UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

BACHELOR'S FINAL THESIS

Study of material interpolation for 3D lightweight structures

Document:

Appendix

Author:

Geli I Cerdan, Jofre

Director:

Ferrer Ferre, Alex

Degree:

Bachelor's Degree in Aerospace Technology Engineering

Examination session:

Spring, 2022.

Contents

List of Figures

List of Tables

1 Git example

In order to show the features of Git within MATLAB, an example has been made. The following example can be divided into two parts, the first one is focused on Git and GitHub commands, whereas the second part is a continuation of the first one but centered on the branches.

1.1 Git and GitHub commands

This example starts with a script named 'Example.m', where the following commentary has been added.

```
%% Example
```

The figure below shows the directory where the script has been stored. It is important to note, that the script is in a folder named 'Git example', and at the same time it is inside the folder 'TFG_Jofre'. This last folder has been tracked by Git and also saved in GitHub, as it can be seen in Figure 2.

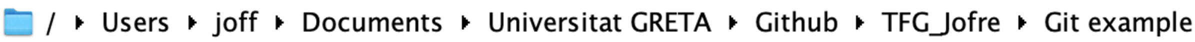


Figure 1: Directory of the 'Example.m' stored in the internal memory of the computer

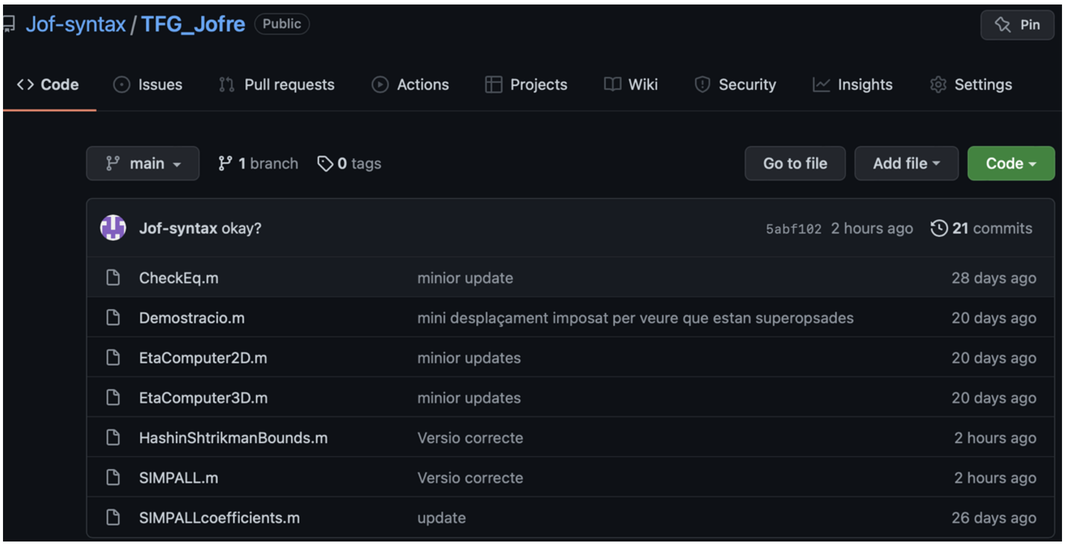


Figure 2: List of files inside the 'TFG_Jofre' in GitHub repository before the example commit

Since 'Example.m' has not been tracked by git yet, it is shown as a dot in the 'Current Folder' section within the MATLAB main window program, see Figure 3.

Figure 3: Status of the 'Example.m' file as null

A right-click has been made over the script name and then a click over the 'Source control' option, and the Figure 4 pop-up has appeared. It should be pointed out that the figure below diverges from

Figure 9 (report), because in this case the file has not been tracked whereas in the other image it has.

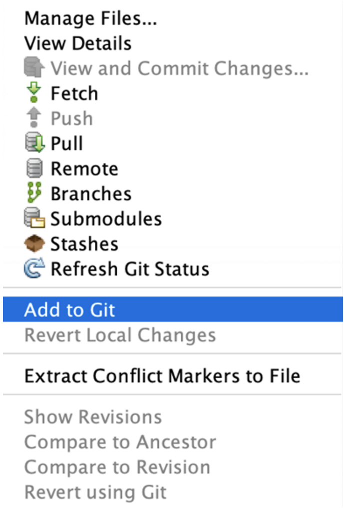


Figure 4: Source control options for a file not tracked by Git

From this point the status has changed to a '+' sign, as seen in image 5. This plus sign means that the file has been added and moved to the staging area, see Figure 1 (report), but not committed.

Figure 5: Status of the 'Example.m' file as added

A commit has been made using the 'Source control' options of Figure 9 (report). It has been named as follows.

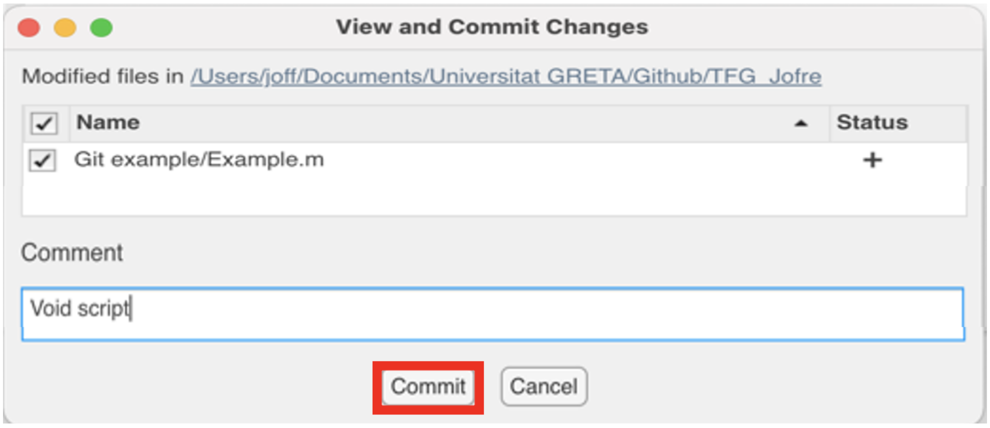


Figure 6: Commit window in Matlab for the first commit

Figure 7: Status of the 'Example.m' after the first commit

The script has been modified as follows, and the status has changed to a blue square. It means that 'Example.m' has been modified but not captured by Git.

```
%% Example

disp('Hello world');
```

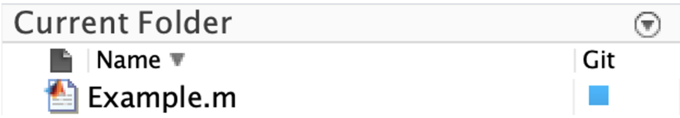


Figure 8: Status of the 'Example.m' file after the modification

Using the same procedure followed in the first commit, a second commit has been realized, see Figures 9 and 10.

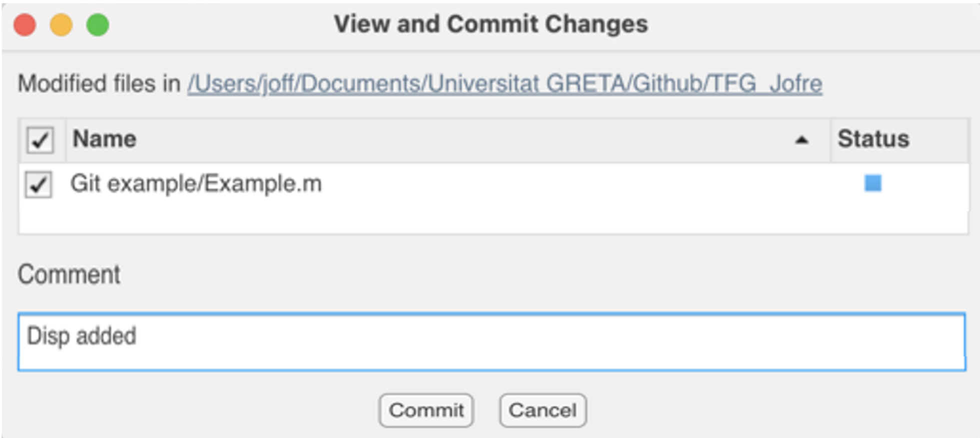


Figure 9: Commit window in Matlab for the second commit

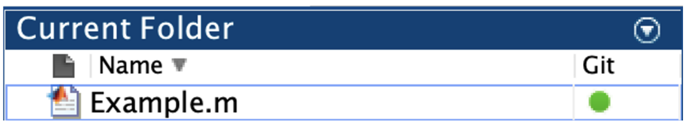


Figure 10: Status of the 'Example.m' file after the second commit

Finally, using once again the menu from Figure 9 (report), a push has been made. It is noteworthy to mention that the files within the 'TFG_Jofre' folder are being tracked by Git and stored in GitHub.

