Writing MATLAB® C/MEX Code

Pascal Getreuer, April 2010

Contents

1 Introduction

It is possible to compile C, C++, or Fortran code so that it is callable from MATLAB. This kind of program is called a Matlab Executable (MEX) external interface function, or more briefly a "MEX-function." MEX enables the high performance of C, C++, and Fortran while working within the MATLAB environment. We will discuss C/MEX functions, which also applies directly to C++/MEX. Fortran/MEX is quite different and we do not discuss it here.

> **Warning** This is not a beginner's tutorial to MATLAB. Familiarity with C and MATLAB is assumed. Use at your own risk.

MEX is often more trouble than it is worth. MATLAB's JIT interpreter in recent versions runs M-code so efficiently that it is often times difficult to do much better with C. Before turning to MEX in an application, optimize your M-code (see my other article, "Writing Fast MATLAB Code"). MEX-functions are best suited to substitute one or two bottleneck M-functions in an application. If you replace all functions in an application with MEX, you might as well port the application entirely to C.

2 Getting Started

The following program demonstrates the basic structure of a MEX-function.

hello.c

```c
#include "mex.h"  /* Always include this */

void mexFunction(int nlhs, mxArray *plhs[],       /* Output variables */
                 int nrhs, const mxArray *prhs[]) /* Input variables  */
{
    mexPrintf("Hello, world!\n");  /* Do something interesting */
    return;
}
```

Copy the code into MATLAB's editor (it has a C mode) or into the C editor of your choice, and save it as `hello.c`.

The next step is to compile. On the MATLAB console, compile `hello.c` by entering the command

```
>> mex hello.c
```

If successful, this command produces a compiled file called `hello.mexa64` (or similar, depending on platform). Compiling requires that you have a C compiler and that MATLAB is configured to use it. MATLAB will autodetect most popular compilers, including Microsoft Visual C/C++ and GCC. As a fallback, some distributions of MATLAB come with the Lcc C compiler. Run `mex -setup` to change the selected compiler and build settings.

Once the MEX-function is compiled, we can call it from MATLAB just like any M-file function:

```
>> hello
Hello, world!
```

Note that compiled MEX files might not be compatible between different platforms or different versions of MATLAB. They should be compiled for each platform/version combination that you need. It is possible to compile a MEX file for a target platform other than the host's using the `-<arch>` option, for example, `mex -win32 hello.c`.

MATLAB comes with examples of MEX in `matlab/extern/examples`. For detailed reference, also see `matrix.h` and the other files in `matlab/extern/include`.

3 Inputs and Outputs

Of course, a function like `hello.c` with no inputs or outputs is not very useful. To understand inputs and outputs, let's take a closer look at the line

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

Here "`mxArray`" is a type for representing a MATLAB variable, and the arguments are:

C/MEX	Meaning	M-code equivalent
`nlhs`	Number of output variables	`nargout`
`plhs`	Array of `mxArray` pointers to the output variables	`varargout`
`nrhs`	Number of input variables	`nargin`
`prhs`	Array of `mxArray` pointers to the input variables	`varargin`

These MEX variables are analogous to the M-code variables `nargout`, `varargout`, `nargin`, and `varargin`. The naming "lhs" is an abbreviation for left-hand side (output variables) and "rhs" is an abbreviation for right-hand side (input variables).

For example, suppose the MEX-function is called as

```
[X,Y] = mymexfun(A,B,C)
```

Then `nlhs` = 2 and `plhs[0]` and `plhs[1]` are pointers (type `mxArray*`) pointing respectively to `X` and `Y`. Similarly, the inputs are given by `rlhs` = 3 with `prhs[0]`, `prhs[1]`, and `prhs[2]` pointing respectively to `A`, `B`, and `C`.

The output variables are initially unassigned; it is the responsibility of the MEX-function to create them. If `nlhs = 0`, the MEX-function is still allowed return one output variable, in which case `plhs[0]` represents the `ans` variable.

The following code demonstrates a MEX-function with inputs and outputs.

`normalizecols.c`

```c
/*  NORMALIZECOLS.C  Normalize the columns of a matrix
Syntax:      B = normalizecols(A)
        or   B = normalizecols(A,p)
The columns of matrix A are normalized so that norm(B(:,n),p) = 1. */
#include <math.h>
#include "mex.h"

#define IS_REAL_2D_FULL_DOUBLE(P) (!mxIsComplex(P) && \
mxGetNumberOfDimensions(P) == 2 && !mxIsSparse(P) && mxIsDouble(P))
#define IS_REAL_SCALAR(P) (IS_REAL_2D_FULL_DOUBLE(P) && mxGetNumberOfElements(P) == 1)

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    /* Macros for the ouput and input arguments */
    #define B_OUT       plhs[0]
```

```c
    #define A_IN        prhs[0]
    #define P_IN        prhs[1]
    double *B, *A, p, colnorm;
    int M, N, m, n;

    if(nrhs < 1 || nrhs > 2)  /* Check the number of arguments */
        mexErrMsgTxt("Wrong number of input arguments.");
    else if(nlhs > 1)
        mexErrMsgTxt("Too many output arguments.");

    if(!IS_REAL_2D_FULL_DOUBLE(A_IN)) /* Check A */
        mexErrMsgTxt("A must be a real 2D full double array.");

    if(nrhs == 1) /* If p is unspecified, set it to a default value */
        p = 2.0;
    else  /* If P was specified, check that it is a real double scalar */
        if(!IS_REAL_SCALAR(P_IN))
            mexErrMsgTxt("P must be a real double scalar.");
        else
            p = mxGetScalar(P_IN);  /* Get p */

    M = mxGetM(A_IN);     /* Get the dimensions of A */
    N = mxGetN(A_IN);
    A = mxGetPr(A_IN);    /* Get the pointer to the data of A */
    B_OUT = mxCreateDoubleMatrix(M, N, mxREAL); /* Create the output matrix */
    B = mxGetPr(B_OUT);   /* Get the pointer to the data of B */

    for(n = 0; n < N; n++)   /* Compute a matrix with normalized columns */
    {
        for(m = 0, colnorm = 0.0; m < M; m++) colnorm += pow(A[m + M*n], p);
        colnorm = pow(fabs(colnorm),1.0/p);  /* Compute the norm of the nth column */

        for(m = 0; m < M; m++) B[m + M*n] = A[m + M*n]/colnorm;
    }

    return;
}
```

Much of the code is spent verifying the inputs. MEX provides the following functions to check datatype, dimensions, and so on:

C/MEX	Meaning	M-code equivalent
`mxIsDouble(A_IN)`	True for a double array	`isa(A,'double')`
`mxIsComplex(A_IN)`	True if array is complex	`~isreal(A)`
`mxIsSparse(A_IN)`	True if array is sparse	`issparse(A)`
`mxGetNumberOfDimensions(A_IN)`	Number of array dimensions	`ndims(A)`
`mxGetNumberOfElements(A_IN)`	Number of array elements	`numel(A)`

The `normalizecols.c` example simplifies input parsing by combining some of these checks into a macro `IS_REAL_2D_FULL_DOUBLE`. Notice how we check `nrhs==1` to see if the function was called as `normalizedcols(A)` or `normalizedcols(A,p)`.

