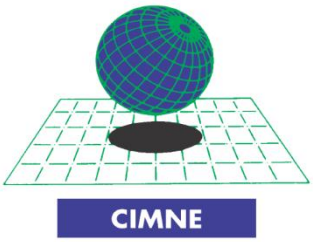


MASTER OF SCIENCE IN COMPUTATIONAL MECHANICS

TECHNICAL UNIVERSITY OF CATALONIA

AN EMBEDDED METHOD FOR CFD

PRESENTED BY:

JOSÉ ENMANUEL AMAYA ARAUJO

SUPERVISORS:

POOYAN DADVAND

RICCARDO ROSSI

JULY 2014

BARCELONA, SPAIN

ABSTRACT

This document represents the final dissertation for the Master of Science in Computational Mechanics, a program organized by the International Center for Numerical Methods in Engineering (CIMNE), and it describes the definition and implementation of an embedded method for Computational Fluid Dynamics (CFD) applications, specifically problems with strong discontinuities, where a fluid in motion interacts with the surface of a solid body.

The proposed method takes advantage of ideas taken from the Level-set Method, the Static Condensation Method, and on to a certain degree from the Extended Finite Element Method to avoid calculating two local systems in all the elements cut by the interface, and instead calculates a single system that takes into consideration the division. In this way, the existence of the interface can be omitted in the meshing process and the assembly process, and the procedure gets considerably simplified. After the method is defined, it gets implemented and incorporated into the Kratos framework, as a new type of element that performs the required procedure, modifying all the necessary files in the framework to make it compatible to the existing modules.

Finally, an initial test example is designed and executed for debugging purposes, to find and correct method definition errors; and a second more complex test to further validate the correct functioning of the method and the implementation.

ACKNOWLEDGMENTS

First of all, I want to thank and dedicate this work to my mother, Maria Elena Araujo, for always being there for me and support me on my decisions and my enterprises, even when she was not in favor of some of them; and my grandmother, Bertha Marina Araujo, may she rest in peace, for also taking care of me since I can remember and for being a great motivation for my career.

I want to thank all of my friends for their support, their trust put in me and for accompanying me during the entire process (some in person and others through the different social media possibilities). I want to explicitly thank my friends from El Salvador: Jorge López, Carlos Melgar, Jesús Peña, Enrique Hernández, Raúl Chinchilla, Herbert Amaya, Claudia Hernández, Jorge Martínez and Bayron Menesses; and my friends from around the world: Tomás Valldeperas, Wenjun Tan, Haiqin Huang, Veronica Manfredini, Stella Mondoví, Foued Ziani, Fırat Ağar, Madara Auzenbaha, Rıza Tarımoğlu and Giulia Giorgini.

I want to thank Prof. Mauricio Pohl and Prof. Guillermo Cortés from El Salvador for their support, advice and help, and for giving me the opportunity to pursue an academic career. Many special thanks to Prof. Kenneth Morgan, Prof. Djordje Peric and Prof. Antonio Gil from Swansea University, and Prof. Riccardo Rossi, Prof. Pooyan Dadvand and Prof. Yongxing Shen from Univertat Politecnica de Catalunya (UPC), for all their valuable teachings, advice and suggestions that made it possible for me to reach this point and to finish this work.

And last but not least, I want to express my thanks to the International Center for Numerical Methods in Engineering (CIMNE) for giving me the opportunity of being part of this program, where I have improved so much, both in my professional career and in my personal life.

CONTENTS

FIGURE INDEX

ABBREVIATIONS

1D, 2D, 3D:
One dimension, two dimensions or three dimensions, usually used to describe the type of problem that is being handled, discussed or described.

BC:
Boundary condition. The constraints imposed on a problem for it to be solvable based on the requirements specified.

DOF:
Degree of freedom. Independent values in a system that can be prescribed or that need to be calculated.

FEM:
Finite Element Method. Procedure in which a domain is subdivided in many small subdomains interconnected by nodes, used to discretize a continuous problem and simplify it to obtain an approximate solution.

ID:
Identifier. Normally used to refer to the sequence of numbers that describe the totality of nodes and/or the totality of elements in a mesh.

LS:
Level Set. Method used to track an interface and its evolution in a function.

PDE:
Partial Differential Equation.

WRM:
Weighted Residual Method. Procedure that multiplies each evaluation in a system by a weight and then averages the result obtaining a good approximate solution.

XFEM:
eXtended Finite Element Method. Modified method that allows the handling of strong discontinuities or voids in a domain by enriching the involved elements with a particular enrichment function.

SYMBOLS AND NOTATION

Δ	:	Laplacian operator.
$\frac{d}{dx}$	:	Leibniz derivative notation.
Γ_D	:	Dirichlet boundary.
Γ_N	:	Neumann boundary.
N	:	Shape functions.
a	:	Additional degrees of freedom used in XFEM.
u	:	System unknown. For the present document, it represents velocity.
K	:	Stiffness matrix of a system.
f	:	Force vector of a system.
w	:	Set of weight used in the WRM.
Φ	:	Level-set function.
ϕ	:	Enrichment function.

PROLOGUE

The present document explains the theory consulted for the implementation of a new approach in the solution of problems in the field of Computational Mechanics characterized by the presence of a solid material interacting with a fluid in motion, where the implementations is in the form of a new element inside the Kratos framework that performs the desired approach. The document also presents the procedures and criteria used to include this new element inside Kratos.

Section 1 presents the context for the document, commenting briefly on the purpose of the implementation and the present document.

Section 2 presents the mathematical tools and methods related to the work done during the project and that are the base for the design of the new approach and element inside the Kratos framework.

Sections 3 and 4 present the goals that the project wants to reach and the computational tools used to make it a reality.

Section 5 presents in detail all the specifications and mechanisms that define the new approach. It also gives context about the Kratos framework, in order to understand how the integration of the element was performed.

Section 6 presents the results of the test examples designed and employed to validate the implementation of the new element.

Section 7 presents some conclusions that could be extracted for the implementation experiences and some recommendations that can serve as ideas to guide future work on this subject.

Section 8 presents a comprehensive glossary of technical terms found throughout the document.

Finally, section 9 presents the bibliographical references consulted during all the development process.

1. INTRODUCTION

The present document constitutes the final dissertation for the Master of Science in Computational Mechanics, organized by the International Center for Numerical Methods in Engineering (CIMNE), and presents the mathematical background for the design and implementation of a proposed embedded method for solving problems involving strong discontinuities taking mainly advantage of techniques such as the Level-set method and the static condensation method.

Fluid-Structure interaction presents a subfield of computational mechanics where implementations for solving problems tend to be quite complex. This subfield is related, but not limited to, cases where there is a solid material inside a fluid in motion or fluid going over a certain surface inside a domain.

When problems of this type are handled with methods that require meshes, there are always a set of elements that gets divided by the line that separates the different materials in the domain, called the Interface. To work with this cut elements can become a tedious task, so since many years ago, research on how to make this kind of work more efficient has been a top priority for scientists.

Most of the current existing methods, modifications and alternatives for solving this type of problems (and many other types as well) are implemented in several software tools, both commercial and open source. One of these tools is the Kratos framework, developed by QUANTECH, a branch of CIMNE.

Kratos provides a software development environment specialized in the field of Computational Mechanics that facilitates to a high degree research processes, method testing and benchmarking, and simulations. Kratos is always being updated to keep up with the state of the art, incorporating new method and new variants of methods into its arsenal.

For all of these reasons, the main goal of the project described in this document is to design a new approach for solving problems that involve a fluid in interaction with the surface of a structure that simplifies considerably the work done, and to implement this approach as a new available alternative inside the Kratos framework, so other developers and researchers can have access to it.

2. MATHEMATICAL BACKGROUND

2.1 The Finite Element Method

The Finite Element Method (FEM) constitutes a procedure used to solve in an approximate way Partial Differential Equations (PDEs). For example, taking into consideration the following simple PDE in 1D to illustrate the method:

$$-\Delta u = f$$

$$-\frac{d}{dx}\left(\frac{d}{dx}u\right) = f$$

Subject to the following boundary conditions:

$$u = u_D \; on \; \Gamma_D; \qquad \frac{d}{dx}u = u_N \; on \; \Gamma_N$$

Where:

- u is the unknown variable that needs to be approximated.
- f is the source term.
- u_D is a prescribed value of the solution.
- Γ_D is the Dirichlet domain of the problem.
- u_N is a prescribed flux.
- Γ_N is the Neumann domain of the problem.

As a requirement for the method, the domain is discretized into a mesh of elements connected by nodes, where the amount of elements and nodes are finite as seen in Figure 1.

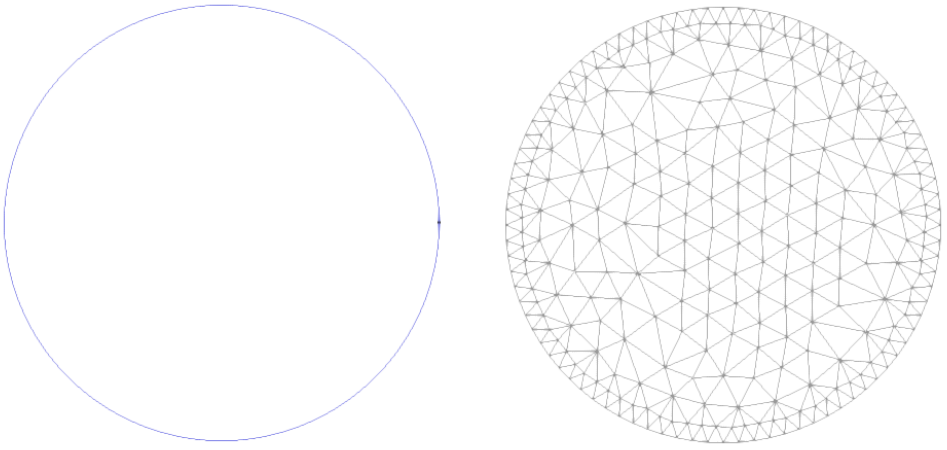


Figure 1 - Domain meshing

Each piece of the mesh is an element, defined by a certain number of nodes: in Figure 1 the elements are triangular and therefore they are defined by a set of three nodes each. There are several more types of elements, for working in 1D, 2D or 3D.

The method turns the solution process into a modular process, where the unknown variable will be calculated at each node of each element, working with the elements separately and then assembling the results into a global solution. In general, the unknown variable needs to be calculated only for the nodes in the mesh (not anymore for the entire continuous of the domain).

The method establishes that the unknown variable is to be interpolated making use of a set of shape functions (from this point forward, the calculation will be done element-wise, also called a $local\ solution$):

$$\hat{u} = u_1 N_1 + u_2 N_2 + \cdots + u_n N_n = \sum_{i=1}^{n} u_i N_i$$

Where:

- $\hat{u}$ is the approximate solution.
- n is the number of nodes in an element (depends on the type of element and the number of dimensions of the problem).
- N_i are the shape functions.
- u_i are the nodal solution values.

The shape functions are defined in such a way that the *i-th* shape function N_i has a value of 1 on the node i and a value of 0 in all the other nodes. They also have to fulfill the Partition of Unity condition:

$$\sum N = 1$$

These functions can be defines in a variety of ways (linear, quadratic, etc.), and for working in 1D, 2D or 3D.

Now, substituting the unknown variable by the approximate solution, the PDE takes the form:

$$-\frac{d}{dx}\left(\frac{d}{dx}\hat{u}\right) = f$$

As the next step, a Weighted Residual approach is implemented on the PDE, multiplying each side of the equation by a weight w and integrating throughout all the element domain:

$$-\int w \frac{d}{dx}\left(\frac{d}{dx}\hat{u}\right) dx = \int w f dx$$

This set of functions w are called test functions, one for each node in the element, and they have to fulfill the following requirements:

- All the functions w must vanish on the boundary.
- The must fulfill the Partition of Unity:

$$\sum w = 1$$

There are several methods to define an appropriate set of test functions, the one that will be discussed is called the Galerkin method. This method takes advantage of the properties of the shape functions used to interpolate the approximate solution, and makes the observation that they already fulfill the requirements of the test functions.

With this, the PDE takes the form:

$$-\int N \frac{d}{dx}\left(\frac{d}{dx}\hat{u}\right) dx = \int N f dx$$

Where N is the set of shape functions being used as test functions. This expression is called the Strong Form of the problem.

To avoid working with a second derivative, the expression can be submitted to an integration by parts, obtaining what is called the Weak Form of the problem:

$$-N\frac{d}{dx}\hat{u}\Big|_{\Gamma_N} + \int \frac{d}{dx} N \frac{d}{dx}\hat{u}\, dx = \int N f dx$$

Making some modifications, and switching to an Einstein notation:

$$\left(\int \frac{d}{dx} N_i \frac{d}{dx} N_j dx\right) u = \int N_i f dx + N\frac{d}{dx}\hat{u}\Big|_{\Gamma_N}$$

In here:

- The approximate solution $\hat{u}$ has been substituted by its equivalent in terms of the shape functions and the nodal solution values (which are independent of the integration variable).
- The term $N\frac{d}{dx}\hat{u}\Big|_{\Gamma_N}$ represents the Neumann boundary condition values that need to be implemented (notice how they appear naturally as part of the method).

The Weak Form can be expressed in a rather simple form using matrix notation, defining:

$$K_{ij} = \int \frac{d}{dx} N_i \frac{d}{dx} N_j dx$$

$$f_i' = \int N_i f dx$$

$$f_{\Gamma_N} = N\frac{d}{dx}\hat{u}\Big|_{\Gamma_N}$$

$$f = f' + f_{\Gamma_N}$$

Where:

- K is called the stiffness matrix.
- f' is the source vector.
- f_{Γ_N} is the Neumann BCs vector.
- f is the total "force vector".

So the problem is simply expressed as:

$$Ku = f$$

As the final step, all the local contributions (the system calculated for each element) are assembled into a global system:

$$K_T = \sum K$$

$$u_T = \sum u$$

$$f_T = \sum f$$

This assembly process adds up each elemental value for a single node into a single member of the global stiffness matrix, the same can be said for the right hand side vector. The global vector u contains all the nodes in the mesh. The global system, the one that is resolved, takes the simple form:

$$K_T u_T = f_T$$

To implement the Dirichlet BCs, the one detail of the problem that remains pending, there are several approaches, the one relevant to the present document is commented on the next section.

2.2 Lagrange Multipliers

The Dirichlet boundary conditions can be considered as constraint relations, since the boundary is required to follow a prescribed value. These types of conditions are usually difficult to impose on a problem, in contrast to the Neumann conditions (also called natural conditions for this same behavior).

For this reason, the method of Lagrange Multipliers is used as it provides great help in a vast variety of cases, all Dirichlet type boundary conditions can be imposed through the use of Lagrange multipliers.

