C/C++ to Matlab conversion

For IPOL algorithms

Paul-Darius Sarmadi

##### Abstract:

IPOL is a research journal of image processing and image analysis. Each article contains a text on an algorithm and its source code, with an online demonstration facility and an archive of experiments. IPOL’s algorithms are developped in C or C++. However many researchers prefer software which offer specific tools for image processing as an environment. The most used is probably Matlab. Thus, in order to increase its visibility, IPOL needs a tool to convert algorithms from C/C++ to Matlab.

Table of Contents

Abstract: 1

1. Introduction 3

1.1. The goal of this document 3

1.2. What is MEX ? 3

2. Files Organization 7

2.1. Introduction 7

2.2. Organizing the files: Practical Example 7

2.3. Organizing the files: Norms 8

2.3.1. Files created from create\_doc.sh 8

2.3.2. Files that should be added or modified 9

3. Checking Dependencies 10

3.1. Checking Dependencies: Practical Example 10

3.2. Checking dependencies: General rules 11

3.2.1. Libraries inclusion 11

3.2.2. compilMex.m generation 11

4. Input and Output with Mexipol 12

4.1. Input and Output: Practical Example 12

4.1.1. Input and Output errors 12

4.1.2. Matrix input 13

4.1.3. String Input 14

4.1.4. Structure Input 14

4.1.5. Matrix Output 14

4.1.6. String Output 14

4.1.7. Structure Output 14

5. Edition of the x.m file 15

5.1. Creation of the document 15

5.2. Adding optional parameter(s) 15

5.3. Creation of help menu 15

6. Other advices 16

6.1. Advices for the user of the library 16

6.2. Advices for the writer of the source code 16

1. Introduction

# The goal of this document

What IPOL needs is a powerful tool for C/C++ to Matlab conversions, and specific norms in order to be sure when a researcher wants to use an algorithm from IPOL in Matlab, he uses the same conventions for each algorithm: he knows in what directory he has to go and what to do there.

This document is a guide on how to make any algorithm developped in C or C++ from IPOL usable within Matlab. The writer of this document decided to take a random algorithm from IPOL and to explain step by step how he created a Matlab version of it. During each step, he describes what he does, and defines general norms that have to be followed by anyone who wants to convert his algorithm from C/C++ to Matlab.

# What is MEX ?

Before we start using the tools developed by the author of this document, we must know what MEX is. MEX is a tool developed by MathWorks to build Matlab functions from C/C++ functions. Taking a basic example is the best way to understand what it is and how it works. Let’s pretend we created three files: addition.h, addition.c and main.c just like this:

addition.h:

#include "stdio.h"

#include "stdlib.h"

double addition(double a, double b);

addition.c:

#include "addition.h"

double addition(double a, double b)

{

return a+b;

}

main.c:

#include "stdio.h"

#include "stdlib.h"

#include "addition.h"

int main()

{

double a,b ;

scanf("%e", &a);

scanf("%e", &b);

printf("%e",addition(a,b));

}

To compile it, we need to use a terminal and type something like “gcc main.c addition.c”. It automatically creates a a.out file that works perfectly when launched.

The algorithm is extremely simple: One main function needs to call a couple of .h/.c files to know how a function “addition” works. This structure is both basic and used in every single algorithm from IPOL. Some of them will need two or three couples of .h/.c functions. It is probably the biggest difference.

Now the issue is to understand how to make it usable within Matlab.

Let’s create a file called MEX\_addition.c:

#include "mex.h"

#include "addition.h"

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

{

if (nlhs!=1)

{

mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs","One output required.");

}

if(nrhs!=2)

{

mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs","Two input required.");

}

int\* a=(int\*)mxGetPr(prhs[0]);

int\* b=(int\*)mxGetPr(prhs[1]);

plhs[0]=mxCreateDoubleMatrix(1,1,mxREAL);

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

pointeur[0]=addition(a[0],b[0]);

}

Now, in Matlab, we type:

**mex MEX\_addition.c addition.c**

to compile it.

Then, for example:

* a=5 ;
* b=4 ;
* c=main(a,b)
  + c = 9

What has been done ?