Another approach to input parsing is to rename this MEX-function as "`normalizecolsmx.c`" and create an M-function wrapper:

normalizecols.m

```
function B = normalizecols(A,p)
% M-function wrapper for parsing the inputs
if nargin < 2
    if nargin < 1
        error('Not enough input arguments');
    end
    p = 2;   % p is unspecified, set default value
end

if ~isreal(A) || ndims(A) ~= 2 || issparse(A) || ~isa(A, 'double')
    error('A must be a real 2D full double array.');
elseif ~isreal(p) || ~isa(p, 'double') || numel(p) ~= 1
    error('P must be a real double scalar.');
end

normalizecolsmx(A, p);   % Call the MEX-function with the verified inputs
```

M-code is much more convenient for input parsing, especially for majestic calling syntaxes like property/value pairs or option structures.

The actual dimensions and data of array `A` are obtained by

```
M = mxGetM(A_IN);      /* Get the dimensions of A */
N = mxGetN(A_IN);
A = mxGetPr(A_IN);     /* Get the pointer to the data of A */
```

Array elements are stored in column-major format, for example, `A[m + M*n]` (where $0 \le \mathtt{m} \le \mathtt{M}-1$ and $0 \le \mathtt{n} \le \mathtt{N}-1$) corresponds to matrix element `A(m+1,n+1)`.

The output `M`×`N` array `B` is created with `mxCreateDoubleMatrix`:

```
B_OUT = mxCreateDoubleMatrix(M, N, mxREAL); /* Create the output matrix */
B = mxGetPr(B_OUT);    /* Get the pointer to the data of B */
```

A double scalar can be created by setting `M = N = 1`, or more conveniently with `mxCreateDoubleScalar`:

```
B_OUT = mxCreateDoubleScalar(Value); /* Create scalar B = Value */
```

4 Numeric Arrays

The previous section showed how to handle real 2D full double arrays. But generally, a MATLAB array can be real or complex, full or sparse, with any number of dimensions, and in various datatypes. Supporting all permutations of types is an overwhelming problem, but fortunately in many applications, supporting only one or a small number of input types is reasonable. MATLAB's Bessel functions do not support `uint8` input, but who cares?

4.1 Complex arrays

If `mxIsComplex(A_IN)` is true, then `A` has an imaginary part. MATLAB represents a complex array as two separate arrays:

```c
double *Ar = mxGetPr(A_IN);  /* Real data      */
double *Ai = mxGetPi(A_IN);  /* Imaginary data */
```

And `Ar[m + M*n]` and `Ai[m + M*n]` are the real and imaginary parts of `A(m+1,n+1)`.

To create a 2-D complex array, use

```c
B_OUT = mxCreateDoubleMatrix(M, N, mxCOMPLEX);
```

4.2 Arrays with more than two dimensions

For 2-D arrays, we can use `mxGetM` and `mxGetN` to obtain the dimensions. For an array with more than two dimensions, use

```c
size_t K = mxGetNumberOfDimensions(A_IN);
const mwSize *N = mxGetDimensions(A_IN);
```

The dimensions of the array are $\texttt{N[0]} \times \texttt{N[1]} \times \cdots \times \texttt{N[K-1]}$. The element data is then obtained as

```c
double *A = mxGetPr(A_IN);
```

or if `A` is complex,

```c
double *Ar = mxGetPr(A_IN);
double *Ai = mxGetPi(A_IN);
```

The elements are organized in column-major format.

	C/MEX	M-code equivalent
3-D array	`A[n0 + N[0]*(n1 + N[1]*n2)]`	`A(n0+1,n1+1,n2+1)`
K-D array	$\texttt{A[}\sum_{k=0}^{K-1} \left(\prod_{j=0}^{k-1} \texttt{N[j]}\right) n_k \texttt{]}$	$\texttt{A(}n_0\texttt{+1,}n_1\texttt{+1,}\ldots\texttt{,}n_{K-1}\texttt{+1)}$

To create an $\texttt{N[0]} \times \texttt{N[1]} \times \cdots \times \texttt{N[K-1]}$ array, use `mxCreateNumericArray`:

```c
B_OUT = mxCreateNumericArray(K, N, mxDOUBLE_CLASS, mxREAL);
```

Or for a complex array, replace `mxREAL` with `mxCOMPLEX`.

4.3 Arrays of other numeric datatypes

Different kinds of MATLAB variables and datatypes are divided into classes.

Class Name	Class ID	Class Name	Class ID
"double"	mxDOUBLE_CLASS	"int8"	mxINT8_CLASS
"single"	mxSINGLE_CLASS	"uint8"	mxUINT8_CLASS
"logical"	mxLOGICAL_CLASS	"int16"	mxINT16_CLASS
"char"	mxCHAR_CLASS	"uint16"	mxUINT16_CLASS
"sparse"	mxSPARSE_CLASS	"int32"	mxINT32_CLASS
"struct"	mxSTRUCT_CLASS	"uint32"	mxUINT32_CLASS
"cell"	mxCELL_CLASS	"int64"	mxINT64_CLASS
"function_handle"	mxFUNCTION_CLASS	"uint64"	mxUINT64_CLASS

The functions **mxGetClassID**, **mxIsClass**, and **mxGetClassName** can be used to check a variable's class, for example,

```
switch(mxGetClassID(A_IN))
{
case mxDOUBLE_CLASS:   /* Perform computation for a double array */
    MyComputationDouble(A_IN);
    break;
case mxSINGLE_CLASS:   /* Perform computation for a single array */
    MyComputationSingle(A_IN);
    break;
default:               /* A is of some other class */
    mexPrintf("A is of %s class\n", mxGetClassName(A_IN));
}
```

For a double array, we can use **mxGetPr** to get a pointer to the data. For a general numeric array, use **mxGetData** and cast the pointer to the type of the array.

```
float *A = (float *)mxGetData(A_IN);                /* Get single data  */
signed char *A = (signed char *)mxGetData(A_IN);    /* Get int8 data    */
short int *A = (short int *)mxGetData(A_IN);        /* Get int16 data   */
int *A = (int *)mxGetData(A_IN);                    /* Get int32 data   */
int64_T *A = (int64_T *)mxGetData(A_IN);            /* Get int64 data   */
```

For a complex array, use **mxGetImagData** to obtain the imaginary part. Aside from the datatype, elements are accessed in the same way as with double arrays.

To create an array of a numeric datatype, use either **mxCreateNumericMatrix** (for a 2-D array) or generally **mxCreateNumericArray**:

```
mxArray* mxCreateNumericMatrix(int m, int n,
        mxClassID class, mxComplexity ComplexFlag)

mxArray* mxCreateNumericArray(int ndim, const int *dims,
        mxClassID class, mxComplexity ComplexFlag)
```

4.4 Sparse arrays

Sparse data is complicated. Sparse arrays are always 2-D with elements of double datatype and they may be real or complex. The following functions are used to manipulate sparse arrays.

