Writing MEX Files for Image Processing

M. P. Foster
mattfoster@clara.co.uk
http://www.mattfoster.clara.co.uk

November 21, 2006

1 Introduction

This brief document describes some of most important aspects of using MEX (MATLAB Extension) files to process images. Most of the information here could also be applied to writing MEX files for any purpose, but specific examples will be given with image processing applications in mind.

MATLAB's External Interfaces guide [Mat02], describes MEX-files as:

> "Dynamically linked subroutines that the MATLAB interpreter can automatically load and execute"

The main advantages of using MEX-files over plain m-files are the fact that loops, and other operations that MATLAB struggles with should be much faster when written in a compiled language than in native MATLAB, and that existing programs can easily be ported to run from within MATLAB. Conversely, if a program is written in say C, with a MEX interface, porting it to other applications should be considerably easier than porting an m-file would be, and should result in much cleaner code than using MATLAB's C engine would.

For a more detailed discussion of where and when it's a good idea to use MEX-files have a look at MATLAB's External Interfaces guide.

The best way of thinking about MEX-files, is to consider them as an interface, with MATLAB on one side, and your program on the other. All you need to do is write a gateway function, to pass data between the two, and you're done. By using the MEX-file as a gateway, and keeping your MEX code and functional code separate, it will be much easier to reuse your code later, or to build a normal programme as a frontend or for debugging. It's also a good idea because it *elimate effects between unrelated things* (one of the Pragmatic Programmer's [HT99] mantras), making it easier to mend and debug things when they break.

2 The Gateway Function

As mentioned above, the gateway function controls the passage of data between MATLAB, and your external program. In many ways, it's like the `main` function of a C program, since it's the first function to be called when invoking a MEX-file, and it always has the same prototype, like so:

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

As shown above, it returns `void`, meaning that all of the input and output variables are passed as pointers, rather than as copies.

Now, consider calling a MEX-file, called `exampleMex` from within MATLAB:

```
>> [a, b] = exampleMex(c, d, e);
```

`nlhs` and `plhs` relate to the arguments on the left hand side of a call to the MEX-file from within MATLAB. `nlhs` gives the number of arguments, and `plhs` is an array of pointers to the variables. Similarly, `nrhs` and `prhs` hold the number of arguments to the function call (the right hand side of the MATLAB expression), and an array of pointers to those variables, respectively. So, using the function call above as an example, `nlhs` would be two, `nhrs` would be three, `plhs` would contain pointers to the (initially empty) variables `a` and `b`, and `prhs` would point to the variables `c`, `d` and `e`, which shouldn't be empty.

3 Data Types

There's only one data type in MATLAB - the `mxArray`. `mxArrays` in which all types of MATLAB variable can be stored. In some ways, this makes life easier, as it reduces the number of data types to you need to remember, but it does mean that you need to be careful, and not make any assumptions about the format of the contents of the array. Thankfully, there are various functions which can help you determine what is inside of an `mxArray` and act accordingly.

To access the data within an `mxArray`, you need to get a pointer to it. The function `mxGetPr` does this, as shown below.

```
double        *a;
a = mxGetPr(phrs[0]);
```

To check if a given `mxArray` contains double precision floats, use:

```
if (!mxIsDouble(prhs[0])) {
    mexErrMsgTxt("Input image should be double.\n");
}
```

Various other functions exist which operate on `mxArrays`, including `mxGetN` and `mxGetN` which get matrix dimensions, and myriad `mxCreate` functions for setting up `mxArrays` of each MATLAB data type. To create a 10-by-10 matrix of real doubles, use:

```
p = mxCreateDoubleMatrix(10, 10, mxREAL);
```

This command will initialise the memory to zero, which is unlike a call to `malloc`, which just allocates a chunk of heap. As an aside, if you normally use `malloc`, then consider trying `calloc` instead, which will zero memory for you.