To illustrate the method, an example problem with a global matrix system of 5x5 will be used:

$$Ku = f$$

Subject to:

$$u_1 = 0; \quad u_2 = u_D; \quad u_4 = u_5$$

These conditions can be written as a system:

$$Bu = h$$

Which has the following description:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{pmatrix} = \begin{pmatrix} 0 \\ u_D \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The matrix B is called the boundary condition matrix. At this point, a set of values called the Lagrange Multipliers are introduced into the problems as follows:

$$\lambda^T (Bu - h) = 0$$

This can be interpreted as the energy that is required by the system to enforce and maintain the boundary conditions. Now, looking at the invariant functional of the original problem:

$$I = (Ku - f)^T \delta x = 0$$

$$I = \delta (u^T Ku - u^T f) = 0$$

From this, the stationary condition is:

$$I = u^T Ku - u^T f$$

Adding the energy term from the boundary conditions:

$$J = u^T Ku - u^T f + \lambda^T (Bu - h)$$

This last expression represents the correct energy in which the total energy of the original system is modified by the boundary conditions imposition. Taking the minimum of this expression with respect to u and λ will produce the solution subject to the boundary conditions:

$$\delta J = \frac{\delta J}{\delta u} du + \frac{\delta J}{\delta \lambda} d\lambda = 0$$

Since du and $d\lambda$ are arbitray:

$$\frac{\delta J}{\delta u} = 0; \quad \frac{\delta J}{\delta \lambda} = 0$$

Producing the final system:

$$\begin{bmatrix} K & B^T \\ B & 0 \end{bmatrix} \begin{pmatrix} u \\ \lambda \end{pmatrix} = \begin{pmatrix} f \\ h \end{pmatrix}$$

The solution of this system, provides the values for the Lagrange multipliers (in the example three values λ_1, λ_2 and λ_3), and also the solution of the original system. The multipliers only serve to impose the boundary conditions and after the resolution of the system has been performed they can be discarded.

As it can be appreciated from the previous exposition, the method of the Lagrange multipliers can provide a high degree of flexibility and scope to impose Dirichlet type boundary conditions, but the following drawbacks can be mentioned:

- The order of the system is increased, and more calculations are required.
- The resulting matrix may lose its strongly banded structure.
- Zero values can appear on the main diagonal and this can affect the solution procedure.

2.3 Level-set Method

Given a function which changes through time, the Level-set method is used to track the evolution of its interface by building the original interface into a surface that intersects the XY plane exactly where the function is located (as seen in Figure 2).

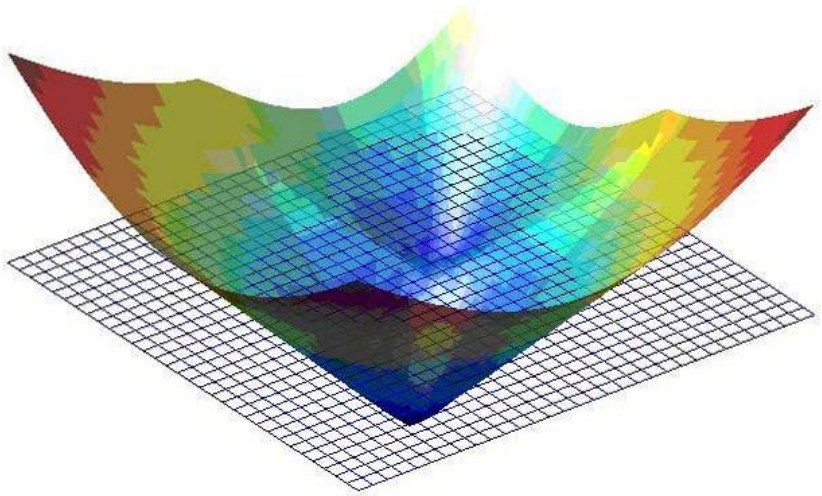


Figure 2 - Level-set Method

The Level-set method is used to describe material interfaces in problems that involve two or more different kinds of materials, voids or holes in problem domains, free-surfaces, etc. Its definition depends exclusively on the type of problem that needs to be solved, the information that is needed to take into consideration on the interface, and the type of operations to be performed on the domain using the information from the interface.

For example, take a look at Figure 3, where the Level-set function can be defined as:

$$\Phi(x) = \begin{cases} > 0 & if\ x \in \Omega_1 \\ < 0 & if\ x \in \Omega_2 \\ 0 & if\ x \in \Gamma \end{cases}$$

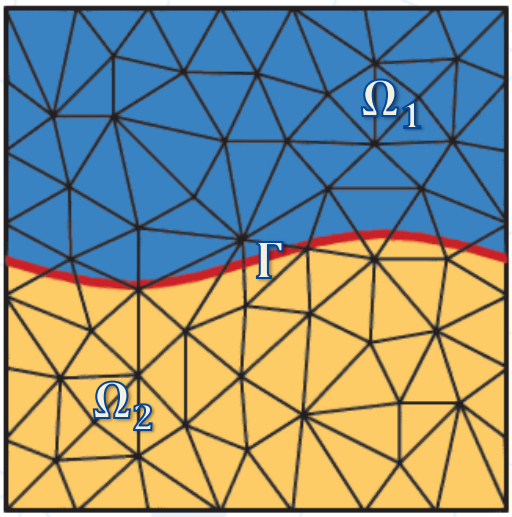


Figure 3 - Divided domain

A level-set function is usually given by its nodal values in the mesh, and if values are needed where there are no nodes, interpolations techniques are implemented. A general expression for the level-set function can be expressed as:

$$\Phi(x) = \sum_{i \in I} N_i(x)\Phi_i$$

Where N_i are a set of shape functions, that can be the same functions used in the Finite Element Method and that have to fulfill the same requirements as them.

Normally, if the requirements for the interface are not complex, a simple sign function can be used for its definition:

$$\Phi(x) = \begin{cases} d & if\ x \in \Omega_1 \\ -d & if\ x \in \Omega_2 \end{cases}$$

Where:

$$d = \min_{\tilde{x} \in \Gamma} \|x - \tilde{x}\|$$

And obviously, the interface Γ would be located at and defined by all the points with a value of $\Phi(x) = 0$.

Formally, the level-set equation is defined as:

$$\frac{D\Phi(x,t)}{Dt} = 0$$

$$\frac{\delta\Phi(x,t)}{\delta x} + v(x,t) \cdot \nabla\Phi(x,t) = 0$$

Where:

- t is the time.
- v is the velocity at which the interface Γ moves in a normal direction.

From this, the normal unit vector can be defined as:

$$n = \frac{\nabla\Phi}{\|\nabla\Phi\|}$$

And the curvature as:

$$\kappa = \nabla \cdot \frac{\nabla\Phi}{\|\nabla\Phi\|}$$

Observe that if the level-set function is a signed distance function, then:

$$\|\nabla\Phi\| = 1$$

2.4 Enriched Finite Element Method

The Enriched Finite Element Method, also called Extended Finite Element Method (XFEM), is a modification to the original Finite Element Method, so that it can additionally:

- Be able to reproduce discontinuities.
- Integrate discontinuous elements.
- Use enrichment functions.
- Allow a variable number of degrees of freedom per element.

It can be implied that the XFEM method is necessary and utilized in problems where an interface is present that defines the discontinuity, which is something not supported by the original method.

Under this method, the expression for the approximated solution in the Finite Element Method now looks like:

$$\hat{u}^h = \sum_{i\in\Omega}^{n} N_i(x)u_i + \sum_{i\in\Omega_{enr}}^{n_{enr}} \phi(x)\tilde{N}_i(x)a_i = u^{std} + u^{enr}$$

The first term corresponds to the normal Finite Element procedure; it presents no changes from what was already explained back in Section 2.1.

The second term exists exclusively for the nodes that need to be enriched, which are usually the nodes influenced by the presence of the interface. This term makes use of:

- An enrichment function $\phi(x)$ that describes the information used at the enriched nodes, basically what is needed to handle the discontinuity.
- A set of shape functions that need to form a partition of unity. This set of shape of functions does not need to be the same as used by the FEM section of the definition.
- These two terms put together can be defines as the XFEM shape functions to form a mirrored term from the FEM one:

$$\hat{u}^h = \sum_{i \in \Omega}^{n} N_i(x) u_i + \sum_{i \in \Omega_{enr}}^{n_{enr}} \hat{N}_i(x) a_i \,; \qquad \hat{N}_i(x) = \phi(x) \tilde{N}_i(x)$$

- a_i are the additional degrees of freedom to be considered at the elements that are cut by the interface. Usually some of these new DOFs are located at the points on the side of the elements where the interface cuts. As seen in Figure 4.

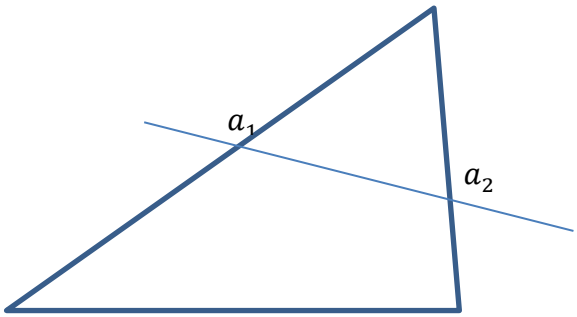


Figure 4 - Additional DOFs

The additional degrees of freedom can also be Gauss points defined in the new elements that will be constructed due to the cut, as seen in Figure 5.

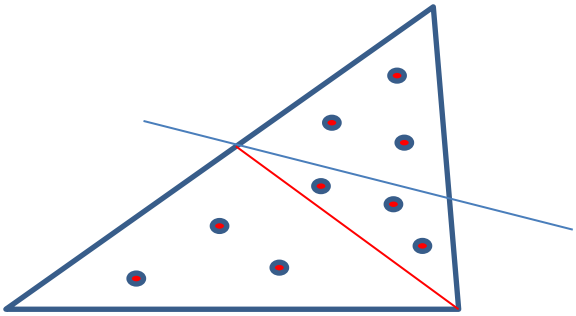


Figure 5 - Gauss points as additional DOFs

So, when defining the vector u of the local system, it will look like (assuming triangular elements and 3 additional degrees of freedom):

$$u^h = [N_1 \quad N_2 \quad N_3 \quad \widehat{N_1} \quad \widehat{N_2} \quad \widehat{N_3}] \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The extra slots are equal to 0 in elements that are not enriched.
The computation of these elements is quite similar to the standard finite element method, the local stiffness matrix will take the form:

$$K = \begin{bmatrix} \int \frac{d}{dx} N_i \frac{d}{dx} N_j dx & \int \frac{d}{dx} N_i \frac{d}{dx} \widehat{N_j} dx \\ \int \frac{d}{dx} \widehat{N_i} \frac{d}{dx} N_j dx & \int \frac{d}{dx} \widehat{N_i} \frac{d}{dx} \widehat{N_j} dx \end{bmatrix}$$

And taking special consideration to the fact that the derivative of the XFEM shape functions has to be calculated as:

$$\frac{d\widehat{N_i}}{dx_k} = \phi \frac{d\widehat{N_i}}{dx_k} + \frac{d\phi}{dx_k} \widehat{N_i}$$

It is extremely frequent to use a Level-set function as the enrichment function in a XFEM implementation, a couple of examples can be mentioned:

a) To handle strong discontinuities

The enrichment function is defined as:

$$\phi(x) = \begin{cases} 1 & if\ x \in \Omega_1 \\ -1 & if\ x \in \Omega_2 \end{cases}$$

Or

$$\phi(x) = \begin{cases} 1 & if\ x \in \Omega_1 \\ 0 & if\ x \in \Omega_2 \end{cases}$$

Where Ω_1 and Ω_2 are the two subdomains defined by the presence on the interface in the problem.

b) To handle voids

The enrichment function is defined as:

$$\phi(x) = \begin{cases} 1 & if\ x \in \Omega \\ 0 & if\ x \notin \Omega \end{cases}$$

This can be interpreted as leaving out of the calculation the part of the domain that is the void/hole.

2.5 Static condensation

The Static condensation procedure is a method that allows transforming a system of equation in matrix form into a simpler but equivalent system. To illustrate the method, let's take into consideration the next example:

$$Ku = f$$

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

If the previous system needs to be simplified, some of the variables can be transformed from explicit to implicit, that is to say, they can be absorbed by the rest. For example, assuming u_3 wants to be turned implicit, the system gets divided as follows:

$$\left[\begin{array}{cc|c} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ \hline K_{31} & K_{32} & K_{33} \end{array}\right] \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{pmatrix} u' \\ b \end{pmatrix} = \begin{pmatrix} g \\ h \end{pmatrix}$$

Where:

$$A = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} K_{13} \\ K_{23} \end{bmatrix}$$

$$C = \begin{bmatrix} K_{31} & K_{32} \end{bmatrix}$$

$$D = [K_{33}]$$

$$u' = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$

$$b = (u_3)$$

$$g = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

$$h = (f_3)$$

Expressing the two equation of the new system:

$$\begin{cases} Au' + Bb = g \\ Cu' + Db = h \end{cases}$$

Solving for b in the second equation:

$$Db = h - Cu'$$

$$b = D^{-1}h - D^{-1}Cu'$$

Plugging this value into the first equation:

$$Au' + Bb = Au' + B(D^{-1}h - D^{-1}Cu') = g$$

$$Au' + BD^{-1}h - BD^{-1}Cu' = g$$

$$(A - BD^{-1}C)u' = g - BD^{-1}h$$

$$K'u' = f'$$

Which is the new system to be solved, the *condensed* system, with:

$$K' = \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} - \begin{bmatrix} K_{13} \\ K_{23} \end{bmatrix} ([K_{33}])^{-1} [K_{31} \quad K_{32}]$$

$$f' = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} - \begin{bmatrix} K_{13} \\ K_{23} \end{bmatrix} ([K_{33}])^{-1} (f_3)$$

Following the same procedure, but condensing not only u_3 but u_2 as well, the values for the condensed system would be:

$$K' = [K_{11}] - [K_{12} \quad K_{13}] \left(\begin{bmatrix} K_{22} & K_{23} \\ K_{32} & K_{33} \end{bmatrix} \right)^{-1} \begin{bmatrix} K_{21} \\ K_{31} \end{bmatrix}$$

$$f' = (f_1) - [K_{12} \quad K_{13}] \left(\begin{bmatrix} K_{22} & K_{23} \\ K_{32} & K_{33} \end{bmatrix} \right)^{-1} \begin{pmatrix} f_2 \\ f_3 \end{pmatrix}$$

$$K'u_1 = f'$$

Which would be a simple equation with one variable.

3. OBJECTIVES

- Define a method that simplifies the solving procedure for problems that have to deal with a solid material interacting with a fluid, able to divide the global domain into two domains, one for the solid material and one for the fluid, in such a way that an Extended Finite Element approach can be applied to it.

- Make the method able to take advantage of the static condensation and level-set methods to include them in the Extended Finite Element approach, so they can handle the divided domain and calculate accurate results for both subdomains, and so that it can be implemented and added into the Kratos framework.

