` (*tmsMatrix *matres*) [Function]

Create an empty matrix array.
Arguments:
matres: initialized destination matrix

`tmsMatrix** tmcCreateRegFrame` (*long len*) [Function]

Create temporary array of matrixes.
Arguments:
len : number of variables to create

`void tmcCreateString` (*tmsMatrix *matres,const char *str*) [Function]

Create a matrix of string type
Arguments:
matres: result
str: character zero-terminated string

`void tmcCreateStringEmpty` (*tmsMatrix *matres*) [Function]

Create an empty matrix of string type
Arguments:
matres: result

`void tmcDisplayMat` (*tmsMatrix *x,short bVerb*) [Function]
Display matrix.
Arguments:
x: matrix to be displayed
bVerb: verbose flag (0: compact printing)

`void tmcFncHandle` (*tmsMatrix *dest,void (*fncptr)(long,long,...),const char *nm*) [Function]
Implements initialization of function reference to a matrix `Y=@F`.
Arguments:
dest: destination matrix
fncptr: assigned function pointer
nm: function name

`void tmcForIterInit` (*tmsMatrix *iteratorM,tmsMatrix *rangeM,tmsMatrix *iteratorvariableM*) [Function]
Implements initialization of iterator type matrix *iteratorM* in '`for`' command by connecting it with iterator variable *iteratorvariableM*. Matrix *iteratorvariableM* is cleared.
Arguments:
iteratorM: destination iterator reference matrix
rangeM: reserved
iteratorvariableM: iterator of for-loop

`short tmcForIterNext` (*tmsMatrix *iteratorM,tmsMatrix *rangeM*) [Function]
Implements increment of for-loop iterator
Arguments:
iteratorM: destination iterator reference matrix
rangeM: range in for command
Iterator variable must be matrix (not cell etc.)
See also: `tmcForIterInit`

`short tmcFreeLib` (*void*) [Function]
Uninitialize TMC run-time (graphics, reference helpers, global variables, string hash, exception handling) Arguments:
NA

`void tmcFreeLocalVar` (*tmsMatrix *src*) [Function]
Clear and destroy temporary variables at function return.
Arguments:
src: matrix to be destroyed

`void tmcFreeRegFrame` (*tmsMatrix **reg*) [Function]
Destroy temporary matrix array.
Argume; nts:
reg : pointer to array to be destroyed.

`tmsMatrix* _tmcGetField` (*tmsMatrix *S,long ind,const char *fname*) [Function]
Get a struct field given by name as a string. Arguments:
S: returned matrix
ind: reserved
fname: field name

`short _tmcGetFieldNumber` (*tmsMatrix *src,const char *fn*) [Function]
Get a struct field order number given by fieled name as a string. Arguments:
src: struct matrix
fn: field name
Return: field number

`short _tmcGetString` (*const tmsMatrix *src , char *str_des , long maxlen*) [Function]
Copy string from string matrix *src* into buffer *str_des* and null-terminate it. If the matrix length is larger than *maxlen*-1, the string is truncated. Arguments:
src: struct matrix
str_des: destination buffer
maxlen: size of destination buffer. Return: 0

`Y=pi`

`void tmcpi` (*long nout,long ninput,tmsMatrix *dest*) [Function]
Return PI constant 3.14159265358979323846264338327 95 . . . .

`void tmci` (*long nout,long ninput,tmsMatrix *dest*) [Function]
Return `i` constant

`void tmcinf` (*long nout,long ninput,tmsMatrix *dest*) [Function]
Return `Inf` constant

`short tmcInitLib` (*const struct CInit_funcs_table * pInit_funcs_table*) [Function]
Initialize TMC run-time. Parameter `pInit_funcs_table` should pass to the initialization function the pointer `&Init_funcs_table` to the global table that is generated by TMC compiler.

