Matlab Integration in Virgo Online applications

D. Sentenac

VIR-0140A-10
Feb15,2010

Abstract

ITF online servers, involved in operational control loops and data analysis, are generally tested before using Matlab simulations. The task consisting in transposing Matlab to C online codes can be tedious and time consuming. In this Note, we propose a method to help commissioning and data analysis people to port their Matlab algorithms directly to Virgo online frame distribution applications. From our performance tests, we conclude that it can be very advantageous to use this technique.

I. Table of Content

II. Reference documents

[1] FrameLib documentation: http://wwwlapp.in2p3.fr/virgo/FrameL/FrDoc.html
[2] The Virgo phase camera: VIR-0442A-09, VIR-0300A-09
[3] Fd documentation: VIR-0092A-08
[4] *Mapp* documentation: http://wwwcascina.virgo.infn.it/sDoc/software/Mapp
[5] *app* documentation: http://wwwcascina.virgo.infn.it/sDoc/software/app
[6] Matlab Help: http://www.mathworks.com/access/helpdesk/help/techdoc/
[7] FFWT documentation: http://www.fftw.org/

III. Acronyms and package meaning

ITF : Interferometer
Matlab: Technical computing software for engineers and scientists
Fd : Virgo Frame Distribution package
app : Matlab frame distribution template package
Mapp : Matlab wrapper library package
CMT : Virgo Component management tool

1. Introduction

Virgo commissioning and data analysis groups make extensively use of Matlab simulations in their research. The frame format containing the ITF data in Virgo-Ligo Collaboration is accessible from Matlab [1]. When the need to adapt such codes in online applications comes, one may want to directly encapsulate the Matlab algorithms in Virgo online applications to gain time and reliability. In this Note we show how to proceed in this direction. We provide a Virgo frame distribution template able to run any Matlab algorithms. This method has been first applied with success in the phase camera project [2]. The Matlab algorithms from the phase camera have been used ‘as it is’ to produce the online amplitudes and phase images. In the next section we describe in details how to use the Matlab frame distribution template. Then we make a few observations about the some performance tests and give a few coding tips.

2. Matlab Frame Distribution Template

2.1 Problem Statement

Virgo online data processing orientted applications are usually based on the Frame Distribution library Fd [3]. This library is intimately related to Virgo DAQ architecture, which is based on a peer-to-peer relation of data exchange servers (producer/consumer). In general, an online application has the following 1Hz cycle constraint:

1. Take a data frame in input (in shared memory or via Cm)
2. Process the data
3. Release a new data frame in output (in shared memory or via Cm)

Our concern is how to encapsulate a Matlab algorithm in step 2. The exercise turns into adjusting the Matlab code *input/output* to the Virgo application. The Matlab algorithm input has to be fed with a set of data from the input Virgo frames. In return a new Virgo frame must be output with a set of data coming from the Matlab code output.

2.2 Method & Implementation

The Virgo template [3] presented here is a C code extending the Fd package. One may eventually choose to write a C++ code template. Matlab provides the possibility to build a C library from a proprietary m-file. After describing what is a m-file, we will show how to build a Matlab wrapper library, and finally how to connect the Matlab input/output in the Virgo application.

2.1.1 Matlab m-file

A m-file contains the Matlab algorithm functions that accept input arguments and produce output data. Matlab has a single input/ouput type of argument which accounts for any type of data (scalar, vector, and array). It is the *mxArray* type. We give an example of m-file to show how are handled the input/output arguments in ANNEX. In example0, the function does simply a matching between the output and input *mxArray* arguments.