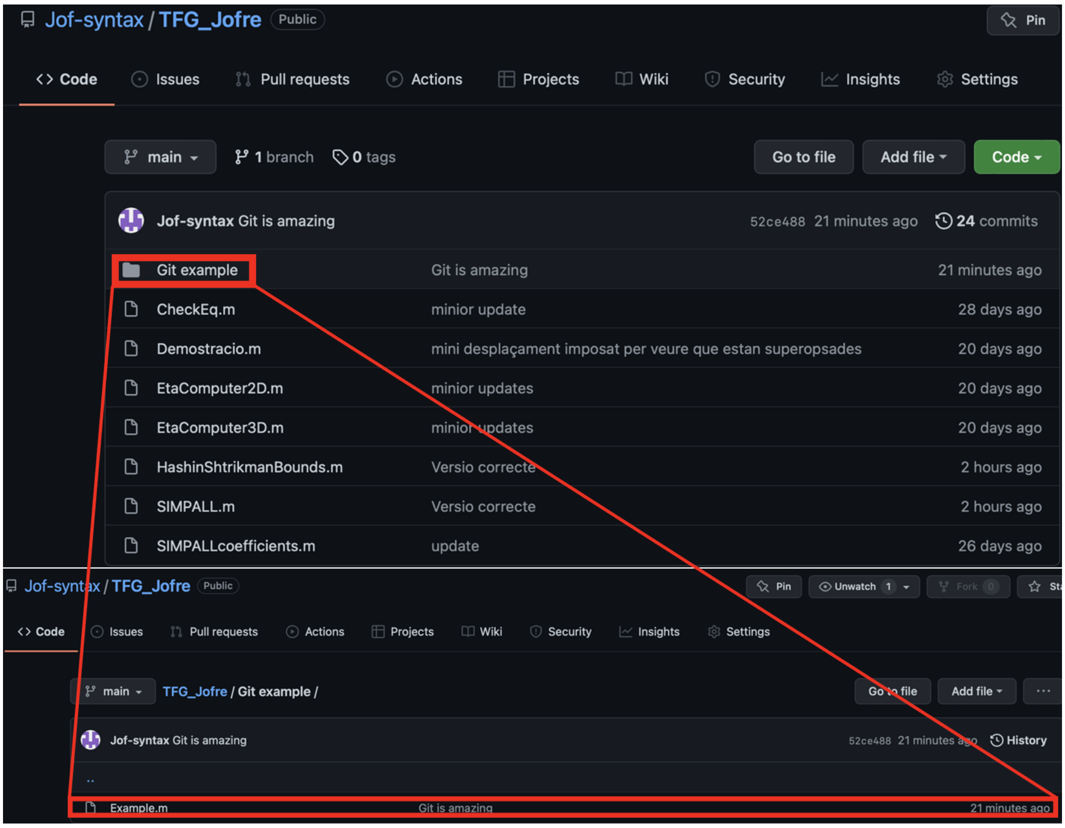


Figure 11: List of files inside the 'TFG Jofre' repository in GitHub after the push action

1.2 Branching

The second part of the example starts once the file has been stored in GitHub. Nevertheless, this part follows the example shown in Figure 7 (report). In this case, the main branch is intended to be indirectly modified by means of a second branch.
In terms of branches, the script only exists in the master branch. Therefore, at the beginning of this part, of the example, the code of the main branch is as follows.
Master branch code:

```
%% Example

disp('Hello world');
```

Moreover, by clicking on the 'Branches' option in the 'Source control' menu, it can be seen that the current branch is the main (master), see Figure 12.

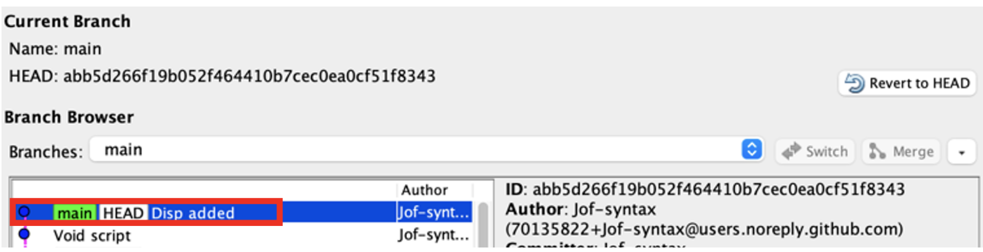


Figure 12: Branches pop-up window, where there is only one branch named 'main'

On the other hand, using the branch creation tool, which is located at lower part of the branches window, a new branch tagged 'BranchA' has been created.

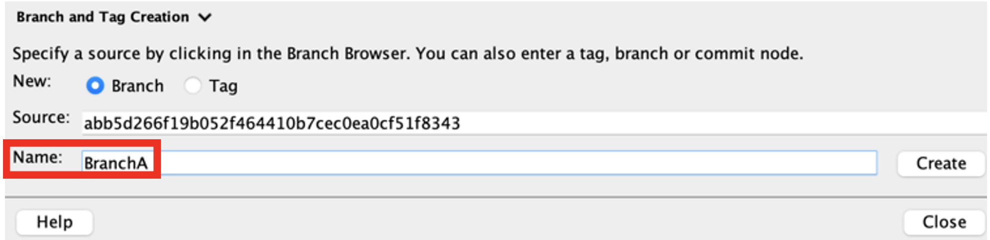


Figure 13: Lower part of the branches window, where a new branch has been created with the name 'BranchA'

Consequently, two branches exist for the file 'Example.m'. In order to modify a branch, the user must switch to the desired one. In this case, it has been switched to the 'BranchA'. It is noteworthy to mention that the code for each of the branches looks as follows.

Master branch code:

```
%% Example

disp('Hello world');
```

Branch 'A' code:

```
%% Example

disp('Hello world');
```

Furthermore, the new branch can be seen in the branches window, Figure 14.

Figure 14: Branches pop-up, after the creation of the master (main) and 'BranchA'

A new modification has been implemented in the branch 'A' code, in consequence, the modified code is:
Branch 'A' code:

```
%% Example

disp('Hello world');

disp('Git is an outstanding tool');
```

This new version of the code has been committed using the same procedure explained in subsection 1.1.

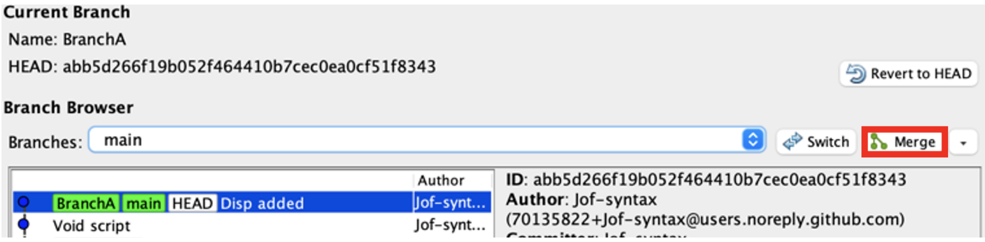


Figure 15: Branches pop-up window of the highlighted merging button

Once the merging process had been successfully completed, the branch 'A' has been deleted using the button of Figure 16.

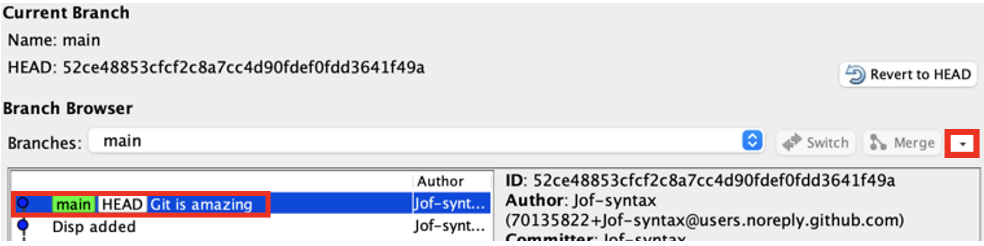


Figure 16: Branches pop-up window, where the only existing branch and the delete button highlighted

Finally, the master branch was modified indirectly, and the resultant code in the main branch is:

Master branch code:

```
%% Example

disp('Hello world');

disp('Git is an outstanding tool');
```

2 Object oriented programming example

The main features of objected oriented programming used in the thesis are shown in this example. It is based on a specific group of classes from 'CodiCante', which can be accessed: `https://github.com/Jof-syntax/CodiCanteUML`. The set of classes create an interesting group to be discussed. The aim of the set is to select and execute an specific solver chosen by the user. Furthermore, the group's structure can be seen in the following UML.

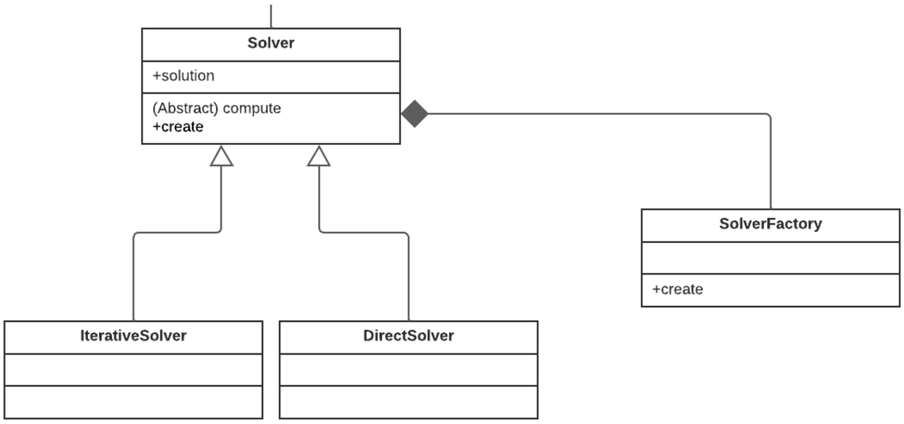


Figure 17: UML for factory group in 'Codi Cante'

In order to fully comprehend the signs and information provided in the previous image, it is important to see the 4.4 subsection (report).

Solver.m

The main class of the group is the Solver.m file. As it can be seen in the following script, the class is composed of three properties and three methods.

Regarding the properties' attributes, since the access is protected, the class and subclasses can access the information. Nevertheless, the 'solution' property has also a public Get-access, it means that Matlab can displays in the command window the name and value of the property.

