Writing MATLAB® C/MEX Code

Pascal Getreuer
What is MEX?

MEX-functions are programs written in C, C++, or Fortran code that are callable from MATLAB. We will focus on C.

MEX provides C functions to manipulate MATLAB variables, for example

<table>
<thead>
<tr>
<th>C/MEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mxCreateDoubleMatrix</code></td>
<td>Create a 2D double array</td>
</tr>
<tr>
<td><code>mxGetPr</code></td>
<td>Get pointer to array data</td>
</tr>
<tr>
<td><code>mxGetDimensions</code></td>
<td>Get array dimensions</td>
</tr>
<tr>
<td><code>mxGetClassID</code></td>
<td>Determine variable’s datatype</td>
</tr>
</tbody>
</table>
Words of Warning

My advice:
1. Try to optimize your M-code first
2. Consider porting the entire application to C/C++
3. Use MEX to substitute only the bottleneck M-functions
Hello, world!

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;
}
```
Getting Started

On the **MATLAB** console, compile `hello.c` with

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

Compiling requires that you have a C compiler and that **MATLAB** is configured to use it. Use “mex -setup” to change compiler settings.

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

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

Beware that compiled MEX files might not be compatible between different platforms or different versions of **MATLAB**.
Inputs and Outputs

Let’s take a closer look at the line

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

“mxArray” is a type for representing a MATLAB variable, and the arguments are:

<table>
<thead>
<tr>
<th>C/MEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlhs</td>
<td>Number of output variables</td>
</tr>
<tr>
<td>plhs</td>
<td>Array of mxArray pointers to the output variables</td>
</tr>
<tr>
<td>nrhs</td>
<td>Number of input variables</td>
</tr>
<tr>
<td>prhs</td>
<td>Array of mxArray pointers to the input variables</td>
</tr>
</tbody>
</table>

Notation: “lhs” = “left-hand side” (output variables)  
“rhs” = “right-hand side” (input variables)
Inputs and Outputs

Let’s take a closer look at the line

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

“mxArray” is a type for representing a MATLAB variable, and the arguments are:

<table>
<thead>
<tr>
<th>C/MEX</th>
<th>M-code equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlhs</td>
<td>nargin</td>
</tr>
<tr>
<td>plhs</td>
<td>varargout</td>
</tr>
<tr>
<td>nrhs</td>
<td>nargin</td>
</tr>
<tr>
<td>prhs</td>
<td>varargin</td>
</tr>
</tbody>
</table>

Notation: “lhs” = “left-hand side” (output variables)
“rhs” = “right-hand side” (input variables)
Inputs and Outputs

Suppose our MEX-function is called as

\[ [X, Y] = \text{mymexfun}(A, B, C) \]

Then \text{mexFunction}

\[
\text{void mexFunction}(\text{int} \ \text{nlhs}, \ \text{mxArray} *\text{plhs}[],
                  \text{int} \ \text{nrhs}, \ \text{const} \ \text{mxArray} *\text{prhs}[])
\]

receives the following information:

\[
\begin{align*}
\text{nlhs} & = 2 \\
\text{plhs}[0] & \text{ points to } X \\
\text{plhs}[1] & \text{ points to } Y \\
\text{nrhs} & = 3 \\
\text{prhs}[0] & \text{ points to } A \\
\text{prhs}[1] & \text{ points to } B \\
\text{prhs}[2] & \text{ points to } C
\end{align*}
\]
Inputs and Outputs

The output variables are initially unassigned; it is the responsibility of the MEX-function to create them. So for the example

\[ [X,Y] = \text{mymexfun}(A,B,C) \]

it is our responsibility to create X and Y.

If \( \text{nlhs} = 0 \), the MEX-function is still allowed return one output variable, in which case \( \text{plhs}[0] \) represents the “\text{ans}” variable.
normalizecols

Objective: Given matrix $A$, construct matrix $B$ according to

$$B_{m,n} = A_{m,n} / \|A_n\|_p$$

where $\|A_n\|_p$ is the $\ell^p$ norm of the $n$th column.

normalizecols.m

