Good practices guide for HiMod Tree

Yves Antonio Brandes Costa Barbosa

January 2017

1 Introduction

The aim of this guide is to provide practical instructions about how to contribute with the HiMod Tree code and keep the structure already present in the current version. In the first part of the manual, some problems found in the previous versions of the code and the measures taken to solve them are illustrated and explained. This way, we expect to give the motivation needed to follow the coding guidelines given in the next sections. Then, we present the rules to program each component of the code.

2 Problem with previous versions

The Hierarchical Model Reduction (HiMod) project is a large research project and scientific endeavor that gathers the contribution of a lot of students, researchers and professors. Like any large project, the development is continuous and new contributions are usually strongly based on the work of past contributors. Hence, the lack of protocols or guidelines to be followed by everyone involved may lead, in the best case, to delays and, in the worse case, a total stop in the progress achieved.

For this reason, the code implementing the HiMod simulation was restructured using the rules presented in this manual. For future contributors, the main objectives of this work is to guarantee that the original code given to you is understandable and to supply a direction to be followed. The new organization was defined to solve the four main problematic areas of the entire project: code architecture, code planning, naming variables and explaining methods.

2.1 Code architecture

The code architecture usually used by Matlab programmers is non hierarchical and function oriented. Briefly, it means that each new function that needs to be implemented will have its own `*.m` file and is placed in the pathway set in the computational environment. This approach is easy and normally does not bring any problem for project with less than thousand lines of code. However,

this is not the case for bigger projects, where the increasing amount of files with methods hierarchically organized in more than 3 levels begin to be very confusing.

The object oriented approach gives the most organized framework to develop projects that are large (usually have more than five thousand lines of code). Even though the object oriented feature of Matlab is not as optimized as in the case of C++ compiler, it is more than enough to the proposes of HiMod Tree, i.e. teach the HiMod technique and have a first glance in the behavior of the code when modifications are inserted. In particular, we can cluster the methods into structural classes and keep track of the functionality without actually reading the description of each method. Besides that the general work flow of the simulation comes to evidence and it becomes easier to identify modification spots and error sources.

2.2 Code planning

In a very simplistic way, code planning is the act of defining what methods are required to perform the simulation and where to correctly place them to improve the overall performance during the computations. This step comes before writing any actual code and is necessary to keep the organization of the methods. In the previous version, we noticed similar calculations being repeated in different functions and loops. This decreases the performance, and, even worse, compromises the understanding of what the algorithm was originally intended to do.

2.3 Variable and function naming

We can summarize the third and fourth big problems as naming problems. In any project, it is necessary to give comprehensible and systematic names for the variables and functions used. In this point, the length of the names are less important than the guarantee of comprehension by the other members of the team. So, do not spare letters and creativity in this one!

3 Coding rules

Before diving into the programming rules to be followed we would like to leave two concepts very the clear. First, good practices in programming is not the same as programming style! Coding style depends on the personal writing preferences of each programmer, whereas good practices are rules accepted by the community as the ideal way to implement a several parts of a program. They are not called rules because they are not necessary to perform the computations, but are essential to guarantee the quality and readability of the code.

Second, if we wanted resume the main message that this manual intend to pass ahead, you could resume it in five words: comment, comment, comment, comment and last, but not least, comment! Any member of the any serious programming community would agree that a code well written keeps the minimum ratio between lines of code and lines of comments as 1 to 3. The reason for this is that projects in the industry rarely begins from scratch. Instead, they often continuous a previous work or recycles a lot of code already used. If the code is not well explained, it is impossible to get the work done.

Now, with this two things in mind, now we came explore in more details the coding rules used in the HiMod Tree project.

3.1 Debug

One non optimality in the objected-oriented feature of Matlab is its inability to keep track of all the variables used in the methods of the classes. In the usual architecture that Matlab programmers use, all the variables are kept available in the workspace area. This saves a good amount of time during debug phase and to check the final results.