`short tmcIsCaseDouble` (*tmsMatrix *expr_code,double x*) [Function]
Returns 1 if *expr_code* equals *x* else returns 0

`short tmcIsCaseString` (*tmsMatrix *expr_code, STRINGCODE n*) [Function]
Returns 1 if *expr_code* contain a string with hash-code *n* else returns 0

`short tmcIsFalse` (*tmsMatrix *x*) [Function]
Returns 1 if *x* has a zero element or empty, otherwise returns 0.

`void tmcIsFieldHcode` (*tmsMatrix *matres,tmsMatrix *src,STRINGCODE hcode*) [Function]
Returns `matres=1` struct *src* has a field with hash-code *hcode*, otherwise returns 0.
May be called without _tmcClearRegister for *matres*.

`short tmcIsTrue` (*tmsMatrix *x*) [Function]
Returns 1 if *x* has all non zero elements and empty, otherwise returns 0.

`void tmcj` (*long nout,long ninput,tmsMatrix *dest*) [Function]
Return `i` constant. The same like `tmci`

`char* _tmcMat2String` (*tmsMatrix *src*) [Function]
Create char buffer containing string *src* and returns pointer to it. The pointer must be free by caller.

`void tmcNaN` (*long nout,long ninput,tmsMatrix *dest*) [Function]
Return `NaN` constant

`tmsMatrix* __tmcNewMatrix` (*void*) [Function]
Create empty matrix and returns pointer to it. The matrix must be free by caller using `tmcFreeLocalVar`.

`short tmcNotCase` (*tmsMatrix *expression,tmsMatrix *case_value*) [Function]
Return 0 is case *expression* equals to *case_value*, otherwise return 1

`long tmcNumElem` (*tmsMatrix *x*) [Function]
Returns number of matrix *x* elements.

`void _tmcRaiseException` (*long errcode,const char *module_name,const char *func_name,const char *errmsg,long numargin,tmsMatrix *x,...*) [Function]
Raises an exception and terminate the program execution.

`void tmcReallocRegister` (*tmsMatrix *src*) [Function]
Main tmsMatrix destructor. Clears x matrix before re-usage. Arguments:
See: _tmcClearRegister

`void tmcScalar` (*tmsMatrix *dest,double x*) [Function]
Creates scalar matrix dest with real value x.
Arguments:
x: real value
dest: result

`void tmcSyntaxError` (*const char *msg*) [Function]
Raise run-time exception. Arguments:
msg: message string

6.4.3 Debugging features

`HANDLE tmcconnectdebugger` (*long pass*) [Function]
Initialize application possibility to be connected to TMC Debugger.
Returns: handle of the window used by TMC debugger. Arguments:
pass: protection code (by default is 1)

`void tmcdbgCloseDebugger` (*void*) [Function]
Un-Initialize application debugging.
Arguments:
none

`long tmcdbgCommonMemConnect` (*void** ptr*) [Function]
Connect to TMC debugger file-mapping.
Arguments:
ptr: pointer to mapping view

`void tmcdbgOpenDebugger` (*void*) [Function]
Initialize application debugging.
Arguments:
none

`void tmcdbgPopStackVar` (*short nVars*) [Function]
Free debugging frame from the variables before the function return.
Arguments:
nVars: number of variables to remove

`void tmcdbgPushStackVar` (*const char *fncname,short nVars,tmsMatrix* var1,const char *varname1,...*) [Function]
Put variables with their names into debugging frame
Arguments:
fncname: currently entered function name
nVars: number of variables to put
var1,varname1,..: pairs matrix and its name

6.4.4 MEX function support

If your project contains MEX-functions with their source available, you should make a number of corrections to the source in order to get it compiled with TMC-compiled application.

