# Matlab demos

Juan Cardelino, José Lezama

June 26, 2014

**Abstract**

In the context of increasing the number of authors publishing on IPOL, it was proposed to reduce the amount of work required by the authors to submit an article for IPOL. Two of the main sources of extra work identified are the level of exigence of the software guidelines for the current language supported (C/C++), and the possibility of including new languages and the particular case of Matlab. In this report we propose an analysis of the problem and a roadmap to implement Matlab support on IPOL. Only the main ideas is discussed here and the technical discussion is addressed in the extended report.

Matlab is a widely used tool in our research community and many researchers students only know how to work in Matlab. Transcribing this code to C/C++ is perceived as a huge barrier to publish in IPOL. This is specially relevant in the Audio Processing Community, where almost everyone codes in Matlab.

From the preliminary test performed, and further discussion with the team, we conclude that it is possible to have Matlab demos using the same server, with a very small increase in the workload of IPOL staff. This extra workload will be handled by and external team. More testing is needed but the main technical problems were tested. In addition, the team should design new Software Guidelines and propose a slightly modified submission process. The main goal is to have some matlab publications in a one year period.

# Chapter 1

# Matlab Publications in IPOL

## 1.1 Introduction

In the context of increasing the number of authors publishing on IPOL, it was proposed to reduce the amount of work required by the authors to submit an article for IPOL. Two of the main sources of extra work identified are the level of exigence of the software guidelines for the current language supported (C/C++), and the possibility of including new languages and the particular case of Matlab. In this report we propose an analysis of the problem and a roadmap to implement Matlab support on IPOL. Only the main ideas is discussed here and the technical discussion is addressed in the referenced document [1].

## 1.2 Analysis

Matlab is a widely used tool in our research community and many researchers students only know how to work in Matlab. Transcribing this code to C/C++ is perceived as a huge barrier to publish in IPOL. This is specially relevant in the Audio Processing Community, where almost everyone codes in Matlab.

## 1.3 Problems to Solve

1. Licensing
2. Software Guidelines
3. Review Process
4. Demo Server Implementation

## 1.4 Team

1. Juan: lead the initiative

2. Some help from JRASP: Haldo and Benoit

## 1.5 Current status

1. Technical:
   (a) During the meeting Juan was able to run a matlab code as a standalone binary.
   (b) The binary was included into a demo that did not require any modification of the IPOL demo server
   (c) The demo is included in the development server purple
2. Guidelines:
   (a) Some rough ideas have been proposed, but more discussion should follow
3. Submission process
   (a) We outlined the main steps but again it should be discussed and put in sync with the new propositions for submission process.

## 1.6 Timeline

1. Propose a structure for the demo system. DONE
2. Put the matlab demo system under test for a short period. 1 month
3. As part of this process some publications or demos are going to be tested by the team (written by the same team). 2 months.
4. Given that test are successful, we will start a proof of concept inviting external authors. We propose to invite three papers this year (6 months).

## 1.7 Scope

The extra work added is the new matlab demos, the rest of the system remains the same. For this reason, this extra work will be handled by the matlab team. The scientific review will be carried by IPOL members as usual. The code review should be also handled by regular IPOL editors/reviewers.

## 1.8 Papers to invite

1. Juan: Fast Marching
2. G. Peyre: TBD
3. S. Mallat: TBD
4. Audio: TBD

## 1.9 Conclusions

From the preliminary test performed, and further discussion with the team, we conclude that it is possible to have Matlab demos using the same server, with a very small increase in the workload of IPOL staff. This extra workload will be handled by and external team. More testing is needed but the main technical problems were tested. In addition, the team should design new Software Guidelines and propose a slightly modified submission process. The main goal is to have some matlab publications in a one year period.

## 1.10 References

[1] "Technical Proposal for an IPOL Matlab Demo Server" B. Petitpas, J. Cardelino.

# Chapter 2

# Technical Proposal for an IPOL Matlab Demo Server

## 2.1 Solution from Juan

:

There's a matlab code submitted by author, with possibility to add some exotic/personnal toolbox

### 2.1.1 Compilation

- a wrapper CPP source file that serves as entry point (main), handles input data and parameters and passes them to the matlab code of the author.
- the server compile the binaries by using a matlab script that compile with "mcc" the matlab code and with "mbuild" construct the execution binaries which NEED a matlab key.

### 2.1.2 Execution

The generated binary depends on Matlab runtime libraries.