For this reason, when programming in this new architecture, one is tempted to use a lot of debugging and double check commands to keep track of the variable during the simulation. For this reason, we defined a systematic way of doing this kind of debug. It goes as follow:

1. **Rule 1:** To keep tracking of the vale of the variables during the simulations using the function `disp`. Do not remove the semicolon from the end of the expression to check the value of a given variable! Use the structure:

```
disp('*NAME OF TRACKED VARIABLE*');
disp(trackedVar);
```

2. **Rule 2:** Always specify if a command or line of code is there only for debug. It saves a lot of time during the cleaning phase and helps separate the main algorithm from the debugging ones. Use the structure:

```
% DEBUG
%-------------------------------------------%
disp('*NAME OF TRACKED VARIABLE*');
disp(trackedVar);
%-------------------------------------------%
```

3.2 Methods

The names used for the functions and variables used need to be clear and signification of role played in the simulation. Do not use incomprehensible or

ambiguous words to reduce the length of the names. In the long run, confusing names lead to more trouble than long ones. Also, er choose to use standard naming structures used in Java and C++ programming languages to name the structure of the code. Thus, we have:

1. **Variables**
    - Start with **lower** case;
    - In the case of composite name, cluster together without space, hyphen or underline;
    - In the case of composite name, use upper case to initiate names after the first one;

```
% EXAMPLE VARIABLES
%-------------------------------------------%
% Number of horizontal discretization
% nodes.
%-------------------------------------------%
numbHorNodes = 100;
```

2. **Methods** (*Functions*)
    - Start with **lower** case;
    - In the case of composite name, cluster together without space, hyphen or underline;
    - In the case of composite name, use upper case to initiate names after the first one;

```
% EXAMPLE METHOD (FUNCTION)
%-------------------------------------------%
% Build method used to assemble the
% stiffness matrix and the right hand
% side component.
%-------------------------------------------%
function [A,b] = buildSystemFEM(obj)
```

Apart from the name, the methods implemented must follow a general structure. The structure is composed by title, call, method explanation, inputs, outputs and computations. Each one of this components is initiated by a section marker %%. The section markers are fundamental in the organization of the code because they allow the code folding property of Matlab interface and give the arrangement to be respected when `publish` is used.

An important difference when using methods in the object-oriented interface instead of functions is that the inputs are not inserted one by one inside parentheses in the function call, but are set as object properties and only the object is

passed inside the function. Also, the size and the complexity of the *computations* component can change drastically depending on what is implemented. For that reason, for every computation or cluster of computations that carry a significant meaning, it must have a title and a brief explanation of what is being computed.

Thus, the general structure of a method should be:

```
%% Method 'Title'

function [output1, output2] = nameMethod(obj)

    %% <-- Note the section marker here!
    %
    % nameMethod - Brief explanation of what the
    % mrthod does and where it is used in the si-
    % mulations.
    %
    % The inputs are:
    %
    %% <-- Note the section marker here!
    %
    %   (1)  input1 : Complete name of input 1
    %   (2)  input2 : Complete name of input 2
    %   (3)  input3 : Complete name of input 3
    %
    % The outputs are:
    %
    %% <-- Note the section marker here!
    %
    %   (1)  output1 : Complete name of output 1
    %   (2)  output2 : Complete name of output 2
    %   (3)  output2 : Complete name of output 3

    %% SUM OF NUMBERS
    %-------------------------------------------------%
    % Compute the sum of the first two inputs. This
    % value is necessary to compute the arithmetic
    % prograssion of defined in the third input.
    %-------------------------------------------------%

    component1 = obj.input1;
    component2 = obj.input2;

    output1 = component1 + component2;

    end
```

3.3 Properties

The *properties* is the equivalent of the function inputs in the object-oriented framework. In this section, the properties used in all the methods of the class are defined in a list. The only specification to be followed here is to specify the complete name of the property and their respective role inside its respective method. This way, whenever the specif role played by the input is not known, the programmer can check the properties section for the complete explanation.

3.4 Classes

The classes are ensembles of methods that have the same central objective. Whenever a group of methods with the same purpose or central role in identified, the programmer can create an additional class in the +Core folder. The classes already created and the their functionality are presented in the `README.pdf` file.

The structure of a general class is composed by the class definition, the class introduction, the properties and the methods. The class definition is responsible for naming the class and the class introduction (initialized by a section mark) gives a general explanation of what the methods inside the class are made for and why they were clustered together. They work as guideline to correctly place new methods inside the class. We also used the C++ and Java standards for class names, so the nomenclature rules are:

Classes

- Start with **upper** case;
- In the case of composite name, cluster together without space, hyphen or underline;
- In the case of composite name, use upper case to initiate names after the first one;

```
% EXAMPLE CLASS
%-------------------------------------------%
% Class containing all the assembling
% methods used in the simulations.
%-------------------------------------------%
classdef AssemblerADRHandler
```

We remark that the methods inside the class can use auxiliary functions, that are not placed in a second class because they share the same role of the class methods, but are also not included as core methods so that the hierarchy inside the class in not broken. In that case, the methods are placed in the end of the class (after the core methods) and function a regular Matlab function.