On the other hand, the first method is static (does not need an obj) and its aim is to call the SolverFactory.m and execute the specific solver type. The second method is intended to be called from the child subclasses, it avoids the definition in each subclass. In case of further modifications, e.g. more inputs needed, the init() Solver.m's function has only to be modified. It avoids the modification of an specific init() function for each subclass. Finally, the last method defines and abstract function with protected access. The objective is to avoid the definition of a generalized computation function, instead a specific compute function for each solver type can be defined in each subclass.

```
classdef Solver < handle

    % solution is the output of the problem
    properties (GetAccess = public, SetAccess = protected)
        solution
    end

```

```
    % A and B are input matrices
    properties (Access = protected)
        A
        B
    end

    methods (Static, Access = public)

       function obj = create(cParams)
              % Run the SolverFactory.create(), which selects the specific solver
              obj = SolverFactory.create(cParams);
              % Execute of the abstract function 'compute()' defined in the
    subclasses
              obj.compute();
       end

    end

    methods (Access = protected)

        function init(obj,cParams)
            % Stores the parameters A and B as object properties
            obj.A   = cParams.A;
            obj.B   = cParams.B;
        end

    end

    methods (Abstract, Access = protected)
        % Function created in the Solver class but defined in another subclass (
    Abstract)
        compute(obj);
    end
end
```

SolverFactory.m

The script below works as a switch, where depending on the user's input the executed function can be the 'DirectSolver' or the 'IterativeSolver' subclasses. It is important to note that the method is static and there is no constructor in the class. Consequently, the class is called as SolverFactory.create(cParams) in the Solver.m class. The output of the class is the execution of the chosen solver.

```
classdef SolverFactory < handle

    methods (Access = public, Static)

        function computeSolver = create(cParams)
            % Switch chooses between 'direct' or 'iterative' option
            switch cParams.type
                case 'direct'
                    % Executes the DirectSolver.m script
                    computeSolver = DirectSolver(cParams);
                case 'iterative'
                    % Executes the IterativeSolver.m script
                    computeSolver = IterativeSolver(cParams);
                otherwise
```

```
                    % Error when the switch do not have to possibility to choose between 'direct' or 'iterative'
                    error('Invalid solver type.');
                end
            end

        end
end
```

<u>*DirectSolver.m & IterativeSolver.m*</u>

The following classes are subclasses of Solver.m, as it can be see in the heading of the scripts (classdef XXX < Solver). They allow the computation of the solution by means of the compute() function. In this case, the input information is provided by the properties of the Solver.m class. Both cases show a similar structure. On the one hand, a public method, which initiates the problem by calling the init() Solver.m's function. On the other hand, a protected method, which computes the actual solution of the problem.

```
classdef DirectSolver < Solver

    methods (Access = public)

        % The constructor initiates the script by calling the init() parent's function, which recovers the A and B matrices from cParams
        function obj = DirectSolver(cParams)
            obj.init(cParams);
        end

    end

    methods (Access = protected)

        function obj = compute(obj)
            A = obj.A;
            B = obj.B;
            % The solution is computed by means of the '/' Matlab operator
            obj.solution = A\B;
        end

    end

end
```

```
classdef IterativeSolver < Solver

    methods (Access = public)

        % The constructor initiates the script by calling the init() parent's function
        function obj = IterativeSolver(cParams)
            obj.init(cParams);
        end

    end

    methods (Access = protected)

```

```
        function obj = compute(obj)
            A = obj.A;
            B = obj.B;
             % The solution is computed by means of the pcg Matlab function (
    iterative function)
            obj.solution = pcg(A, B);
        end

    end

end
```

It is important to note that the classes are defined in a general way, so as to allow the re-usability in other projects. The intention to write a generalized code is further discussed in the Clean Code section, see section 5 (report).

3 Testing example

In order to show the testing techniques used in MATLAB, an example is provided. The following example can be divided into three parts. The first one is focused on how to write tests, the second part is an example of code coverage, and finally, the third one shows two examples of UMLs.

3.1 Testing

During the refactoring of the 'CodiCante', tests for each class have been created. The tests can be seen in: `https://github.com/Jof-syntax/CodiCanteUML` . It is remarkable that these tests follow the *static stored data* scheme described in subsection 4.1 (report). On the other hand, the script structure of each test has been done following the next three points:

- Compute current results: By means of the tested class, the outputs are obtained using the same inputs that had been used to compute the expected results.
- Load expected results: These outputs are the reference values to check the current outputs results. They are obtained at the beginning of the refactoring and are considered as the correct values of the program.
- Verify results: Comparison of the current and expected results, where the comparison's answer is a string in the command window.

In order to better comprehend the previous points, the example of the 'GliderAnalyser.m' test function is provided below.

```
function TestGliderAnalyser()
    % Obtain current results
    InputData = load('TestData/Input data/TestClassDataGliderAnalyser.mat').cParams;
    test = GliderAnalyser(InputData);
    test.compute();
    % Load stored results (expected)
    expectedResult = load('TestData/Outputdata/ResultTestGliderAnalyser.mat').test;
    % Verify the expected and current results
    computeError(test, expectedResult, 'TestGliderAnalyser');
end
```

Depending on the comparison result, the outputs of this test function can be Figure 16 or 17 (report) from Section 4 (report). Furthermore, the called function 'computeError(test, expectedResult, Name)' is defined as follows.

```
function computeError(test, expectedResult, Name)
    % test: Object of current results
    % expectedResult: Object of expected results
    % Name: String that appears to indicate the executed test
    if isequaln(test, expectedResult)
        % Satisfactory message in command window
        cprintf('green',[Name,' --> ']);
        cprintf('green',' PASSED. \n');
    else
        % Unsatisfactory message in command window
        cprintf('red',[Name,' --> ']);
        cprintf('red',' FAILED. \n');
    end
end
```

3.2 Code coverage

The coverage of the tests presented in subsection 3.1 has been done by means of the functions shown in section 4.2 (report). For this specific example, a repository containing the repository from subsection 3.1 and the two new functions has been created. It can be found in: `https://github.com/Jof-syntax/TestRunnerCodiCante` . The aim of this new repository is to allow the easy execution of the 'TestRunner.m', so as to avoid the unnecessary move of scripts inside the folders.

Since all the tests are specific cases of the 'testGliderAnalyser.m' test function, the coverage has been made tracking it. On the other hand, the results from this coverage are provided below:

SolverTest.m From this function, the results are shown in the command window, see the figure below.

Name	Passed	Failed	Incomplete	Duration	Details
{'SolverTest/testGliderAnalyser'}	true	false	false	0.62801	{1×1 struct}

Figure 18: Result from 'SolverTest.m' function

From figure 18, it can be seen that the test has been satisfactory. It means that the test has been executed without problems and the output results from the expected and current are the same.

TestRunner.m This function has generated a plugin, whose results are recovered in the following table.

FILE NAME	LINE COVERAGE	EXECUTABLE LINES	EXECUTED LINES	MISSED LINES
CGComputer.m	100%	22	22	0
CheckSafety.m	100%	14	14	0
ConservativeForcesComputer.m	100%	49	49	0
DOFSplitterComputer.m	100%	24	24	0
Dimension.m	100%	11	11	0
DirectSolver.m	100%	4	4	0
DisplacementComputer.m	100%	39	39	0
DynamicSolver.m	100%	54	54	0
ExternalInfluence.m	100%	15	15	0
ForcesComputer.m	100%	47	47	0
GliderAnalyser.m	100%	44	44	0
GliderData.m	100%	40	40	0
GliderGeometry.m	100%	24	24	0
GliderMass.m	100%	22	22	0
GliderMaterial.m	100%	29	29	0
IterativeSolver.m	0%	4	0	4
MatrixAndVectorSplitter.m	100%	13	13	0
NodeForceComputer.m	100%	68	68	0
NotConservativeForcesComputer.m	100%	45	45	0
PlotBarStress.m	100%	47	47	0
ResultComputer.m	100%	17	17	0
Solver.m	100%	4	4	0
SolverFactory.m	42.85%	7	3	4
SolverTest.m	100%	8	8	0
StiffnessMatrixComputer.m	100%	55	55	0
StressComputer.m	100%	44	44	
TOTAL FILES	**LINE COVERAGE**	**EXECUTABLE LINES**	**EXECUTED LINES**	**MISSED LINES**
27	97.74%	755	738	17

Table 1: Percentage results from 'TestRunner.m' function

Looking at the previous table, the global coverage has only been 97.74%. Nevertheless, this result can be explained as a consequence of the 'inputs'. They only take into consideration the 'direct' solving method. It can be seen in the table, specifically the files 'SolverFactory.m' and 'IterativeSolver.m'. These functions are shown below for reference proposes. Furthermore, the executable lines have been highlighted depending on the case with green or red.

```
     classdef SolverFactory < handle

         methods (Access = public, Static)

             function computeSolver = create(cParams) % la funcio es diu create
1                switch cParams.type
1                    case 'direct'
1                        computeSolver = DirectSolver(cParams);
0                    case 'iterative'
0                        computeSolver = IterativeSolver(cParams);
0                    otherwise
0                        error('Invalid solver type.');
                 end
             end

         end
     end
```

Figure 19: Source coverage display of 'SolverFactory.m' function

```
     classdef IterativeSolver < Solver

         methods (Access = public)

             function obj = IterativeSolver(cParams)
0                obj.init(cParams);
             end

         end

         methods (Access = protected)

             function obj = compute(obj)
0                A = obj.A;
0                B = obj.B;
0                obj.solution = pcg(A, B);
             end

         end

     end
```

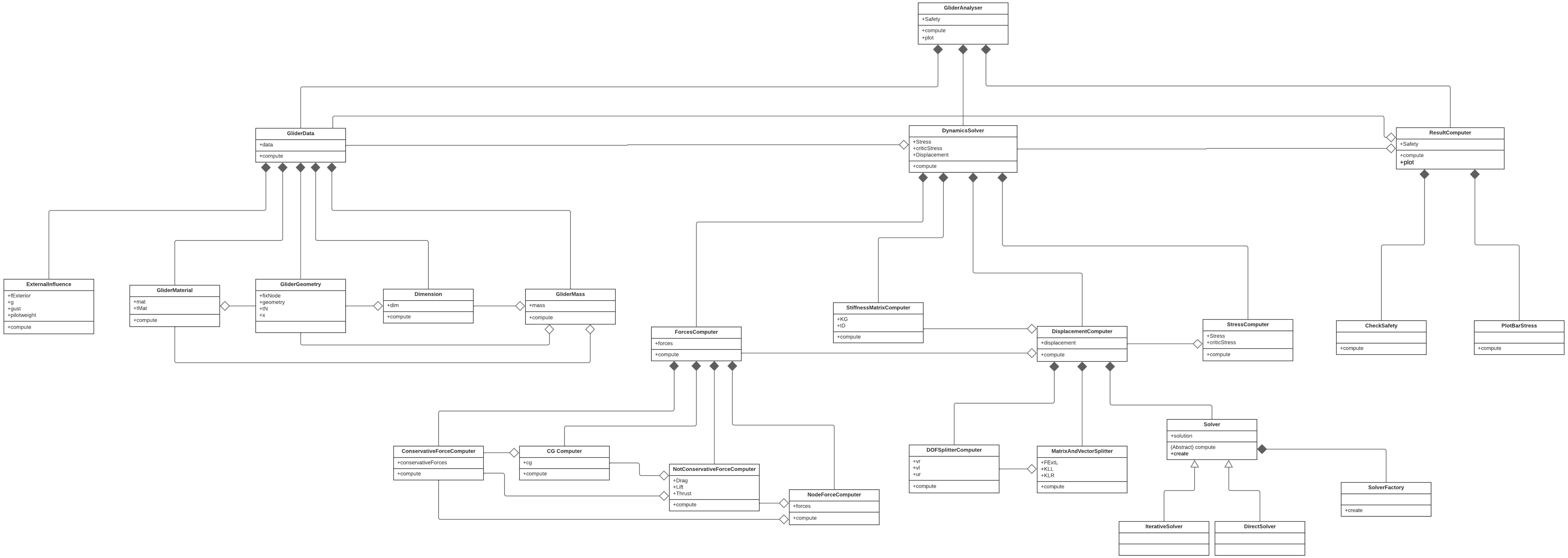
Figure 20: Source coverage display of 'IterativeSolver.m' function