C/MEX	Meaning
`mwIndex *jc = mxGetJc(A)`	Get the **jc** indices
`mwIndex *ir = mxGetIr(A)`	Get the **ir** indices
`mxGetNzmax(A)`	Get the capacity of the array
`mxSetJc(A, jc)`	Set the **jc** indices
`mxSetIr(A, ir)`	Set the **ir** indices
`mxSetNzmax(A, nzmax)`	Set the capacity of the array

See the example MEX-function **fulltosparse.c** in **matlab/extern/examples/refbook** to see how to create a sparse matrix.

5 Creating an Uninitialized Numeric Array

This trick is so effective it gets it own section. The array creation functions (e.g., `mxCreateDoubleMatrix`) initialize the array memory by filling it with zeros. This may not seem so serious, but in fact this zero-filling initialization is a significant cost for large arrays. Moreover, initialization is usually unnecessary.

Memory initialization is costly, and can be avoided.

The steps to creating an uninitialized array are:

1. Create an empty 0×0 matrix
2. Set the desired dimensions (`mxSetM` and `mxSetN` or `mxSetDimensions`)
3. Allocate the memory with `mxMalloc` and pass it to the array with `mxSetData`. Repeat with `mxSetImagData` if creating a complex array.

For example, the following creates an uninitialized M×N real double matrix.

```
mxArray *B;
B = mxCreateDoubleMatrix(0, 0, mxREAL);     /* Create an empty array         */
mxSetM(M);                                  /* Set the dimensions to M x N   */
mxSetN(N);
mxSetData(B, mxMalloc(sizeof(double)*M*N)); /* Allocate memory for the array */
```

This code creates an uninitialized N[0] × N[1] × · · · × N[K-1] complex single (float) array:

```
mxArray *B;
B = mxCreateNumericMatrix(0, 0, mxSINGLE_CLASS, mxREAL);  /* Create an empty array                    */
mxSetDimensions(B, (const mwSize *)N, K);                 /* Set the dimensions to N[0] x ... x N[K-1] */
mxSetData(B, mxMalloc(sizeof(float)*NumEl));              /* Allocate memory for the real part        */
mxSetImagData(B, mxMalloc(sizeof(float)*NumEl));          /* Allocate memory for the imaginary part   */
```

where above `NumEl = N[0]*N[1]* · · · *N[K-1]`.

Often it is useful to create an uninitialized array having the same dimensions as an existing array. For example, given `mxArray *A`, an uninitialized `int32` array of the same dimensions is created by

```
mxArray *B;
B = mxCreateNumericMatrix(0, 0, mxINT32_CLASS, mxREAL);             /* Create an empty array */
mxSetDimensions(B, mxGetDimensions(A), mxGetNumberOfDimensions(A)); /* Set the dimensions    */
mxSetData(B, mxMalloc(4*mxGetNumberOfElements(A)));                 /* Allocate memory       */
```

If you want to initialize `B` as a copy of `A`, just use `mxDuplicateArray`:

```
mxArray *B = mxDuplicateArray(A);  /* Create B as a copy of A */
```

Section 9.2 will explain `mxMalloc` and other memory allocation functions in more detail.

6 Calling a MATLAB function from MEX

6.1 mexCallMATLAB

MEX-functions are useful because they enable calling C code from MATLAB. The reverse direction is also possible: **mexCallMATLAB** enables a MEX-function to call a MATLAB command. The syntax is

```
int mexCallMATLAB(int nlhs, mxArray *plhs[], int nrhs,
  mxArray *prhs[], const char *functionName);
```

The first four arguments are the same as those for **mexFunction** (see section 3). The fifth argument is the MATLAB function to call. It may be an operator, for example **functionName = "+"**.

callqr.c

```c
#include <string.h>
#include "mex.h"

void DisplayMatrix(char *Name, double *Data, int M, int N)
{   /* Display matrix data */
    int m, n;
    mexPrintf("%s = \n", Name);
    for(m = 0; m < M; m++, mexPrintf("\n"))
        for(n = 0; n < N; n++)
            mexPrintf("%8.4f ", Data[m + M*n]);
}

void CallQR(double *Data, int M, int N)
{   /* Perform QR factorization by calling the MATLAB function */
    mxArray *Q, *R, *A;
    mxArray *ppLhs[2];

    DisplayMatrix("Input", Data, M, N);
    A = mxCreateDoubleMatrix(M, N, mxREAL); /* Put input in an mxArray   */
    memcpy(mxGetPr(A), Data, sizeof(double)*M*N);

    mexCallMATLAB(2, ppLhs, 1, &A, "qr");   /* Call MATLAB's qr function */
    Q = ppLhs[0];
    R = ppLhs[1];
    DisplayMatrix("Q", mxGetPr(Q), M, N);
    DisplayMatrix("R", mxGetPr(R), M, N);

    mxDestroyArray(R);                      /* No longer need these      */
    mxDestroyArray(Q);
    mxDestroyArray(A);
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    #define M_IN   prhs[0]
    if(nrhs != 1 || mxGetNumberOfDimensions(M_IN) != 2 || !mxIsDouble(M_IN))
        mexErrMsgTxt("Invalid input.");

    CallQR(mxGetPr(M_IN), mxGetM(M_IN), mxGetN(M_IN));
}
```

```
>> M = round(rand(3)*3);
>> callqr(M)
Input =
   2.0000    2.0000    0.0000
   2.0000    1.0000    1.0000
   1.0000    2.0000    0.0000
Q =
  -0.6667    0.1617   -0.7276
  -0.6667   -0.5659    0.4851
  -0.3333    0.8085    0.4851
R =
  -3.0000   -2.6667   -0.6667
   0.0000    1.3744   -0.5659
   0.0000    0.0000    0.4851
```

It is possible during **mexCallMATLAB** that an error occurs in the called function, in which case the MEX-function is terminated. To allow the MEX-function to continue running even after an error, use **mexCallMATLABWithTrap**.

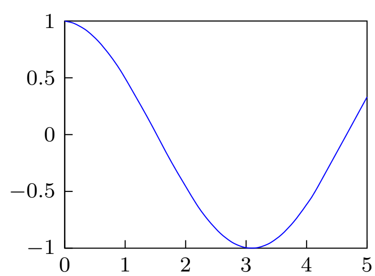
6.2 mexEvalString

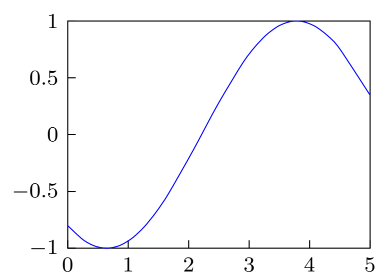
Two related functions are **mexEvalString** and **mexEvalStringWithTrap**, which are MEX versions of MATLAB's **eval** command. They accept a **char*** string to be evaluated, for example

```
eval('x = linspace(0,5); for k = 1:200, plot(x, cos(x+k/20)); drawnow; end');
```

can be performed in MEX as

```
mexEvalString("x = linspace(0,5); for k = 1:200, plot(x, cos(x+k/20)); drawnow; end");
```





7 Calling a MATLAB function handle from MEX

This example solves the partial differential equation

$$\partial_t u - \partial_{xx} u = 0, \qquad u(0,t) = u(1,t) = 0,$$

and plots the solution at every timestep. It demonstrates passing a function handle to a MEX-function and allocating temporary work arrays with `mxMalloc`.

