Tutorial: C mex matrix demo

December 2007
Nick Clark & James Tursa

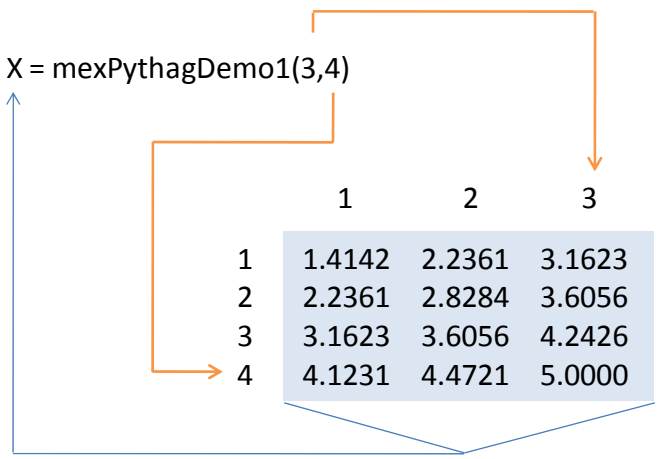
Introduction:

This submission was created to help other programmers, who are familiar with MATLAB, be able to process data from MATLAB in a C program, and pass it back to MATLAB in an elegant and intuitive manner. The example contained is a trivial function, but should provide a good stepping stone for others entering this field.

This file was originally uploaded to the Mathworks website on the 11th December 2007 and after just a few days of it being in place there, James Tursa contacted me via email with numerous additions, improvements and suggestions showing a number of ways of thinking about the fundamentally simple act of passing multi-dimensional data between MATLAB functions and compiled C functions in mex format. This document is a copy of the email conversations between us, so that the reader is able to follow the modifications between each of the source code files contained in the zip.

The Function:

The function calculates two dimensional Pythagorean distances between two axis ($z = (x^2 + y^2)^{1/2}$) where the user can specify the length of each axis individually. The following diagram illustrates the behaviour of the mex functions contained in this script. The box highlighted in blue is the output of the function when supplied with the input parameters as shown.



The same result could just as easily be obtained using the MATLAB command (as long as you are using a version no older than R2007a):

```
sqrt(bsxfun(@plus,[1:3].^2,[1:4]'.^2))
```

Users of previous Matlab versions can obtain this result using the following, much slower antiquated code:

```
sqrt( repmat([1:4]',1,3).^2 + repmat(1:3,4,1).^2 )
```

Now would be a good point to compile and run the first of the mex functions and familiarise yourself with the behaviour. The following section is a list of email correspondence between James and I on various improvements. The updated C code in the other files in this zip correspond to these discussions.

Emails:

Hi, I downloaded your file and gave it a look over. Nice! Cute trick with the indexing. I do have some comments that you may find useful improvements. Enclosed is a modified version of your file. By allocating all of the memory at once you ensure contiguous memory for the matrix, that way you can just do a block memory copy, and freeing the memory is slightly simpler. You kind of glossed over the fact that your mxArray is the transpose of the c matrix. Also The Mathworks recommends using mxCalloc and mxFree instead of calloc and free to ensure their memory manager can free all memory if there is an early error exit from the mex program. And in general, if you *do* use calloc and free you should be including the headers stdlib.h and malloc.h also. Finally, I don't understand the comment you made on this line: release_matrix_space(a, xDim); //Not necessary and slower than MATLAB! You used calloc to allocate the memory from the heap, so why do you make the comment that you don't really need to free it? I will wait a couple of days in case you have any comments back for me, and then I will post a response on the file sharing site (good comments and a good rating, I promise!) James Tursa

Wow! Thankyou for taking the time to make some really useful comments. This is what the MATLAB community should be all about (rather than people asking me to do their signal processing homework LOL). The memcopy idea is way better. I realised the transpose thing after I uploaded (oops). My two main questions on your updates are:

```
a = (SAMPLE **) mxCalloc(xDim, sizeof(SAMPLE *)); // Use mxCalloc instead of calloc

  --a; //Here we offset the pointer for the array of arrays

  a[1] = (SAMPLE *) mxCalloc(xDim*yDim, sizeof(SAMPLE)); // Get all of the memory at once

  --a[1]; // Offset the first a[x]
```

Why do we re-offset a[1] at the bottom (I get the memory at once thing, nice, but don't get this – I'm probably being dumb)? My second question is with regard to freeing memory. I only picked up a book on C for the first time last week, and it says to make sure I free stuff when I've used

malloc/calloc. Is this not necessary then if the function is going to terminate (will memory be freed before it is called again?). If I use mxCalloc, does the memory freeing become necessary now or is it just good practice? Sorry for my totally newbie questions!