- Implement an element inside the FluidDynamics application of the Kratos framework that calculates the local system of an element subjected to the specifications of the method defined, implementing the calculation of the subdomains systems and the condensation specifications, so the framework can now provide support of a new way for solving this type of problems.

- Test the implemented element with a set of customized examples to obtain basic information of the physical behavior of the fluid defined in them and in this way validate the definition and the implementation of the method and leave it ready for future improvements and escalations at later phases of development.

4. TOOLS AND SOFTWARE

Applications and Frameworks

- Python IDLE
- GiD
- Microsoft Visual Studio 2010
- Kratos

Programming Languages

- Python
- C++

Operative System

- Windows 7, 64 bits

5. DESIGN AND IMPLEMENTATION

5.1 The proposed embedded method

The proposed embedded method has been defined for a domain with a strong discontinuity, specifically, a domain with two kinds of material: one solid material in interaction with a fluid. This can occur in different ways:

- A fluid going over a defined surface.
- A fixed solid submerged in a fluid.
- A hole in a structure where fluid is flowing (which can be understood as a hollow fixed submerged solid).

Two subdomains are properly defined, a positive domain Ω^+ for the fluid part of the problem, and a negative domain Ω^- for the solid part of the problem. Figure 6 illustrates this in 2D:

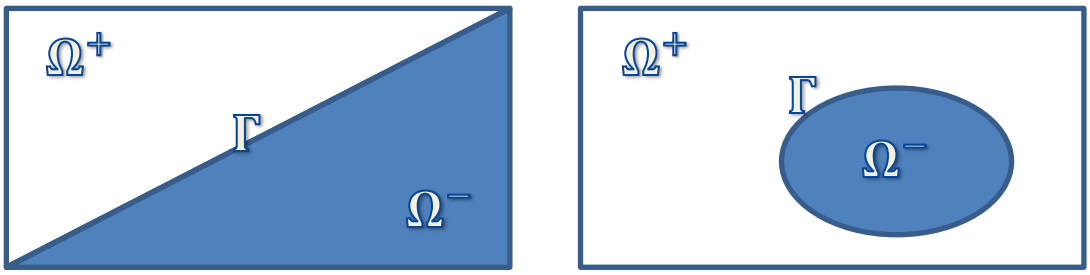


Figure 6 - Two-domain problem examples

The domain is meshed as if the interface doesn't exist and it was all the same material. After meshing, a signed distance function is defined as a Level-set function:

$$\Phi(x) = \begin{cases} d & if\ x \in \Omega_1 \\ -d & if\ x \in \Omega_2 \end{cases}$$

Where:

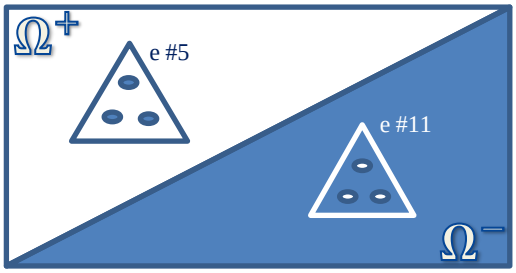
$$d = \min_{\tilde{x} \in \Gamma} \|x - \tilde{x}\|$$

This Level-set function will be used as the enrichment function to determine the behavior of all the elements cut by the interface. It is important to notice that the possibility $d = 0$ is not considered in the definition, the reason for this is that the method will make sure during the meshing procedure that no node belongs to the interface and that no node will be at a distance from the interface small enough to be considered zero.

For all the elements that belong completely to Ω^+ or completely to Ω^-, the procedure will be the same as a normal Finite Element Method implementation (see Figure 7). This means:

- The integration procedure will make use of the Gauss points of the element.
- For each Gauss point, the values for the shape functions will be interpolated using their values on the nodes of the element.

- For each Gauss point, the values for the derivatives of the shape functions will be interpolated using their values on the nodes of the element.
- All the contributions calculated for each Gauss point are assembled into the local stiffness matrix K and local force vector f.



For element #5:
- Using all its Gauss points, a system $Ku = f$ is created.

For element #11:
- Using all its Gauss points, a system $Ku = f$ is created.

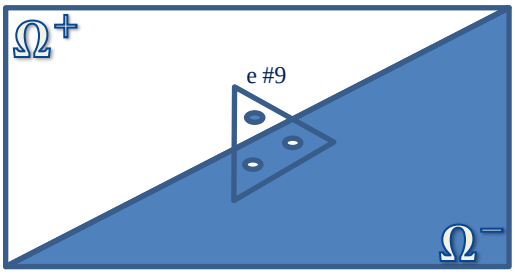
Figure 7 - Two subdomains, elements not cut example.

For the elements cut by the interface, the element will be considered divided in a positive part and a negative part; however, no sub elements will be calculated (no mesh refinement due to the cut of the interface will be performed).

Instead of a mesh refinement, some modifications to the normal procedure are made:

- The integration procedure will make use of the Gauss points of the element.
- For each Gauss point in the positive part of the element, the values for the shape functions will be interpolated using their values on the nodes of the element.
- For each Gauss point in the positive part of the element, the values for the derivatives of the shape functions will be interpolated using their values on the nodes of the element.
- All the contributions calculated for each Gauss point in the positive part of the element are assembled into a positive local stiffness matrix K^+ and positive local force vector f^+.
- For each Gauss point in the negative part of the element, the values for the shape functions will be interpolated using their values on the nodes of the element.
- For each Gauss point in the negative part of the element, the values for the derivatives of the shape functions will be interpolated using their values on the nodes of the element.
- All the contributions calculated for each Gauss point in the negative part of the element are assembled into a negative local stiffness matrix K^- and negative local force vector f^-.

In this way, for each cut element, two systems are defined, as seen in *Figure 8*:



For element #9:
- One Gauss point is used to create the system $K^+u = f^+$.

- The other Gauss points are used to create the system $K^-u = f^-$.

Figure 8 - Two domains, cut elements example.

Notice how each of the systems use the complete vector of unknowns. Expanding the positive system:

$$\begin{bmatrix} K_{11}^{+} & K_{12}^{+} & K_{13}^{+} \\ K_{12}^{+} & K_{22}^{+} & K_{23}^{+} \\ K_{13}^{+} & K_{23}^{+} & K_{33}^{+} \end{bmatrix} \begin{pmatrix} u_1 \\ u_2^{*} \\ u_3^{*} \end{pmatrix} = \begin{pmatrix} f_1^{+} \\ f_2^{+} \\ f_3^{+} \end{pmatrix}$$

u_2 and $u_3$have an asterisk as a superscript to indicate that they are "phantom variables", they do not really belong to the system since all the values were calculated for the positive part of the element, the real u_2 and u_3 are not in this system.

To correct this situation, the system will be condensed, specifically, u_2^{*} and u_3^{*} will be condensed into u_1, giving rise to the new system:

$$K_{+} u_{+} = f_{+}$$

Where:

$$K_{+} = [K_{11}^{+}] - \begin{bmatrix} K_{12}^{+} & K_{13}{}^{+} \end{bmatrix} \left(\begin{bmatrix} K_{22}^{+} & K_{23}^{+} \\ K_{32}^{+} & K_{33}^{+} \end{bmatrix} \right)^{-1} \begin{bmatrix} K_{21}^{+} \\ K_{31}^{+} \end{bmatrix} = [K_{aa}]$$

$$f_{+} = (f_1^{+}) - \begin{bmatrix} K_{12}^{+} & K_{13}{}^{+} \end{bmatrix} \left(\begin{bmatrix} K_{22}^{+} & K_{23}^{+} \\ K_{32}^{+} & K_{33}^{+} \end{bmatrix} \right)^{-1} \begin{pmatrix} f_2^{+} \\ f_3^{+} \end{pmatrix} = (f_a)$$

$$u_{+} = (u_1)$$

The same reasoning is then applied to the negative system:

$$\begin{bmatrix} K_{11}^{-} & K_{12}^{-} & K_{13}^{-} \\ K_{12}^{-} & K_{22}^{-} & K_{23}^{-} \\ K_{13}^{-} & K_{23}^{-} & K_{33}^{-} \end{bmatrix} \begin{pmatrix} u_1^{*} \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1^{-} \\ f_2^{-} \\ f_3^{-} \end{pmatrix}$$

The "phantom variable" in this case is u_1^{*}, so it will be condensed into u_2 and u_3, resulting in the new system:

$$K_{-} u_{-} = f_{-}$$

Where:

$$K_{-} = \begin{bmatrix} K_{22}^{-} & K_{23}^{-} \\ K_{23}^{-} & K_{33}^{-} \end{bmatrix} - \begin{bmatrix} K_{12}^{-} \\ K_{13}^{-} \end{bmatrix} ([K_{11}^{-}])^{-1} [K_{12}^{-} \quad K_{13}^{-}] = \begin{bmatrix} K_{bb} & K_{bc} \\ K_{cb} & K_{cc} \end{bmatrix}$$

$$f_{-} = \begin{pmatrix} f_2^{-} \\ f_3^{-} \end{pmatrix} - \begin{bmatrix} K_{12}^{-} \\ K_{13}^{-} \end{bmatrix} ([K_{11}^{-}])^{-1} (f_1^{-}) = \begin{pmatrix} f_b \\ f_c \end{pmatrix}$$

$$u_{-} = \begin{pmatrix} u_2 \\ u_3 \end{pmatrix}$$

After this, there are two partial systems, one for the positive part of the element, and one for the negative part. The final system for the element is the assembly of these two partial systems:

$$\begin{bmatrix} K_+ & 0 \\ 0 & K_- \end{bmatrix} \begin{pmatrix} u_+ \\ u_- \end{pmatrix} = \begin{pmatrix} f_+ \\ f_- \end{pmatrix}$$

$$\begin{bmatrix} K_{aa} & 0 & 0 \\ 0 & K_{bb} & K_{bc} \\ 0 & K_{cb} & K_{cc} \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_a \\ f_b \\ f_c \end{pmatrix}$$

$$Ku = f$$

Lastly, it is important to mention that to define if an element is cut, it is sufficient to evaluate the Level-set function at each of its nodes, it at least one of these values has a different sign than the rest, then the element has been cut.

5.1.1 Imposing Dirichlet boundary conditions

Now that the main procedure of the proposed embedded method has been presented, it is needed to comment about how the Dirichlet boundary conditions will be imposed, but first it is needed to specify which prescribed values are required.

Since the interface is representing the surface of the solid material, and is where the main contact with the fluid will be, it is necessary to make $u = 0$ all along the interface. In terms of a cut element, this prescribed value has to be imposed on the points where the interface cuts the element.

To do this, the method of Lagrange multipliers will be employed. A system of the following form is needed:

$$\begin{bmatrix} K & H^T \\ H & 0 \end{bmatrix} \begin{pmatrix} u \\ \lambda \end{pmatrix} = \begin{pmatrix} f \\ h \end{pmatrix}$$

In such a way that the following is true:

$$Hu = h = 0$$

For this problem, $h = 0$ since all the prescribed values are 0. The definition of H is then as follows:

$$H = [w_1 N_1 \quad w_2 N_2 \quad w_3 N_3]$$

Where:

- w_i is a set of weights for the expression. As weights, the values for the areas of the cut edges will be used (as presented in the next subsection).

- N_i is a set of shape functions that need to be evaluated at the points where the interface cuts the element.

Since the interpolated values are calculated in two separate scenarios, the positive and negative parts of the element, H needs to be calculated for each of these cases.

Also, since H affects u, and members of this vector are getting condensed in each of those cases, the imposition of the BCs will depend on which nodes are being condensed and which are not.

Figure 9 helps visualize the situation and the way it will be handled:

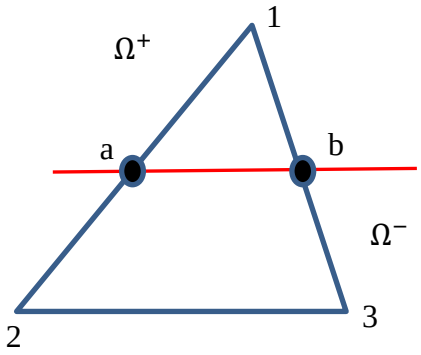


Figure 9 - Imposition of BCs

5.1.2 First case: positive system

In this scenario, nodes 2 and 3 will be condensed into node 1, the influence of these two nodes on the points a and b needs to remain after the condensation, so two different values for H are needed: H_2 and H_3.

Since node 2 only influences the edge 1-2, and node 3 only influences the edge 1-3, H_2 and H_3 will be defined as (using the respective edge areas as weights):

$$H_2 = [A^{12}N_1^{12} \quad A^{12}N_2^{12} \quad 0]$$

$$H_3 = [A^{13}N_1^{13} \quad 0 \quad A^{13}N_3^{13}]$$

Where:

- A^{12} is the area of the cross section of the edge 1-2.
- A^{13} is the area of the cross section of the edge 1-3.
- N_i^{12} is the shape function calculated at the point a, interpolated using the values of the shape functions at nodes 1 and 2.
- N_i^{13} is the shape function calculated at the point b, interpolated using the values of the shape functions at nodes 1 and 3.

In this case, the final system takes the form (before condensation):

$$\begin{bmatrix} K^+ & H_2^T & H_3^T \\ H_2 & 0 & 0 \\ H_3 & 0 & 0 \end{bmatrix} \begin{pmatrix} u \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} f^+ \\ 0 \\ 0 \end{pmatrix}$$

5.1.3 Second case: negative system

In this scenario, node 1 will be condensed into nodes 2 and 3, the influence of this node on the points a and b needs to remain after the condensation, so just one value for H is needed: H_1.