4 Addressing Matrices

Data within `mxArrays` is stored as one blob of memory, *columnwise*. This means that a matrix is addressed as shown in figure 1:

This is probably contrary to most people's expectations, and so is something to watch out for. For example, to address the element in figure 1 containing 3 (0, 1), you'd need to look at the second element.

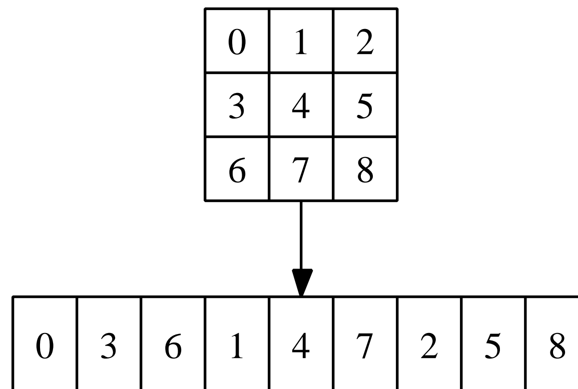

Figure 1: mxArray memory layout.

To convert from x and y coordinates to linear columnwise addressing, use: $i = x \times r + y$ Where r is the number of rows in the matrix.

5 A Small Example: Sobel Edge Detection

Sobel edge detectors work by convolving an image with two masks, which give a measure of the strength of vertical and horizontal edges. These can be combined to give magnitude and direction, which can then be used for segmentation, filter steering etc. For more information on edge detection filters, refer to an image processing textbook (e.g: [GW01]).

This example program Sobel filters a a given image, and returns two double matrices containing the vertical and horizontal edge strengths. To build and run the example code on a *NIX machine, with GCC and make, and the prototype generator `cproto`, save all of this code in a directory, using the filenames given above the code, and type `make`. If everything builds successfully, it should be possible to call the function from within MATLAB using the command:

```
>> [x, y] = eg(img);
```

To rename the function within MATLAB, rename `eg.c` and alter the makefile accordingly (or just change the `FUNCTION` variable).

Gateway function `eg.c`:

```
/* Sobel filter MEX-gateway */

#include <mex.h>
#include "sobel.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
        const mxArray *prhs[])
{
    double          *img, *out_h, *out_v;
    int                 rows, cols;

    /* Declarations*/
    if (nrhs != 1) {
```

```
        mexErrMsgTxt("One input arguments required.\n");
    } else if (nlhs > 2) {
        mexErrMsgTxt("Too many output arguments.\n");
    }


    /* We only handle doubles */
    if (!mxIsDouble(prhs[0])) {
        mexErrMsgTxt("Input image should be double.\n");
    }

    img   = mxGetPr(prhs[0]);
    cols  = mxGetN(prhs[0]);
    rows  = mxGetM(prhs[0]);

    /* Create output matrices: */
    plhs[0] = mxCreateDoubleMatrix(cols, rows, mxREAL);
    out_h   = mxGetPr(plhs[0]);
    plhs[1] = mxCreateDoubleMatrix(cols, rows, mxREAL);
    out_v   = mxGetPr(plhs[1]);

    sobel(img, rows, cols, out_h, out_v);

    return;
}
```