It is remarkable that the number of executed and not executed lines coincide with the 'Executed lines' and 'Missed lines' respectively, from table 1. Moreover, it is noteworthy to mention that these test mistakes have been made in purpose in order to show the scope of the source coverage in MATLAB. Nevertheless, the not executed lines can be avoided by adding two extra tests. One that covers the iterative case (cParams.type='iterative') and another test that executes the otherwise option (cParams.type!='iterative' AND cParams.type!='direct'). Consequently, the covered lines from 'TestRunner.m' function ought to be 100 %.

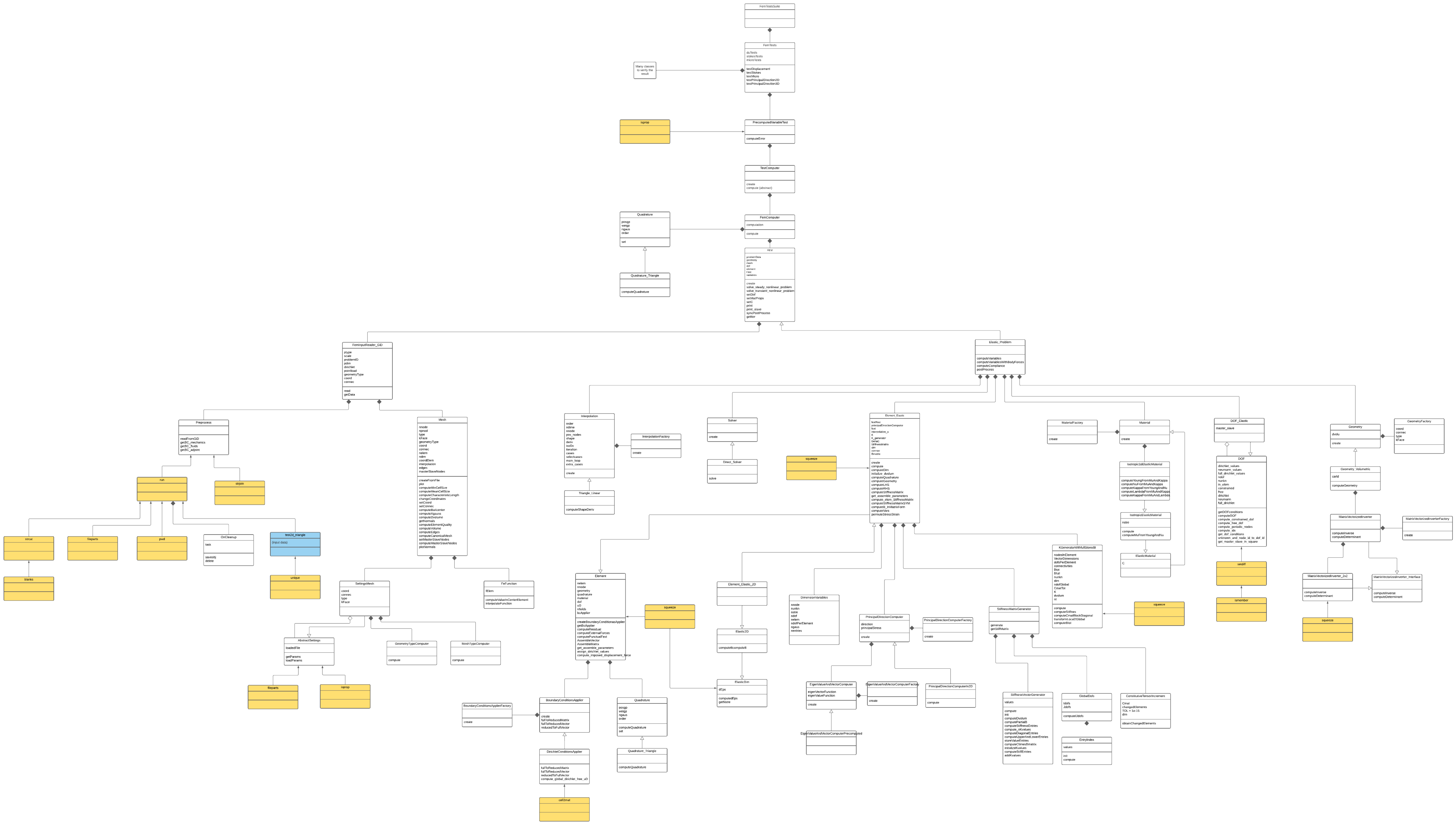
3.3 UMLs

In the following pages, two UML examples are provided. The first UML example corresponds to the final version of "Codi Cante", which can be found in: `https://github.com/Jof-syntax/CodiCanteUML` . It allowed the refactoring of 'codi cante' in a swift manner. On the other hand, the second example corresponds to the FEM functionality of the Swan code, which can be found in: `https://github.com/SwanLab/Swan/blob/master/FEM/FEM.m` . This UML helped in the process of understanding the way Swan code works. It is remarkable that some of the boxes are highlighted in yellow, in order to shown that these are functions instead of classes.

Codi Cante UML



Swan's FEM functionality UML



Note: Since the number of classes that appear is considerable, the association relation has been omitted.

4 Hashin Shtrikman Bounds

The purpose of the present section consists on the derivation of bounds for the effective elastic moduli of multiphase materials in 'N' dimensions using the original work "A variational approach to the theory of elastic behaviour of multiphase materials" by Z. Hashin and S. Shtrikman [1].

Introduction to the variational principle

Let an elastic deformed body with known stress (σ_{ij}^o) and strain (ϵ_{ij}^o) tensor fields have N dimensions as volume (V) and $N-1$ dimensions as boundery (Γ). Hooke's law is given by

$$\sigma_{ij}^o = \lambda_o\, \epsilon_{kk}^o\, \delta_{ij} + 2\, \mu_o\, \epsilon_{ij}^o = L_o : \epsilon_{ij}^o \tag{1}$$

The Lamé parameters λ_o and μ_o represent the Lamé constant and shear modulus in 'N' dimensions, respectively, which for simplicity are taken constant throughout the body. Besides that, the subscripts are whole numbers from 1 to N, a repeated subscript denotes summation and δ_{ij} is the Kronecker delta. On the other hand, the strains are given in terms of displacements by

$$\epsilon_{ij}^o = \frac{1}{2}\left(\frac{\partial u_i^o}{\partial x_j} + \frac{\partial u_j^o}{\partial x_i}\right) \tag{2}$$

Let part or whole of the body be modified with a new material and $u_i^o(\Gamma)$ be held fixed. As a consequence, the new stress and strains fields are unknown matrices in the new body.

Also allow the Hooke's law for the new body be defined as

$$\sigma_{ij} = \lambda\, \epsilon_{kk}\, \delta_{ij} + 2\, \mu\, \epsilon_{ij} \tag{3}$$

Furthermore, let the stress polarization tensor (p_{ij}) be defined as follows

$$\sigma_{ij} = L : \epsilon_{ij} = L_o : \epsilon_{ij} + p_{ij} \tag{4}$$

where L and L_o are given by (1) and (3), respectively. It is remarkable that p_{ij} gives information of both, the new stress caused by the insertion of the new material and the loss of it when mass is subtracted from the original one. Define also

$$\begin{aligned} u_i' &= u_i - u_i^o \\ u_j' &= u_j - u_j^o \end{aligned} \tag{5}$$

Consequently

$$\epsilon_{ij}' = \epsilon_{ij} - \epsilon_{ij}^o \tag{6}$$

It is remarkable that σ_{ij} and ϵ_{ij} can be found from (4) and (6) once p_{ij} and ϵ_{ij}' are found.

Let the variational principle be defined as sum of volume integrals

$$U_p = U_o + U_{p\,\epsilon^o} \tag{7}$$

where

$$U_o = \frac{1}{2}\int \sigma_{ij}^o : \epsilon_{ij}^o\, dV \tag{8}$$

$$U_{p\,\epsilon^o} = \frac{1}{2}\int p_{ij} : \epsilon_{ij}^o\, dV \tag{9}$$

While the first equation (8) is the strain energy associated to the original body, equation (9) is the energy caused by p_{ij} that would have the original body if the stress p_{ij} had been applied.

On the other hand, by means of equation (6), U_p can be expanded as follows

$$U_p = U_o - \frac{1}{2}\int -p_{ij} : (-\epsilon_{ij} + \epsilon'_{ij} + 2\,\epsilon^o_{ij})\, dV \tag{10}$$

Taking into consideration (4), the variational principal involving p_{ij} and ϵ'_{ij} can be formulated as

$$U_p = U_o - \frac{1}{2}\int \left(p_{ij} : \frac{1}{L - L_o} : p_{ij} - p_{ij} : \epsilon'_{ij} - 2\,p_{ij} : \epsilon^o_{ij}\right) dV \tag{11}$$

which is subjected to

$$\frac{\partial(L_o : \epsilon'_{ij} + p_{ij})}{\partial x_j} = 0 \tag{12}$$

and

$$u'_i(\Gamma) = 0 \tag{13}$$

It can be proven that the variational principle is stationary for

$$p_{ij} = L : \epsilon_{ij} - L_o : \epsilon_{ij} \tag{14}$$

It is noteworthy to mention that the result match with equation (4).

