MEX-File Programming

for Image Processing Using DIP*image*

dr. ir. Cris L. Luengo Hendriks

Quantitative Imaging Group,
Department of Applied Sciences,
Delft University of Technology

Delft
May 11, 2007

Contents

Chapter 1

Introduction

1.1 MATLAB Scripting Language Versus C

MATLAB scripts are very slow when it comes to loops. In image processing, it sometimes is necessary to visit each pixel in turn to compute something. This will often require a loop or a set of loops. Sometimes, however, the loops can be avoided by *vectorizing* the code. This means that a function is applied to all pixels at once; MATLAB does the looping implicitly. In the cases where a computation is not vectorizable, *and* speed is important, it is possible to re-write the MATLAB script in C. A MATLAB function written in a compiled language is called a MEX-file (for MATLAB executable file).

1.2 Possibilities Within a MEX-File

By making the step to C, complex mathematical expressions cannot be written in a simple manner any more. However, since it is possible to call back to MATLAB, you can ask it to evaluate any MATLAB command. This enables the programmer to write efficient code in C without loosing the flexibility of the interpreted language. It also means that everything you can do from an M-file, you can also do from a MEX-file.

1.3 The MATLAB Compiler

The MATLAB Compiler can generate C code from M-files, which can, in turn, be compiled into MEX-files. This looks interesting, but it lacks a feature (at the time of this writing), which makes it useless in combination with DIP*image*: it is not possible to compile M-files that use objects and object methods. I hope this changes soon. The MathWorks, creators of MATLAB, have been promising for two years that a future version of their compiler will be able to handle objects.

If you want to use the MATLAB Compiler to generate C code for you, don't use the `dip_image` object or any of the DIP*image* functions. Also do not call any MEX-file (such as the DIP*lib* functions).

1.4 Structure of This Document

The next chapter deals in avoiding loops in MATLAB scripts. It gives a set of tips and hints towards vectorizing your algorithms. Chapter 3 shows how to write and compile MEX-files, and Chapter 4 deals with the `dip_image` object. Chapter 5 explains some advanced topics

concerning memory management, compiling complex projects and debugging them. Finally, Chapter 6 explains how to call your own DIP*lib* code from MATLAB, and how to use calls to DIP*lib* functions in your MEX-files. This chapter is only intended for people that have read the DIP*lib Programmers Guide*.

1.5 Documentation Conventions

The following conventions are used throughout this manual:

- Example code: in `typewriter` font
- File names and URLs: in `typewriter` font
- Function names/syntax: in `typewriter` font
- Keys: in **bold**
- Mathematical expressions: in *italic*
- Menu names, menu items, and controls: "inside quotes"
- Description of incomplete features: in *italic*

Chapter 2

Vectorizing Algorithms

2.1 General Guideline

It is worth the effort to try to vectorize an algorithm only if it takes too long to run. The term "too long" is subjective, but should be related to the time that you need to rewrite the algorithm.

A piece of code that is called many times is more interesting to optimize than one that is called only once. A piece of code that takes up a large portion of the total time of the algorithm is more interesting to optimize than the rest.

The functions `tic` and `toc` can be used to measure the time spend by a function or group of commands. `toc` returns the amount of time (in seconds) elapsed since the last call to `tic`. The function `profile` provides a more comprehensive way of profiling your functions.

The general idea of vectorization is that writing

```
a = a*2;
```

is much better than writing

```
for ii=1:prod(size(a))
   a(ii) = a(ii)*2;
end
```

In this case it is obvious how to write the vectorized form of the expression, but in other cases it is less so. Take as an example a neighborhood operation. A uniform filter can be written in this way (note that the indexing used assumes `in` and `out` are `dip_image` objects):

```
for ii=1:size(a,1)-2
   for jj=1:size(a,2)-2
      out(ii,jj) = (in(ii-1,jj-1)+in(ii,jj-1)+in(ii+1,jj-1)+...
              in(ii-1,jj)+in(ii,jj)+in(ii+1,jj)+...
              in(ii-1,jj+1)+in(ii,jj+1)+in(ii+1,jj+1))/9;
   end
end
```

This is very slow. Another way to do it is this:

```
out = (in(0:end-2,0:end-2)+in(1:end-1,0:end-2)+in(2:end,0:end-2)+...
       in(0:end-2,1:end-1)+in(1:end-1,1:end-1)+in(2:end,1:end-1)+...
       in(0:end-2,2:end)+in(1:end-1,2:end)+in(2:end,2:end))/9;
```

The second method is much quicker, but also requires more memory. In the example above, 9 temporary images were made, against none in the first method.