Thanks again so much for using your time to look over this, I'll be sure to update the submission with your masterful version next week (got an auditory model which needs to be set up today before the weekend) and remove the transpose.

> "I realised the transpose thing after I uploaded (oops)."

I didn't mean to imply that this was necessarily a bad thing. Using the transpose allows you to do a block memory copy, which is good. I just meant that it was not pointed out in your code or comments that this was taking place, so a reader might miss this very important point. But if you want a same shape matrix in c you will need code similar to your original code that does the copy element-by-element. This is simply the price you have to pay. The user simply has to pick their poison, so to speak.

> "Why do we re-offset a[1] at the bottom (I get the memory at once thing, nice, but don't get this – I'm probably being dumb)?"

I decremented a[1] here simply to keep the code consistent with your offset indexing scheme. No other reason.

> "My second question is with regard to freeing memory. I only picked up a book on C for the first time last week, and it says to make sure I free stuff when I've used malloc/calloc. Is this not necessary then if the function is going to terminate (will memory be freed before it is called again?). If I use mxCalloc, does the memory freeing become necessary now or is it just good practice?"

Wow! You wrote this mex routine after only a week of looking at a C book???!!! That's impressive. Most people coming out of a regular 1st semester C class in college can't write code this good. Anyway, to answer your question, here are some examples that explain it.

Example 1)
Heap memory lost because you didn't call free

```
#include "mex.h"
void mexFunction(int hlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
char *cp;
cp = (char *) malloc(1000000);
mexPrintf("cp = %p\n",cp);
}
```

Call this file heaplost.c and mex it, then run it several times. You will see the memory address of the allocated memory keep going up. This is because the allocated heap memory from previous calls was never freed and becomes leaked when the mex file exits. Not good. If you keep running heaplost enough times (e.g., put it in a loop in an m-file) you will exhaust the heap and crash MATLAB.

Example 2)
Heap memory saved because you called free

```
#include "mex.h"
void mexFunction(int hlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
char *cp;
cp = (char *) malloc(1000000);
mexPrintf("cp = %p\n",cp);
free(cp);
}
```

Call this file heapsave.c and mex it, then run it several times. You will see the memory address of the allocated memory stay constant. This is because the allocated heap memory from previous calls was freed and becomes available for the next mex routine call. Good. You can run this indefinitely and not exhaust the heap. MATLAB will not crash.

Example 3)
Heap memory lost anyway because an error occurred before you called free

```
#include "mex.h"
void mexFunction(int hlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
char *cp;
cp = (char *) malloc(1000000);
mexPrintf("cp = %p\n",cp);
mexErrMsgTxt("malloc error return without free(cp)");
free(cp);
}
```

Call this file heaplost2.c and mex it, then run it several times. You will see the memory address of the allocated memory keep going up. This is because the allocated heap memory from previous calls was never freed and becomes leaked when the mex routine exits. The reason it was not freed is because an error occurred that caused the mex routine to exit before you reached the free(cp) line, which was never executed. Since the MATLAB memory manager doesn't know anything about memory allocated with malloc, calloc, or realloc, this memory gets leaked. Not good. If you keep running

heaplost2 enough times you will exhaust the heap and crash MATLAB.

Example 4)
Heap memory saved even though an error occurred before you called mxFree

```
#include "mex.h"
void mexFunction(int hlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
char *cp;
cp = (char *) mxMalloc(1000000);
mexPrintf("cp = %p\n",cp);
mexErrMsgTxt("mxMalloc error return without mxFree(cp)");
mxFree(cp);
}
```

Call this file heapsave2.c and mex it, then run it several times. You will see the memory address of the allocated memory stay constant. This is because the allocated heap memory from the mxMalloc call is registered with the MATLAB memory manager, so when the error occurrs the MATLAB memory manager automatically frees it for you even though you never executed the mxFree(cp) call. This is good. You can run this indefinitely and not exhaust the heap. MATLAB will not crash. This is why you should always use mxMalloc, mxCalloc, and mxRealloc instead of the malloc, calloc, and realloc routines.

I always put mxFree calls in my mex routines for good programming practice, even though there is a safety net here in case of an early error exit. Also, if you are working with very large arrays if can become important to free the memory as soon as you are done with it so that other parts of your code that need heap memory have access to it.

Finally, I recommend that you exit MATLAB and restart it after doing these tests because you will have a very large bunch of leaked heap memory that needs fixed up.

Cheers,

James Tursa

One more version for you to consider. This one doesn't do any memory copy at all ... it simply writes directly into the data area of the mxArray itself. For this particular example it is a bit more efficient.

James Tursa

(This final email corresponds to the third version of the source code)