The first point is that a file called “mex.h” is used. This file gives the possibility to use mexfunctions in the .c file.

Then, we can see the line:

-**nlhs** (Number of arguments on Left-Hand Side) is the number of output the Matlab user chose.

-**plhs** (Pointer to arguments on Left-Hand Side) is an array where each element contains one output, which must be made of mxArray data.

-**nrhs** (Number of arguments on Right-Hand Side) is the number of input the Matlab user chose.

-**prhs** (Pointer to arguments on Right-Hand Side) is an array where each element contains one output, which must be made of mxArray data.

This line replaces the call to main. Thus, **nlhs** and **nrhs** are similar to **argc** and **plhs** and **prhs** are similar to **argv**.

After that, two tests are made to check if the number of input and output are valid. If they are not, an error message appears within Matlab during the use of the function and the process is stopped. For more information about messages and error messages within Matlab, the best place to go is Matlab’s mex documentation.

Then, we define two variables:

**double\*a=mxGetPr(prhs[0]);**

**double\*b=mxGetPr(prhs[1]);**

The first point is **Matlab’s input and output are almost always double arrays. Matlab makes no difference between arrays and numbers. A value is an array made of one element. mxGetPr(prhs[X])** is a pointer to double to the array contained in the field n°X of **prhs**. This means it refers to the value contained in the input n°X. For example, to get the first value contained in **prhs[X]**, we have to type **mxGetPr(prhs[X])[0]***,* and to get the second value contained in **prhs[X]**, we type **mxGetPr(prhs[X])[1]***.* In our case we just need the first value, as we just entered one number: so we will use **mxGetPr(prhs[0])[0] (=a[0])** and **mxGetPr(prhs[1])[0] (=b[0])**.

The next line is:

**plhs[0]=mxCreateDoubleMatrix(1,1,mxREAL);**

This means we decide that the first output will be a Matrix made of double real values, and of size 1x1 (we just need one output value, not a bigger array or matrix). Many other possibilities exist: mxCreateString, mxCreateNumericArray, etc. Again, it is easy to check on MathWork’s documentation to get more information. To modify the content of the output the only way is to create a pointer on it (the penultimate line) and to modify the pointer’s content (the last line). There is no need to allocate space for the output, it is automatically made by Matlab during the call to mxCreateXXX. The user just have to add the content he wants in the pointer.

NOTE: The content is added like in an array ( pointer[0] = xx ; pointer[1] = xxx ; (...) ). Thus, if the variable plhs[X] is declared as a matrix, notice that Matlab considers you will add your content with a column-major process. Which means: top to bottom, then left to right. See figure 1.

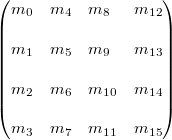


figure 1: A column-major process

In conclusion, Mex is a library that is used to make C/C++ algorithms available within Matlab. The biggest point is to understand that the algorithm itself will not be modified –in the example, we did not have to modify the function “addition”. In fact, we did not even need to know what the work of the function “addition” was. The only thing we did was to list the needed input and output, and to adapt them in order to make Matlab able to understand what we were refering to thanks to mex functions. The purpose of the mex-ipol library is to make it easier: thanks to high-level functions, the user does not need to think about pointers anymore during the conversion, and will avoid many segmentation faults.

1. Files Organization

# Introduction

This section’s purpose is to explain the general organization of the files that will have to be followed by anybody who wants to convert his C/C++ algorithm to Matlab. Making a consistent structure which always remain the same regardless of the algorithm is very important for the user: each time one wants to try a new algorithm with Matlab one knows where to find it, how to compile it, where to learn how tu use it, etc. In a first time a practical example of use is shown. Then, general rules are given. Those rules must be strictly followed.

# Organizing the files: Practical Example

The Non-Local Means Denoising article will be taken as an example for this document. First of all we need to download it. The article will be found here: [www.ipol.im/pub/art/2011/bcm\_nlm/](http://www.ipol.im/pub/art/2011/bcm_nlm/)

In the same page, the source code is available: inside the folder the author called “nlmeansC”, we can see many files: makefile(s), .cpp files, .h files, README(s). In an annex of this document, there is a script called “create\_doc.sh”. We copy this file in the directory called “nlmeansC”. We execute it by typing “./create\_doc.sh nlmeans». Its work is to create the directories and files needed (figure 2) to respect the standards I suggest concerning C/C++ to Matlab conversion. Note that the argument “nlmeans” is necessary to generate the README.md file. Then, the file “create\_doc.sh” is automatically removed.