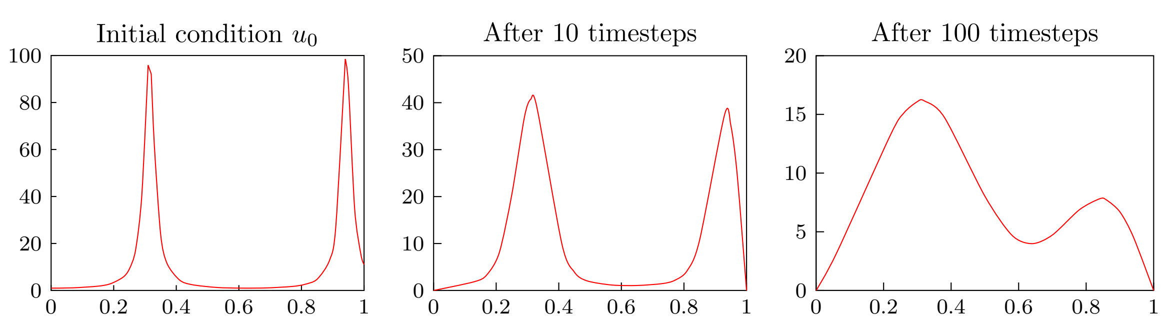
heateq.c

```c
#include "mex.h"

#define IS_REAL_2D_FULL_DOUBLE(P) (!mxIsComplex(P) && \
mxGetNumberOfDimensions(P) == 2 && !mxIsSparse(P) && mxIsDouble(P))
#define IS_REAL_SCALAR(P) (IS_REAL_2D_FULL_DOUBLE(P) && mxGetNumberOfElements(P) == 1)

mxArray *g_PlotFcn;

void CallPlotFcn(mxArray *pu, mxArray *pt)
{   /* Use mexCallMATLAB to plot the current solution u */
    mxArray *ppFevalRhs[3] = {g_PlotFcn, pu, pt};

    mexCallMATLAB(0, NULL, 3, ppFevalRhs, "feval");  /* Call the plotfcn function handle */
    mexCallMATLAB(0, NULL, 0, NULL, "drawnow");      /* Call drawnow to refresh graphics */
}

void SolveHeatEq(mxArray *pu, double dt, size_t TimeSteps)
{   /* Crank-Nicolson method to solve u_t - u_xx = 0, u(0,t) = u(1,t) = 0 */
    mxArray *pt;
    double *u, *t, *cl, *cu, *z;
    double dx, lambda;
    size_t n, N = mxGetNumberOfElements(pu) - 1;

    pt = mxCreateDoubleMatrix(1, 1, mxREAL);
    u = mxGetPr(pu);
    t = mxGetPr(pt);
    u[0] = u[N] = 0.0;
    *t = 0.0;
    CallPlotFcn(pu, pt);                  /* Plot the initial condition */

    dx = 1.0/N;                           /* Method initializations */
    lambda = dt/(dx*dx);
    cl = mxMalloc(sizeof(double)*N);      /* Allocate temporary work arrays */
    cu = mxMalloc(sizeof(double)*N);
    z = mxMalloc(sizeof(double)*N);

    cl[1] = 1.0 + lambda;
    cu[1] = -lambda/(2.0*cl[1]);
    for(n = 2; n <= N-1; n++)
    {
        cl[n] = 1.0 + lambda + cu[n-1]*(lambda/2.0);
        cu[n] = -lambda/(2.0*cl[n]);
    }

    while(TimeSteps--)                    /* Main computation loop */
    {
```

```
        z[1] = ((1.0-lambda)*u[1] + (lambda/2.0)*u[2]) / cl[1];
        for(n = 2; n <= N-1; n++)
            z[n] = ((1.0-lambda)*u[n] + (lambda/2.0)*(u[n+1] + u[n-1] + z[n-1])) / cl[n];
        for(u[N-1] = z[N-1], n = N-2; n >= 1; n--)
            u[n] = z[n] - cu[n]*u[n+1];

        *t += dt;
        CallPlotFcn(pu, pt);             /* Plot the current solution */
    }

    mxFree(z);                           /* Free work arrays */
    mxFree(cu);
    mxFree(cl);
    mxDestroyArray(pt);
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray*prhs[])
{   /* MEX gateway */
    #define U0_IN           prhs[0]
    #define DT_IN           prhs[1]
    #define TIMESTEPS_IN    prhs[2]
    #define PLOTFCN_IN      prhs[3]
    #define U_OUT           plhs[0]

    if(nrhs != 4)                        /* Input checking */
        mexErrMsgTxt("Four input arguments required.");
    else if(nlhs > 1)
        mexErrMsgTxt("Too many output arguments.");
    else if(!IS_REAL_2D_FULL_DOUBLE(U0_IN) || !IS_REAL_SCALAR(DT_IN) || !IS_REAL_SCALAR(TIMESTEPS_IN))
        mexErrMsgTxt("Invalid input.");
    else if(mxGetClassID(PLOTFCN_IN) != mxFUNCTION_CLASS && mxGetClassID(PLOTFCN_IN) != mxCHAR_CLASS)
        mexErrMsgTxt("Fourth argument should be a function handle.");

    U_OUT = mxDuplicateArray(U0_IN);     /* Create output u by copying u0 */
    g_PlotFcn = (mxArray *)PLOTFCN_IN;
    SolveHeatEq(U_OUT, mxGetScalar(DT_IN), mxGetScalar(TIMESTEPS_IN));
    return;
}
```

In MATLAB, **heateq** can be used as

```
N = 100;
x = linspace(0,1,N+1);
u0 = 1./(1e-2 + cos(x*5).^2);      % Create the initial condition
plotfcn = @(u,t) plot(x, u, 'r');  % Create plotting function
heateq(u0, 1e-4, 100, plotfcn);    % Solve heat equation
```

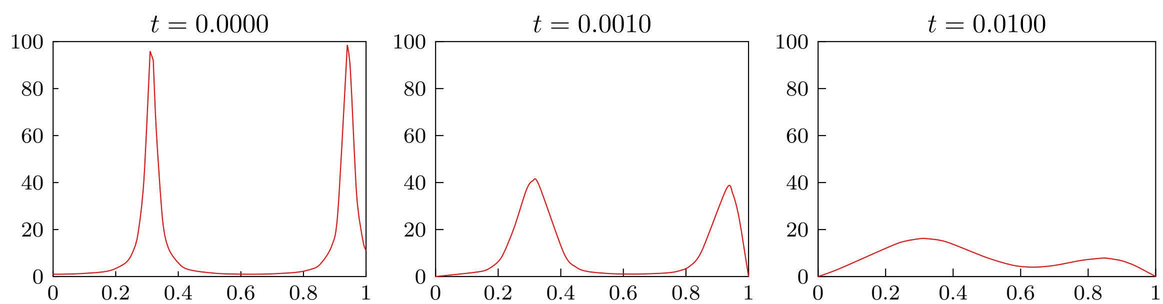


For more control over the plot function, we can write it as an M-function:

myplot.m

```
function myplot(u,t)
% Plotting function for heateq.c
plot(linspace(0,1,length(u)), u, 'r');
ylim([0,100]);                   % Freeze the y axis
title(sprintf('t = %.4f',t));   % Display t in the plot title
```

Calling `heateq(u0, 1e-4, 100, 'myplot')` produces



The actual figure is animated—try it to get the full effect.