2.1.2 Matlab library wrapper

In order to access the Matlab code in the Virgo template, one has to link the application to the Matlab code C library. Matlab offers command line tools to build C library wrappers, called *mcc*. Therefore it is straightforward to integrate the building process in the commonly used Virgo Component Management Tool (CMT). The only requirement is to use the Matlab CMT wrapper package. A typical CMT requirements file for building a Matlab C wrapper library can be found in the *Mapp* template package [4] and looks like:

requirements

```
package Mapp

use Matlab v7r0p5

macro Mapp_linkopts "-L${MAPPROOT}/${Mapp_tag} -lapp "

macro+ Mapp_linkopts "" \
  OSF1  "-Wl,-rpath,$(RPATH) " \
  Linux "-Wl,-rpath,${MAPPROOT}/${Mapp_tag} " \
  RIO806X " "


path_prepend LD_LIBRARY_PATH "${MAPPROOT}/${UNAME}"

document mcc libapp example0.m example1.m example2.m example3.m
```

A simple ‘make’ at the command line will produce automatically the expected *libapp.so* library in the usual bin directory (${UNAME}), and the header *libapp.h* in the src directory.

2.1.3 Application Initialization & Termination

Integrating Matlab code in the application requires some initialization. Let’s have a look inside the *app* template [5]. The first thing to do is to include in the Virgo application *requirements* file the freshly built Matlab wrapper library *Mapp*, in addition to the *Fd* package:

use Mapp v0r0

then, including the header file generated in the *Mapp* package*:*

Include <libapp.h>

In order to set up the MATLAB Component Runtime (MCR), the initialization function must be invoked:

mclInitializeApplication(NULL,0);

The MCR can be seen as a virtual machine in which the Matlab library wrapper can be run.

Then one or more wrapper libraries can be initialized dynamically by invoking the library specific functions holding the library name for each of them, like:

libappInitialize();

Note that in order to use the library dynamically during execution, it is necessary to add the library path in the LD_LIBRARY_PATH environment variable. This can be done automatically by setting up the *app* package CMT environment, before execution:

cmt config
source setup.cs

The application code can forward info messages coded in the Matlab functions, in particular useful for debugging purposes. The method to do it is to use the *disp* Matlab function in the m-files (see ANNEX) and declare them in the Matlab initialization to redirect the output to a custom print_handler function.
libappInitializeWithHandlers((mclOutputHandlerFcn)print_handler,
(mclOutputHandlerFcn)print_handler);

Before terminating the application, one may invoke the library termination routine:

libappTerminate();

Then any Matlab memory used should be freed:

mxDestroyArray(inputs);

And finally the application closed:

mclTerminateApplication();

2.1.4 Input/Output Connections

We will see now how to translate the Matlab outputs and inputs arguments in the Virgo application. The Matlab algorithm inputs and outputs have to be defined as pointers:

```
mxArray     *inputs;
mxArray     *outputs;
```

To account for a scalar, the input (output) must be instantiated as follow:

```
inputs = mxCreateDoubleMatrix (1,1,mxREAL);
```

To account for a vector, the input must be instantiated as follow:

```
inputs = mxCreateDoubleMatrix (n,1,mxREAL);
```

Where n is the vector size.

To transform a *mxArray* into a double type, one has to apply the following:

```
double *d;
d = mxGetPr(outputs);
```

Inversely to copy a C data pointer to a *mxArray* data object, one has to do the following:

```
double *d;
memcpy (mxGetPr(inputs),d, sizeof(double) * n);
```

Where n is the size of the double pointer array d.

Finally, the general call to the Matlab algorithm function would read, in case of calling the function defined in example0.m:

```
(*mlfExample0 )(n,&output1,..,&outputn,input1,…,inputp);
```

where n stands for the number of outputs. Note that one can use as many as output & input arguments. The function name is defined automatically by the wrapper. It is always preceded with the *mlf* prefix.

3. Code performances & coding tips

We do not intend to provide here pure Matlab coding tips to improve performances and the read will refer to the Matlab Help section [6]. On the other hand we draw the attention of the reader on the consequences of integrating Matlab m-files in a frame

distribution application. We recall that the reader will find a full example of the implementation of a frame distribution application in the *Mapp* [3] and *app* [4] packages.