Notice how node 1 influences both the edge 1-2 and the edge 1-3, H_1 will be defined as (using the respective edge area as weight):

$$H_1 = [A^{12}N_1^{12} + A^{13}N_1^{13} \quad A^{12}N_2^{12} \quad A^{13}N_3^{13}]$$

Where A^{12}, A^{13}, N_i^{12} and N_i^{13} are the same as in the previous case. Now, the final system takes the form (before condensation):

$$\begin{bmatrix} K^- & H_1^T \\ H_1 & 0 \end{bmatrix} \begin{pmatrix} u \\ \lambda_1 \end{pmatrix} = \begin{pmatrix} f^- \\ 0 \end{pmatrix}$$

5.1.4 Calculation of the shape functions in the points where the interface cuts the element.

Figure 10 shows an edge of an element that has been cut by the interface, focusing on the values of the Level-set function at each of the nodes that define said edge:

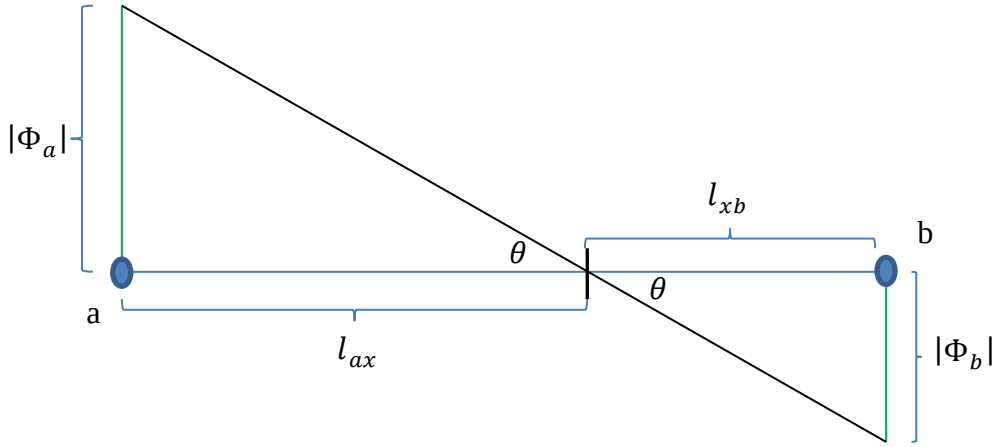


Figure 10 - Relation between the level-set function values in a cut edge

Calculating the tangent of the angle θ:

$$\tan\theta = \frac{|\Phi_a|}{l_{ax}} = \frac{|\Phi_b|}{l_{xb}}$$

From there, the following can be said:

$$\frac{l_{xb}}{l_{ax}} = \frac{|\Phi_b|}{|\Phi_a|}$$

Now, Figure 11 shows an edge of an element that has been cut by the interface, focusing on the values of the shape functions at each of the nodes that define said edge:

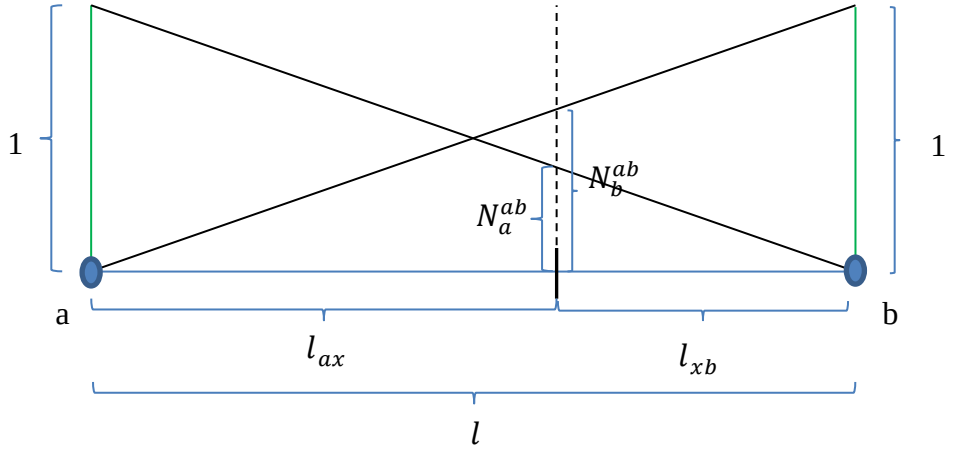


Figure 11 - Interpolation of shape functions at the cut of an edge

Using the principles of the similarity of triangles, the following expressions can be obtained from Figure 11:

$$\frac{1}{l} = \frac{N_a^{ab}}{l_{xb}} = \frac{N_b^{ab}}{l_{ax}}$$

From there, solving for N_a^{ab}:

$$N_a^{ab} = N_b^{ab} \frac{l_{xb}}{l_{ax}}$$

The shape functions N_a^{ab} and N_b^{ab} must form a partition of unity, so:

$$N_a^{ab} + N_b^{ab} = 1$$

$$N_b^{ab} \frac{l_{xb}}{l_{ax}} + N_b^{ab} = N_b^{ab} \left(\frac{l_{xb}}{l_{ax}} + 1 \right) = 1$$

$$N_b^{ab} = \frac{1}{\frac{l_{xb}}{l_{ax}} + 1}$$

It is known that $\frac{l_{xb}}{l_{ax}} = \frac{|\Phi_b|}{|\Phi_a|}$, so:

$$N_b^{ab} = \frac{1}{\frac{l_{xb}}{l_{ax}} + 1} = \frac{1}{\frac{|\Phi_b|}{|\Phi_a|} + 1} = \frac{1}{\frac{|\Phi_b| + |\Phi_a|}{|\Phi_a|}} = \frac{|\Phi_a|}{|\Phi_b| + |\Phi_a|}$$

So the shape functions are calculated using the level-set function values on the nodes that define the cut edges:

$$N_b^{ab} = \frac{|\Phi_a|}{|\Phi_b| + |\Phi_a|}$$

$$N_a^{ab} = 1 - N_b^{ab}$$

5.2 The Kratos framework

Kratos is a framework that has support for solving a great variety of problems in the field of Computational Mechanics. Figure 12 shows a brief exposition of its architecture:

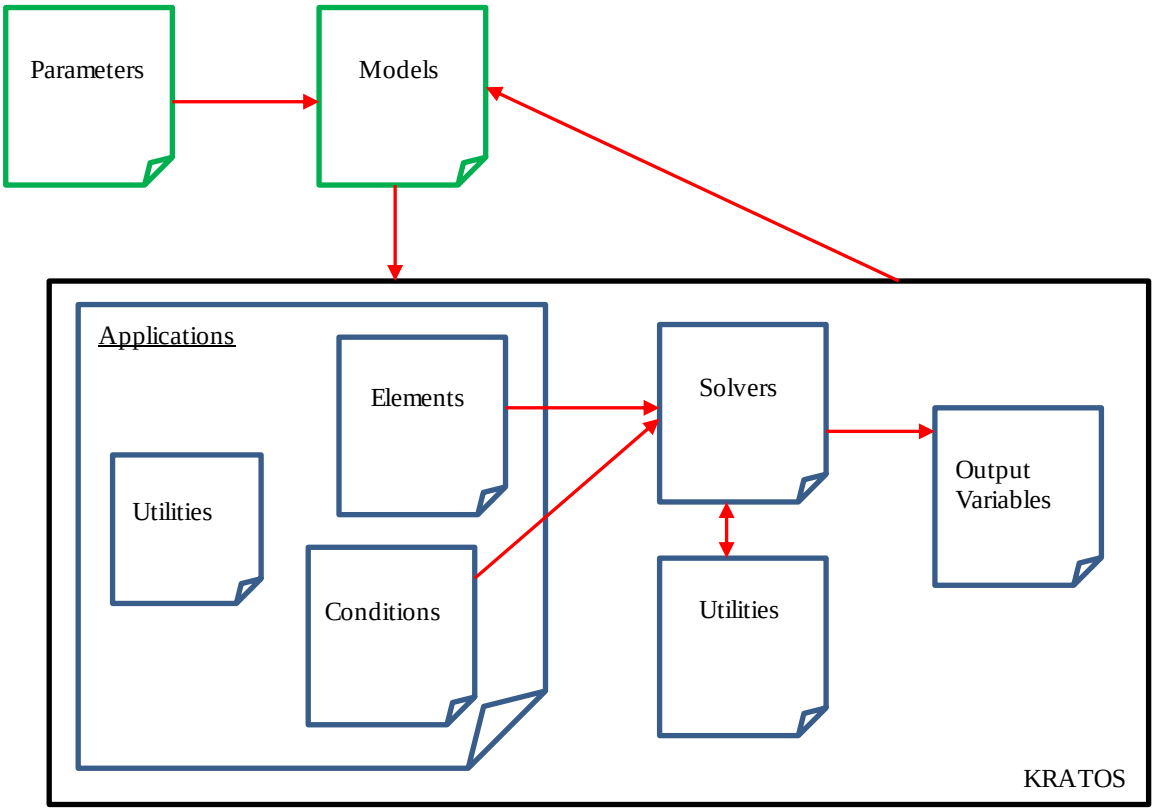


Figure 12 - Kratos architecture

Each section of Figure 12 represents one of the different modules required to work with Kratos, the ones in green represent modules where the user works mainly with the Python scripting language, while the modules in blue represent pre-compiled modules written in C++ that constitute the core of the framework and that the user calls in its Python scripts to perform the solution of any problem supported by Kratos.

Detailed next, is a brief description of all the modules depicted on Figure 12:

- **Parameters**: set of Python files that specify control variables that the Models need to perform their operations. Strictly speaking, it is an optional module, since all this code could be incorporated in the same files as the Models, but it is recommended to use it to give a high degree of legibility to a work and for easy access to critical variables that usually need to be altered during tests.

- **Models**: set of Python files that interact with the Kratos core, providing to it all the necessary input data to run a particular solver to obtain simulation results. In this module the geometry of the problem is defined, the domain, the boundary conditions, the type of problem, the type of solver needed, the type of results required, and so on. After the results are obtained, they can be printed in any format the user finds appropriate.

- **Applications:** set of pre-compiled C++ files that contain all the necessary tools to solve a particular kind of problem in the field of Computational Mechanics. There are applications for solving solid problems, fluid problems, thermal problems, electrostatic problems, etc. For each applications there a set of internal modules such as:
  - **Elements:** set of pre-compiled C++ files that describe possible implementations for the calculation of the local system in each element of the problem mesh. Some implementations may respond to a classic FEM, others may implement several XFEM approaches; some might be defined for very particular elements present in certain problems or situations.

  - **Conditions:** set of pre-compiled C++ files that describe possible implementations for the imposing of particular conditions on the local system of each element of the problem mesh. As with the elements, these implementations can respond to different kinds of implementations and requirements.

  - **Utilities:** set of pre-compiled C++ files that contain specialized functions needed to ease the construction of local systems for each element of the problem mesh. Theses utilities can usually be applied to very particular cases or problems, they are not meant to be used in a general way.

- **Solvers:** set of pre-compiled C++ files that contain all the possible implementations for solving a Computational Mechanics problem given the assembly of local systems of each element in the problem mesh. Most of these implementations are of iterative nature, and their goal is to obtain the best possible solution for a particular problem based on the input data.

- **Utilities:** set of pre-compiled C++ files that contain specialized functions needed to ease the solving procedures of Computational Mechanics problems. These functions are of a more general perspective than the ones located in each application, and they can be used by any of them, if necessary.

- **Output variables:** set of information variables manipulated by all the pre-compiled C++ files that constitute the core of Kratos, which are defined by the Model files and that will be handled by them to present the results in the most convenient way. These variables' values are obtained after a convergent and successful solving procedure.

5.3 Implementation of the proposed embedded method

For the implementation of the proposed embedded method, a new element called CondensatedTwoFluidVMS was created as part of the FluidDynamics application of the Kratos framework.

This element calculates the system of each element based on all the specifications explained throughout Section 5.1; however, with the considerations that the calculations will be performed in 3D, so the elements will be tetrahedras.

For this reason, a local system will take the form:

$$Ku = f$$

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{21} & K_{22} & K_{23} & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & K_{43} & K_{44} \end{bmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}$$

The structure of each u_i is as follows:

$$u_i = \begin{pmatrix} v_i \\ p_i \end{pmatrix}$$

Where:
- v_i is the velocity at the node.
- p_i is the pressure at the node.

Every v has the following structure:

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

For this reason, each K_{ij} in the stiffness matrix is a submatrix with dimensions 4x4; and in the same way, each f_i in the force vector is a subvector with dimensions 4x1.

Detailed next, is a description of the algorithms implemented in each of the main functions of the CondensatedTwoFluidVMS element, so it can be understood how it performs the proposed embedded method explained in Section 5.1:

5.3.1 CalculateLocalSystem

This function is present in all elements implemented inside the Kratos framework, it is the starting point for the construction of the local system of each element (prepares the stiffness matrix and the force vector according to the method(s) employed).

For the CondensatedTwoFluidVMS element, it determines whether the element is cut or not, and constructs the left hand side matrix and the right hand side vector of the local system accordingly.

Parameters:

- *rLeftHandSideMatrix*: matrix of dimensions 16x16 that serves as the container for the final result of the stiffness matrix of the local system.
- *rRightHandSideVector*: vector of dimensions 16x1 that serves as the container for the final result of the force vector of the local system.
- *rCurrentProcessInfo*: structure that contains important information about the problem. This structure is initialized in the Model files and passed down to Kratos to use for solving.

Algorithm:

i. The local size of the problem is determined (this is used to know the proper number of nodes and dimension of the problem).

ii. *rLeftHandSideMatrix* is initialized with values of 0 in all its members.

iii. *rRightHandSideVector* is initialized with values of 0 in all its members.

iv. *rLeftHansSideMatrixPositive* and *rLeftHandSideMatrixNegative* are created as the containers for the positive and negative parts in the cut elements.

v. Similarly, *rRightHandSideVectorPositive* and *rRightHandSideVectorNegative* are created as the containers for the positive and negative parts in the cut elements.

vi. The variable *MassMatrix* is created to perform the construction of the left-hand side matrix before storing it in *rLeftHandSideMatrix* at the end of the procedure. This is to be used with the non-cut elements.

vii. Similarly, *MassMatrixPositive* and *MassMatrixNegative* are created for the same purpose when the element is cut.

viii. Using the geometry of the problem, the shape functions, their derivatives, and the area are calculated for the element.

ix. The level-set function values for the nodes of the element are obtained.

x. The exact coordinates for all the element nodes are calculated.

xi. Using the previous information, other variables are calculated: shape function values on the Gauss points, their location, and the edge areas of the element. If the element is cut, additionally, it is determined for each gauss point if it is located in the positive part or the negative part of the element; the element also gets divided into partitions used in the integration process.

xii. If the element is not cut, *CalculateLocalSystemAux* is called once, *MassMatrix* is used for the calculation of the left-hand side of the local system using the whole element, and the result will be stored in *rLeftHandSideMatrix* and *rRightHandSideVector*.

xiii. If the element is cut, a vector of signs for the nodes of the elements is calculated, to easily identify the ones located on the positive part of the element, and the ones located on the negative part.

xiv. If the element is cut, *CalculateLocalSystemAux* is called twice: first, *MassMatrixPositive* is used for the calculation of the left-hand side of the local system using only the positive part of the element, and the result will be stored in *rLeftHandSideMatrixPositive* and *rRightHandSideVectorPositive*; then, *MassMatrixNegative* is used for the calculation of the left-hand side of the local system using only the negative part of the element, and the result will be stored in *rLeftHandSideMatrixNegative* and *rRightHandSideVectorNegative*.

xv. Finally, *condensate* is called twice: first, condensing the positive part of the element and allocating the result on *rLeftHandSideMatrix* and *rRightHandSideVector*. Then, condensing the negative part of the element and allocating the result on the updated *rLeftHandSideMatrix* and *rRightHandSideVector*, this will be the final local system.

5.3.2 CalculateLocalSystemAux

This function does all the necessary calculations to create both the left-hand side and the right-hand side of the local system for the element. It stores all the calculations in the *MassMatrix*, *rLeftHandSideMatrix* and *rRightHandSideVector* containers using all the existing Gauss points if the element is not cut; and only a portion of the Gauss points is the element is cut (either the ones on the positive part or the negative part), while the rest are skipped.