So it needs :

- the sources
- the cpp wrapper (can be generated)
- the script (can be generate)
- a Matlab Compiler Runtime Environment (MCR) that need nothing special to run
- a person that can compile WITH a matlab token for compilation with the same version AS the MCR. This happens only one time, offline and can wait for license token.

Note: at the end for running a submission, it needs only one key to compile and none to run.

## 2.2 Server modifications

The server needs to have MCR installed no further modifications are required.

## 2.3 Distribution of workload

### 2.3.1 Author job

- Put his sources into a way that it is easy to use
  - a scr folder
  - a toolbox folder (with all the .m files that is needed for execution but is not the algorithm)
- They will be probably required to comply with a certain matlab version.
- They will have to comply with a certain input/ouput interface in their matlab main function.

### 2.3.2 Editor job (matlab team)

- generate or write the cpp file AND the script - compile with the same version as the MCR the code for generating the binaries

### 2.3.3 Reviewer job

-The scientific review stays unchanged -The software review

## 2.4 Update policy

Any given matlab usually survives quite well for many years (3 or 4 server OS(linux) versions).

## 2.5 Problems to Solve

- how do we update the MCR ?
- if it not update :
  - no author in the future
  - One solution is to update and if a demo doesn't work anymore let it die, or put in a queue for update.
- if it updates : will the old demo work?
- Licensing do we need to RE-compile every programs everytime we update the MCR ?
  Is it possible to run concurrent matlab processes without a license token? Yes, using the MCR (Matlab Compiler Runtime) running the demos is free and legal. Tokens are needed only to compile (one token 1 time).

## 2.6 How it will works (submission)

Steps :

1. We receive a matlab code with article
2. An editor analyses the code according to the software guidelines (to be defined)
3. The editor writes (or automatically generates) the wrapper cpp file, compiles and generates the final binary
4. The editor or author writes a demo calling the generated binary
5. The rest of the process is the same as a regular IPOL publication

## 2.7 Software Guidelines

This has to be defined, but some hints are provided:

- Matlab users are used to plot data interleaved in the code. Authors should be required to use a flag to turn on/off the plots. It is advised not to remove them, because it might be useful for review.
- The input/output interface of the main matlab function has to comply with a certain structure.

## 2.8 Compilation How To

### 2.8.1 Introduction

MATLAB programs can be compiled into standalone executable applications using the MATLAB compiler. The MATLAB compiler can be invoked from within MATLAB or from the command line. For example, in the *purple* server the location of the MATLAB compiler is `/usr/local/MATLAB/bin/mcc`.

### 2.8.2 Compiling a simple script

Compiling a simple MATLAB script is very easy. Suppose we have created the following simple script, and saved it as `test_script.m`

```
1 function test_script()
2
3 a = 1;
4
5 disp('Hello World.');
```

First version of `test_script.m`

Compiling it from within MATLAB would require executing the following command:

```
mcc -m test_script
```

Compiling it from outside MATLAB (from command line) would require calling the `mcc` script. In *purple* this would be done as:

```
/usr/local/MATLAB/bin/mcc -m test_script
```

Once the MATLAB compiler has finished, **it will produce two important files**: One is the executable binary file, that in this case would be called `test_script`. The other one is a bash script, which in this case would be called `run_test_script.sh`. The bash script sets the enviornment variables needed to load the necessary libraries for running the script. It will be this bash script that will need to be called from the IPOL demo. The bash script takes as first argument the path to the MATLAB root directory (`/usr/local/MATLAB` in *purple*). The rest of the arguments are passed to the MATLAB script. Our simple script does not require any argument, so if we were in *purple*, it would be simply called by:

```
./run_test_script.sh /usr/local/MATLAB
```

### 2.8.3 Passing arguments to the script

So far our example script was very simple, but a useful script will probably require passing arguments to it. Fortunately this is also very simple, since arguments passed to the bash script are directly passed to the compiled MATLAB

script. The only important thing to have in mind is that **a compiled MATLAB script receives the arguments as character strings**. See the following snippet of code for a new version of our test script.

```
1 function y = test_script(x)
2
3 if isdeployed
4     x = str2num(x);
5 end
6
7 y = 2 * x;
8
9 disp(sprintf('Your result is %f',y));
```

Second version of `test_script.m`

Let us examine lines 3 to 5. The condition tested in line 3 uses the `isdeployed` MATLAB function, which yields a `true` value when the script is run a compiled code, and yields `false` when it is run as a normal `.m` script inside MATLAB. Since compiled MATLAB code receives the arguments as strings of characters, the sentence in line 4 converts the argument to a numeric variable.