Sobel filtering function `sobel.c`:

```
/* Sobel filter an image */

int coord(int i, int j, int cols)
{
    return j * cols + i;
}

/* Perform a very simple unrolled convolution: */
double sobel_v(double *img, int x, int y, int cols)
{
        double val = 0;

        val += img[coord(x-1, y-1, cols)] * -1;
        val += img[coord(x  , y-1, cols)] * -2;
        val += img[coord(x+1, y-1, cols)] * -2;

        val += img[coord(x-1, y+1, cols)] *  1;
        val += img[coord(x  , y+1, cols)] *  2;
        val += img[coord(x+1, y+1, cols)] *  2;

        return val;
}

double sobel_h(double *img, int x, int y, int cols)
{
```

```
        double val = 0;

        val += img[coord(x-1, y-1, cols)] * -1;
        val += img[coord(x-1, y  , cols)] * -2;
        val += img[coord(x-1, y+1, cols)] * -2;

        val += img[coord(x+1, y-1, cols)] *  1;
        val += img[coord(x+1, y  , cols)] *  2;
        val += img[coord(x+1, y+1, cols)] *  2;

        return val;
}

void sobel(double *img, int rows, int cols, double *out_h, double *out_v)
{
    int i, j;

    /* Loop over the image. Note the bounds start at one so we don't read
     * off the edge. */
    for (i = 1; i < rows - 1; i++) {
        for (j = 1; j < cols - 1; j++) {
            out_h[coord(i, j, cols)] = sobel_h(img, i, j, cols);
            out_v[coord(i, j, cols)] = sobel_v(img, i, j, cols);
        }
    }

}
```

Sobel header file —`sobel.h`:

```
/* sobel.c */
int coord(int i, int j, int cols);
double sobel_v(double *img, int x, int y, int cols);
double sobel_h(double *img, int x, int y, int cols);
void sobel(double *img, int rows, int cols, double *out_h, double *out_v);
```

`Makefile`:

```
##
## Mex Options:
##

MATLAB           = /usr/local/matlab7.1
OSTYPE     = linux
CC         = cc
CMEX       = $(MATLAB)/bin/mex
DEBUG      = -g -O -v
MEXSUFFIX  = mexglx

##
## Linker/Loader flags
##
```

```
LDFLAGS    =
INCLUDES   = -lm
CFLAGS     = -g $(INCLUDES)
PROGRAM          = eg
FUNCTION   = $(PROGRAM).$(MEXSUFFIX)
MAKEFILE   = Makefile
HDRS           =
SRCS           = sobel.c
MAINSRC          = $(PROGRAM).c
HEADER     = include.sh
LIBS       = $(LOCAL_LIBS) $(SYS_LIBS)

##
## Targets
##

all : include $(FUNCTION)

$(FUNCTION): $(MAKEFILE)

sources : $(SRCS) $(HDRS)

include: $(SRCS)
        @echo 'Generating headers for $+'
        $(HEADER) $+

$(FUNCTION): $(MAINSRC) $(SRCS) $(HDRS) Makefile
        $(CMEX) $(DEBUG) $(CFLAGS) $(LDFLAGS) $(SRCS) $(MAINSRC) \
        $(LIBS) -o $(FUNCTION)

standalone: main.c $(SRCS)
        $(CC) -o anisoGauss -Wall -ansi $(CFLAGS) $(LDFLAGS) $+

delmex:
        rm -f $(FUNCTION)

clean : delmex
        rm -f *~ core .depend.$(OSTYPE)

tags : sources
        etags -t $(SRCS) $(HDRS)

.PHONY:    all sources delmex full
.PHONY:    clean tags
```

Frontend for cproto, `include.sh`:

```
#! /bin/sh

for x in $*; do
                name=`basename $x .c`
                cproto $name.c > $name.h
```

```
done
```

6 Conclusions

Hopefully, this short report will have shown how simple using MEX-functions can be, as well as a few pitfalls to avoid. Whilst the code has only been tried under Linux, there's not reason at all why it should also compile and run within Windows, but you may need to adjust the Makefile for your build environment.

Finally, this document is licensed under the Creative Commons Attribution 2.5 License, which means you're pretty much free to do as you like with it, and the code contained within, provided you mention me as the original source. For more information, and the full license, see:
http://creativecommons.org/licenses/by/2.5/.

The original document will always be available from:
http://www.mattfoster.clara.co.uk/files/image_mex.pdf.

References

[GW01] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing 2/E*. Prentice Hall, 2001.

[HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley, 1999.

[Mat02] Mathworks. *MATLAB External interfaces*, 2002.