Finally, regarding equations (1) and (3), the variational principal (U_p) is an absolute maximum when

$$\lambda > \lambda_o\,, \qquad \mu > \mu_o \tag{15}$$

and an absolute minimum when

$$\lambda < \lambda_o\,, \qquad \mu < \mu_o \tag{16}$$

It has been proven that the variational principle has an absolute maximum and an absolute minimum when part or whole of the original body is modified with another material.

General bounds for the effective moduli of multi-phase materials

Let the quasi-homogenity of the multi-phase material be obtained from any reference cube in the composite material which is large compared to the size of the non-homogeneities, but small compared to the hole body. Consequently, the volume average of a quantity such as strain, displacement, stress or phase volume fraction is the same for the whole body and the reference cube.

Furthermore, let the elastic strain energy in a reference cube of unit volume be represented as follows.

$$U = \frac{1}{2}(N^2\kappa^*\,\epsilon^{o2} + 2\mu^*\,e^o_{ij}\,e^o_{ij}) \tag{17}$$

where the mean strains ϵ^o_{ij} are split into isotropic and deviatoric forms

$$\epsilon^o_{ij} = \epsilon^o\,\delta_{ij} + e^o_{ij} \tag{18}$$

being

$$\epsilon^o = \frac{\epsilon_{kk}}{N} \tag{19}$$

Finally, κ^* and μ^* are the effective bulk and shear moduli.

Besides that, the bulk parameter (κ) for the problem is defined as

$$\kappa = \lambda + \frac{2}{N}\mu \tag{20}$$

Imposing the variational principle from the previous part to the unit volume and assuming that the displacements $u_i^o(\Gamma)$ are impressed on a homogeneous body whose elastic moduli are κ_o and μ_o, from the theory of elasticity follows that the strains throughout the body are constant and equal to ϵ_{ij}^o. Therefore, if $u_i^o(\Gamma) = \epsilon_{ij}^o\, x_j$ (being ϵ_{ij}^o constant and x_j the Cartesian co-ordinates of an inertial system) in composite body is prescribed, the strains have to be

$$\epsilon_{ij} = \epsilon_{ij}^o + \epsilon_{ij}' \tag{21}$$

By definition of (21), since ϵ_{ij}' is a deviation from the mean strain ϵ_{ij}^o, the following statement must be true.

$$\bar{\epsilon}_{ij}' = 0 \tag{22}$$

Then, the volume is divided into r^{th} phases, where each division is denoted by V_r. Therefore,

$$\sum_{r=1}^{r=n} \frac{V_r}{V} = 1 \tag{23}$$

Moreover, the polarization tensor p_{ij} is considered constant within the division.

$$p_{ij} = p_{ij}^r \quad \text{in} \quad V_r \tag{24}$$

Transforming all tensors from (11) into isotropic and deviatoric parts, introducing (24) and using (22) yields the result

$$U_p = U_o + U' - \frac{1}{2}\sum_{r=1}^{r=n}\left[\frac{(p^r)^2}{\kappa_r - \kappa_o} + \frac{f_{ij}^r\, f_{ij}^r}{2(\mu_r - \mu_o)} - 2N\, p^r\, \epsilon^o - 2 f_{ij}^r\, e_{ij}^o\right]\frac{V_r}{V} \tag{25}$$

where

$$U_o = \frac{1}{2}(N^2 \kappa_o\, \epsilon^{o2} + 2\mu_o\, \epsilon_{ij}^o\, \epsilon_{ij}^o) \tag{26}$$

$$U' = \frac{1}{2}\int (N\, p\epsilon' + f_{ij}\, e_{ij}')dV \tag{27}$$

By means of the Fourier methods and making use of (13) and (51), U' can be evaluated in terms of the polarization field (24). The resultant equation is provided below

$$U' = \frac{1}{2}\left(\alpha_o\left[\sum_{r=1}^{r=n}(p^r)^2\frac{V_r}{V} - \left(\sum_{r=1}^{r=n} p^r\frac{V_r}{V}\right)^2\right] + \beta_o\left[\sum_{r=1}^{r=n} f_{ij}^r\, f_{ij}^r\frac{V_r}{V} - \sum_{r=1}^{r=n} f_{ij}^r\frac{V_r}{V}\sum_{r=1}^{r=n} f_{ij}^r\frac{V_r}{V}\right]\right) \tag{28}$$

where α_o and β_o are

$$\alpha_o = -\frac{1}{2\mu_o + \left(\kappa_o - \frac{2}{N}\mu_o\right)} \tag{29}$$

$$\beta_o = -\frac{(\kappa_o + 2\mu_o)\,(N-1)}{(N^2 + N - 2)\,\mu_o\,\left(\kappa_o - \frac{2}{N}\mu_o\right)} \tag{30}$$

Introducing (28) into (25), U_p becomes

$$U_p = U_o - \frac{1}{2}\sum_{r=1}^{r=n}\left[\frac{(p^r)^2}{\kappa_r-\kappa_o} - \alpha_o\,(p^r)^2 + \frac{f_{ij}^r\,f_{ij}^r}{2(\mu_r-\mu_o)} - \beta_o\,f_{ij}^r\,f_{ij}^r - 2N\,p^r\,\epsilon^o + ...\right.$$

$$\left.... - 2f_{ij}^r\,e_{ij}^o\right]\frac{V_r}{V} - \frac{\alpha_o}{2}\left(\sum_{r=1}^{r=n} p^r\,\frac{V_r}{V}\right)^2 - \frac{\beta_o}{2}\sum_{r=1}^{r=n} f_{ij}^r\,\frac{V_r}{V}\sum_{r=1}^{r=n} f_{ij}^r\,\frac{V_r}{V} \tag{31}$$

Regarding the maximum condition (15), whenever κ_o and μ_o satisfy the following inequalities, the energy inequality is also satisfied

$$\kappa_r > \kappa_o\,, \qquad \mu_r > \mu_o \qquad \text{which implies} \qquad U_p < U \tag{32}$$

Similarly, from (16), whenever κ_o and μ_o are lower than κ_r and μ_r, the following inequalities are satisfied

$$\kappa_r < \kappa_o\,, \qquad \mu_r < \mu_o \qquad \text{which implies} \qquad U_p > U \tag{33}$$

From this point a myraid of bounds on the effective elastic moduli have been found. Nevertheless, in order to obtain the best bounds for a polarization tensor of the type (24), U_p must be maximized for condition (32) and minimized for condition (33). Therefore,

$$\frac{\partial U_p}{\partial p^r} = 0 \qquad \Longrightarrow \qquad \sum_{r=1}^{r=n}\left[-\frac{p^r}{\kappa_r-\kappa_o} + \alpha_o\,p^r\left(1-\frac{V_r}{V}\right) + N\,\epsilon^o\right]\frac{V_r}{V} = 0 \tag{34}$$

$$\frac{\partial U_p}{\partial f_{ij}^r} = 0 \qquad \Longrightarrow \qquad \sum_{r=1}^{r=n}\left[-\frac{f_{ij}^r}{2(\mu_r-\mu_o)} + \beta_o\,f_{ij}^r\left(1-\frac{V_r}{V}\right) + e_{ij}^o\right]\frac{V_r}{V} = 0 \tag{35}$$

Introducing (34) and (35) into (31), U_p can be refactored for the extreme cases as follows

$$\hat{U_p} = U_o + \frac{1}{2}(N\hat{p}\,\epsilon^o + \hat{f}_{ij}\,e_{ij}^o) \tag{36}$$

where the mean values $\hat{p}$ and $\hat{f}$ can be determined from (34) and (35) by solving for p^r and f_{ij}^r and obtaining the mean values by means of (24). The results are

$$\hat{p} = \epsilon^o\frac{N\,A}{1+\alpha_o\,A} \tag{37}$$

$$\hat{f_{ij}} = e_{ij}^o\frac{B}{1+\beta_o\,B} \tag{38}$$

where

$$A = \sum_{r=1}^{r=n}\frac{\frac{V_r}{V}}{\frac{1}{\kappa_1-\kappa_o} - \alpha_o} \tag{39}$$

$$B = \sum_{r=1}^{r=n}\frac{\frac{V_r}{V}}{\frac{1}{2(\mu_1-\mu_o)} - \beta_o} \tag{40}$$

In order to find bounds for the effective bulk modulus let a mean strain system of the form

$$\epsilon_{ij}^o = \epsilon^o\,\delta_{ij} \tag{41}$$

be applied to the composite material. Then, it can be proven from (17), (32), (33), (36) and (37) that

$$\kappa^* \lesseqgtr \kappa_o + \frac{A}{1 + \alpha_o\, A} \tag{42}$$

In line with the previous deduction, the bounds for the effective shear can be found imposing a purely deviatoric form

$$\epsilon^o_{ij} = e^o_{ij}, \quad e^o_{kk} = 0 \tag{43}$$

then from (17), (32), (33), (36) and (38),

$$\mu^* \lesseqgtr \mu_o + \frac{1}{2}\frac{B}{1 + \beta_o\, B} \tag{44}$$

where the upper inequality signs applies for the conditions (32) and the lower signs for the conditions (33).

