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

C/MEX	Meaning
`mxCreateDoubleMatrix`	Create a 2D double array
`mxGetPr`	Get pointer to array data
`mxGetDimensions`	Get array dimensions
`mxGetClassID`	Determine variable's datatype

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

Getting Started

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

```
>> 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:

```
>> 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

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

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

C/MEX	Meaning
`nlhs`	Number of output variables
`plhs`	Array of `mxArray` pointers to the output variables
`nrhs`	Number of input variables
`prhs`	Array of `mxArray` pointers to the input variables

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

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

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

C/MEX	M-code equivalent
`nlhs`	`nargout`
`plhs`	`varargout`
`nrhs`	`nargin`
`prhs`	`varargin`

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] = mymexfun(A,B,C)
```

Then `mexFunction`

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

receives the following information:

`nlhs` = 2	`nrhs` = 3
`plhs[0]` points to `X`	`prhs[0]` points to `A`
`plhs[1]` points to `Y`	`prhs[1]` points to `B`
	`prhs[2]` points to `C`

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] = mymexfun(A,B,C)
```

it is our responsibility to create `X` and `Y`.

If `nlhs = 0`, the MEX-function is still allowed return one output variable, in which case `plhs[0]` represents the “`ans`” variable.

normalizecols

Objective: Given matrix A, construct matrix B according to

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

where $\left\| A_n \right\|_p$ is the ℓ^p norm of the nth column.

normalizecols.m — M-function implementation

```
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
```

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

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

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

normalizecols: Converting to MEX

The first step is to figure out the calling syntax. Let's look at

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

This calling syntax in MEX representation is

nlhs = 1	nrhs = 2
plhs[0] points to B	prhs[0] points to A
	prhs[1] points to p

For clarity, it is a good idea to define

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

normalizecols: Converting to MEX

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

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

This calling syntax in MEX representation is

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

For clarity, it is a good idea to define

```
#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

```c
#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.

C/MEX	Meaning
`mxIsDouble(A_IN)`	True for a double array
`mxIsComplex(A_IN)`	True if array is complex
`mxIsSparse(A_IN)`	True if array is sparse
`mxGetNumberOfDimensions(A_IN)`	Number of array dimensions
`mxGetNumberOfElements(A_IN)`	Number of array elements

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

```
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

```
B = normalizecols(A)      % nrhs == 1  (use p = 2.0)
B = normalizecols(A, p)   % nrhs == 2
```

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

```
double p;

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: Input Checking

`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  */
{
    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.

```
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]` 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

```
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!

normalizecols: Interface Complete

`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  */
{
    double p, *A, *B;
    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);
}
```

normalizecols: DoComputation

All that remains is to code the computation itself.

```
#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;
    }
}
```

normalizecols: M-code vs. MEX

The MEX implementation is certainly more complicated than the M-function. So did our hard work pay off?

Runtime (s) with $p = 2.7$:

Input A	M-function	MEX-function	MEX-function*
1000×1000	0.2388	0.0963	0.0927
2000×2000	0.9479	0.3656	0.3599
3000×3000	2.1976	0.8896	0.8550
4000×4000	3.9033	1.5816	1.5128

*With minor optimizations

More Information

Please see Writing MATLAB C/MEX Code on my webpage
www.math.ucla.edu/~getreuer