In the example, the `plotfcn` function handle is called in `CallPlotFcn`:

```
mxArray *g_PlotFcn;

void CallPlotFcn(mxArray *pu, mxArray *pt)
{   /* Use mexCallMATLAB to plot the current solution u */
    mxArray *ppFevalRhs[3] = {g_PlotFcn, pu, pt};

    mexCallMATLAB(0, NULL, 3, ppFevalRhs, "feval");  /* Call the plotfcn function handle */
    mexCallMATLAB(0, NULL, 0, NULL, "drawnow");      /* Call drawnow to refresh graphics */
}
```

The trick is to pass the function handle to `feval`, which in turn evaluates the function handle. The first `mexCallMATLAB` call is equivalent to `feval(plotfcn, u, t)`. The second `mexCallMATLAB` calls `drawnow` to refresh graphics; this is necessary to watch the plot change in realtime during the computation. See section 9.3 for another example of calling a function handle.

8 Calling MATLAB from a non-MEX Program

We have discussed using **mexCallMATLAB** to call a MATLAB command from within a MEX-function. But is it possible to call MATLAB from a program that is *not* a MEX-function? The answer is yes, it is possible! But beware my approach is quite inefficient and roundabout.

The key is that MATLAB can be started to run a command, for example

```
matlab -r "x=magic(6);save('out.txt','x','-ascii');exit"
```

This starts MATLAB, creates a 6×6 magic matrix, saves it to out.txt, then promptly exits. More practically, you should start a script containing the actual commands.

```
matlab -r makemagic
```

Under UNIX, you can add the **-nodisplay** flag to hide the MATLAB window.

The following is a simple C program that calls MATLAB to create magic matrices of a specified size:

makemagic.c

```c
/* Run as "makemagic N" to make an NxN magic matrix */
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    float v;
    int n, N  = (argc > 1) ? atoi(argv[1]) : 6;

    /* Write a MATLAB script */
    FILE *fp = fopen("makemagic.m", "wt");
    fprintf(fp, "x = magic(%d);\n"           /* Make an NxN magic matrix */
                "save magic.txt x -ascii\n"  /* Save to magic.txt        */
                "exit;", N);                 /* Exit MATLAB              */
    fclose(fp);

    /* Call MATLAB to run the script */
    printf("Calling MATLAB...\n");
    system("matlab -r makemagic");

    /* Read from the output file */
    fp = fopen("magic.txt", "rt");
    while(!feof(fp))
    {
        for(n = 0; n < N && fscanf(fp, "%f", &v) == 1; n++)
            printf("%4d ", (int)v);
        printf("\n");
    }
    fclose(fp);

    return 0;
}
```

9 Memory

9.1 Remembering variables between calls

C variables that are declared globally are remembered between calls. The following MEX-function counts the number of times it was called.

remember.c

```
#include "mex.h"

/* Count is a global variable, so it will be remembered between calls */
static int Count = 1;

void MyExit()
{
    mexPrintf("MyExit() called!\n");
    /* Do cleanup here ... */
    return;
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    mexAtExit(&MyExit); /* Register MyExit() to run when MEX-function is cleared */
    mexPrintf("Count=%d\n", Count);
    Count++;  /* Increment Count */
    return;
}
```

This MEX-function also demonstrates `mexAtExit`, which allows us to run a cleanup function when the MEX-function is cleared or when MATLAB exits.

```
>> remember
Count=1
>> remember
Count=2
>> remember
Count=3
>> clear remember
MyExit() called!
>> remember
Count=1
```

A MEX-function can be explicitly cleared by `clear` *function* or `clear all`.

9.2 Dynamic memory allocation

The MEX interface provides several functions for managing dynamic memory:

C/MEX	Meaning	Standard C equivalent
`mxMalloc(Size)`	Allocate memory	`malloc(Size)`
`mxCalloc(Num, Size)`	Allocate memory initialized to zero	`calloc(Num, Size)`
`mxRealloc(Ptr, NewSize)`	Change size of allocated memory block	`realloc(Ptr, NewSize)`
`mxFree(Ptr)`	Release memory allocated by one of the above	`free(Ptr)`

It is also possible to use the standard C `malloc`, etc., or C++ `new` and `delete` within a MEX-function. The advantage of `mxMalloc`, etc. is that they use MATLAB's internal memory manager so that memory is properly released on error or abort (Ctrl+C).

A useful function when allocating memory is `mxGetElementSize`, which returns the number of bytes needed to store one element of a MATLAB variable,

```
size_t BytesPerElement = mxGetElementSize((const mxArray *)A);
```

9.3 Persistent Memory

By default, memory allocated by `mxMalloc`, etc. is automatically released when the MEX-function completes. Calling `mexMakeMemoryPersistent(Ptr)` makes the memory persistent so that it is remembered between calls.

The following MEX-function wraps `feval` to remember function evaluations that have already been computed. It uses `mexMakeMemoryPersistent` to store a table of precomputed values.

pfeval.c

```
#include "mex.h"

#define INITIAL_TABLE_CAPACITY   64

/* Table for holding precomputed values */
static struct {
    double *X;    /* Array of x values                  */
    double *Y;    /* Array of corresponding y values    */
    int Size;     /* Number of entries in the table     */
    int Capacity; /* Table capacity                     */
} Table = {0, 0, 0, 0};

void ReallocTable(int NewCapacity)
{   /* (Re)allocate the table */
    if(!(Table.X = (double *)mxRealloc(Table.X, sizeof(double)*NewCapacity))
        || !(Table.Y = (double *)mxRealloc(Table.Y, sizeof(double)*NewCapacity)))
        mexErrMsgTxt("Out of memory");
    mexMakeMemoryPersistent(Table.X);  /* Make the table memory persistent */
    mexMakeMemoryPersistent(Table.Y);  /* Make the table memory persistent */
    Table.Capacity = NewCapacity;
}

void AddToTable(double x, double y)
{   /* Add (x,y) to the table */
    if(Table.Size == Table.Capacity)
        ReallocTable(2*Table.Capacity);
    Table.X[Table.Size] = x;
    Table.Y[Table.Size++] = y;
}

mxArray* EvaluateFunction(const mxArray *pFunction, const mxArray *px)
{   /* Evaluate function handle pFunction at px */
    const mxArray *ppFevalRhs[2] = {pFunction, px};
```

```c
    mxArray *py;
    mexPrintf("Evaluating f(x = %g)\n", mxGetScalar(px));
    mexCallMATLAB(1, &py, 2, (mxArray **)ppFevalRhs, "feval");
    return py;
}

void MyExit()
{   /* Clean up */
    mxFree(Table.X);
    mxFree(Table.Y);
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    #define FCN_IN   prhs[0]
    #define X_IN     prhs[1]
    #define Y_OUT    plhs[0]
    double x, *y;
    int i;

    if(nrhs != 2)
        mexErrMsgTxt("Two input arguments required.");
    else if(mxGetClassID(FCN_IN) != mxFUNCTION_CLASS && mxGetClassID(FCN_IN) != mxCHAR_CLASS)
        mexErrMsgTxt("First argument should be a function handle.");
    else if(!mxIsDouble(X_IN) || mxGetNumberOfElements(X_IN) != 1)
            mexErrMsgTxt("X must be a real double scalar.");

    x = mxGetScalar(X_IN);
    mexAtExit(&MyExit); /* Register MyExit() to run when MEX function is cleared */

    if(!Table.X || !Table.Y)                  /* This happens on the first call    */
        ReallocTable(INITIAL_TABLE_CAPACITY); /* Allocate precomputed values table */

    /* Search for x in the table */
    for(i = 0; i < Table.Size; i++)
        if(x == Table.X[i])
        {
            mexPrintf("Found precomputed value for x = %g\n", x);
            y = mxGetPr(Y_OUT = mxCreateDoubleMatrix(1, 1, mxREAL));
            *y = Table.Y[i];
            return;
        }

    /* x is not yet in the table */
    Y_OUT = EvaluateFunction(FCN_IN, X_IN);  /* Evaluate the function         */
    AddToTable(x, mxGetScalar(Y_OUT));       /* Make a new entry in the table */
    return;
}
```