# Organizing the files: Norms

Let x be the name of the given algorithm.

## Files created from create\_doc.sh

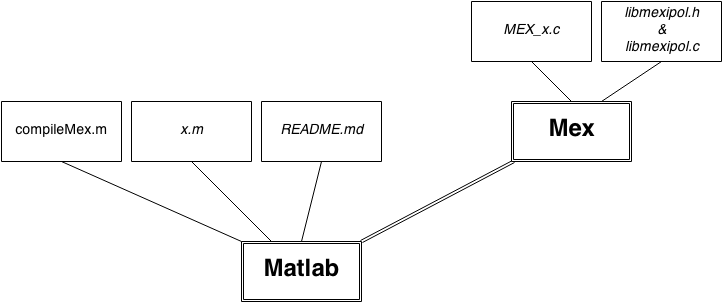


figure 2: Files and Directories made when create\_doc.sh is executed

When the user of libmexipol copies “create\_doc.sh” in the main directory of his project and then types «./create\_doc.sh x” in a shell in this directory, some files are created.

Those files are:

-A directory called “Matlab” inside the main directory.

Inside the “Matlab” directory:

-A file called “README.md”: This file tells the user what he has to do to make the algorithm work within Matlab: First, compile it with the file “compileMex.m” by typing “compileMex” in Matlab and then use it with the file “x.m” by typing “x” within Matlab.

-An empty file called “compileMex.m”.

-An empty file called “x.m”.

-A directory called “MEX”

Inside the “MEX” directory:

-A file called libmexipol.h and a file called libmexipol.c: This is the library containing the mex high-level functions that will be used for the conversion

-A file called MEX\_x.c.

## Files that should be added or modified

In the “MEX” directory:

- The MEX\_x.c file is the main file. When compiled, a MEX\_x.mexmaci64 file is created. This last file can be used as the x algorithm directly within Matlab by typing something similar to output=MEX\_x(input). Thus, the biggest work is to build this file (see Section 3, 5, 6).

In the “Matlab” directory:

-The compileMex.m file contains the compilation instructions, and is analogous to the Makefile of the project. (see Section 3.2.2)

-The x.m file properly calls the MEX\_x function and is the place for a help section in Matlab (see Section 5.3)

Notice that the final user of the soft would appreciate a test\_x.m file in the Matlab directory which gives an example of use of the algorithm within Matlab (see Section 6.2)

1. Checking Dependencies

# Introduction

In this third part, general rules that do not really change from a project to an other are shown. As in the previous paragraph, we will utilize a practical example in order to find general rules.

# Checking Dependencies: Practical Example

First, let’s remember that our work is not to change the algorithm itself but to redirect its input and output. Thus, it is not a “.png” or “.jpeg” file that will be sent to the algorithm but an image as a matrix from Matlab. Then basically there is two kinds of dependencies: the png/jpeg libraries that are not used anymore in a mex file, and all the other libraries that are absolutely needed to make the algorithm work: mainly .c file where the main algorithm is defined and other .c files containing computing subfunctions that the algorithm calls.

In our case, the directory nlmeansC contains three kind of files:

Files that are useless png/jpeg libraries and other files: io\_png.c, io\_png.h, README.txt and Makefile.

Files that are necessary libraries: libauxiliar.cpp, libauxiliar.h, libdenoising.cpp, libdenoising.h, mt19937ar.c, mt19937ar.h.

Files that call “main” functions: img\_diff\_ipol.cpp, img\_mse\_ipol.cpp and nlmeans\_ipol.cpp. Only the last one interests us -the others basically compute the difference between noisy and denoised images: they are not important here.