Parameters:

- *rLeftHandSideMatrix*: matrix of dimensions 16x16 that serves as the container for the final result of the stiffness matrix of the local system.

- *rRightHandSideVector*: vector of dimensions 16x1 that serves as the container for the final result of the force vector of the local system.
- *MassMatrix*: matrix of dimensions 16x16 that serves as the medium to calculate the final result for the stiffness matrix of the local system.
- *flag*: flag variable that determines is the process is to be done for a non-cut element (0), the positive part of a cut element (1), or the negative part of a cut element (2).
- *rCurrentProcessInfo*: structure that contains important information about the problem. This structure is initialized in the Model files and passed down to Kratos to use for solving.
- *ndivisions*: number of partitions of the element, related to the number of Gauss points and whether or not the element is cut.
- *Area*: the area of the element.
- *volumes*: the volumes for each of the partitions specified.
- *N*: shape functions for the element.
- *DN_DX*: derivative of the shape functions for the element.
- *Ngauss*: set of Gauss points for the element and the integration procedure to be performed.
- *signs*: the location for each of the Gauss points: 1 if they are on the positive part of the element, -1 if they are on the negative part (used for cut elements).

Algorithm:

i. The local Body Force vector is obtained from the process information.

ii. If there is only one partition (the element is not cut), the Gauss points for an exact calculation of a tetrahedral element are obtained, as well as its weights, shape function values and volumes.

iii. If the element is cut, the total volumes for the positive and negative parts of the element are calculated.

iv. For each Gauss point in the element (when it is not cut) or in the specified part of the element (if it is cut):

   a. The density is interpolated on the Gauss point.
   b. The kinetic viscosity is interpolated on the Gauss point.
   c. The Body Force is interpolated on the Gauss point.
   d. The effective viscosity is calculated.
   e. The momentum is added to the right-hand side vector of the local system.
   f. The advective velocity is calculated.
   g. The stabilization parameters are calculated.
   h. The integration point vector contribution is added to the left-hand side matrix of the local system.
   i. The consistent mass matrix contribution is added to MassMatrix.

v. The Mass matrix gets lumped.

vi. For each Gauss point in the element (when it is not cut) or in the specified part of the element (if it is cut):

   a. The density is interpolated on the Gauss point.
   b. The kinetic viscosity is interpolated on the Gauss point.
   c. The Body Force is interpolated on the Gauss point.
   d. The effective viscosity is calculated.
   e. The momentum is added to the right-hand side vector of the local system.
   f. The advective velocity is calculated.
   g. The stabilization parameters are calculated.
   h. The mass stabilization terms are added to MassMatrix.

vii. The left-hand side matrix of the local system is updated with the product between the body force vector and the mass matrix.

viii. The right-hand side vector of the local system gets updated with the contributions of Mass using the body force vector and the previous computations of the solution.

ix. The right-hand side vector of the local system gets updated with the residual of the local system.

5.3.3 condensate:

Given a left-hand side matrix and a right-hand side vector, and a flag specifying either the positive part or the negative part of a cut element, this function condensates the part not specified into the specified part, and stores the result into the final *rLeftHandSideMatrix* and *rRightHandSideVector* variables.

Parameters:

- *LHSMatrixT*: matrix of dimensions 16x16 that serves as the container for the final result of the stiffness matrix of the local system.
- *LHSMatrix*: matrix of dimensions 16x16 that serves as the container for the partial result of the stiffness matrix of the local system (either the positive or the negative part).
- *RHSVectorT*: vector of dimensions 16x1 that serves as the container for the final result of the force vector of the local system.
- *RHSVector*: vector of dimensions 16x1 that serves as the container for the partial result of the force vector of the local system (either the positive or the negative part).
- *pon*: vector that contains the signs for each node of the element (1 if it is on the positive part, and -1 if it is on the negative part).
- *ponflag*: flag variables that specify which part will be condensed. 1 if the negative part is to be condensed into the positive part, or 2 for the opposite process.
- *eltaldistances*: level-set function values for all the nodes in the element.
- *edge_areas*: cross-section areas for all the edges of the element.

Algorithm:

i. Two vectors are created: one containing the IDs of the nodes that belong in the part of the element specified by *ponflag* (*in*), and another one to contain the IDs of the nodes on the other part (*out*).

ii. The necessary shape function are calculated evaluated on the points where the interface cuts the element, one set for each point on the out vector, and then they are multiplied by the corresponding edge area, in order to construct the necessary matrices H for the imposing of BCs (as explained back in Section 5.1).

iii. Following the exposition of the static condensation method explained in Section 2.5, four matrix containers are created: one for each of the four identifiable sections in the method (the matrices *A*, *B*, *C* and *D*).

iv. In the same manner, two vector containers are created: one for each of the two sections identifiable in the static condensation method (the vector *g* and *h*).

v. The input matrix *LHSMatrix* is conceptually considered as a macro-matrix of dimensions 4x4, where each member is a matrix also of dimensions 4x4. One by one, the members of the macro-matrix are evaluated to know in which of the four containers they belong based on the value specified by *ponflag*, and then they are copied into the one they correspond to.

vi. Then, the input vector *RHSVector* is considered as a macro-vector of dimensions 4x1, where each member is a vector also of dimensions 4x1. One by one, the members of the macro-vector are evaluated to know in which of the two containers they belong based on the value specified by *ponflag*, and then they are copied into the one they correspond to.

vii. A matrix of dimensions 4x4, where the first three columns correspond to an identity matrix of dimension 3 and the last column is full of zeroes, is prepared to be multiplied and copied by the values of H so they can be appropriate located inside the containers B, C and D.
Figure 13 shows the final state for an example case after reaching this point.

viii. The vector *g* is updated with the residual of the local system.

ix. The static condensation procedure is performed.

x. The condensed result is stored in its respective location inside the input matrix *LHSMatrixT* and the input vector *RHSVectorT*.

Annex A shows the code for these three functions.

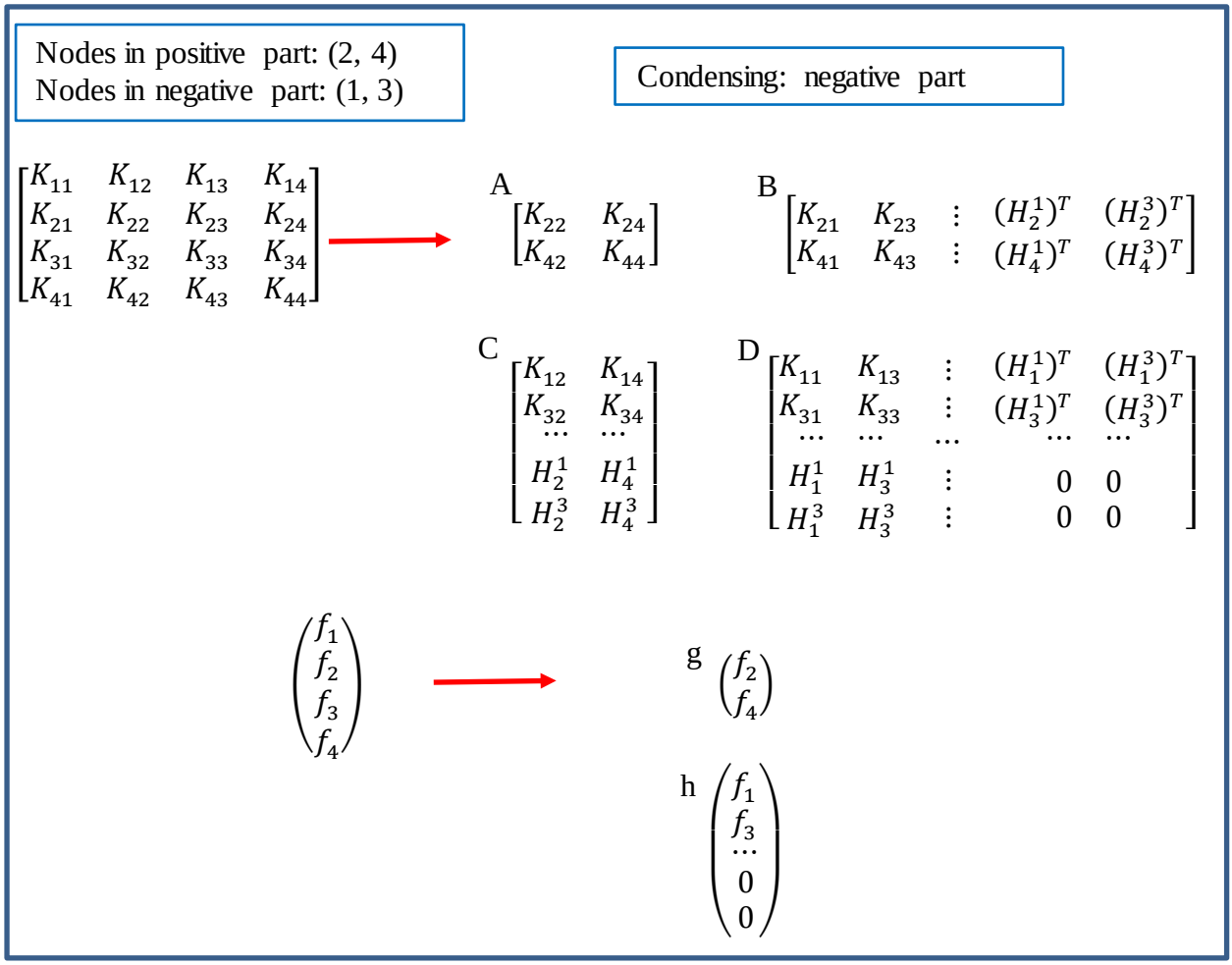


Figure 13 - Example of condensation

5.4 Other files modified

To fully integrate the implementation into the Kratos framework, a few already existing files needed to be slightly modified:

- *enrichment_utilities_duplicate_dofs.h*: this files is one the general purpose utilities of the Kratos architecture. A function to calculate the cross-section areas of the element was added, since it was needed as part of the method to calculate the shape functions in the points where the element is cut.

- *fluid_dynamics_application.h*: this file and the next one serve as the main definition of all the elements, conditions and tools that constitute the Fluid Dynamics application inside the Kratos architecture. Some lines of code were added to make the application recognize the new implemented element: specifically the name of the element and if it works on 2D or 3D.

- *fluid_dynamics_application.cpp*: as with the previous file, this one registers information about all the implementations inside the Fluid Dynamics application. Some lines of code were added for the application to recognize the new implemented element: specifically the explicit registration and the type, name and number of parameters the element will need to calculate the local systems.

- *vms_monolithic_solver_condensated.py*: this file is one of the auxiliary solvers inside the Fluid Dynamics application, and it is a copy an already existing file (vms_monolithic_solver.py). Some control parameters were modified to make it compatible with the new implemented element; a copy was made to avoid making the existing version incompatible with other implemented elements.

6. TEST RESULTS

After the implementation, a couple of test examples were defined to validate the design of the proposed embedded method and its integration into the Kratos framework. Detailed next, both of these test examples with the results obtained.

6.1 Initial test

This first example simulates a small cube domain of dimensions 1x1x1, with a plane cutting the cube diagonally from the middle of one side to the middle of the bottom of the cube, so a prism of triangular base is form and becomes a sub-volume. From the side of the cube opposite to the entry of the cutting plane, a fluid with a constant velocity of 1m/s is entering parallel to the base of the cube, and the outlet of the system is on the opposite side of the inlet.

As boundary conditions, the velocity is enforced as 0 on the top and bottom of the cube and on the surface of the cutting plane. Figure 14 shows the model in a graphical way:

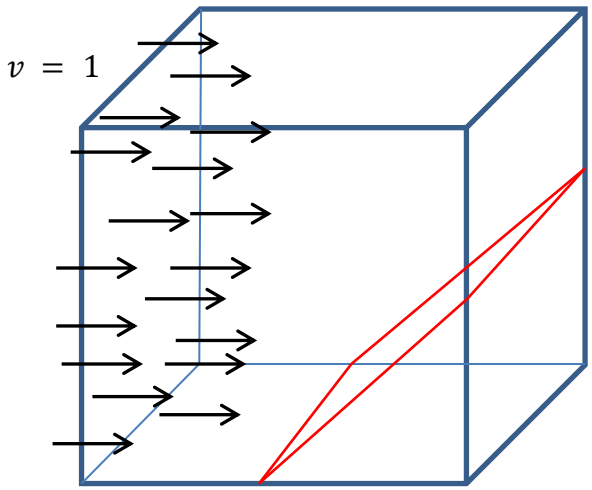


Figure 14 - Test example#1

The detailed control parameters used for this example are:

- AMGL Solver.
- Tolerance of 1e-5.
- Maximum of 100 iterations for the linear solver.
- Diagonal preconditioner.
- GMRES Krilov solver.
- Dynamic τ of 0.01.
- Δt of 0.03s.
- Simulation from $t = 0s$ to $t = 10s$.
- 500 steps for the simulation.
- 3 levels of refinement.
- Number of elements of 20x20x10.