The MATLAB MEX-function that has the prototype

```
void mexFunction ( int nlhs, mxArray *plhs[], int nrhs,
const mxArray*prhs[])
```

should be replaced by mex-tmc-function with prototype

```
 void tmcFuncName (long nlhs,long nrhs,tmsMatrix *lhsMatrix1,...
tmsMatrix *lhsMatrixM, tmsMatrix *rhsMatrix1,..tmsMatrix *rhsMatrixN)
```

Any reference to left-hand-side arguments

```
plhs[K]
```

should be replaced by refernce to

```
lhsMatrixK
```

A principle difference between MEX and TMC calling convention is that that in TMC the output argument matrix should be created before calling a function while in MATLAB MEX the variable is created inside the function itself.
Thus, i.e., the code

```
plhs[0] = mxCreateDoubleMatrix(m , n  , mxREAL );
```

should be replaced by the following:

```
TMCMEX_CREATE_DOUBLE_MATRIX(lhsMatrix1,m, n , mxREAL );
```

that is expanded to

```
_tmcCreateMatrix(lhsMatrix1, m , n , mxREAL )
```

Macroses for supported external interface functions The following macroses have the same prototype as the corresponding functions in MATLAB MEX and do not demand the change of user source code.

`void mexErrMsgTxt` (*const char *message*) [Function]
Display error message and raise an exception.

`void mexPrintf` (*const char *message*) [Function]
Display string message.

`char* mxArrayToString` (*tmsMatrix *mX*) [Function]
Creates zero-terminated char buffer initialized by the string stored in matrix mX. Returns pointer to the string that must be freed by caller.

`tmsMatrix* mxGetCell` (*tmsMatrix *mX,int k*) [Function]
Returns *k*-th zero-based element of cell array matrix *mX*.

`int mxGetM` (*const tmsMatrix *mX*) [Function]
Return number of rows of matrix mX.

`int mxGetN` (*const tmsMatrix *mX*) [Function]
Return number of columns of matrix mX.

`double* mxGetPi` (*const tmsMatrix *mX*) [Function]
Return pointer to imagine part data of matrix mX.

`double* mxGetPr` (*const tmsMatrix *mX*) [Function]
Return pointer to real part data of matrix mX.

`int mxGetString` (*const tmsMatrix *mX,const char *S,int len*) [Function]
Create matrix of string type. Returns zero on success.

`void mxFree` (*tmsMatrix *mX*) [Function]
Free matrix memory. The matrix itself is not destroyed.

`void mxSetCell` (*tmsMatrix *mX,int k,tmsMatrix *mA*) [Function]
Assign a matrix mA to k-th zero-based element of cell array matrix mX.
Arguments:
mX: cell array matrix to be modified
k: element index to assign
mA: matrix to be assigned

`tmsMatrix* mxGetField` (*tmsMatrix *mX,int k,const char *fieldname*) [Function]
Return field with name fieldname of k-th zero-based element of structure matrix mX.
Arguments:
mX: structure array matrix
k: element index of structure array (reserved, must be 0)
fieldname: field name

`int mxGetFieldNumber` (*tmsMatrix *mX,const char *fieldname*) [Function]
Return field number 0-based index of a field with name fieldname of structure matrix mX. If the field does not exist returns -1.
Arguments:
mX: structure array matrix
fieldname: field name

`int mxGetNumberOfElements` (*tmsMatrix *mX*) [Function]
Return number of elements in array matrix mX.
Arguments:
mX: structure array matrix

`int mxIsChar` (*tmsMatrix *mX*) [Function]
Return 1 if the matrix array mX has string type.
Arguments:
mX: array matrix

`int mxIsComplex` (*tmsMatrix *mX*) [Function]
Return 1 if the matrix array mX has imagine part.
Arguments:
mX: array matrix

`int mxIsEmpty` (*tmsMatrix *mX*) [Function]
Return 1 if the matrix array mX has no elements.
Arguments:
mX: array matrix

`int mxIsStruct` (*tmsMatrix *mX*) [Function]
Return 1 if the matrix array mX is of structure type.
Arguments:
mX: matrix

TMC MEX Macroses for initialization of output arguments

`TMCMEX_CREATE_CELL_MATRIX` (*tmsMatrix *mX,int M,int N*) [Macro]
Initialize 2-dim cell array matrix. Each cell should be assigned by mxSetCell.
Matrix *mX* should be a MEX function argument created by caller.
Arguments:
mX: initialized matrix
M: number of rows
N: number of columns

`TMCMEX_CREATE_DOUBLE_MATRIX` (*tmsMatrix *mX,int M,int N,short tmcType*) [Macro]
Initialize 2-dim matrix.
Matrix *mX* should be a MEX function argument created by caller.