Functions such as `find`, `repmat` and `reshape` are often used to avoid writing loops. Much like the example above, the drawback of using them is the need for more memory.

2.2 Using repmat and reshape

Imagine you have a MATLAB array in which each row represents a histogram of some sort. You want to normalize those histograms by dividing each row in the array by its sum. The direct (non-vectorized) way of doing this is:

```
for ii=1:size(h,1)
   h(ii,:) = h(ii,:)/sum(h(ii,:));
end
```

This can be vectorized by using `repmat`. `repmat` will replicate or tile an array a number of times in any direction. For example, `repmat(x,1,2)` is the same as `[x,x]`, and `repmat(x,100,300)` is the same as `[x,x,x,...;x,x,x,...;...]`, an array of size `size(x).*[100,300]`. Thus, the above example can be written as

```
h = h./repmat(sum(h,2),1,size(h,2));
```

`sum(h,2)` is the sum over the rows of `h`, and the result is replicated to the same size as `h`. This matrix can be used to divide `h` by.

Another example for the use of repmat is subtracting from each slice in a 3D image the same 2D image:

```
a = readim
b = scalespace(a)
b - repmat(a,1,1,size(b,3))
```

`reshape` is used to change the shape of a matrix (or image). This can be useful in many cases, as in the next example. Imagine you want to flip every block of 8 rows in an image. The simplest way to do this is to reshape the image to 8 rows, flip them, and then reshape the image to its original size:

```
b = reshape(a,prod(size(a))/8,8);
b = flipud(b);
b = reshape(b,size(a))
```

Note that `reshape` takes rows from the image (that is, the second dimension (y) is taken first). This is consistent with indexing using a single index, where that index also increases first along the y-dimension.

2.3 Generating Coordinate Images

Especially when creating test images, it is required to do some computations that involve the coordinates of a pixel. These operations can be vectorized by creating images that contain these coordinates. Like with the use of `repmat`, this speeds up calculations at the expense of larger memory requirements. The following functions fall in this category:

`xx`	distance along the x-axis from the center of the image.
`yy`	distance along the y-axis from the center of the image.
`zz`	distance along the z-axis from the center of the image.
`rr`	distance from the center of the image.
`phiphi`	angle around the center of the image from the horizontal (0 being to the right).

All of these have the same syntax. They allow both a size vector for the output image, or an image whose size is to be taken. The center of the image is defined consistently with the Fourier transform in DIP*lib*, to the right of the true center if the size is even:

```
[-4,-3,-2,-1,0,1,2,3]
```

For example, to create concentric circles, use `cos(rr)`.

The functions `meshgrid` and `ndgrid` can be used to accomplish similar things with MATLAB arrays, instead of `dip_image` objects.

2.4 Using `find` and Mask Images

To apply operations selectively to certain pixels only, a mask image can be created and used to index the image. For example, setting all negative pixels to 0 is easy this way:

```
a(a<0) = 0;
```

Another example is to multiply the values of all non-zero pixels:

```
prod(double(a(a~=0)));
```

(since `prod` is not defined for the `dip_image` object, we convert the pixel data to doubles first). Note that by indexing using a mask image, we create a 1D image with the selected pixel values.

```
m = a~=0;
a(m)
```

is the same as

```
I = find(a~=0);
a(I)
```

The array `I` above contains indices into the image `a`, and can be used to index it in the same way as a mask image. Note that the indices in `I` go down and then to the right (`y+x*size(a,2)`). `I` can be used when looping is unavoidable, and you want to address each of the pixels in the mask. Thus, instead of:

```
m = a~=0; Q = 1;
for ii=0:prod(size(m))-1
   if m(ii)
      a(ii) = Q;
      Q = Q+1;
   end
end
```

you can write

```
I = find(a~=0);
for ii=1:length(I)
   a(I(ii)) = ii;
end
```

(Note that the above actually is vectorizable, but I couldn't think of a simple application that is not.)

Chapter 3

A Basic MEX-File

This chapter shows the basic form and elements of a MEX-file written in C. You can also write MEX-files in C++ and FORTRAN, which is very similar.