Finally, we know what files we have to link to MEX\_nlmeans.c: libauxiliar.cpp, libdenoising.cpp, and mt19935.c. As we can see in nlmeans\_ipol.cpp –the file containing the main function- the main algorithm only needs to include libdenoising.h. This means libdenoising probably includes libauxiliar.h and mt19935.h itself.

Now, we know what to include in MEX\_nlmeans.c:

**#include “../../libdenoising.h”**

And of course libmexipol.h as its functions will be very useful in part 4:

**#include “libmexipol.h”**

Note that this library contains the mex.h inclusion.

And we know how to compile the mex file:

**mex MEX/MEX\_nlmeans.cpp MEX/libmexipol.c ../libdenoising.cpp ../libauxiliar.cpp ../mt19935.cpp**

This line has to be added to compileMex.m. It is basically the only needed line for this file.

NOTE: We had to change the file format of MEX\_nlmeans.c to .cpp in order to make mex compile the project as a C++ project. Indeed, some libraries used C++ functions and then, a C++ compiler was required.

# Checking dependencies: General rules

## Libraries inclusion

There are basically two kinds of libraries:

-Those that are not necessary within Matlab (png, jpeg, tiff libraries).

-Those that are needed for computations.

Among the necessary ones, the user only has to include the libraries that are needed in the file that contains the main functions.

## compilMex.m generation

The general syntax that will be used is:

**mex <main file> <main dependencie> <second dependencie or first dependencie of the main dependencie> <third dependencie or second dependencie of the main dependencie or first dependencie of the second dependencie> etc.**

This line must be copied in compileMex.m, a file situated in the directory called “Matlab” (figure 2).

1. Input and Output with Mexipol

# Input and Output: Practical Example

## Input and Output errors

As we are discovering the nlmeans algorithm, we do not know what its input and output are. Thus we have to check the main functions contained in nlmeans\_ipol.cpp.

We can see this line:

**printf("usage: nlmeans\_ipol image sigma noisy denoised \n");**

Then obviously the first argument is an image. Concerning the sigma value, an attentive reading of the main function made me understand that its value must be between 0 and 100. The noisy and denoised parameters are the adress of the two images that are output.

The structure of the function is this one:

Thus, the number of input is two. The number of output is also two.

To test if it is the case, I use my first function from libmexipol:

**number\_io(nlhs,2,prhs,2) ;**

The first input must be a double matrix.

I use:

**is\_type("double",prhs,0) ;**

The second input must be a value between 0 and 100.

**is\_type("double",prhs,1) ;**

Both output are double matrix.

The beginning of the code will be:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "libmexipol.h"

#include "../../libdenoising.h"

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

{

number\_io(nlhs,2,nrhs,2) ;

is\_type("double",prhs,0) ;

is\_type("double",prhs,1) ;

}

With those three lines, all the problems linked to input/output number or input type are resolved.

## Matrix input

Now that I have checked the typical errors a user can make, I need to create the image matrix that is required from the first matlab’s input instead of a registered file. In the main file called “nlmeans\_ipol.cpp”, the creator of the algorithm wrote this:

size\_t nx,ny,nc;

float \*d\_v = NULL;

d\_v = io\_png\_read\_f32(argv[1], &nx, &ny, &nc);

-**d\_v** is a float\* pointer that will contain all the needed data.

-**nx** is the image’s width.

-**ny** is the image’s height.

-**nc** is the number of channel ;

It seems that the io\_png\_read\_f32 function defines those three last values.

I modify those lines in order to provide compatibility with Matlab using an important function from the mexipol library called “image\_matlab\_to\_c\_malloc”:

int nx, ny, nc;

float \*d\_v = [image\_matlab\_to\_c](libmexipol_8h.html#ad81ec3c17f5ef5f617d151a44c4a2cce)\_malloc(prhs[0], &nx, &ny, &nc);

At the end of the algorithm, **d\_v** must be freed:

free(d\_v);

NOTE: I could also have used a function called image\_matlab\_to\_c.

I also need to transform the sigma argument from this:

float fSigma = atof(argv[2]);

to this:

float\* pointer = mxGetData(prhs[1]);

float fSigma = pointer[0];

image\_matlab\_to\_c may have worked too.