As a simple example, here is **pfeval** applied to $f(x) = x^2$:

```
>> f = @(x) x.^2;  % Define the function to evaluate
>> pfeval(f, 4)
Evaluating f(x = 4)
ans =
    16
>> pfeval(f, 10)
Evaluating f(x = 10)
ans =
   100
```

```
>> pfeval(f, 4)
Found precomputed value for x = 4
ans =
    16
```

Remark: I do not recommend using `pfeval` in practice. It does not work correctly if called with more than one function handle. Additionally, it would be better to use a binary search on a sorted table and to do error checking when calling the function handle.

Persistent memory should be used in combination with `mexAtExit`. You should write a cleanup function that releases the persistent memory and use `mexAtExit` to register it. If you do not do this, persistent memory is never released, and MATLAB will leak memory!

9.4 Locking

A MEX-function is "unlocked" by default, meaning it may be cleared at any time. To prevent the MEX-function from being cleared, call `mexLock`. Call the function `mexUnlock` to unlock it again. If `mexLock` is called n times, `mexUnlock` must be called n times to unlock it. The `mexIsLocked` function checks whether the MEX-function is locked.

10 Non-Numeric Variables

There are specialized interface functions for handling non-numeric classes. Non-numeric variables are still represented with `mxArray` objects, and some functions like `mxDestroyArray` work on any class. Use `mxGetClassID` or `mxGetClassName` as described in section 4.3 to determine the class of a variable. You can alternatively use `mxIsLogical`, `mxIsChar`, `mxIsCell`, `mxIsStruct`, or `mxIsFunctionHandle`.

10.1 Logicals

Logicals are not so different from numeric classes. Logical elements are represented in C/MEX as type `mxLogical` (a typedef for `bool` or `unsigned char` depending on platform) and are arranged in column-major organization. The following functions are used to create and handle logical arrays.

Function	Description
`L = mxCreateLogicalScalar(Value)`	Create a logical `L = Value`
`L = mxCreateLogicalMatrix(M,N)`	Create an M×N logical matrix
`L = mxCreateLogicalArray(K,(const mwSize *)N)`	Create an `N[0]` × ··· × `N[K-1]` logical array
`mxLogical *Data = mxGetLogicals(L)`	Get pointer to the logical data
`mxIsLogicalScalar(L)`	True if `L` is logical and scalar
`mxIsLogicalScalarTrue(L)`	True if logical scalar `L` equals true

10.2 Char arrays

Char arrays (strings) are represented as UTF-16 `mxChar` elements. For convenience, there are functions to convert between char arrays and null-terminated C `char*` strings in the local codepage encoding.

Function	Description
`S = mxCreateString("My string")`	Create a 1×N string from a `char*` string
`S = mxCreateCharMatrixFromStrings(M, (const char **)Str)`	Create matrix from `Str[0]`,...,`Str[M-1]`
`S = mxCreateCharArray(K, (const mwSize *)N)`	Create an `N[0]` × ··· × `N[K-1]` char array
`mxGetString(S, Buf, BufLen)`	Read string `S` into `char*` string `Buf`

Warning: `mxGetString` will truncate the result if it is too large to fit in `Buf`. To avoid truncation, `BufLen` should be at least `mxGetNumerOfElements(S)*sizeof(mxChar) + 1`.

stringhello.c

```
/* A string version of the "Hello, world!" example */
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    plhs[0] = mxCreateString("Hello, world!");
    return;
}
```

The UTF-16 data may also be manipulated directly if you are determined to do so. Use **mxGetData** to get an **mxChar*** pointer to the UTF-16 data. For example, the following MEX-function creates the string “明天.txt” containing Chinese characters.

mingtian.c

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    #define M_OUT   plhs[0]
    mxChar *Data;
    /* UTF-16LE data */
    unsigned short MingTian[] = {0xFEFF, 0x660E, 0x5929, 0x002E, 0x0074, 0x0078, 0x0074};
    int k, Size[] = {1, 7};

    M_OUT = mxCreateCharArray(2, (const int*)Size);
    Data = mxGetData(M_OUT);

    for(k = 0; k < 7; k++)
        Data[k] = MingTian[k];
}
```

Running MEX-function **mingtian** on the console will show non-ASCII characters as boxes (the data is there, but the console is limited in what it can display).

```
>> mingtian

ans =

□□.txt
```

So to see this string, we need to get it out of MATLAB. This script attempts to write a file with **mingtian**’s output as the filename:

```
Status = {'Succeeded', 'Failed'};
s = mingtian;
fid = fopen(s, 'w');
fprintf('Creating file with UTF-16 filename:  %s\n', Status{(fid == -1)+1});
fclose(fid);
fid = fopen(s, 'r');
fprintf('Opening file with UTF-16 filename:   %s\n', Status{(fid == -1)+1});
fclose(fid);
```

If successful, there should be a new file called 明天.txt in the current directory.

Remark: You don’t need to use MEX to create exotic UTF-16 characters. The example above can be reproduced on the console as