3.1 The Gateway Routine: `mexFunction`

Each MEX-file must have a function called `mexFunction`, with a pre-defined set of parameters. This is the function that MATLAB calls when you type the name of the MEX-file at the command prompt. This is the smallest MEX-file:

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
}
```

which might or might not compile depending on your compiler. It obviously does nothing. To compile it, type

```
mex mymexfile.c
```

at the MATLAB command prompt. This will create a MEX-file called `mymexfile`, which you can then execute in the same way you would execute an M-file. Read "Application Program Interface Guide" (in the MATLAB manual set) for instructions on customizing the `mex` script.

The four parameters to `mexFunction` are `nlhs`, the number of left-hand side parameters, `plhs[]`, the array of left-hand side parameters, `nrhs`, the number of right-hand side parameters, and `prhs[]`, the array of right-hand side parameters. `plhs[]` and `prhs[]` are arrays of pointers to `mxArray` structures, described in the next section. It is very important to note the const qualifier on the right-hand side parameter array. You are not supposed to change the input arrays. Create a new array to write values to. You can pass an input array as output, but you should not change it.

3.2 The `mxArray`

The `mxArray` is the structure that encapsulates a MATLAB array. All of the array types can be represented in such a structure. The "Application Program Interface Reference" lists many functions to deal with the `mxArray`, including functions to create and destroy all types of arrays, and to fill elements of the structure and cell arrays. These all start with `mx`, and are too many to mention here. Just read the online reference.

A second set of functions available to the MEX-file programmer are those that start with `mex`.

They can be used for the interaction with MATLAB, and contain things like `mexErrMsgTxt`, `mexCallMATLAB`, and `mexPrintf`. Again, see the reference manual.

The best way to show the usage of the `mxArray` is through an example. This function is called `rmsd`, and calculates the root mean square value of the input data. The call `rmsd(a)` is the same as `sqrt(sum(a(:).^2))`.

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
   double res = 0;
   double *in;
   int ii, nel;
   /* Check for proper number of input and output arguments */
   if (nrhs != 1) {
      mexErrMsgTxt("One input argument required.");
   }
   if (nlhs > 1) {
      mexErrMsgTxt("Too many output arguments.");
   }
   /* Check data type of first input argument */
   if (!mxIsDouble(prhs[0]) || mxIsComplex(prhs[0])) {
      mexErrMsgTxt("Input argument must be a real double.");
   }
   /* Do the calculations */
   nel = mxGetNumberOfElements(prhs[0]);
   in = mxGetPr(prhs[0]);
   for (ii=0; ii<nel; ii++) {
      res += *in * *in;
      in++;
   }
   res = sqrt(res);
   /* Create an output matrix and put the result in it. */
   plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
   mxGetPr(plhs[0])[0] = res;
}
```

The function first checks the number of input and output arguments. Note that `nlhs` can be zero, but `plhs[0]` is always defined. This is because, if no output arguments are given by the user, the output argument is put into `ans`.

The second part of the function checks the type of the input matrix. It must be `double` or else we cannot handle it (we could, if we wanted, though!)

Next we retrieve the number of elements in the array, and the pointer to the first element. We loop over all elements, adding their square up as we go. Finally, we take the square root.

The last part of the function allocates an `mxArray` structure, assigns it to the output and puts the result in it.

By examining the list of `mx...` functions, you should be able to get an idea of the possibilities open to you. However, for image processing, not much more than the above statements are necessary.

Chapter 4

Using dip_image Objects in a MEX-File

4.1 Calling Back to MATLAB

The dip_image object is seen as a structure array inside the MEX-file, but the class name is recognizable. Thus, the call

```
if (mxIsClass(prhs[0], "dip_image")) ...
```

can distinguish if the input is an object of type dip_image. It is then possible to extract each of the elements of the structure. But you shouldn't do this, since it would make your code less robust against changes in the internal definition of the dip_image class. Correct would be to call double in MATLAB to get an array with doubles. This is then easy to process using mxGetPr:

```
mxArray *mxdata;
double *data
mexCallMATLAB(1, &mxdata, 1, &prhs[0], "double");
data = mxGetPr(mxdata);
```

If mxdata contains complex data, you also would want to call mxGetPi.