Figure 15 shows the results for the simulation focusing on the velocity vector; Figure 16 shows the results for the simulation focusing on the velocity contour lines; and Figure 17 shows the results for the simulation focusing on the pressure contour lines:

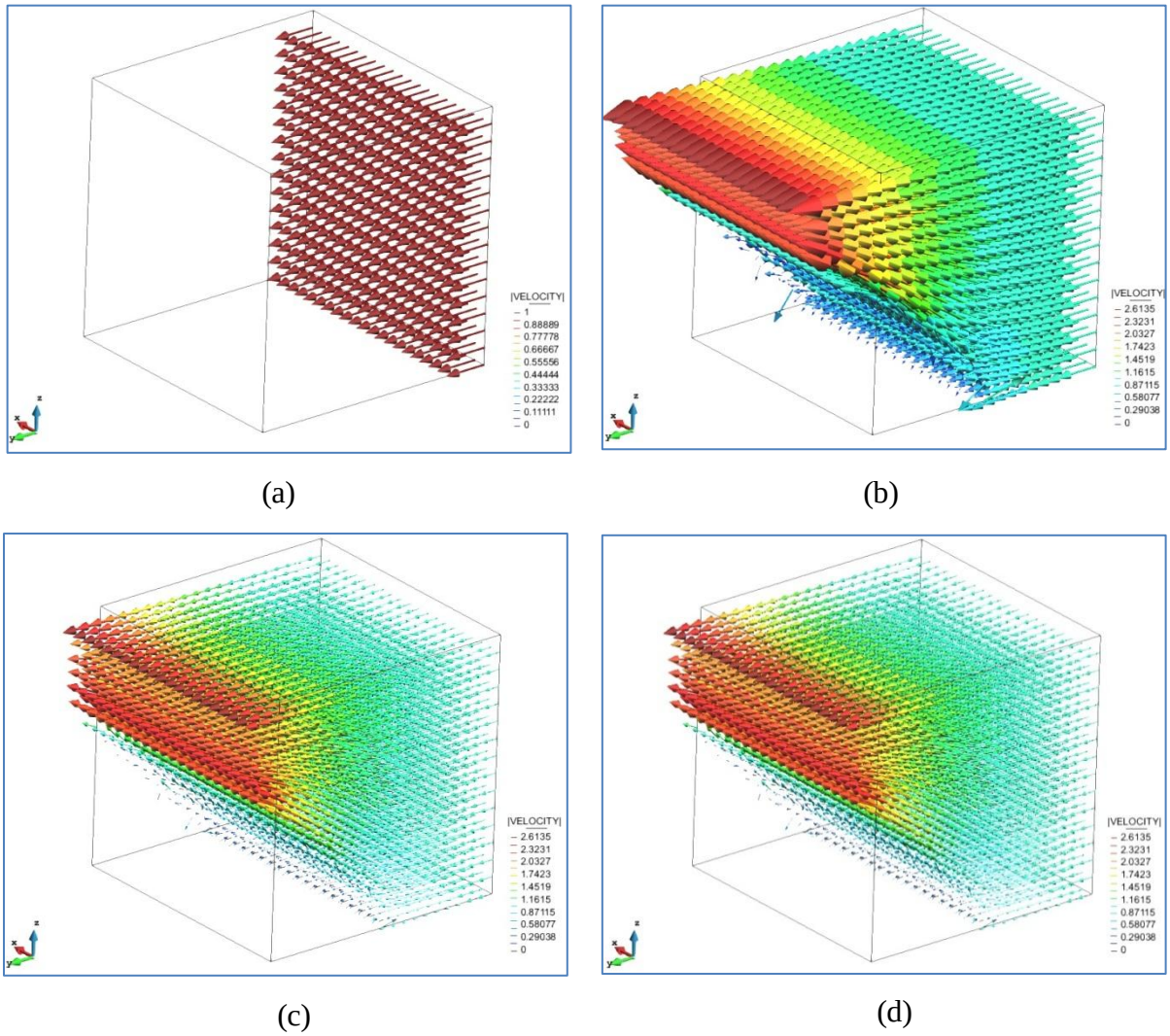


Figure 15 - Results for the test example #1. (a) t=0s. (b) t=3.33s. (c) t=6.66s. (d) t=10s.

As it can be observed on the results:

- The expected behavior has been achieved throughout all the simulation time.
- Certain stability is achieved in the movement of the fluid with the passing of time.
- The fluid goes over the cutting plane towards the outlet.
- The fluid approximately doubles its velocity by the time it reaches the outlet.
- The boundary conditions on the top and bottom of the cube are being enforced and maintained.

- The pressure concentrates at the bottom of the cutting plane, as it is the point of first contact with the fluid.

Therefore the test can be considered a success, and the implementation is ready to be subjected to a more complex problem.

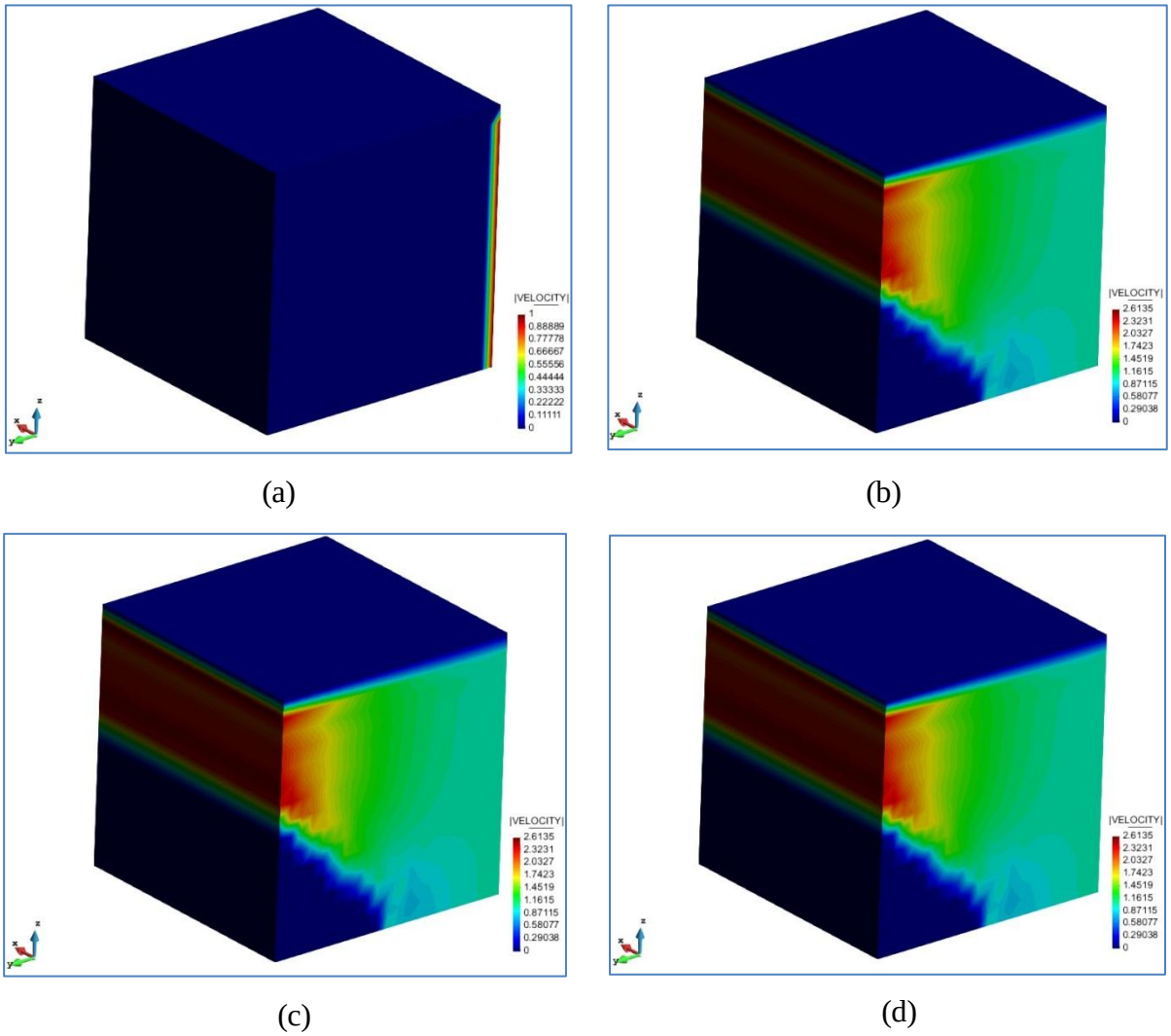


Figure 16 - Results for the test example#1 (contour lines). (a) t=0s. (b) t=3.33s. (c) t=6.66s. (d) t=10s.

6.2 Further testing

This second example simulates a bigger rectangular domain 60x60x30, with a smaller solid cube located at the bottom of the rectangular box and it is considered as a sub-volume, virtually cutting the bigger space. From the side of the rectangular box a fluid with a constant velocity of 1m/s is entering parallel to the base of the box, and the outlet of the system is on the opposite side of the inlet. The fluid moves horizontally in such a way that it will collide with the cube and it will have to find its way around it while surrounding it.

As boundary conditions, the velocity is enforced as 0 on the top and bottom of the rectangular domain and on the surface of the cube. Figure 18 shows the model in a graphical way.

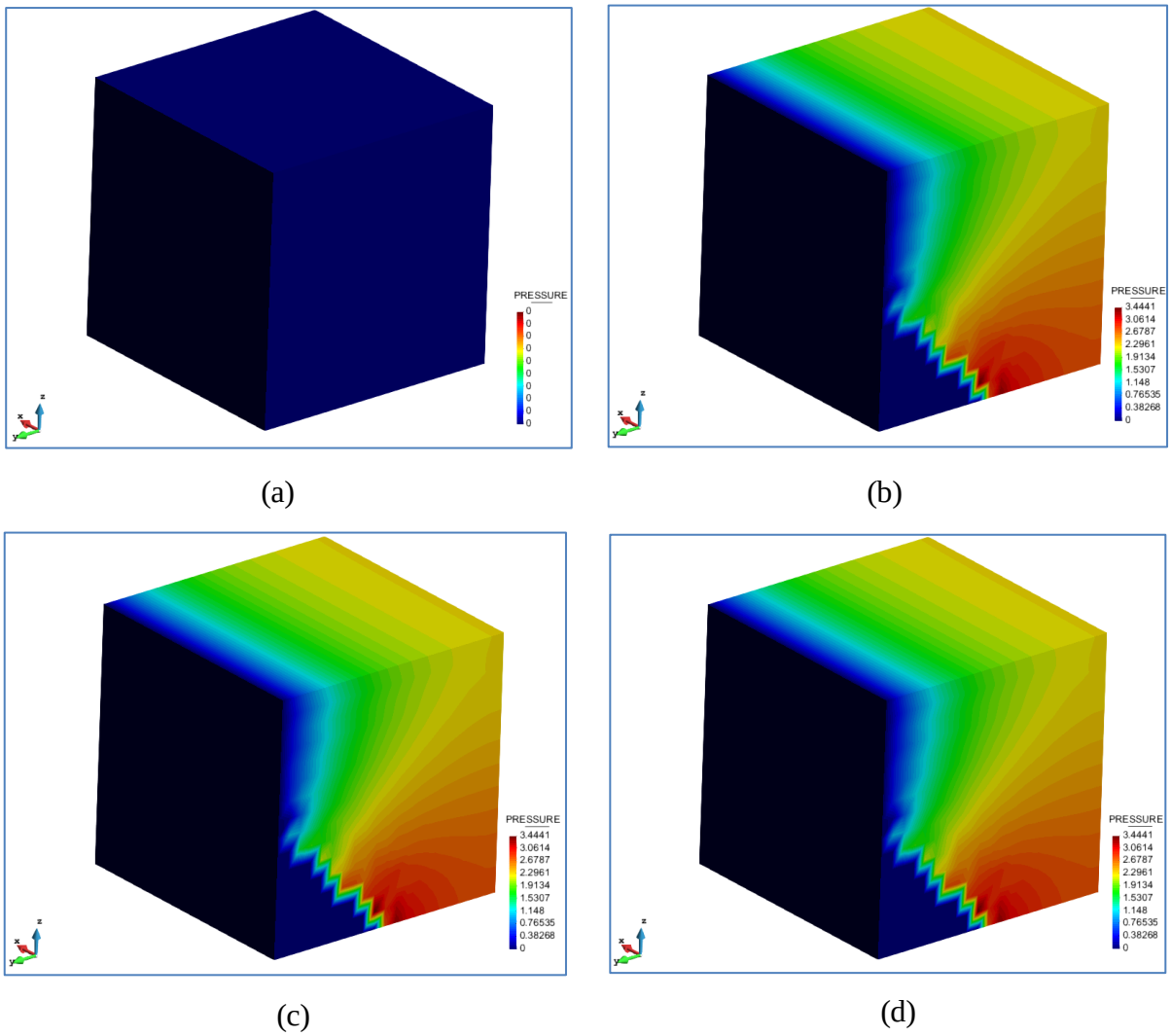


Figure 17 - Results for the test example#1 (pressure). (a) t=0s. (b) t=3.33s. (c) t=6.66s. (d) t=10s.

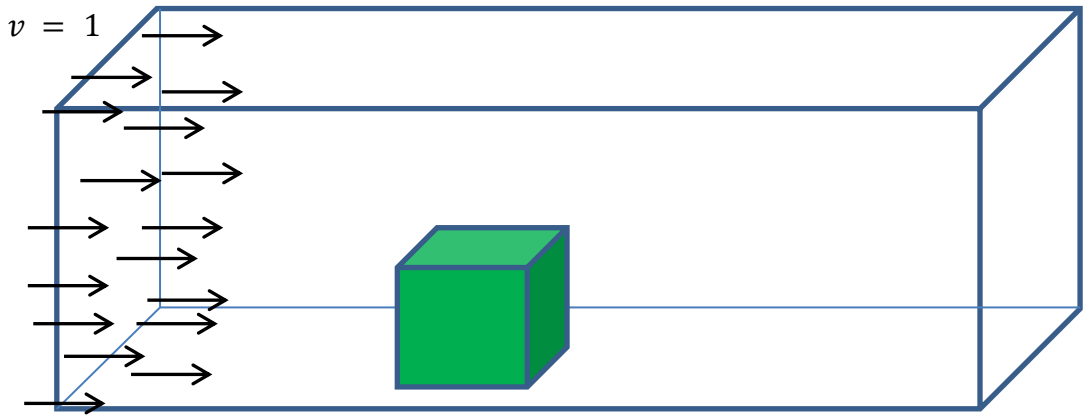


Figure 18 - Further testing model

The detailed control parameters used for this example are the same as in the previous problem, except for the number of elements, which this time is of 40x40x20.

Figure 20 shows the results for the simulation focusing on the velocity vectors from a top view and a lateral vies; Figure 21 shows the results for the simulation focusing on the velocity contour lines (also from a top view and a lateral view); and Figure 22 shows the results for the simulation focusing on the pressure contour lines (from a top vies and a lateral view as well). Firstly, Figure 19 shows a top view and lateral view combined.

Once again, it can be inferred from the results:

- The expected behavior has been achieved throughout all the simulation time.
- Certain stability is achieved in the movement of the fluid with the passing of time.
- The fluid goes over and around the cube towards the outlet.
- The fluid increases its velocity accordingly by the time it reaches the cube and has to find its way around it.
- The boundary conditions on the top and bottom of the box and on the surface of the cube are being enforced and maintained.
- The pressure concentrates accordingly around the surface of the cube, as it is the area of maintained contact with it.

Therefore the test can be considered a success, and the implementation can be considered correct and ready for future improvements.