Arguments:
mX: initialized matrix
M: number of rows
N: number of columns
tmcType: matrix type (mxREAL=0,mxCOMPLEX=1)

`TMCMEX_NEW_DOUBLE_MATRIX` (*tmsMatrix *mX,int M,int N,short tmcType*) [Macro]
Create and initialize 2-dim matrix.
Matrix *mX* should be an uninitalized local variable.
Arguments:
mX: initialized matrix
M: number of rows
N: number of columns
tmcType: matrix type (mxREAL=0,mxCOMPLEX=1)

6.4.4.1 MEX example

```
/*==========================================================
 * function Y = ExMex1(X1,X2);
 example function
Y= X1+X2
 *==========================================================*/

#include <math.h>
#include "mex.h"

#define TMCMEX_DLL // must be defined for TMC and undefined for MATLAB mex compilation


#ifndef TMCMEX_DLL

#include "mexport.h"
#define mY plhs[0]
#define mX1  prhs[0]
#define mX2  prhs[1]

void mexFunction( int nlhs, mxArray *plhs[],
int nrhs, const mxArray*prhs[] )

#else
#define nrhs nargin
#define nlhs nargout
void tmcExMex1(int nargout, int nargin,tmsMatrix *mY,
 tmsMatrix *mX1,tmsMatrix *mX2)
#endif
```

```
{
    int mp, np;
    int  ind;
    double * X1_rPtr;
    double * X1_iPtr;
    double * Y_rPtr;
    double * Y_iPtr;
    double * X2_rPtr;
    double * X2_iPtr;

    /* Check for proper number of arguments */
    if (nrhs != 2)
    {
        mexErrMsgTxt("Two input arguments required.");
    }
    if (nlhs > 1) {
        mexErrMsgTxt("One output argument required.");
    }
    /* Check the dimensions */
mp = mxGetM(mX1); /* Number of rows in the first input argument */
    np = mxGetN(mX2); /* Number of rows in the first input argument */
    if ( mp != np )
    {
        mexErrMsgTxt("Matrix dimensions do not match.");
    }
    /* Create a matrix for the return argument */
    if ( mxGetPi ( mX1 ) == NULL && mxGetPi ( mX2 ) == NULL)
    {
        TMCMEX_CREATE_DOUBLE_MATRIX(mY,mp, np , mxREAL );
    }
    else
    {
        TMCMEX_CREATE_DOUBLE_MATRIX(mY,mp, np , mxCOMPLEX );
    }
    X1_rPtr = mxGetPr ( mX1 );
    X1_iPtr = mxGetPi ( mX2 );
    X2_rPtr = mxGetPr ( mX2 );
    X2_iPtr = mxGetPi ( mX2 );
    Y_rPtr = mxGetPr ( mY  );
Y_iPtr = mxGetPi ( mY  ) ;
if ( X1_iPtr == NULL )
{
if ( ( Y_iPtr != NULL ) && ( X2_iPtr != NULL ) )
{
for ( ind = 0 ; ind < mp ; ind++ )
{
                    Y_rPtr[ ind ] = X2_rPtr[ ind ] + X1_rPtr[ ind ];
                    Y_iPtr[ ind ] = X2_rPtr[ ind ] ;
            }
}
else
        {
for ( ind = 0 ; ind < mp ; ind++ )
{
                    Y_rPtr[ ind ] = X2_rPtr[ ind ] + X1_rPtr[ ind ];
}
        }
```

7 TMC Code debugging

The simplest way of the code debugging is the usage of `save` function to save intermediate variables. Too additional tools are provided (for MS Windows):

- TMC Debugger.