This way, a general shape for the Matlab classes in HiMod Tree must have the following structure:

```
classdef ClassName

    %% Introduction to 'ClassName'
    %-----------------------------------------%
    % Explanation of the main methods
    % and purpose of the class.
    %-----------------------------------------%

    properties (SetAccess = public, GetAccess = public)

        %% 'ClassName' - OBJECT PROPERTIES
        %-----------------------------------------%
        % Brief explanation of the properties
        % used in the methods.
        %-----------------------------------------%

        % TYPE 1 PROPERTIES

        property1;  % Complete name and
                    % explanation of property1;

        property2;  % Complete name and
                    % explanation of property2;

        % TYPE 2 PROPERTIES

        property3;  % Complete name and
                    % explanation of property3;

        property4;  % Complete name and
                    % explanation of property4;

    end

    methods (Access = public)

        %% 'ClassName' - CONSTRUCT METHOD
        %-----------------------------------------%
        % Definition of the construct method.
        %-----------------------------------------%

        %% 'ClassName' - TYPE 1 METHODS
```

```
43          %-------------------------------------------%
44          % Explanation of the methods of TYPE 1.
45          %-------------------------------------------%
46
47              %% Method 'exampMethod1'
48
49          %% 'ClassName' - TYPE 2 METHODS
50          %-------------------------------------------%
51          % Explanation of the methods of TYPE 2.
52          %-------------------------------------------%
53
54              %% Method 'exampMethod2'
55
56              %% Method 'exampMethod3'
57
58      end
59  end
```

3.5 Demos

The demonstration file is the test file used to run the simulation in HiMod Tree. It is formed several sections that can usually be separated in two parts. The specifications and the solver. In the specifications section we assign all the values necessary to define the differential problem, the discretization parameters, convergence parameters, among others. It is important to notice, that all parameters need to be assigned only here and passed along by the solver method to the respective core methods. Assigning variables inside the methods makes it very hard to analyse the simulation and possible errors. That is the reason why the 'demo' file was created. Finally, we have the call of the solver method that organize the whole simulation. In the README.pdf file you can find some common specifications used in the project.

A general structure of the demonstration file is as follows:

```
1  %% DEMONTRATION NAME
2  %-------------------------------------------%
3  % Specifies what simulation will be performed,
4  % the differential problem solved and the re-
5  % sults that will be displayed.
6  %-------------------------------------------%
7
8      %% IMPORT CORE CLASSES
9
10     import Core.Class1
11     import Core.Class2
12     import Core.Class3
```

```
13
14      %% Simulation case
15
16      case  = 1;       % Analysed Case
17
18      %% Discrtization parameters
19      %-------------------------------------------------%
20      % Description of what specifications are in-
21      % serted here.
22      %-------------------------------------------------%
23
24      domainLimitInX       = [minX,maxX];
25      domainLimitInY       = [minY,maxY];
26
27      numbModes        = 5;
28      numbDim          = length(numbModes);
29      stepHorMesh      = 0.001;
30      numbElements     = round((maxX-minX)/stepHorMesh);
31
32      %% Solver
33      %-------------------------------------------------%
34      % Definition of the solver to be called and
35      % the properties necessary to run the inter-
36      % nal methods.
37      %-------------------------------------------------%
38
39      objClass1 = Class1();
40
41      % Properties Assignment
42
43      objClass1.numbModes = numbModes;
44      objClass1.stepHorMesh = stepHorMesh;
45      objClass1.numbElements = numbElements;
46
47      tic;
48      [output1,output2] = coreMethod2(objClass1);
49      toc;
```

4 Who do I talk to?

In case you have any doubts about this guide or the code please take a look in the `README.pdf` file present in the same folder of this document. In the case of further doubts, please contact the code administrator.

CODE ADMINISTRATOR

- Name: Yves Antonio BRANDES COSTA BARBOSA
- Email: yvesantonio.brandes@mail.polimi.it

PROJECT ADMINISTRATOR

- Name: Simona PEROTTO
- Email: simona.perotto@polimi.it