3.1 Performance Tests

In the app package is shown a simple performance test using different Matlab functions doing the same computation in different ways, and a comparison is made with the same computation done in the application itself. It consists simply of assigning the input vector to the output vector values. The performance results are shown in the following table:

Algorithm	Example0	Example1	Example2	C-code
Average computation time (seconds)	3E-03	1.7E-01	3.3E-04	2.4E-04

We make the following observations:

- The best way to assign the inputs to the outputs is using the concept of vectorization (see ANNEX) as in Example2.
- If it is not possible, it is important to allocate the output vector before entering the loop using the *zeros* function, as in Example0. The worst is clearly Example1.
- The C-code shows better performances respect to best Matlab Example2 by a factor ~ 1.4.

We can explain the differences between Example2 and C-code in terms of memory allocation time efficiency. We indeed checked independently that memory allocation is more efficient in the C-code than in Matlab. In addition, consider that the Matlab inputs memory allocation can be done prior calling the m-file, at the Virgo application initialization. But we cannot avoid that the outputs generated in the m-file function are allocated at each cycle. This is demonstrated by comparing Example0 to Example1. This memory drawback can be compensated by using as much as possible the vectorization techniques, which appears by far the most efficient way of calculation Matlab offers. Where vectorization is not possible, memory allocation will have to be minimized, i.e reusing the same outputs as much as possible. To further test the efficiency of Matlab algorithms, we push the comparison computing a fast Fourier transform (FFT). In the following table are shown the FFT performances obtained for a FFT done on a photodiode channel at 10 kHz:

Algorithm	Example3	C-code
Average computation time (seconds)	1.0E-03	6.9E-04

Example3 uses fully the vectorization technique to compute the FFT (see ANNEX). The C-code uses the FFTW library which is known as one of the fastest FFT

software [7]. As before, the C-code appears more efficient than the Matlab code by a factor ~1.4. This shows that the difference is merely due to the outputs memory allocation issue pointed in the first test. Considered alone, Matlab computational FFT gives similar performances results to FFTW. If we suppose that the Matlab functions are of the same quality as the FFT, it can be very advantageous to use directly Matlab code in Virgo application, especially if the algorithm use complex functions that would be tedious to transpose in C.

3.2 Frame frequency limitation

In an online application, a frame is produced at 1Hz. This fixes the upper limit for frame production of one second for output frame computation. However computing tasks may require more than one frame in input (like in time Fourier transform), and computing may grow with time, exceeding the maximum time allowed to output frames. To overcome this issue, multithreading the application becomes necessary. The algorithm computation thread can be separated from the main thread dealing only with frame I/O. In offline simulations, the user will get an estimation of the Matlab algorithm computing time, and decide if multithreading is necessary or not.

4 Conclusion

In this Note we have given the user a fast method to implement his Matlab algorithms in online applications. It is based on two template packages, *Mapp* gathering the m-files and wrapper libraries, and *app* being the Virgo front end application. We focused on frame distribution aspects which is most adapted to online control loops and data analysis needs. We have shown that when code optimization techniques are correctly applied in Matlab, we obtain very good performances respect to pure C-code. The overall code efficiency may depend essentially on how and how much memory is allocated for the Matlab outputs.

ANNEX

The Matlab package *Mapp* contains the following example m-files used in the performance test done in 3.1:

example0.m

```
function [Outputs] = example0(Inputs);

warning off;
disp('running example0');
[m,n]=size(Inputs);
Outputs = zeros(m,n);
for i=1:m,
  for j=1:n,
      Outputs(i,j)= Inputs(i,j);
  end;
end;
```

example1.m

```
function [Outputs] = example1(Inputs);

warning off;
disp('running example1');
[m,n]=size(Inputs);
for i=1:m,
  for j=1:n,
      Outputs(i,j)= Inputs(i,j);
  end;
end;
```

example2.m

```
function [Outputs] = example2(Inputs);

warning off;
disp('running example2');
Outputs=Inputs;
```

example3.m

```
function [Outputs] = example3(Inputs);

warning off;
disp('running example3');
Outputs = fft(Inputs);
```