4.2 Retrieving Pixel Data in a Specific Format

If you don't want to convert the pixel data into doubles, but single float values, call the function single instead of double. Other available functions are: uint8, uint16, uint32, int8, int16 and int32. Finally, the function dip_array will return an array of the type that was originally there, without any conversions. Next call mxGetClassID to extract the data type, mxIsLogical will return true if the image was binary, and mxGetData returns a void pointer to the data (you will have to cast it to the appropriate data type). Note that it is difficult to write a function that can handle any data type; it is better to convert to singles or doubles. Finally, mxIsComplex returns true if there is an imaginary part, which can be obtained with mxGetImagData.

4.3 Returning dip_image Objects from a MEX-file

Using the same syntax as before, we can convert any numerical array into an object of type dip_image by calling MATLAB:

```
mexCallMATLAB(1, &plhs[0], 1, &mxdata, "dip_image");
```

To create a `dip_image` of another type is a bit more tricky:

```
mxArray *args[2];
args[0] = mxdata;
args[1] = mxCreateString("sfloat");
mexCallMATLAB(1, &plhs[0], 2, args, "dip_image");
```

Furthermore, things like concatenating images into an image array, creating color images, doing arithmetic with images or applying any previously defined function to an image should be done through callbacks to MATLAB. This is the more efficient way of doing them (in terms of your time).

It is best if you only re-write in C that portion of you algorithm that is too slow. Make a private MEX-file that you call at the substituted portion of your algorithm. This way, only you have to call it, and you don't need waste too much time on correctly parsing all the (possibly wrong) input values. Besides, MATLAB gracefully kills any MEX-file that produces segmentation errors, so you really don't need to worry about inputs.

Chapter 5

Other Topics

5.1 MATLAB Memory Management

MATLAB should handle all memory for you. All you need to do is create arrays; they will be destroyed automatically when your function returns or when you call `mxErrMsgTxt`, which quits your function. However, it is possible to delete temporary arrays halfway a calculation to free up memory for other arrays. It is also possible to create static arrays, which stay in memory from one call to the next.

5.1.1 Removing Arrays from Memory

A call to `mxDestroyArray` removes both the `mxArray` structure and the associated data. You should never destroy an array passed to you by MATLAB in the `prhs[]` array, and neither should you destroy an array you want to pass back to MATLAB through the `plhs[]` array. Arrays that you don't destroy explicitly will be destroyed by MATLAB upon finishing the execution of your `mexFunction`.

5.1.2 Allocating Memory for Other Purposes

`mxMalloc`, `mxCalloc`, `mxRealloc` and `mxFree` should be used inside MEX-files instead of their C counterparts (`malloc`, `calloc`, `realloc` and `free`). Memory allocated through these functions will be freed automatically when your function ends.

Memory allocated using these functions can be inserted into an `mxArray` as the real or imaginary part of the data (see `mxSetPr` or `mxSetData` and `mxSetN` or `mxSetDimensions`).

5.1.3 Making Arrays Persistent (Static)

In the event that you want some data to be available from one function call to the next, you can create a persistent `mxArray`, which won't be freed until you do so explicitly. You are responsible for doing so, if you don't, MATLAB will leak memory. By registering a function with `mexAtExit`, you can make sure that the `mxArray` will be freed when your MEX-file is cleared (which happens when the user types `clear mex` or `clear all` at the MATLAB command prompt). If you don't want your function to be cleared, you can lock it with `mexLock`. The next example illustrates this. It is a function that generates a random value (calling MATLAB to do so!), and stores it in a persistent array. Every time the function is called, the same value is returned. However, after clearing the MEX-file, the value is destroyed, and a new one must be generated. Enabling the call to `mexLock` causes the array never to be cleared (until MATLAB is closed, that is).

Take care with locked MEX-files: since MATLAB cannot clear them, it is not possible to recompile them in the same MATLAB session (you have to quit MATLAB to free the MEX-file). Call `mexUnlock` to unlock the file (it's a good idea to have a special syntax for your function that causes it to unlock itself, so that you can recompile it during development).

```
#include "mex.h"

mxArray* value=NULL;

void AtExit(void)
{
   if (value) {
      mexPrintf("Clearing data!\n");
      mxDestroyArray(value);
   }
}

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
   if (!value) {
      mexCallMATLAB(1,&value,0,NULL,"rand");
      mexMakeArrayPersistent(value);
      mexAtExit(AtExit);
      /* optionally: mexLock(); */
   }
   plhs[0] = value;
}
```

Note how MATLAB correctly handles indirect copies to the array value (it is both put into the output variable and stored in the MEX-file itself).