```
>> s = char([65279, 26126, 22825,'.txt']);
>> fid = fopen(s, 'w'); Status = {'Succeeded', 'Failed'}; Status{(fid == -1)+1}, fclose(fid);

ans =
Succeeded
```

10.3 Cell arrays

A cell array is essentially an array of `mxArray` pointers. The functions `mxGetCell` and `mxSetCell` are used to access or change the `mxArray*` of a cell.

Function	Description
`C = mxCreateCellMatrix(M, N)`	Create an `M`×`N` cell array
`C = mxCreateCellArray(K, (const mwSize *)N)`	Create an `N[0]` × ⋯ × `N[K-1]` cell array
`mxArray *A = mxGetCell(C, i)`	Get contents of the `i`th cell, `A = C{i+1}`
`mxSetCell(C, i, A)`	Set contents of the `i`th cell, `C{i+1} = A`

The index `i` in `mxGetCell` and `mxSetCell` is zero based.

10.4 Structs

Like a cell array, a struct array is essentially an array of `mxArray` pointers, but with an additional dimension indexed by field names. Each field name has a corresponding field number; the fields are numbered in the order in which they were added to the struct. Fields may be referenced either by name or by number.

A struct array is manipulated as a whole by the following functions.

Function	Description
`mxGetNumberOfFields(X)`	Get the number of fields in struct `X`
`X = mxCreateStructMatrix(M, N, NumFields, (const char **)Str)`	Create an `M`×`N` struct
`X = mxCreateStructArray(K, N, NumFields, (const char **)Str)`	Create an `N[0]` × ⋯ × `N[K-1]` struct
`mxAddField(X, (const char *)Str)`	Add a new field to struct `X`
`mxRemoveField(X, k)`	Remove the `k`th field (if it exists)
`int k = mxGetFieldNumber(X, "myfield")`	Get the field number of a field name
`const char *Name = mxGetFieldNameByNumber(X, k)`	Get the field name of the `k`th field

In the creation functions `mxCreateStructMatrix` and `mxCreateStructArray`, the field names are given by null-terminated `char*` strings `Str[0]`, . . . , `Str[NumFields-1]`.

Individual fields are accessed and changed with the following functions.

Function	Description
`mxArray *F = mxGetField(X, i, "myfield")`	Get the `i`th element's field `F = X(i+1).myfield`
`mxArray *F = mxGetFieldByNumber(X, i, k)`	Get the `i`th element's `k`th field
`mxSetField(X, i, "myfield", F)`	Set the `i`th element's field `X(i+1).myfield = F`
`mxSetFieldByNumber(X, i, k, F)`	Set the `i`th element's `k`th field

The index `i` above is zero based.

10.5 Function handles

Function handles are slippery creatures with very little support within MEX. Even in M-code, they have some surprising limitations:

```
>> f = @(x)cos(2*x); g = @(x)cos(2*x);
>> isequal(f,g)

ans =
     0
```

At least a function handle is equal to itself:

```
>> isequal(f,f)

ans =
     1
```

To identify an `mxArray*` as a function handle, use its class ID or `mxIsFunctionHandle`.

To execute a function handle, use `mexCallMATLAB` to call `feval`. For example, the following evaluates a function handle $y = f(x)$:

```
mxArray* EvaluateFunction(const mxArray *f, const mxArray *x)
{   /* Evaluate function handle by caling y = feval(f,x) */
    mxArray *y;
    const mxArray *ppFevalRhs[2] = {f, x};
    mexCallMATLAB(1, &y, 2, (mxArray **)ppFevalRhs, "feval");
    return y;
}
```

A function handle with multiple arguments can be evaluated similarly (see section 7 for an example).

Similarly, `mexCallMATLAB` may be used to perform other operations with function handles.

MATLAB function	Description
`y = feval(f,x)`	Evaluate a function handle
`functions(f)`	Get information about a function handle
`s = func2str(f)`	Convert function handle to string
`f = str2func(s)`	Convert string to function handle (see below)

In MATLAB 2009a and newer, it is possible to create a function handle in MEX by calling `str2func`. Some older versions of MATLAB have this command but do not support anonymous function creation.

```
mxArray* CreateFunctionFromString(const char *Str)
{   /* Create a function handle from a string */
    mxArray *f, *str2func = mxCreateString("str2func"), *s = mxCreateString(Str);
    const mxArray *ppFevalRhs[2] = {str2func, s};
    mexCallMATLAB(1, &f, 1, (mxArray **)ppFevalRhs, "feval");
    mexDestroyArray(s);
    return f;
}
```

For example, `f = CreateFunctionFromString("@(x) x^2")` creates the square function.

11 FFTs with FFTW

To perform an FFT within a MEX-function, you could use `mexCallMATLAB` to call MATLAB's `fft` command. However, this approach has the overhead that `fft` must allocate a new mxArray to hold the resulting FFT, as well as the overhead and nuisance of wrapping up the data in mxArray objects. It is more efficient to perform FFTs directly by calling the FFTW library.

11.1 A brief introduction to FFTW3

The FFTW3 library is available on the web at `www.fftw.org`. The library can perform FFTs of any size and dimension. It can also perform related trigonometric transforms.

To perform a transform, the type, size, etc. are specified to FFTW to create a *plan*. FFTW considers many possible algorithms and estimates which will be fastest for the specified transform. The transform itself is then performed by executing the plan. The plan may be executed any number of times. Finally, the plan is destroyed to release the associated memory.

Two common ways to store complex arrays are *split format* and *interleaved format*. Section 4.1 explained how complex MATLAB arrays are represented with two separate blocks of memory, one for the real part and the other for the imaginary part,

$$\text{Split format:} \quad r_0, r_1, r_2, \ldots, r_{N-1}, \quad \ldots \quad i_0, i_1, i_2, \ldots, i_{N-1}.$$

In FFTW, such a two-block organization is called *split format*. Another common way to arrange complex data is to interleave the real and imaginary parts into a single contiguous block of memory,

$$\text{Interleaved format:} \quad r_0, i_0, r_1, i_1, r_2, i_2, \ldots, r_{N-1}, i_{N-1}.$$

FFTW can handle both interleaved and split formats. Complex MATLAB arrays are always split format, so you must use split format to store a complex FFT output directly in a MATLAB array.

11.2 FFTW3 examples

To use FFTW3, we need to include `fftw3.h`. We also need to include option `-lfftw3` when calling the `mex` command to link the MEX-function with the FFTW3 library:

```
mex mymexfunction.c -lfftw3
```

Additional options may be necessary depending on how FFTW3 is installed on your system; see the `-l` and `-L` options in `help mex`.

It is helpful to define `DivideArray`, which we will use to normalize results after inverse transforms.

```
#include <fftw3.h>

/* Divide an array per-element (used for IFFT normalization) */
void DivideArray(double *Data, int NumEl, double Divisor)
{
    int n;
    for(n = 0; n < NumEl; n++)
        Data[n] /= Divisor;
}
```

We first consider transforms with the interleaved format since FFTW3's interface is simpler in this case. The following function computes the 1D FFT of length `N` on complex array `X` to produce complex output array `Y`.