```matlab
function B = normalizecols(A, p)

if nargin < 2 % Was p not specified?
    p = 2; % Set default value for p
end

[M, N] = size(A);
B = zeros(M, N);

for n = 1:N % Build matrix B column-by-column
    B(:,n) = A(:,n) / norm(A(:,n), p);
end
```

M-function implementation
normalizecols: Converting to MEX

We will convert normalizecols.m to MEX. Let’s start with hello.c as a template...

normalizecols.c

MEX-function implementation

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

void mexFunction(int nlhs, mxArray *plhs[], /* Outputs */
                 int nrhs, const mxArray *prhs[]) /* Inputs */
{
    /* TODO ... */
}
```

Pascal Getreuer (UCLA)
The first step is to figure out the calling syntax. Let’s look at

\[ B = \text{normalizecols}(A, p) \]

This calling syntax in MEX representation is

- \( \text{nlhs} = 1 \)
- \( \text{nrhs} = 2 \)
- \( \text{plhs}[0] \) points to \( B \)
- \( \text{prhs}[0] \) points to \( A \)
- \( \text{prhs}[1] \) points to \( p \)

For clarity, it is a good idea to define

```c
#define A_IN prhs[0]
#define P_IN prhs[1]
#define B_OUT plhs[0]
```
normalizecols: Converting to MEX

What if `normalizecols` is called without `p`?

```matlab
B = normalizecols(A)
```

This calling syntax in MEX representation is

- `nlhs = 1`
- `plhs[0]` points to `B`
- `nrhs = 1`
- `prhs[0]` points to `A`
- `prhs[1]` doesn’t exist

For clarity, it is a good idea to define

```c
#define A_IN prhs[0]
#define P_IN prhs[1]
#define B_OUT plhs[0]
```
normalizecols: Input Checking

Be on the defensive and check inputs. We can use mexErrMsgTxt to abort with an error message.

normalizecols.c  MEX-function implementation

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

#define A_IN prhs[0]
#define P_IN prhs[1]
#define B_OUT plhs[0]

void mexFunction(int nlhs, mxArray *plhs[], /* Outputs */
                 int nrhs, const mxArray *prhs[]) /* Inputs */
{
    if(nrhs < 1 || nrhs > 2)
        mexErrMsgTxt("Must have either 1 or 2 input arguments.");
    if(nlhs > 1)
        mexErrMsgTxt("Too many output arguments.");

    /* ... */
}
normalizecols: Input Checking

We should also verify the datatype of the input variables.

<table>
<thead>
<tr>
<th>C/MEX</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxIsDouble(A_IN)</td>
<td>True for a double array</td>
</tr>
<tr>
<td>mxIsComplex(A_IN)</td>
<td>True if array is complex</td>
</tr>
<tr>
<td>mxIsSparse(A_IN)</td>
<td>True if array is sparse</td>
</tr>
<tr>
<td>mxGetNumberOfDimensions(A_IN)</td>
<td>Number of array dimensions</td>
</tr>
<tr>
<td>mxGetNumberOfElements(A_IN)</td>
<td>Number of array elements</td>
</tr>
</tbody>
</table>

For simplicity, let’s require that A is a real 2D full double matrix.

```c
if (mxIsComplex(A_IN) || mxGetNumberOfDimensions(A_IN) != 2
    || mxIsSparse(A_IN) || !mxIsDouble(A_IN))
    mexErrMsgTxt("Sorry! A must be a real 2D full double matrix.");
```
normalizecols: Input Checking

We want to allow two calling syntaxes

\[
\begin{align*}
B &= \text{normalizecols}(A) \quad \% \text{nrhs} == 1 \quad (\text{use } p = 2.0) \\
B &= \text{normalizecols}(A, p) \quad \% \text{nrhs} == 2
\end{align*}
\]

We can determine which way \texttt{normalizecols} was called by checking \texttt{nrhs}. If \texttt{nrhs} = 2, then we should also check the datatype of \texttt{p}.

double p;

\[
\begin{align*}
\text{if}(\text{nrhs} == 1) & \quad /\!\!\!\!\text{ Was p not specified? } /\!\!\!\!\text{*} \\
& \quad p = 2.0; \quad /\!\!\!\!\text{ Set default value for p } /\!\!\!\!\text{*} \\
\text{else} & \\
& \quad \text{if} (\text{mxIsComplex(P\_IN)} \mid\mid !\text{mxIsDouble(P\_IN)} \\
& \quad \mid\mid \text{mxGetNumberOfElements(P\_IN)} != 1) \\
& \quad \quad \text{mexErrMsgTxt("p must be a double scalar.");} \\
& \quad \text{else} \\
& \quad \quad p = \text{mxGetScalar(P\_IN)}; \quad /\!\!\!\!\text{ Get the value of p } /\!\!\!\!\text{*}
\end{align*}
\]
#include "mex.h" /* Always include this */