It is also possible to make memory allocated with `mxMalloc`, `mxCalloc`, etc. to be persistent.

5.2 Compiling MEX-files that Call Libraries

If your MEX-file depends on other C sources or libraries, add their names on the call to `mex`. Libraries can be added under UNIX with the familiar `-l<file>` syntax. The first C source dictates the name of the resulting MEX-file. Use the `-output <name>` option to specify a function name.

5.3 Debugging Your MEX-file

Compile your MEX-file with the `-g` option (debugging symbols enabled).

On UNIX machines, load MATLAB with the `-D` switch:

```
matlab -Ddbx
```

(or whatever name of debugger you use). This causes MATLAB to be loaded within the

debugger. Now issue a "run" command to let MATLAB start. In MATLAB, now type

```
dbmex on
```

When you call a function in a MEX-file, the debugger will appear. You can also set breakpoints and what not.

On Windows machines, start your debugger, and run MATLAB from it (MATLAB has no debugging symbols, you will get a warning for this). Now set breakpoints (either in the code, or "on image load" or something). You also need to set the debugger to stop on handled exceptions, since MATLAB handles all exceptions your MEX-file generates (segmentation violations and the like). If you now run your function from MATLAB, the debugger should come into action.

The book "Application Program Interface Guide" from the MATLAB manual set gives more detailed information on this topic.

Chapter 6

Using DIP*lib* in Your MEX-file

This chapter assumes you are somewhat familiar with DIP*lib*. We recommend that you first read the "DIP*lib Programmers Guide*". This chapter deals with two seprate problems: adapting your MEX-file to call a function in DIP*lib* (which involves everything from converting MATLAB `mxArray` objects to the appropriate DIP*lib* structures, to linking DIP*lib* with your MEX-file), and writing your own DIP*lib*-style code within a MEX-file.

6.1 An Interface Between MATLAB and DIP*lib*

When linking a MEX-file to DIP*lib*, it is necessary to link to `libdml` as well. It contains all the functionality needed to link MATLAB and DIP*lib*.

This section discusses some of the functions defined in this interface.

6.1.1 DIP*lib*-Style Resource Management and Error Handling

When creating a `mexFunction` that uses DIP*lib* functionality, it is recommended to use a set of macros that reproduces DIP*lib*'s own resource management and error handling. This is, of course, not necessary, but makes things a bit easier. The code would look like this:

```
#include "dml_dipmex.h"

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

   int arg1, arg2;
   ...
   DML_ERROR_INIT;

   ...
   DMLXJ ( dip_function (arg1, arg2, ...) );
   ...

dml_error:
   DML_ERROR_EXIT;
}
```

You will immediately see the similarity to DIP*lib* code.

`DML_ERROR_INIT` initializes the error management structures and allocates a resources structure named `rg`. `DMLXJ` is similar to `DIPXJ`, and jumps to the `dml_error` label if the DIP*lib*

function returns an error state. `DML_ERROR_EXIT` will deallocate any memory registered to `rg` and print the function call stack to the MATLAB window in case an error occurred. There is also a `DMLSJ` defined, to substiture `DIPSJ`. Do not use any of the `DIPxx` macros, though, since they assume different internal variable names and labels.

`dml_dipmex.h` includes `diplib.h` and `mex.h`, so you do not need to include these explicitly.

6.1.2 Converting MATLAB `mxArray` Objects to DIP*lib* Objects

There are various data types that are interesting to convert to and fro between DIP*lib* and MATLAB. Foremost are images. There are four macros that deal with this, assuming you are using the abovementioned `DML_ERROR_INIT` and related macros. `DML_MEX2DIP(ma,im)` will convert the MATLAB array or `dip_image` object `ma` into a DIP*lib* `dip_Image` structure `im`. To convert it back to a MATLAB array use `DML_DIP2MLA(im,ma)`, and to convert it back to a `dip_image` object use `DML_DIP2MEX(im,ma)`. Finally, to create an (unspecified) DIP*lib* image use `DML_GENDIP_IMAGE(im)` (which calls `dip_ImageNew()` in such a way that the image, when forged, will be allocated by MATLAB). All of these macros avoid copying of the image data by making DIP*lib* use memory allocated by MATLAB. This is sadly not possible for complex image data. MATLAB and DIP*lib* differ too much in the way this data is stored to be able to share it directly. Therefore, complex images are copied each time one of these macros is called.

