**ENSC 180 – Introduction to Engineering Analysis Tools**

**Assignment #7: MEX files**

## Background

Functions written in C/C++ can easily be called from MATLAB as if they were MATLAB functions by interfacing through a MEX file. This MEX file is named after the function we wish to implement. It contains the gateway subroutine that passes all the inputs from MATLAB into our C/C++ programming environment. Special built-in functions from the MEX function library are used inside the gateway subroutine to perform this conversion. After this is done, we are free to use these inputs in any C/C++ code. Once our C/C++ subroutines are finished, we send their outputs back to the MATLAB workspace by converting the C/C++ data into MATLAB data through our gateway subroutine. Just as before, this involves using the built-in functions from the MEX function library. The ability to interface C/C++ with MATLAB allows us to overcome many of MATLAB’s shortcomings regarding non-vectorized code.

## Objectives

The task given in this assignment is fairly trivial and does not benefit much from implementation in C/C++ since it can be performed very efficiently with MATLAB vector operations. Nonetheless, the goal here is simply to provide a practical walkthrough on how to interface MATLAB with C++ through MEX. Upon finishing, you will have a working template for integrating C/C++ with MATLAB that is suitable for most situations.

## Procedure

You are given a MATLAB script “demoThreshold\_slow.m”, which reads an RGB color image, converts it to grayscale, and then applies a threshold to the grayscale image. You should run this script and observe the outputs of each of the three steps above. The thresholding operation in the third step is simply:

where is the input image, is the thresholded image, and is the threshold. The function that performs this step is “applyThreshold\_slow” which is purposely written in an inefficient manner (in the context of MATLAB) to make it easier to port to C++.

We wish to design a function:

imgBinary = applyThreshold\_mex(img, T);

with inputs:

* img – The input grayscale image. This is a matrix of size *M*-by-*N*, where *M* is the height of the image and *N* is the width.
* T – The threshold value. It should be an integer between 0 and 255.

In order to do this, we need to create a .cpp file called “applyThreshold\_mex.cpp”. This file will serve as a wrapper. It will pass on the inputs img and T onto our C++ code when we perform the function call shown above in our MATLAB script.

From here on **keep in mind that C++ indexes arrays starting with 0, not 1** and proceed as follows:

**1.** Create folder called “mex” in the directory containing “demoThreshold\_slow.m” and “applyThreshold\_slow”.

**2.** Inside the folder “mex”, create a blank .cpp file called “applyThreshold\_mex.cpp”. Inside this file, include the following,  
  
#include "mex.h"

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

{

}

**3.** The inputs to our function can be fetched using the array of pointers prhs[]. To be specific, prhs[0] is a pointer to the matrix img and prhs[1] is a pointer to the scalar T. We will begin mexFunction by fetching the variable T as follows:

double T = mxGetScalar(prhs[1]);

Since mxGetScalar returns a double, our variable T must also be defined as a double.

**4.** Fetching the matrix img is a little bit more involved. First, we need to find its dimensions. Using the function mxGetDimensions we can obtain a pointer to an array of dimensions. A quick glance at the documentation of mxGetDimensions shows that its output is a pointer of type const int. Thus, we can gain access to this array of dimensions with:

const int \*pSize;

pSize = mxGetDimensions(prhs[0]);

The dimensions in this array are organized in the same order as MATLAB, i.e., pSize[0] is the number of rows, pSize[1] is the number of columns, pSize[2] is the number of pages, etc. Since our input image is simply a 2D matrix, we only have pSize[0], which is the height of our image, and pSize[1], which is the width. Store pSize[0] in a variable IMG\_H and pSize[1] in a variable IMG\_W.

**5.** Initialize a dynamic 2D array double \*\*img with the dimensions from the previous step. We will store the input image in this array.

**6.** The function mxGetPr will give a pointer to the input image. This function only works on arrays of type double, so use it as shown below:

double \*pData;

pData = mxGetPr(prhs[0]);

**7.** Next, we must transfer the data pointed to by pData into the array img. This is to be done using a nested for-loop:

for

{

for

{

img[ ][ ] = pData[ ];

}

}

Note that pData is 1-dimensional. The pixel in the image located at (row, col) would be pointed to by pData at the location corresponding to the linear index of (row, col). For example,

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **j** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| **pData[j]** | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t |

would correspond to the matrix

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | e | i | m | q |
| b | f | j | n | r |
| c | g | k | o | s |
| d | h | l | p | t |

**8.** Just like in Step 5, Initialize a dynamic 2D array double \*\*imgBinary with the dimensions IMG\_H and IMG\_W. We will store the output image to this array.

**9.** At this point, create a new file “threshold.h”. Inside it, place the following code

using namespace std;

void threshold(double \*\*imgBinary, double \*\*img, double T, const int IMG\_H, const int IMG\_W);

Although this seems redundant for such a simple implementation, more complex projects benefit from this sort of organization.

**10.** The function that performs the thresholding procedure will be defined in a separate file “threshold.cpp”. Create a blank .cpp file with this title and place the following in this blank file:

#include "threshold.h"

using namespace std;

void threshold(double \*\*imgBinary, double \*\*img, double T, const int IMG\_H, const int IMG\_W)

{

}

**11.** Implement the thresholding procedure using a nested for-loop exactly as shown in “applyThreshold\_slow.m”.

**12.** Returning back to our wrapper file “applyThreshold\_mex.cpp”, add #include "threshold.h" to the top, before mexFunction begins.

**13.** Continuing on after the code from Step 8, make a call to the threshold function with the necessary inputs.

**14.** To output the thresholded image, we need to allocate space for an array of real values using the mxCreateDoubleMatrix function:

plhs[0] = mxCreateDoubleMatrix(IMG\_H, IMG\_W, mxREAL);

double \*outputMatrix = mxGetPr(plhs[0]);

(Note: The thresholded image is binary and hence can be represented as a Boolean array. However, to avoid issues related to type-casting and to simplify things, we mostly just use the double data type in this assignment). Recall that plhs[0] is a pointer to the first output. In the above code, outputMatrix points to the first element of the array we created with mxCreateDoubleMatrix.

**15.** Finally, we need to transfer the contents of imgBinary to the locations pointed at by outputMatrix. This is essentially the reverse of Step 7:

for

{

for

{

outputMatrix[ ][ ] = imgBinary[ ];

}

}

**16.** Save all .cpp and .h files. In order to compile these files, make sure you have a default compiler selected by using the command mex -setup in the MATLAB terminal. The command to compile our files (from the directory in which “demoThreshold\_slow.m” is located) is

mex '.\mex\applyThreshold\_mex.cpp' '.\mex\threshold.cpp'

You may use the file “threshold\_compile.m” to do this for you. Simply run this file in MATLAB. Remember that every time a change is made in the C++ code, it has to be recompiled in MATLAB, so a file like this is convenient to have.

**17.** Create a new MATLAB script called “demoThreshold\_mex.m” and copy the contents of “demoThreshold\_slow.m” directly into it. Then, replace the following line

imgBinary = applyThreshold\_slow(imgGray, T);

with our newly created MEX function

imgBinary = applyThreshold\_mex(imgGray, T);

Ensure that you obtain the same results with the MEX function as you do with the original MATLAB implementation.