#define A_IN prhs[0]
#define P_IN prhs[1]
#define B_OUT plhs[0]

void mexFunction(int nlhs, mxArray *plhs[], /* Outputs */
                 int nrhs, const mxArray *prhs[]) /* Inputs */
{
    double p;

    /*** Check inputs ***/
    if(nrhs < 1 || nrhs > 2)
        mexErrMsgTxt("Must have either 1 or 2 input arguments.");
    if(nlhs > 1)
        mexErrMsgTxt("Too many output arguments.");

    if(mxIsComplex(A_IN) || mxGetNumberOfDimensions(A_IN) != 2
        || mxIsSparse(A_IN) || !mxIsDouble(A_IN))
        mexErrMsgTxt("Sorry! A must be a real 2D full double matrix.");

    if(nrhs == 1)        /* Was p not specified? */
        p = 2.0;       /* Set default value for p */
    else
        if(mxIsComplex(P_IN) || !mxIsDouble(P_IN)
            || mxGetNumberOfElements(P_IN) != 1)
            mexErrMsgTxt("p must be a double scalar.");
        else
            p = mxGetScalar(P_IN);    /* Get the value of p */
}
normalizecols: Reading Input A

Now that the inputs are verified, we can safely interpret $A$ as a real 2D full double matrix.

```c
int M, N;
double *A;

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

The elements of $A$ are stored contiguously in memory in column-major format,

$$A[m + M*n] \text{ corresponds to } A(m+1,n+1)$$
normalizecols: Creating Output B

Output variables must be created by the MEX-function. We can create B by

```c
double *B;

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

The interface is now set up!
#include "mex.h" /* Always include this */

#define A_IN prhs[0]
#define P_IN prhs[1]
#define B_OUT plhs[0]

void mexFunction(int nlhs, mxArray *plhs[], /* Outputs */
                 int nrhs, const mxArray *prhs[]) /* Inputs */
{
    int M, N;

    /*** Check inputs ***/
    /* ... */

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

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

    /* TODO: Do the computation itself */
    DoComputation(B, A, M, N, p);
}

Pascal Getreuer (UCLA)
normalizecols: DoComputation

All that remains is to code the computation itself.

```c
#include <math.h>

void DoComputation(double *B, double *A, int M, int N, double p) {
    double colnorm;
    int m, n;

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

        colnorm = pow(fabs(colnorm), 1.0/p);

        /* Fill the nth column of B */
        for(m = 0; m < M; m++)
            B[m + M*n] = A[m + M*n]/colnorm;
    }
}
```
The MEX implementation is certainly more complicated than the M-function. So did our hard work pay off?

Runtime (s) with $p = 2.7$:

<table>
<thead>
<tr>
<th>Input $A$</th>
<th>M-function</th>
<th>MEX-function</th>
<th>MEX-function*</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000 \times 1000$</td>
<td>0.2388</td>
<td>0.0963</td>
<td>0.0927</td>
</tr>
<tr>
<td>$2000 \times 2000$</td>
<td>0.9479</td>
<td>0.3656</td>
<td>0.3599</td>
</tr>
<tr>
<td>$3000 \times 3000$</td>
<td>2.1976</td>
<td>0.8896</td>
<td>0.8550</td>
</tr>
<tr>
<td>$4000 \times 4000$</td>
<td>3.9033</td>
<td>1.5816</td>
<td>1.5128</td>
</tr>
</tbody>
</table>

*With minor optimizations
More Information

Please see **Writing MATLAB C/MEX Code** on my webpage
[www.math.ucla.edu/~getreuer](http://www.math.ucla.edu/~getreuer/)

**Tutorials**

- **Writing MATLAB C/MEX Code**
  MEX: Combine the power of MATLAB and C.

- **Writing Fast MATLAB Code**
  Profiling, JIT, vectorization, and more.

- **Image Processing with MATLAB**
  Reading and writing image files, basic operations, and filtering

- **TikZ for High-Quality LaTeX Pictures**