Finally, it can be proven that equations (42) and (44) are equivalent to inequalities (45) when $\kappa_o = \kappa_A$, $\mu_o = \mu_A$ for the lower bounds and $\kappa_o = \kappa_B$, $\mu_o = \mu_B$ for the upper bounds being $n = 2$.

$$\begin{aligned}
\frac{\theta}{2\,\mu_B - 2\,\mu^*_{UB}} &\leqslant \frac{1}{2\,\mu_B - 2\,\mu_A} + \frac{(1-\theta)\,(\kappa_B + 2\,\mu_B)\,(N-1)}{\mu_B\,(N^2+N-2)\left(\kappa_B + 2\,\mu_B - \frac{2\,\mu_B}{N}\right)} \\
\frac{1-\theta}{2\,\mu^*_{LB} - 2\,\mu_A} &\leqslant \frac{1}{2\,\mu_B - 2\,\mu_A} + \frac{\theta\,(\kappa_A + 2\,\mu_A)\,(N-1)}{\mu_A\,(N^2+N-2)\left(\kappa_A + 2\,\mu_A - \frac{2\,\mu_A}{N}\right)} \\
\frac{\theta}{\kappa_B - \kappa^*_{UB}} &\leqslant \frac{1}{\kappa_B - \kappa_A} + \frac{1-\theta}{\kappa_B + 2\,\mu_B - \frac{2\,\mu_B}{N}} \\
\frac{1-\theta}{\kappa^*_{LB} - \kappa_A} &\leqslant \frac{1}{\kappa_B - \kappa_A} + \frac{\theta}{\kappa_A + 2\,\mu_A - \frac{2\,\mu_A}{N}}
\end{aligned} \tag{45}$$

Proofs and relations of the variational principle

In order to fully understand the steps carried out in the previous parts, the following relations and proofs are provided.

Strain relation

First, let (5) be differentiated with its orthogonal direction as follows.

$$\begin{aligned}
\frac{\partial u'_i}{\partial x_j} &= \frac{\partial u_i}{\partial x_j} - \frac{\partial u^o_i}{\partial x_j} \\
\frac{\partial u'_j}{\partial x_i} &= \frac{\partial u_j}{\partial x_i} - \frac{\partial u^o_j}{\partial x_i}
\end{aligned} \tag{46}$$

Second, combine the previous equations and multiply the resultant expression by a factor of 0.5.

$$\frac{1}{2}\left(\frac{\partial u'_i}{\partial x_j} + \frac{\partial u'_j}{\partial x_i}\right) = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{1}{2}\left(\frac{\partial u^o_i}{\partial x_j} + \frac{\partial u^o_j}{\partial x_i}\right) \tag{47}$$

Finally, regarding equation (2), equation (47) becomes (6).

The stationary value

The proof relies on the derivation of the Variational principle (U_p). It can be done with an extra parameter λ following the restricted optimization method. In this differentiation the parameter λ is maximized, while p and ϵ minimized. Therefore

$$\max \lambda \quad \min p\,\epsilon \qquad U_o - \frac{1}{2}\int \left(p_{ij} : \frac{1}{L - L_o} : p_{ij} - p_{ij} : \epsilon'_{ij} - 2\,p_{ij} : \epsilon^o_{ij} \right) dV + ...$$
$$... + \lambda\,\frac{\partial (L_o : \epsilon'_{ij} + p_{ij})}{\partial x_j} \tag{48}$$

In order to simplify the calculus, $\lambda\,\frac{\partial}{\partial x_j}$ is replaced by ϵ_λ.

The next consist on the differentiation of U_p for each of the terms.

$$\begin{aligned}
\frac{\partial U_p}{\partial p_{ij}} = 0 \quad &\Longrightarrow \quad \frac{-p_{ij}}{L - L_o} + \frac{\epsilon'_{ij}}{2} + 2\,\epsilon^o + \epsilon_\lambda = 0 \\
\frac{\partial U_p}{\partial \epsilon'_{ij}} = 0 \quad &\Longrightarrow \quad \frac{p_{ij}}{2} + \epsilon_\lambda\,L_o = 0 \\
\frac{\partial U_p}{\partial \epsilon_\lambda} = 0 \quad &\Longrightarrow \quad p_{ij} + L_o\,\epsilon'_{ij} = 0
\end{aligned} \tag{49}$$

Solving the system for p_{ij} and applying the relation (6), the stationary condition (14) is obtained.

<u>*Subsidiary condition in terms of u'_i*</u>

On the one hand, the subsidiary condition (12) can be expanded by means of equation (2) as

$$\frac{\partial}{\partial j}\left(L_o : \frac{1}{2}\left(\frac{\partial u'_j}{\partial ji} + \frac{\partial u'_i}{\partial jj} \right) \right) + \frac{\partial p_{ij}}{\partial j} = 0 \tag{50}$$

On the other hand, replacing L_o with its definition in (1), the following equation is obtained.

$$\left(\frac{\lambda_o}{2} + \mu_o \right) \frac{\partial u'_j}{\partial ji} + \mu_o\,\frac{\partial u'_i}{\partial jj} + \frac{\partial p_{ij}}{\partial j} = 0 \tag{51}$$

<u>*$\hat{U_p}$ deduction from U_p*</u>

Insulating ϵ^o and e^o_{ij} from (34) and (35) respectively, and introducing the results in (31) yields

$$\hat{U_p} = U_o - \frac{1}{2}\sum_{r=1}^{r=n}\left(N\hat{p}^r\,\frac{V_r}{V} \right)\epsilon^o + N\hat{p}\,\epsilon^o - \frac{1}{2}\sum_{r=1}^{r=n}\left(\hat{f}^r_{ij}\,\frac{V_r}{V} \right)\epsilon^o + \hat{f}_{ij}\,e_{ij} \tag{52}$$

Simplifying the summation expressions as the mean values $\hat{p}$ and $\hat{f}_{ij}$

$$\hat{U_p} = U_o - \frac{N}{2}\hat{p}\,\epsilon^o + N\hat{p}\,\epsilon^o - \frac{1}{2}\hat{f}_{ij}\,e_{ij} + \hat{f}_{ij}\,e_{ij} \tag{53}$$

Then, equation (53) can be simplified into (36).

<u>*HS inequalities deduction from absolute energy*</u>

By means of the inequalities (32) and (32), and making use of expressions (17) and (36), the equation below is obtained

$$U \lesseqgtr U_o + \frac{1}{2}(N\hat{p}\,\epsilon^o + \hat{f}_{ij}\,e^o_{ij}) \tag{54}$$

Expanding this expression with the bulk and shear elastic strain energy

$$\frac{1}{2}(N^2\kappa^*\,(\epsilon^o)^2 + 2\mu^*\,e^o_{ij}\,e^o_{ij}) \lesseqgtr \frac{1}{2}(N^2\kappa^o\,(\epsilon^o)^2 + 2\mu_o\,e^o_{ij}\,e^o_{ij}) + \frac{1}{2}(N\hat{p}\,\epsilon^o + \hat{f}_{ij}\,e^o_{ij}) \tag{55}$$

Introducing (37) in the previous equation and splitting it into its isotropic part as

$$N^2\kappa^*\,(\epsilon^o)^2 \lesseqgtr N^2\kappa^o\,(\epsilon^o)^2 + N\frac{N\,A}{1+\alpha_o\,A}\,(\epsilon^o)^2 \tag{56}$$

Using the same deduction with (38), the deviatoric part is obtained

$$2\mu^*\,e^o_{ij}\,e^o_{ij} \lesseqgtr 2\mu_o\,e^o_{ij}\,e^o_{ij} + \frac{B}{1+\beta_o\,B}e^o_{ij}\,e^o_{ij} \tag{57}$$

Finally, the previous equations can be simplified and consequently (42) and (44) are obtained.

Deduction of Allaire inequalities from HS inequalities

First, recovering the shear equation (44) and replacing the parameter μ_o for μ_A

$$\mu^* - \mu_A = \frac{1}{2}\frac{B}{1+\beta_o\,B} \tag{58}$$

Introducing (40), imposing the case $v_r = 1-\theta$ (being θ the porosity) and expanding the denominator from the left hand side

$$(\mu^* - \mu_A)\left(1+\beta_o\,\frac{1-\theta}{\frac{1}{2(\mu_B-\mu_A)}-\beta_o}\right) = \frac{1}{2}\frac{1-\theta}{\frac{1}{2(\mu_B-\mu_A)}-\beta_o} \tag{59}$$

being μ_B the shear from the other material. Then, simplifying and ordering the equation

$$\frac{1-\theta}{2(\mu^*-\mu_A)} = \frac{1}{2(\mu_B-\mu_A)} - \theta\,\beta_o \tag{60}$$

Finally, replacing β_o for (30) and taking into account the inequalities from (33)

$$\frac{1-\theta}{2\,\mu^*_{LB} - 2\,\mu_A} \leqslant \frac{1}{2\,\mu_B - 2\,\mu_A} + \frac{\theta\,(\kappa_A + 2\,\mu_A)\,(N-1)}{\mu_A\,(N^2+N-2)\,\left(\kappa_A + 2\,\mu_A - \frac{2\,\mu_A}{N}\right)} \tag{61}$$

Similarly, the upper bound can be found by means of equation (44), replacing the parameter μ_o for μ_B, imposing the case $v_r = \theta$ and following the same procedure used for the lower bound. The result is provided below

$$\frac{\theta}{2\,\mu_B - 2\,\mu^*_{UB}} \leqslant \frac{1}{2\,\mu_B - 2\,\mu_A} + \frac{(1-\theta)\,(\kappa_B + 2\,\mu_B)\,(N-1)}{\mu_B\,(N^2+N-2)\,\left(\kappa_B + 2\,\mu_B - \frac{2\,\mu_B}{N}\right)} \tag{62}$$

On the other hand, the bulk equation (42) needs to be modified by changing the parameter κ_o for κ_A

$$\kappa^* - \kappa_A = \frac{A}{1+\alpha_o\,A} \tag{63}$$

Introducing (39), imposing the case $v_r = 1-\theta$ and expanding the denominator from the left hand side

$$(\kappa^* - \kappa_A)\left(1 + \alpha_o \frac{1-\theta}{\frac{1}{\kappa_B - \kappa_A} - \alpha_o}\right) = \frac{1-\theta}{\frac{1}{2(\kappa_B - \kappa_A)} - \alpha_o} \tag{64}$$

being κ_B the bulk from the other material. Then, simplifying and ordering the equation

$$\frac{1-\theta}{\kappa^* - \kappa_A} = \frac{1}{\kappa_B - \kappa_A} - \theta\,\alpha_o \tag{65}$$

Then, replacing α_o for (29) and taking into account the inequalities from (32)

$$\frac{1-\theta}{\kappa^*_{LB} - \kappa_A} \leqslant \frac{1}{\kappa_B - \kappa_A} + \frac{\theta}{\kappa_A + 2\,\mu_A - \frac{2\,\mu_A}{N}} \tag{66}$$