```
/* FFT1DInterleaved  1D FFT complex-to-complex interleaved format
   Inputs:
   N      Length of the array
   X      Input array, X[2n] = real part and X[2n+1] = imag part of the nth element (n = 0, ..., N - 1)
   Sign   -1 = forward transform, +1 =  inverse transform

   Output:
   Y      Output array, Y[2n] = real part and Y[2n+1] = imag part of the nth element (n = 0, ..., N - 1)
*/
void FFT1DInterleaved(int N, double *X, double *Y, int Sign)
{
    fftw_plan Plan;
    if(!(Plan = fftw_plan_dft_1d(N, (fftw_complex *)X, (fftw_complex *)Y, Sign, FFTW_ESTIMATE)))
        mexErrMsgTxt("FFTW3 failed to create plan.");
    fftw_execute(Plan);
    fftw_destroy_plan(Plan);

    if(Sign == 1)  /* Normalize the result after an inverse transform */
        DivideArray(Y, 2*N, N);
}
```

Similarly, we can compute 2D FFTs as

```
/* FFT2DInterleaved  2D FFT complex-to-complex interleaved format */
/* X[2*(m + M*n)] = real part and X[2*(m + M*n)+1] = imag part of the (m,n)th element, and similarly for Y */
void FFT2DInterleaved(int M, int N, double *X, double *Y, int Sign)
{
    fftw_plan Plan;
    if(!(Plan = fftw_plan_dft_2d(N, M, (fftw_complex *)X, (fftw_complex *)Y, Sign, FFTW_ESTIMATE)))
        mexErrMsgTxt("FFTW3 failed to create plan.");
    fftw_execute(Plan);
    fftw_destroy_plan(Plan);

    if(Sign == 1)
        DivideArray(Y, 2*M*N, M*N);
}
```

Now we consider split format. Performing FFTs on split format arrays requires using FFTW3's more involved guru interface.

```
/* FFT1DSplit  1D FFT complex-to-complex split format
   Inputs:
   N      Length of the array
   XReal  Real part of the input array, XReal[n] = real part of the nth element
```

```c
   XImag  Imaginary part of the input array, XImag[n] = imag part of the nth element
   Sign   -1 = forward transform, +1 =  inverse transform

   Output:
   YReal  Real part of the output array
   YImag  Imaginary part of the output array
*/
void FFT1DSplit(int N, double *XReal, double *XImag, double *YReal, double *YImag, int Sign)
{
    fftw_plan Plan;
    fftw_iodim Dim;

    Dim.n = N;
    Dim.is = 1;
    Dim.os = 1;

    if(!(Plan = fftw_plan_guru_split_dft(1, &Dim, 0, NULL,
        XReal, XImag, YReal, YImag, FFTW_ESTIMATE)))
        mexErrMsgTxt("FFTW3 failed to create plan.");

    if(Sign == -1)
        fftw_execute_split_dft(Plan, XReal, XImag, YReal, YImag);
    else
    {
        fftw_execute_split_dft(Plan, XImag, XReal, YImag, YReal);
        DivideArray(YReal, N, N);
        DivideArray(YImag, N, N);
    }

    fftw_destroy_plan(Plan);
}
```

Finally, here is a general function for the N-D FFT with split format:

```c
/* FFTNDSplit  ND FFT complex-to-complex split format
   Inputs:
   NumDims  Number of dimensions
   N        Array of dimension sizes
   XReal    Real part of the input, an N[0] x N[1] x ... x N[NumDims-1] array in column-major format
   XImag    Imaginary part of the input
   Sign     -1 = forward transform, +1 =  inverse transform

   Output:
   YReal    Real part of the output array
   YImag    Imaginary part of the output array
*/
void FFTNDSplit(int NumDims, const int N[], double *XReal, double *XImag, double *YReal, double *YImag, int Sign)
{
    fftw_plan Plan;
    fftw_iodim Dim[NumDims];
    int k, NumEl;

    for(k = 0, NumEl = 1; k < NumDims; k++)
    {
        Dim[NumDims-k-1].n = N[k];
        Dim[NumDims-k-1].is = Dim[NumDims-k-1].os = (k == 0) ? 1 : (N[k-1] * Dim[NumDims-k].is);
        NumEl *= N[k];
    }

    if(!(Plan = fftw_plan_guru_split_dft(NumDims, Dim, 0, NULL,
        XReal, XImag, YReal, YImag, FFTW_ESTIMATE)))
```

```
        mexErrMsgTxt("FFTW3 failed to create plan.");

    if(Sign == -1)
        fftw_execute_split_dft(Plan, XReal, XImag, YReal, YImag);
    else
    {
        fftw_execute_split_dft(Plan, XImag, XReal, YImag, YReal);
        DivideArray(YReal, NumEl, NumEl);
        DivideArray(YImag, NumEl, NumEl);
    }

    fftw_destroy_plan(Plan);
    return;
}
```

Remark: To perform transforms in-place, simply set `Y = X` (or `YReal = XReal` and `YImag = XImag`) when calling the above functions.

Remark: The `DivideArray` function scales the result by $1/N$. It is often possible to absorb this scale factor elsewhere to avoid this computation.

Remark: There are many possibilities in FFTW3 beyond the scope of this document. It is possible to perform multiple FFTs in a single plan, which may be more efficient than performing multiple plans. Aside from complex-to-complex transforms, FFTW3 can also perform real-to-complex, complex-to-real, and real-to-real transforms. See `www.fftw.org/fftw3_doc` for more details.

12 Miscellaneous

There are a few other interface functions in MEX that we haven't discussed yet. They are mostly analogues of basic M-code commands.

C/MEX	Meaning	M-code equivalent
`mexPrint("Hello")`	Print a string	`disp('Hello')`
`mexPrintf("x=%d", x)`	Print a formatted string	`fprintf('x=%d', x)`
`mexWarnMsgTxt("Trouble")`	Display a warning message	`warning('Trouble')`
`mexErrMsgTxt("Abort!")`	Display a error message	`error('Abort!')`
`mexFunctionName()`	Get the MEX-function's name	`mfilename`
`mexGet(h,"Prop")`	Get a property on object `h`	`get(h,'Prop')`
`mexSet(h,"Prop",Value)`	Set a property on object `h`	`set(h,'Prop',Value)`
`mexGetVariable`	Copy variable from a workspace	`evalin(WS,'Var')`
`mexGetVariablePtr`	Get variable read-only pointer	—
`mexPutVariable`	Create variable in a workspace	`assignin`

There are also functions for manipulating MATLAB MAT files. An object of type `MATFile*` represents a handle to an open MAT file.

C/MEX	Meaning
`MATFile *mfp = matOpen("my.mat", Mode)`	Open a MAT file
`matClose(mfp)`	Close MAT file
`const char *Str = matGetDir(mfp, &Num)`	Get list of variable names in the file
`mxArray *V = matGetVariable(mfp, Name)`	Get the variable named `Name`
`mxArray *V = matGetNextVariable(mfp, &Name)`	Get the next variable and its name
`matGetNextVariableInfo`	Get header info about a variable
`matPutVariable(mfp, Name, V)`	Write variable `V` with name `Name`
`matPutVariableGlobal`	Write variable as global
`matDeleteVariable(mfp, Name)`	Delete the variable named `Name`

13 Further Reading

All MEX interface functions are documented on the web in online function references (search for example "matlab mxSetPr"). Information is otherwise limited, but there is MEX wisdom to be found scattered through forums and buried within the dark source code of existing MEX-functions.

For more MEX-function examples, study the files in `matlab/extern/examples`. For instance, the example "explore.c" shows how to read the data of a variety of different MATLAB variables. You can open this file by entering the following on the MATLAB console:

```
>> edit([matlabroot '/extern/examples/mex/explore.c'])
```

For detailed reference, study the files in `matlab/extern/include`, particularly `matrix.h`. These files are thoroughly commented and explain many of the inner workings of MEX.