After compilation, an invocation to this script in *purple*, passing the number 3.5 as argument would be done as:

```
./run_test_script.sh /usr/local/MATLAB 3.5
```

And the output should be:

```
Your result is 7.000000
```

Note that other useful types of arguments are matrices or filenames. Matrices need to be specified inside double quotes. The following example call would pass as arguments the filename "lena.png" and the matrix [3, 2, 1]:

```
./run_test_script.sh /usr/local/MATLAB lena.png "[3 2 1]"
```

### 2.8.4 Using multiple `.m` files

A normal MATLAB project will proably consist of multiple `.m` files, and a main file for executing the program. This needs to be communicated to the MATLAB compiler `mcc` using the `-a` option. For example, if our script `test_script.m` calls other `.m` scripts located in a local folder called `toolbox`, the compiler invocation would need to look like this:

```
/usr/local/MATLAB/bin/mcc -a ./toolbox -m test_script
```

This is telling the compiler to add all files in the `toolbox` folder and all subfolders. Note that the behaviour of compiled code is different than when running a normal `.m` script inside MATLAB. If we were running the script inside MATLAB, we would probably use the command `addpath ./toolbox` to

tell MATLAB to look for `.m` files inside the `toolbox` folder. This is not possible in compiled code. In fact using the `addpath` function inside a script that will be compiled will probably produce an error at runtime causing the termination of the program. The workaround for this is once again to use the `isdeployed` function to check if the script is run as compiled code or as a regular MATLAB script.

Suppose we create a folder called `toolbox` and inside it a file named `multiply.m` containing:

```
1 function c = multiply(a,b)
2
3 c = a * b;
```

`multiply.m`

The new version of our `test_script.m` would look like this:

```
1  function y = test_script(x)
2
3  if isdeployed
4      x = str2num(x);
5  else
6      addpath ./toolbox
7  end
8
9  y = multiply(2,x);
10
11 disp(sprintf('Your result is %f',y));
```

Third version of `test_script.m`

And the compilation could be done as any of the following:

```
/usr/local/MATLAB/bin/mcc -a ./toolbox -m test_script
```

```
/usr/local/MATLAB/bin/mcc -a ./toolbox/*.m -m test_script
```

```
/usr/local/MATLAB/bin/mcc -a ./toolbox/multiply.m -m test_script
```

### 2.8.5 Displaying figures

If your MATLAB code uses the `figure()` function to display plots, it is possible that the compiled program when executed, waits for the closing of the figures windows to terminate. Since in *purple* you do not have access to the graphical interface, we recommend turning off the displaying of figures in the compiled program, by adding the following option to the compilation command `mcc`:

```
-R -nodisplay
```

The `-R` option tells the compiler the MATLAB options that should be used at runtime.

## 2.9 Matlab Demo How To

### 2.9.1 Introduction

Let MATLABROOT be the path to the matlab instalation, thus MATLAB's compiler will be located in `MATLABROOT/bin`. In this document we treat matlab as another compiler thus it should be added to the user/system path.

### 2.9.2 Steps

1. Demo system configuration:
   Edit the demo.conf file and add the change the variable matlab.path to poin to you MATLABROOT directory.

   ```
   matlab.path='/usr/local/MATLAB'
   ```

   Note: In the demo server, this is already configured.

2. Set up Matlab Compiler
   Add Matlab's Compiler to system path

   ```
   export PATH=$PATH:/usr/local/MATLAB
   ```

   Note: In *purple* server, this is already added to system path.

3. Makefile:
   The makefile is very simple, just invoke mcc as explained in previous section

   ```
   PROG=gauss
   all:
       mcc -m ${PROG}.m -a lib -R -nodisplay
   ```

4. `build` method:
   The build method is the same as a regular demo. Note: the difference is that it has to copy the executable (`gauss`) and the script (`run_gauss.sh`) to the ./bin directory.

5. `run_algo` method:
   As explained in section 2.8 the script should be executed as follows

   ```
   ./run_test_script.sh /usr/local/MATLAB lena.png "[3 2 1]"
   ```

   thus, in the `run_algo` method you should invoke `run_proc` like this

   ```
   p = self.run_proc(['run_gauss.sh', self.matlab_path,
                      'input_0.png','output_1.png', str(arg1)],
   ```

   the `self.matlab_path` stores the path to matlab installation directory.

# Bibliography