Finally, the bulk upper bound can be found by means of equation (42), replacing the parameter κ_o for κ_B, imposing the case $v_r = \theta$ and following the same procedure used for the lower bound. The result is provided below

$$\frac{\theta}{\kappa_B - \kappa^*_{UB}} \leqslant \frac{1}{\kappa_B - \kappa_A} + \frac{1-\theta}{\kappa_B + 2\,\mu_B - \frac{2\,\mu_B}{N}} \tag{67}$$

5 Swan code modified scripts

```
classdef MaterialInterpolation < handle

   properties (Access = protected)
        nstre
        ndim
        pdim
        nElem
        muFunc
        dmuFunc
        kappaFunc
        dkappaFunc
        matProp
        isoMaterial
    end

    methods (Access = public, Static)

        function obj = create(cParams)
            f   = MaterialInterpolationFactory;
            obj = f.create(cParams);
        end

    end

    methods (Access = public)

        function mp = computeMatProp(obj,rho)
            mu      = obj.muFunc(rho);
            kappa   = obj.kappaFunc(rho);
            dmu     = obj.dmuFunc(rho);
            dkappa  = obj.dkappaFunc(rho);
            mp.mu     = mu;
            mp.kappa  = kappa;
            mp.dmu    = dmu;
            mp.dkappa = dkappa;
        end

    end

    methods (Access = protected)

        function init(obj,cParams)
            obj.nElem = cParams.nElem;
            obj.pdim  = cParams.dim;
            obj.computeNDim(cParams);
            obj.saveYoungAndPoisson(cParams);
            obj.createIsotropicMaterial();
            obj.computeMuAndKappaIn0();
            obj.computeMuAndKappaIn1();
        end

        function computeNDim(obj,cParams)
            switch cParams.dim
                case '2D'
                  obj.ndim = 2;
                case '3D'
                  obj.ndim = 3;
            end
```

```
        end

        function saveYoungAndPoisson(obj,cParams)
            cP = cParams.constitutiveProperties;
            obj.matProp.rho1 = cP.rho_plus;
            obj.matProp.rho0 = cP.rho_minus;
            obj.matProp.E1   = cP.E_plus;
            obj.matProp.E0   = cP.E_minus;
            obj.matProp.nu1  = cP.nu_plus;
            obj.matProp.nu0  = cP.nu_minus;
        end

        function createIsotropicMaterial(obj)
            s.pdim  = obj.pdim;
            s.ptype = 'ELASTIC';
            obj.isoMaterial = Material.create(s);
        end

        function computeMuAndKappaIn0(obj)
            E0  = obj.matProp.E0;
            nu0 = obj.matProp.nu0;
            obj.matProp.mu0    = obj.computeMu(E0,nu0);
            obj.matProp.kappa0 = obj.computeKappa(E0,nu0);

        end

        function computeMuAndKappaIn1(obj)
            E1  = obj.matProp.E1;
            nu1 = obj.matProp.nu1;
            obj.matProp.mu1    = obj.computeMu(E1,nu1);
            obj.matProp.kappa1 = obj.computeKappa(E1,nu1);
        end

        function computeSymbolicInterpolationFunctions(obj)
            [muS,dmuS,kS,dkS] = obj.computeSymbolicMuKappa();
            obj.muFunc        = matlabFunction(muS);
            obj.dmuFunc       = matlabFunction(dmuS);
            obj.kappaFunc     = matlabFunction(kS);
            obj.dkappaFunc    = matlabFunction(dkS);
        end

        function [muS,dmuS,kS,dkS] = computeSymbolicMuKappa(obj)
            [muS,dmuS] = obj.computeMuSymbolicFunctionAndDerivative();
            [kS,dkS]   = obj.computeKappaSymbolicFunctionAndDerivative();
        end

        function mu = computeMu(obj,E,nu)
            mat = obj.isoMaterial;
            mu  = mat.computeMuFromYoungAndNu(E,nu);
        end

        function k = computeKappa(obj,E,nu)
            mat = obj.isoMaterial;
            k   = mat.computeKappaFromYoungAndNu(E,nu);
        end

        function computeNstre(obj)
            ndim      = obj.ndim;
            obj.nstre = 3*(ndim-1);
        end
```

```

    end

    methods (Access = protected, Abstract)
        computeMuSymbolicFunctionAndDerivative(obj)
        computeKappaSymbolicFunctionAndDerivative(obj)
    end

end
```

```
classdef MaterialInterpolationFactory < handle

    methods (Access = public, Static)

        function obj = create(cParams)
            switch cParams.typeOfMaterial
                case 'ISOTROPIC'
                    switch cParams.interpolation
                        case 'SIMPALL'
                            if ~isfield(cParams,'simpAllType')
                                cParams.simpAllType = 'EXPLICIT';
                            end
                            switch cParams.simpAllType
                                case 'EXPLICIT'
                                    obj = SimpAllInterpolationExplicit(cParams);
                                case 'IMPLICIT'
                                    switch cParams.dim
                                        case '2D'
                                            obj = SimpAllInterpolationImplicit2D(
    cParams);
                                        case '3D'
                                            obj = SimpAllInterpolationImplicit3D(
    cParams);
                                        otherwise
                                            error('Invalid problem dimension.');
                                    end
                                otherwise
                                    error('Invalid SimpAll type.');
                            end
                        case 'SIMP_Adaptative'
                            obj = SimpInterpolationAdaptative(cParams);
                        case 'SIMP_P3'
                            obj = SimpInterpolationP3(cParams);
                        otherwise
                            error('Invalid Material Interpolation method.');
                    end
                otherwise
                    error('Invalid type of material');
            end

        end

    end

end
```

```
classdef SimpAllInterpolationExplicit < MaterialInterpolation

    methods  (Access = public)

```

```
        function obj = SimpAllInterpolationExplicit(cParams)
            obj.init(cParams);
            obj.computeNstre();
            obj.computeSymbolicInterpolationFunctions();
        end

    end

    methods (Access = protected)

        function [mS,dmS] = computeMuSymbolicFunctionAndDerivative(obj)
            mS  = obj.computeSymMu();
            dmS = diff(mS);
        end

        function [kS,dkS] = computeKappaSymbolicFunctionAndDerivative(obj)
            kS  = obj.computeSymKappa();
            dkS = diff(kS);
        end

        function mu = computeSymMu(obj)
            m0   = obj.matProp.mu0;
            m1   = obj.matProp.mu1;
            k0   = obj.matProp.kappa0;
            k1   = obj.matProp.kappa1;
            eta0 = obj.computeEtaMu(m0,k0);
            eta1 = obj.computeEtaMu(m1,k1);
            c    = obj.computeCoeff(m0,m1,eta0,eta1);
            mu   = obj.computeRationalFunction(c);
        end

        function kappa = computeSymKappa(obj)
            m0    = obj.matProp.mu0;
            m1    = obj.matProp.mu1;
            k0    = obj.matProp.kappa0;
            k1    = obj.matProp.kappa1;
            eta0  = obj.computeEtaKappa(m0);
            eta1  = obj.computeEtaKappa(m1);
            c     = obj.computeCoeff(k0,k1,eta0,eta1);
            kappa = obj.computeRationalFunction(c);
        end

        function etaMu = computeEtaMu(obj,mu,kappa)
            N   = obj.ndim;
            num = -mu*(4*mu - kappa*N^2 - 2*mu*N^2 + 2*mu*N);
            den = 2*N*(kappa + 2*mu);
            etaMu = num/den;
        end

        function etaKappa = computeEtaKappa(obj,mu)
            N   = obj.ndim;
            num = 2*mu*(N-1);
            den = N;
            etaKappa = num/den;
        end

    end

    methods (Access = protected, Static)

```

```
        function c = computeCoeff(f0,f1,eta0,eta1)
            c.n01 =  -(f1 - f0)*(eta1 - eta0);
            c.n0  =  f0*(f1 + eta0);
            c.n1  =  f1*(f0 + eta1);
            c.d0  =  (f1 + eta0);
            c.d1  =  (f0 + eta1);
        end

        function f = computeRationalFunction(s)
            rho = sym('rho','positive');
            n01 = s.n01;
            n0  = s.n0;
            n1  = s.n1;
            d0  = s.d0;
            d1  = s.d1;
            num = n01*(1-rho)*(rho) + n0*(1-rho) + n1*rho;
            den = d0*(1-rho) + d1*rho;
            f   = num/den;
        end

    end

end
```

```
classdef SimpAllInterpolationImplicit < MaterialInterpolation

    properties (Access = protected)
       dmu0
       dmu1
       dk0
       dk1
    end

    methods (Access = protected)

        function [mS,dmS] = computeMuSymbolicFunctionAndDerivative(obj)
            s.f0      = obj.matProp.mu0;
            s.f1      = obj.matProp.mu1;
            s.df0     = obj.dmu0;
            s.df1     = obj.dmu1;
            [mS,dmS] = obj.computeParameterInterpolationAndDerivative(s);
        end

        function [kS,dkS] = computeKappaSymbolicFunctionAndDerivative(obj)
            s.f0      = obj.matProp.kappa0;
            s.f1      = obj.matProp.kappa1;
            s.df0     = obj.dk0;
            s.df1     = obj.dk1;
            [kS,dkS] = obj.computeParameterInterpolationAndDerivative(s);
        end

        function [f,df] = computeParameterInterpolationAndDerivative(obj,s)
            c      = obj.computeCoefficients(s);
            rho    = sym('rho','positive');
            fSym   = obj.rationalFunction(c,rho);
            dfSym  = obj.rationalFunctionDerivative(c,rho);
            f      = simplify(fSym);
            df     = simplify(dfSym);
        end