  This tool (tmcdbgW.exe) enables to view variables that are stored in the debugging frame by `tmcdbgPushStackVar` function. Calls to this function are generated by TMC Converter if it was called with switch `-d`. The application should call `tmcdbgOpenDebugger` function at initialisation and be stopped by a debugger (GDB or other one, depending on the host compiler).

  Note, that call

  ```
  call tmcdisp(0,1,0,variable)
  ```

  or

  ```
  call tmcdisp(0,1,0,address)
  ```

  during GDB debugging session should print the value of the intermediate *variable* (like reg[n]). In the following `GDB` session aplication `Ex1_w32_shared.exe` is loaded for debugging, a breakpoint is set at the entry to function `tmcmyeq` (generated from `myeq`) and the application run. Then execution is stopped at the breakpoint, some steps are performed and then variable `reg[5]` is displayed. The the address of local variable x is requested and the variable x is printed using its address.

  ```
  >gdb .\bin\Ex1_w32_shared.exe
  (gdb) b tmcmyeq
  Breakpoint 1 at 0x4013e0: file myeq.c, line 19.
  (gdb) r
  Breakpoint 1, tmcmyeq (nargout=1, nargin=1, y=0x56ae38, x=0x565708)
      at myeq.c:19
  19      ,tmsMatrix *x) {
  (gdb) s
  20      tmsMatrix **reg=tmcCreateRegFrame(25);
  (gdb) s
  19      ,tmsMatrix *x) {
  (gdb) s
  20      tmsMatrix **reg=tmcCreateRegFrame(25);
  (gdb) s
  23      TMC_DBG_PUSH_STACK_VAR("myeq",2,
  (gdb) s
  20      tmsMatrix **reg=tmcCreateRegFrame(25);
  (gdb) s
  23      TMC_DBG_PUSH_STACK_VAR("myeq",2,
  (gdb) s
  30      tmcScalar(reg[2],2.000000000000000e+000);
  (gdb) s
  31      tmcPowerScalar(reg[3],x,reg[2]);
  (gdb) s
  34      tmcReallocRegister(reg[5]);
  (gdb) s
  35      tmcsin(1,1, reg[5], x);
  (gdb) s
  36      tmcMulScalar(reg[6],reg[3],reg[5]);
  (gdb) s
  ```

```
38      tmcScalar(reg[8],3.000000000000000e+000);
(gdb) call tmcdisp(0,1,0,reg[5])
Matrix(1,1) =[
0.958924,       ;
];
$1 = 0
(gdb) p x
$2 = (tmsMatrix *) 0x565708
(gdb) call tmcdisp(0,1,0,0x565708)
Matrix(1,1) =[
-5,     ;
];
$4 = 0
```

If TMC Debugger is started at this point, it connects to the application (the appication process ID is found by its window) and reads its process memory from the debugging stack and accepts the variable addresses. Then it reads the variables from these addresses. TMC Debugger displays calling stack; when a called function from the stack is selected, a list/tree of local variables is displayed. When a local variable is selected in the tree, its content is displayed. If some of variables are structures, the corresponded symbols hash table `hash_initx.dat` should be loaded into TMC Debugger. In the current version the table is compiled statically.

- TMC Graph Viewer.

  This tool implements a graphic server that provides a minimal support for functions like `figure`, `plot`, `subplot`. This server updates the graphics even when the application is stopped at a breakpoint. The executable (tmcgra.exe) should be put in the application running directory.

  The first call to `function` or `plot` functions start the server. The data of the plot to be displayed is passed to the server by a file in a predefined format. The file is put in the current directory and display is synchronized by a window message. Actually user may use its own implementation of the graphics using this data format.

Concept Index

Function Index

L

M

N

O

P

Q

R

S

T

U

W

X

Y

Z