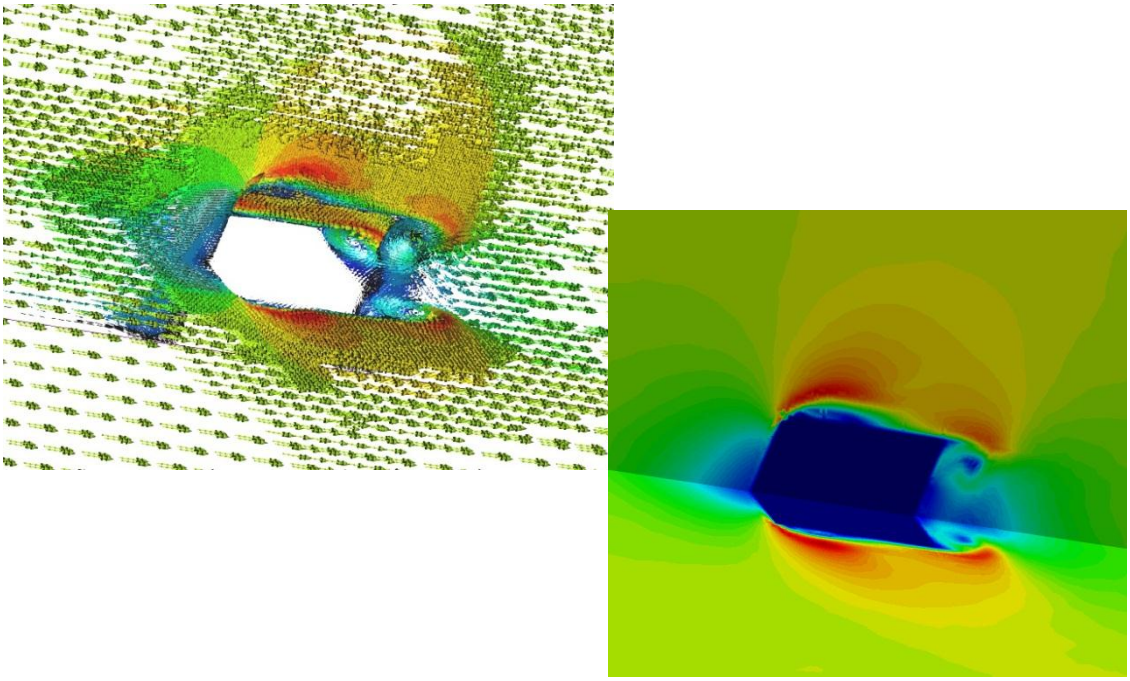
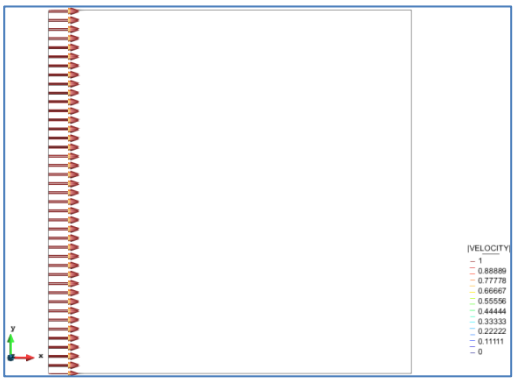
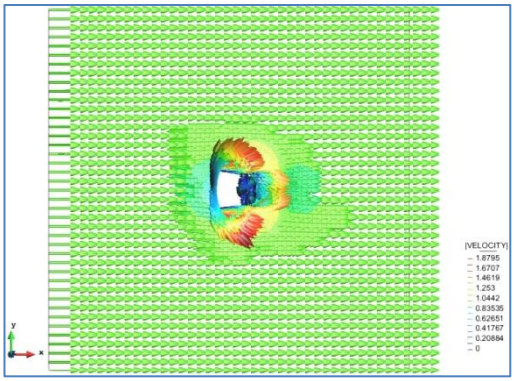


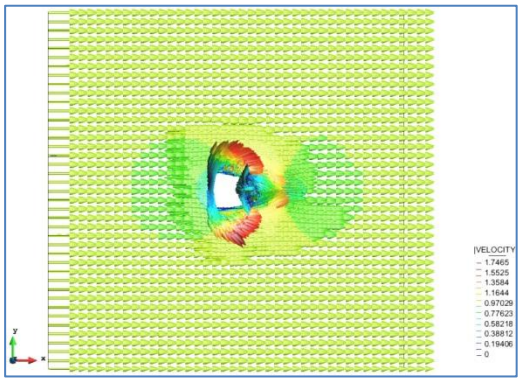
Figure 19 - Top and lateral view combined.



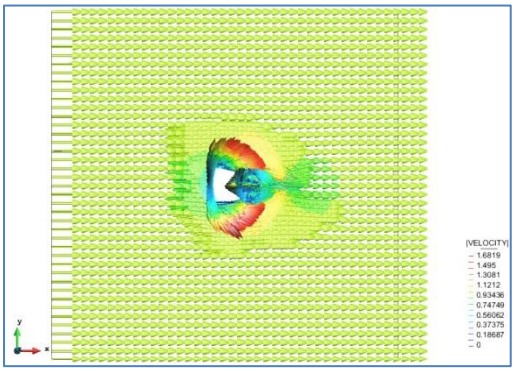
(a)



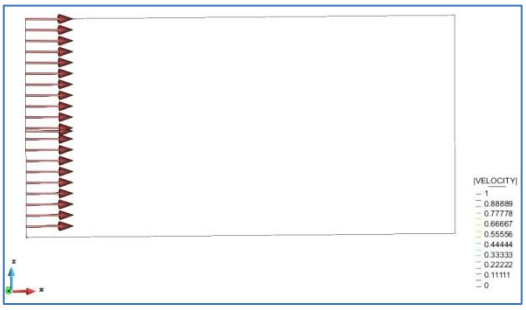
(b)



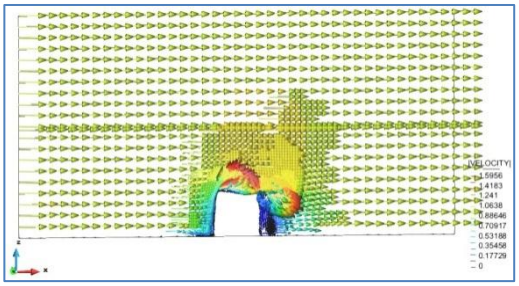
(c)



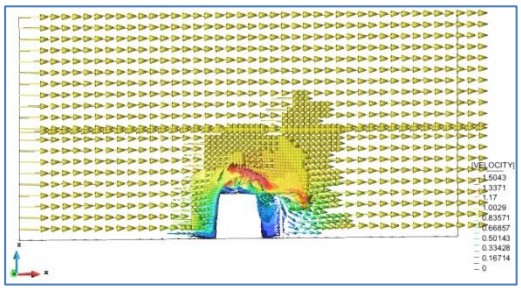
(d)



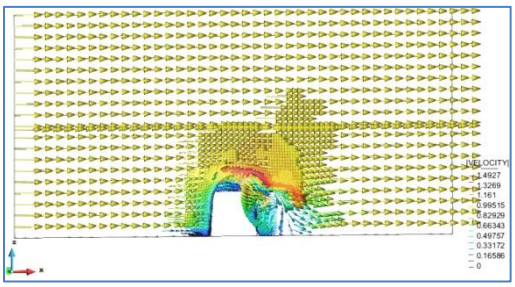
(e)



(f)

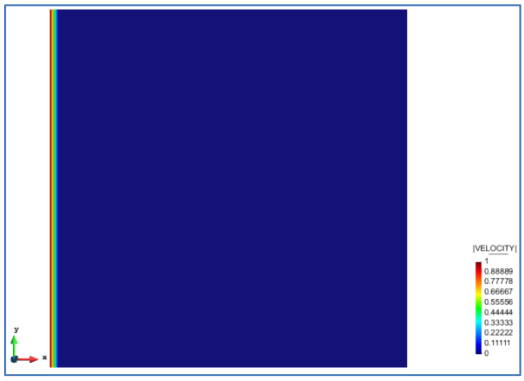


(g)

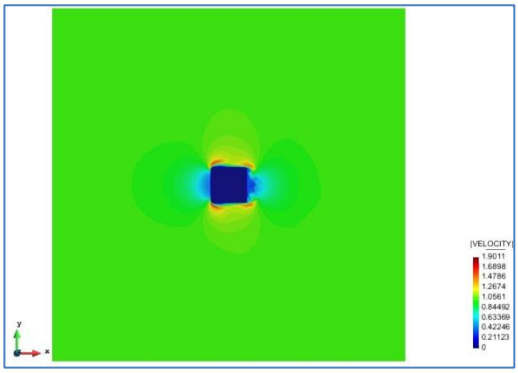


(h)

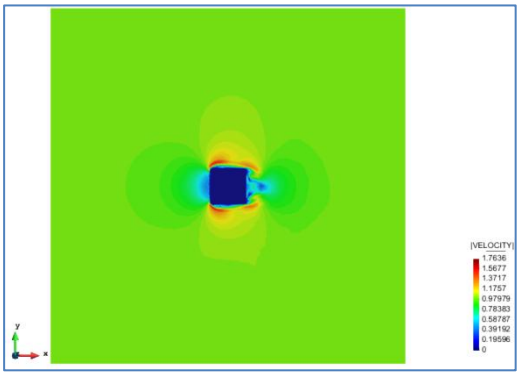
Figure 20 – Velocity vectors (a)-(d): Top view from t=0s to t=10s. (e)-(h): Lateral view from t=0s to t=10s.



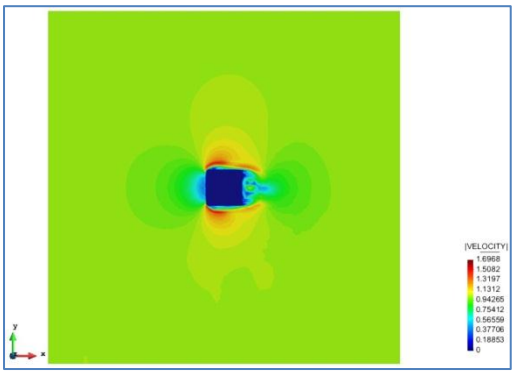
(a)



(b)

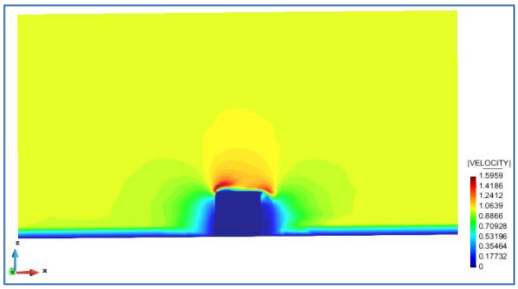


(c)

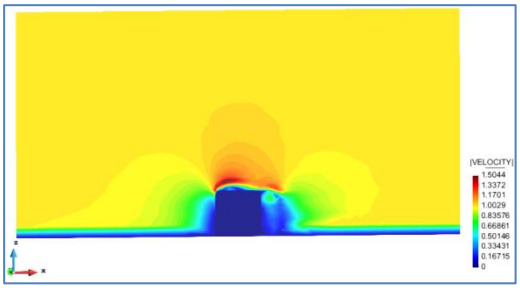


(d)

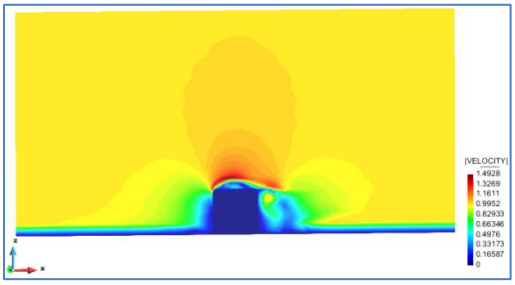
(e)



(f)



(g)



(h)

Figure 21 - Velocitycontour lines (a)-(d): Top view from t=0s to t=10s. (e)-(h): Lateral view from t=0s to t=10s.

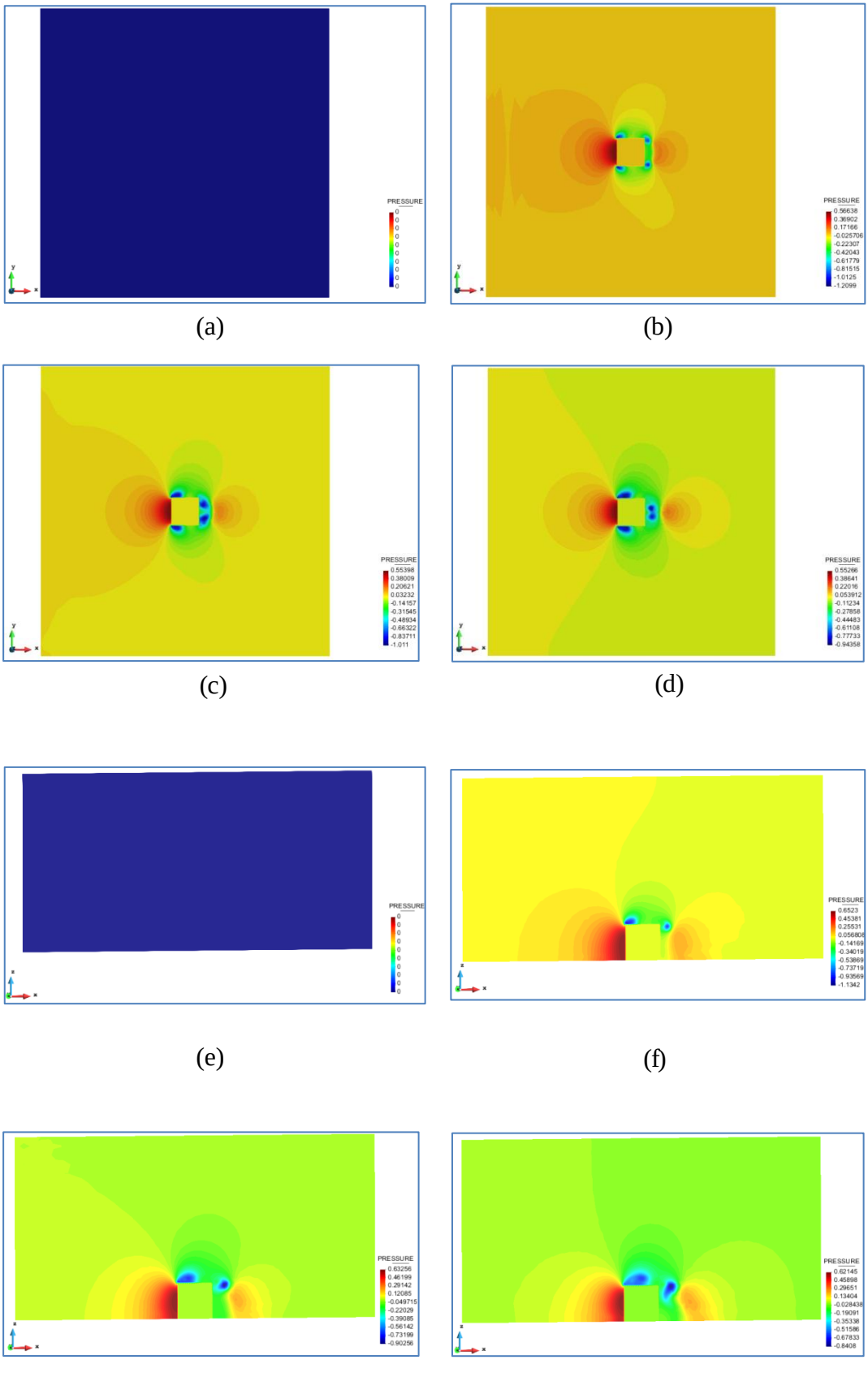


Figure 22 - Pressure (a)-(d): Top view from t=0s to t=10s. (e)-(h): Lateral view from t=0s to t=10s.

7. CONCLUSIONS AND RECOMMENDATIONS

7.1 Conclusions

- Based on the results, as exposed in Section 6, the proposed embedded method can be considered a valid alternative to solve problems that have strong discontinuities due to the interaction between a fluid in motion and the surface of a solid object. It can be clearly observed how, without the need of calculating two local systems for the same element, both subdomains can be correctly handled by one single system.