```

```
        function c = computeCoefficients(obj,s)
            f1     = s.f1;
            f0     = s.f0;
            df1    = s.df1;
            df0    = s.df0;
            c1     = sym('c1','real');
            c2     = sym('c2','real');
            c3     = sym('c3','real');
            c4     = sym('c4','real');
            coef   = [c1 c2 c3 c4];
            r1     = obj.matProp.rho1;
            r0     = obj.matProp.rho0;
            eq(1)  = obj.rationalFunction(coef,r1) - f1;
            eq(2)  = obj.rationalFunction(coef,r0) - f0;
            eq(3)  = obj.rationalFunctionDerivative(coef,r1) - df1;
            eq(4)  = obj.rationalFunctionDerivative(coef,r0) - df0;
            c      = solve(eq,[c1,c2,c3,c4]);
            c      = struct2cell(c);
            c      = [c{:}];
        end

    end

    methods (Access = protected, Static)

        function r = rationalFunction(coef,rho)
            c1   = coef(1);
            c2   = coef(2);
            c3   = coef(3);
            c4   = coef(4);
            num = (c1*rho^2 + c2*rho + 1);
            den = (c4 + rho*c3);
            r    = num/den;
        end

        function dr = rationalFunctionDerivative(coef,rho)
            c1   = coef(1);
            c2   = coef(2);
            c3   = coef(3);
            c4   = coef(4);
            n1   = c2 + 2*rho*c1;
            d1   = c4 + rho*c3;
            dr1  = n1/d1;
            n2   = -c3*(c1*rho^2 + c2*rho + 1);
            d2   = (c4 + rho*c3)^2;
            dr2  = n2/d2;
            dr   = dr1 + dr2;
        end

    end

end
```

```
classdef SimpAllInterpolationImplicit2D < SimpAllInterpolationImplicit

   methods  (Access = public)

        function obj = SimpAllInterpolationImplicit2D(cParams)
            obj.init(cParams);
            obj.computeNstre();
```

```
            obj.computeSymbolicInterpolationFunctions();
            obj.dmu0 = obj.computeDmu0();
            obj.dmu1 = obj.computeDmu1();
            obj.dk0  = obj.computeDKappa0();
            obj.dk1  = obj.computeDKappa1();
        end

    end

     methods  (Access = protected)

        function [pMu,pKappa] = computePolarizationTensorAsMuKappa(eMatrix,eInclusion,nuMatrix,nuInclusion)
            coef = obj.compute2Dcoefficients(eMatrix,eInclusion,nuMatrix,nuInclusion);
            [p1,p2] = obj.computeP1P2(coef);
            [pMu,pKappa] = obj.computePkappaPmu(p1, p2);
        end

        function dmu0 = computeDmu0(obj)
            E1  = obj.matProp.E1;
            E0  = obj.matProp.E0;
            nu1 = obj.matProp.nu1;
            nu0 = obj.matProp.nu0;
            mu0 = obj.matProp.mu0;
            [pMu0,~] = obj.computePolarizationTensorAsMuKappa(E0,E1,nu0,nu1);
            dmu0     = -mu0*pMu0;
        end

        function dmu1 = computeDmu1(obj)
            E1  = obj.matProp.E1;
            E0  = obj.matProp.E0;
            nu1 = obj.matProp.nu1;
            nu0 = obj.matProp.nu0;
            mu1 = obj.matProp.mu1;
            [pMu1,~] = obj.computePolarizationTensorAsMuKappa(E1,E0,nu1,nu0);
            dmu1     = mu1*pMu1;
        end

        function dkappa0 = computeDKappa0(obj)
            E1     = obj.matProp.E1;
            E0     = obj.matProp.E0;
            nu1    = obj.matProp.nu1;
            nu0    = obj.matProp.nu0;
            kappa0 = obj.matProp.kappa0;
            [~,pKappa0] = obj.computePolarizationTensorAsMuKappa(E0,E1,nu0,nu1);
            dkappa0     = -kappa0*pKappa0;
        end

        function dkappa1 = computeDKappa1(obj)
            E1     = obj.matProp.E1;
            E0     = obj.matProp.E0;
            nu1    = obj.matProp.nu1;
            nu0    = obj.matProp.nu0;
            kappa1 = obj.matProp.kappa1;
            [~,pKappa1] = obj.computePolarizationTensorAsMuKappa(E1,E0,nu1,nu0);
            dkappa1     = kappa1*pKappa1;
        end

     end
```

```
    methods  (Access = protected, Static)

        function coef = compute2Dcoefficients(eMatrix,eInclusion,nuMatrix,
    nuInclusion)
            coef.a    = (1 + nuMatrix)/(1 - nuMatrix);
            coef.b    = (3 - nuMatrix)/(1 + nuMatrix);
            coef.gam  = eInclusion/eMatrix;
            coef.tau1 = (1 + nuInclusion)/(1 + nuMatrix);
            coef.tau2 = (1 - nuInclusion)/(1 - nuMatrix);
            coef.tau3 = (nuInclusion*(3*nuMatrix - 4) + 1)/(nuMatrix*(3*nuMatrix -
     4) + 1);
        end

        function [p1,p2] = computeP(s)
            a    = s.a;
            b    = s.b;
            gam  = s.gam;
            tau1 = s.tau1;
            tau2 = s.tau2;
            tau3 = s.tau3;
            p1   = 1/(b*gam+tau1)*(1+b)*(tau1-gam);
            p2   = 0.5*(a-b)/(b*gam+tau1)*(gam*(gam-2*tau3)+tau1*tau2)/(a*gam+tau2
    );
        end

        function [pMu,pKappa] = computePKappaMu(p1, p2)
            pMu    = p1;
            pKappa = 2*p2 + p1;
        end

    end

end
```

```
classdef SimpAllInterpolationImplicit3D < SimpAllInterpolationImplicit

   methods  (Access = public)

        function obj = SimpAllInterpolationImplicit3D(cParams)
            obj.init(cParams);
            obj.computeNstre();
            obj.dmu0 = obj.computeDmu0();
            obj.dmu1 = obj.computeDmu1();
            obj.dk0  = obj.computeDKappa0();
            obj.dk1  = obj.computeDKappa1();
            obj.computeSymbolicInterpolationFunctions();
        end

   end

    methods  (Access = protected)

        function [dMu,dKappa] = computePolarizationParametersAsMuKappa(obj,
    eMatrix,eInclusion,nuMatrix,nuInclusion)
            [m1,m2] = obj.compute3Dcoefficients(eMatrix,eInclusion,nuMatrix,
    nuInclusion);
            [dMu,dKappa] = obj.computeDkappaDmu(m1,m2);
        end

```

```
        function dmu0 = computeDmu0(obj)
            E1  = obj.matProp.E1;
            E0  = obj.matProp.E0;
            nu1 = obj.matProp.nu1;
            nu0 = obj.matProp.nu0;
            mu0 = obj.matProp.mu0;
            mu1 = obj.matProp.mu1;
            [dMu0,~] = obj.computePolarizationParametersAsMuKappa(E0,E1,nu0,nu1);
            qmu0 = dMu0/(mu0*(mu1-mu0));
            dmu0 = mu0*(mu1-mu0)*qmu0;
        end

        function dmu1 = computeDmu1(obj)
            E1  = obj.matProp.E1;
            E0  = obj.matProp.E0;
            nu1 = obj.matProp.nu1;
            nu0 = obj.matProp.nu0;
            mu0 = obj.matProp.mu0;
            mu1 = obj.matProp.mu1;
            [dMu1,~] = obj.computePolarizationParametersAsMuKappa(E1,E0,nu1,nu0);
            qmu1 = dMu1/(mu1*(mu0-mu1));
            dmu1 = mu1*(mu1-mu0)*qmu1;
        end

        function dkappa0 = computeDKappa0(obj)
            E1     = obj.matProp.E1;
            E0     = obj.matProp.E0;
            nu1    = obj.matProp.nu1;
            nu0    = obj.matProp.nu0;
            kappa0 = obj.matProp.kappa0;
            kappa1 = obj.matProp.kappa1;
            [~,dKappa0] = obj.computePolarizationParametersAsMuKappa(E0,E1,nu0,nu1);
            qKappa0 = dKappa0/(kappa0*(kappa1-kappa0));
            dkappa0 = kappa0*(kappa1-kappa0)*qKappa0;
        end

        function dkappa1 = computeDKappa1(obj)
            E1     = obj.matProp.E1;
            E0     = obj.matProp.E0;
            nu1    = obj.matProp.nu1;
            nu0    = obj.matProp.nu0;
            kappa0 = obj.matProp.kappa0;
            kappa1 = obj.matProp.kappa1;
            [~,dKappa1] = obj.computePolarizationParametersAsMuKappa(E1,E0,nu1,nu0);
            qKappa1 = dKappa1/(kappa1*(kappa0-kappa1));
            dkappa1 = kappa1*(kappa1-kappa0)*qKappa1;
        end

    end

    methods (Access = protected, Static)

        function [m1,m2] = compute3Dcoefficients(eMatrix,eInclusion,nuMatrix,nuInclusion)
            mu      = eMatrix/(2*(1+nuMatrix));
            muI     = eInclusion/(2*(1+nuInclusion));
            mu2     = mu - muI;
            lambda  = eMatrix*nuMatrix/((1+nuMatrix)*(1-2*nuMatrix));
```

```
            lambdaI  = eInclusion*nuInclusion/((1+nuInclusion)*(1-2*nuInclusion));
            lam2     = lambda - lambdaI;
            m1n      = 15*mu*mu2*(nuMatrix-1);
            m1d      = 15*mu*(1-nuMatrix)+2*mu2*(5*nuMatrix-4);
            m1       = m1n/m1d;
            m2n      = lam2*(15*mu*lambda*(1-nuMatrix) + 2*lambda*mu2*(5*nuMatrix
    -4)) - 2*mu2*(lambda*mu2-5*mu*nuMatrix*lam2);
            m2d      = 5*mu2*(3*mu*lambda*(1-nuMatrix)-3*mu*nuMatrix*lam2-lambda*
    mu2*(1-2*nuMatrix));
            m2       = m2n/m2d;
        end

        function [dMu,dKappa] = computeDkappaDmu(m1,m2)
            dMu    = m1;
            dKappa = m1*m2+2/3*m1;
        end

    end

end
```

References

[1] Shtrikman S Hashin Z. A variational approach to the theory of the elastic behaviour of multiphase materials. J Mech Phys Solids., (1st edition) edition, 1963, Vol. 11, pp. 127to 140.