For other common DIP*lib* tipes, such as arrays and enumeration types, there are also macros. However, these were written to simplify the task of creating a glue layer between DIP*lib* and MATLAB. Therefore, they are not as flexible as the macros mentioned above. The ones converting MATLAB data to DIP*lib* require a input parameter number (the `n` in `prhs[n]`) and a variable name. See the file `dml_macros.h` for a complete listing. You can also copy code from this file instead of using the macros, which will prove more flexible.

As an example, this is the code for the MEX-file `dip_gauss`:

```
#include "dml_dipmex.h"
#include "dip_linear.h"
#include "dip_globals.h"

void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[]) {
   dip_Image in;
   dip_Image out;
   dip_BooleanArray process;
   dip_FloatArray sigmas;
   dip_IntegerArray parOrder;
   DML_ERROR_INIT;
   DML_CHK_NARGSIN (4);
   DML_CHK_NARGSOUT (1);
   DML_2DIP_IMAGE (0, in);
   DML_GENDIP_IMAGE (out);
   DML_2DIP_BOOLEANARRAY (1, process);
   DML_2DIP_FLOATARRAY (2, sigmas);
   DML_2DIP_INTEGERARRAY (3, parOrder);
   DMLXJ ( dip_Gauss (in, out, NULL, process, sigmas, parOrder, -1) );
   DML_2MEX_IMAGE (0, out);
dml_error:
   DML_ERROR_EXIT;
}
```

6.2 Writing a DIP*lib* Function Within a MEX-file

The C source file for your MEX-file can, of course, contain other functions definitions as well. You can write a DIP*lib* function within the same source file without any problem. You can also write this function in its own C source file, and link the two together when creating the MEX-file.

The DIP*lib* function should not use the `DML...` macros described above, but the `DIP...` macros described in the "DIP*lib Programmers Guide*". It should also stay away from any MATLAB functions.

6.3 Combining DIP*lib* Calls and MATLAB Functionality in the Same Function

If you have a function that does much more processing than only calling a DIP*lib* function, you might not want to use the `dml_error` label at the end. In this case, you will not be able

to use the DMLXJ macro. Do this instead:

```
void mexFunction (int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[]) {
   int arg1, arg2;
   DML_ERROR_DECLARE

   ... /* some non-DIPlib code */

   DML_ERROR_START
   *errorNext = dip_function ();
   *errorNext = dip_function ();
   *errorNext = dip_function ();
   DML_ERROR_FINISH

   ... /* some more non-DIPlib code */

   DML_ERROR_START
   *errorNext = dip_function ();
   DML_ERROR_FINISH

   ... /* even more non-DIPlib code */
}
```

DML_ERROR_FINISH causes your function to exit with an error message, and clears all resources registered in rg.

Also use dml_mex2dip(ma,&im,rg), dml_dip2mla(im,&ma), dml_dip2mex(im,&ma) and dml_newdip(&im,rg) instead of DML_MEX2DIP(ma,im), DML_DIP2MLA(im,ma), DML_DIP2MEX(im,ma) and DML_GENDIP_IMAGE(im).

Otherwise it is not a problem combining DIP*lib* code and MATLAB code in one function.

6.4 Linking Your MEX-file to DIP*lib*

The mex command, as discussed in Chapter 3, will compile your MEX-file. You will need to give it some extra parameters so that it will link to DIP*lib* and the libdml interface library, and so that it will be able to find the DIP*lib* include files:

```
mex mymexfile.c -I/dip/Linux/include -L/dip/Linux/lib ...
                -ldml -ldipio -ldip
```

(make sure you use the right paths). You might also need to add the math library, with -lm. Under Windows, this is a bit different:

```
mex mymexfile.c -Ic:\dip\include c:\dip\bin\libdml.lib ...
                 c:\dip\bin\libdip.lib c:\dip\bin\libdipio.lib
```

(again, fill in the right paths).

If you have more than one C source file, just put them all on the mex command line:

```
mex mymexfile.c somefunction.c morestuff.c    etc.
```

The MEX-file will get the name of the first C-file. If you want to change this name, use the -output option:

```
mex file1.c file2.c file3.c -output mymexfile    etc.
```