## String Input

If the user needs a string input, he should use the function get\_string\_malloc.

## Structure Input

If the user needs a structure input, he should use the function get\_structure\_malloc

## Matrix Output

At the end of the algorithm I can see:

// save noisy and denoised images

if (io\_png\_write\_f32(argv[3], noisy, (size\_t) d\_w, (size\_t) d\_h, (size\_t) d\_c) != 0)

{

printf("... failed to save png image %s", argv[3]);

}

if (io\_png\_write\_f32(argv[4], denoised, (size\_t) d\_w, (size\_t) d\_h, (size\_t) d\_c) != 0)

{

printf("... failed to save png image %s", argv[4]);

}

noisy and denoised have been defined here:

int d\_whc = d\_c \* d\_w \* d\_h;

(...)

float \*noisy = new float[d\_whc];

float \*denoised = new float[d\_whc];

With

int d\_w = (int) nx;

int d\_h = (int) ny;

int d\_c = (int) nc;

The best solution is to use **image\_c\_to\_matlab**, which converts a float\* array to an output.

**image\_c\_to\_matlab(noisy, d\_w, d\_h, d\_c, plhs, 0) ;**

**image\_c\_to\_matlab(denoised, d\_w, d\_h, d\_c, plhs, 1) ;**

NOTE: I could also have used a function called “image\_c\_to\_matlab”. For more information, see .........

## String Output

If the user needs a string output, he should use the function set\_string

## Structure Output

If the user needs a string output, he should use the function set\_structure.

1. Edition of the x.m file

# Creation of the document

The only absolutely needed line that has to be added is:

**function [output1, ..., outputN]=x(input1, ..., inputM)**

**[output1, ..., outputN]=MEX\_x(input1, ..., inputM)**

# Adding optional parameter(s)

If the user wants a parameter inputK to be optional, a function called “exist” will be very useful.

For example:

function out=x(input1, input2, optional)

if(~exist('optional', 'var'))

out=x(input1, input2)

else

out=x(input1, input2, optional)

end

# Creation of help menu

Typing “help x” will show the parts of the x.m file that contain comments –starting with %. The global syntax that will be used will be this one:

**% X short description of x**

**% (...)**

**% end of short description**

**% INPUT:**

**% =======**

**% (...)**

**% Information concerning the compulsory input**

**% (...)**

**% OPTIONS:**

**% =========**

**% (...)**

**% Information concerning the optional input**

**% (...)**

**% OUTPUT:**

**% =========**

**% (...)**

**% Information concerning the output**

**% (...)**

Examples are available on GitHub.

1. Other advices

# Advices for the user of the library

-Adding a matlab file with a .m extension providing an example of use of the function is a very important work to make things clearer for the future user of the software.

-A copyright with the name of the author of the file should not be forgotten.

# Advices for the writer of the source code

-When someone wants to convert an algorithm from C/C++ to Matlab, he basically needs to know where the algorithm is defined, what its dependencies are, and above all what the input, the options, and the output name and type are supposed to be.

However, to precisely get these information, the only solution has been shown before: groping in the source files and mainly in the file that contains the main function. Things could be done easierly –and why not automatically ?- if for each article a file with a name similar to “inoutx” -standing for “Input and Output for the algorithm X” was available within the sources. It should not care about the type of data a shell user faces -”.png” or “.jpeg” files- but of the types the algorithm directly needs (“float\*”, “double”, “char\*” ).

This file has to look like the model given as an annex to this document.

-An other limitation for automation is that in many “main” functions, a part of the algorithm is outside the main algorithm’s call. Let’s schematize:

**main()**

**{**

**float\* image = get\_image\_from\_file(file) ; //1st Part**

**modification\_on\_image(image) ; //2nd Part:**

**main\_algorithm(image) ; //3rd Part**

**}**

The second part -which is generally not a call to a specific function but rather successive instructions containing several “for” loops- should be inside the main\_algorithm function, not outside. Because then, knowing how to use the main\_algorithm is not enough, and the user must go back to the file containing the “main” function. And this should be avoided in order to promote automation: as long as the user will need to check the main source file, and not only “inoutx”, no simple conversion algorithm will be writeable.