- In a related point, using a customized mix of the static condensation, level-set and extended finite element methods has proved to be a successful strategy in the design of the proposed embedded method for solving the desired type of problems.

- During the development process experience, it could be recognized how the Kratos framework allowed for a transparent and manageable way to incorporate the new implemented element in terms of construction and communication with already existing components, it was not difficult to know which files needed to be modified and the locations where the new files should be placed. However, in terms of providing enough information for facilitating the debugging process, sometimes it was very confusing to understand what and where the error was, turning the debugging phase into one of the phases where the majority of the time for the project was invested.

- During the testing phase experience, it became evident how a good test design is valuable to find errors in the definition of an implementation. During this phase several problems and missing aspects of the proposed embedded method were identified and then solved until the corrected version was complete.

7.2 Recommendations

- To perform a benchmarking and comparison process against other methods for solving this type of problems, so more input about its validity can be obtained.

- To improve the error notification during execution time for the Kratos framework, so debugging processes can be a little more user-friendly.

- To implement parallelization for the element, so the calculation of the models can be done faster and more efficiently.

- To perform further testing with more complex geometries, finer meshes and/or more powerful hardware, to make sure the method gives support to a big variety of cases.

- To expand the method to give support to cases where the strong discontinuity is not enclosed, where it has open ends well defined inside the domain.

- To expand the proposed embedded method to include additional boundary conditions of the form $\boldsymbol{u} \cdot \boldsymbol{n} = 0$, where u is the velocity and n is the normal vector. This is to be applied where found appropriate depending on the problem at hand.

8. GLOSSARY

- **Assembly**: process performed to unify all the local systems for all the elements in a mesh during the application of the Finite Element Method in order to construct the global system for it to be resolved and find the solution.

- **Boundary Condition**: constraints required to be imposed on certain sections of the boundary of a problem so a particular partial differential equation can be solved. This constraint can take usually the form of a prescribed value for the solution or a prescribed flux.

- **C++**: programming language based on the C programming language that allows the implementation of an Object Oriented paradigm, mostly the use of Classes, Interfaces, and Methods.

- **Computational Mechanics**: discipline concerned with the use of computational methods to study phenomena governed by the principles of mechanics. Sometimes considered a sub-discipline of Computational Science.

- **Degree of freedom**: an independent physical parameter in a physical system. Usually the unknown to be found in the system through the application of a particular method.

- **Dirichlet domain**: part of the boundary of the domain in a problem where a finite number of prescribed values for the solutions are to be imposed. Usually represented by Γ_D.

- **Discretization**: process in which a continuous domain where an unknown needs to be calculated exactly at all points is transformed into a net of discrete points where the unknown is to be calculated in an approximate way. The value of the unknown at any other point is to be interpolated.

- **Element**: each of the sections in which a domain is divided after a meshing process. Each of these elements is defined by a set of nodes in the mesh, and they constitute the main element for calculation in the Finite Element Method.

- **Enrichment**: process in which elements cut by an interface between two materials in a single domain have their shape functions modified with a certain function in order to be able to include said discontinuity into the calculation.

- **Extended Finite Element Method**: modification of the Finite Element Method in which elements cut by an interface between two materials in a single domain have their shape functions modified with a certain function called the Enrichment Function. The rest of the process proceeds as normally.

- **Finite Element Method**: method that takes a domain discretized into a mesh where a particular partial differential equation needs to be solved, and for each element constructs a local system for said PDE as if the rest of the domain does not exist; then, it takes all the local systems contribution to construct a global system.

- **Framework**: a universal, reusable software platform to develop software applications, products and solutions. A framework can include programs, compilers, libraries, tool sets, application programming interfaces, and so on.

- **Galerkin Method**: in an application of the Finite Element Method, it is the procedure that uses as a test function in the weighted residual step the same shape functions used to interpolate the solution in each element.

- **Kratos**: a framework for building multi-disciplinary finite element programs. It provides several tools for easy implementation of finite element applications and a common platform providing effortless interaction between them.

- **Lagrange Multiplier**: method used to impose Dirichlet boundary conditions on a problem where a particular partial differential equation needs to be solved, in which the prescribed values are added to the invariant functional of the system through the use of auxiliary variables called the Lagrange multipliers.

- **Level-set Method**: numerical technique for tracking interfaces and shapes for a function through time. Specially used to follow shaped that change topology, like splitting shapes, holes, etc.

- **Mesh**: a domain that has been discretized into a finite set of partitions called elements that are defined and interconnected by nodes, which are the finite set of points that the domain has been discretized into.

- **Neumann domain**: part of the boundary of the domain in a problem where a finite number of prescribed fluxes for the solutions are to be imposed. Usually represented by Γ_N.

- **Node**: set of finite number of points in which a continuous domain has been discretized into, they define each of the partitions of the discretized domain (called elements).

- **Partial Differential Equation**: a differential equation that contains unknown multivariable functions and their partial derivatives. Often used to formulate problems involving functions of several variables.

- **Partition of Unity**: condition that a certain set of values must fulfill, it is described as two aspects: all values must be in the interval [0,1], and the sum of all the values must be equal to 1.

- **Python**: a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in other languages.

- **Shape function**: set of functions used in the finite element method to interpolate the values of the unknown to be found using the nodes in each of the elements of the mesh that discretizes the problem domain.

- **Static condensation**: method used to simplify a system of equations in matrix form in which a part of the variables is absorbed by the rest of the variables. This results in a smaller but equivalent system.

- **Stiffness matrix**: a matrix that represents the system of linear equations that must be solved in order to ascertain an approximate solution to the differential equation in an application of the finite element method.

- **Strong Form**: set of governing partial differential equations with boundary conditions that define and describe a problem to be solved with the finite element method.

- **Test function**: arbitrary function used to multiply the weak form in order to solve the global system by using a weighted residual approach before integrating.

- **Weak Form**: a variational statement of the problem to be solved in a finite element implementation in which the equations are integrated against a test function. The choice of test function is arbitrary and it has the effect of relaxing the problem.

- **Weighted Residual Method**: method for solving differential equations in which the solution is assumed to be well approximated by a function of a particular form having a finite set of degrees of freedom that it depends on. In the finite element method, it is applied by multiplying the weak form of the problem by a test function and then integrating throughout the entire domain.

9. BIBLIOGRAPHY

- Spencer, A. J. M. *Continuous Mechanics*. England, UK: Dover Publications Inc. 2004.

- Zienkiewicz, O. C.; Morgan, K. *Finite Elements & Approximation*. New York, USA: Dover Publications Inc. 2006.

- Anderson Jr., J. D. *Computational Fluid Dynamics*. Singapore: McGraw Hill Inc. 1995.

- Bonet, J; Wood, R. D. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Second Edition. England, UK: Cambridge University Press. 2008.

- Donea, J.; Huerta, A. *Finite Element Methods for Flow Problems*. England, UK: Wiley Editorials. 2003.

- Greenough, C.; Robinson, L. *The Finite Element Library – Theory and Programming Techniques* [online]. England, UK: Chris Greenough, December 2000. [Consulted on: 05/04/2014]. Available on:
  < http://www.softeng.rl.ac.uk/st/projects/felib3/Docs/html/Intro/intro.html>.

- *Kratos, Multi-Physics* [online]. Barcelona, Spain: CIMNE, October 2012. [Consulted on: 20/01/2014]. Available on:
  < http://kratos-wiki.cimne.upc.edu/index.php/Main_Page>.

- *Python Documentation* [online]. USA: Python Software Foundation, 2014. [Consulted on: 17/03/2014]. Available on:
  < https://www.python.org/doc/>

- Dawes, B.; Abrahams, D.; Rivera, R. *boost, C++ libraries* [online]. USA, 2013. [Consulted on: 20/04/2014]. Available on:
  < http://www.boost.org/>.

- LAPACK — Linear Algebra PACKage [online]. USA: Univ. of Tennessee; Univ. of California, Berkeley; Univ. of Colorado Denver; and NAG Ltd. November, 2013. [Consulted on: 12/05/2014]. Available on:
  < http://www.netlib.org/lapack/>.

- Dadvand, P.; Rossi, R.; Oñate, E. "An object-oriented environment for developing finite element codes for multi-disciplinary applications". *Archives of computational methods in engineering* 17-3 (2010), p253-297.

- Osher, S.; Fedkiw, R. P. "Level Set Methods: An Overview and Some Recent Results". *Journal of Computational Physics* 169-2 (2001), p463-502.

- Adalsteinsson, D.; Sethian, J. A. "A Fast Level Set Method for Propagating Interfaces". *Journal of Computational Physics* 118-2 (1995), p269-277.

- Peng, D.; Merriman, B.; Osher, S.; Zhao, H.; Kang, M. "A PDE-Based Fast Local Level Set Method". *Journal of Computational Physics* 155-2 (1999), p410-438.

- Belytschko, T.; Gracie, R. "On XFEM applications to dislocations and interfaces". *International Journal of Plasticity* 23-10 (2007), p1721-1738.

- Yvonnet, J.; Le Quang, H.; "An XFEM/level set approach to modeling surface/interface effects and to computing the size-dependent effective properties of nanocomposites". *Computational Mechanics* 42-1 (2008), p119-131.

- Benson, D. J.; Bazilevs, Y.; De Luycker, E.; Hsu, M.; Hughes, T. J. R.; Belytschko, T. "A generalized finite element formulation for arbitrary basis functions: From isogeometric analysis to XFEM". *International Journal for Numerical Methods in Engineering* 83-6 (2010), p765-785.

- Wilson, E. L. "The static condensation algorithm". *International Journal for Numerical Methods in Engineering* 8-1 (1974), p198-203.

- Belytschko, T. "Fluid-structure interaction". *Computers & Structures* 12-4 (1980), p459-469.

- Franca, L. P.; Frey, S. L.; Hughes, T. J. R. "Stabilized finite element methods: I. Application to the advective-diffusive model". *Computer Methods in Applied Mechanics and Engineering* 95-2 (1992), p253-276.

- Franca, L. P.; Frey, S. L. "Stabilized finite element methods: II. The incompressible Navier-Stokes equations". *Computer Methods in Applied Mechanics and Engineering* 99-2 (1992), p209-233.

ANNEXES

ANNEX A

Code implemented, as described in Section 5.3:

```
virtual void CalculateLocalSystem(MatrixType& rLeftHandSideMatrix,
                                  VectorType& rRightHandSideVector,
                                  ProcessInfo& rCurrentProcessInfo)
{

    const unsigned int LocalSize = (TDim + 1) * TNumNodes;

    //*************************************************
    // Resize and set to zero the RHS
    if(rRightHandSideVector.size() != LocalSize)
        rRightHandSideVector.resize(LocalSize,false);
    noalias(rRightHandSideVector) = ZeroVector(LocalSize);

    Vector rRightHandSideVectorPositive(LocalSize);
    noalias(rRightHandSideVectorPositive) = ZeroVector(LocalSize);

    Vector rRightHandSideVectorNegative(LocalSize);
    noalias(rRightHandSideVectorNegative) = ZeroVector(LocalSize);

    // Resize and set to zero the LHS
    if (rLeftHandSideMatrix.size1() != LocalSize)
        rLeftHandSideMatrix.resize(LocalSize, LocalSize, false);
    noalias(rLeftHandSideMatrix) = ZeroMatrix(LocalSize, LocalSize);

    Matrix rLeftHandSideMatrixPositive(LocalSize, LocalSize);
    noalias(rLeftHandSideMatrixPositive) = ZeroMatrix(LocalSize, LocalSize);

    Matrix rLeftHandSideMatrixNegative(LocalSize, LocalSize);
    noalias(rLeftHandSideMatrixNegative) = ZeroMatrix(LocalSize, LocalSize);

    Matrix MassMatrix(LocalSize,LocalSize,0.0);
    Matrix MassMatrixPositive(LocalSize,LocalSize,0.0);
    Matrix MassMatrixNegative(LocalSize,LocalSize,0.0);

    //Splitting the element
    double Area;
   array_1d<double, TNumNodes> N;
   boost::numeric::ublas::bounded_matrix<double, TNumNodes, TDim> DN_DX;
   GeometryUtils::CalculateGeometryData(this->GetGeometry(), DN_DX, N, Area);


   //input data for enrichment function
   Vector& eltaldistances = this->GetValue(ELEMENTAL_DISTANCES);
    array_1d<double,6> edge_areas;
   Matrix coords(TNumNodes, TDim);

   //output data for enrichment function
   Matrix Nenriched;
   Vector volumes;
   Matrix Ngauss;
   Vector signs(0);
    //signs = ZeroVector(6);
   std::vector< Matrix > gauss_gradients;


   //fill coordinates
   for (unsigned int i = 0; i < TNumNodes; i++)
   {
        const array_1d<double, 3 > & xyz = this->GetGeometry()[i].Coordinates();

        for (unsigned int j = 0; j < TDim; j++)
            coords(i, j) = xyz[j];
   }

    unsigned int ndivisions =
    EnrichmentUtilitiesDuplicateDofs::CalculateTetrahedraEnrichedShapeFuncions(coords,
            DN_DX, eltaldistances, volumes, Ngauss, signs, gauss_gradients, Nenriched,
            edge_areas);
```

```
        bool split_element = this->GetValue(SPLIT_ELEMENT);

        if (split_element == true){

                CalculateLocalSystemAux(rLeftHandSideMatrixPositive,
                        rRightHandSideVectorPositive,MassMatrixPositive,1,rCurrentProcessInfo,
                        ndivisions,Area,volumes,N,DN_DX,Ngauss,signs);
                CalculateLocalSystemAux(rLeftHandSideMatrixNegative,
                        rRightHandSideVectorNegative,MassMatrixNegative,2,rCurrentProcessInfo,
                        ndivisions,Area,volumes,N,DN_DX,Ngauss,signs);

        //Determining signs of the nodes
        Vector positiveOrNegative(TNumNodes);
        for(unsigned int i=0; i< TNumNodes; i++)
                if(eltaldistances[i]>0)
                        positiveOrNegative[i]=1;
                else
                        positiveOrNegative[i]=0;

                condensate(rLeftHandSideMatrix, rLeftHandSideMatrixPositive, rRightHandSideVector,
                        rRightHandSideVectorPositive, positiveOrNegative, 1,
                        eltaldistances, edge_areas);
                condensate(rLeftHandSideMatrix, rLeftHandSideMatrixNegative, rRightHandSideVector,
                        rRightHandSideVectorNegative, positiveOrNegative, 0,
                        eltaldistances, edge_areas);
        }else{
                CalculateLocalSystemAux(rLeftHandSideMatrix,
                        rRightHandSideVector,MassMatrix,0,rCurrentProcessInfo,ndivisions,
                        Area,volumes,N,DN_DX,Ngauss,signs);
                }

    }

void CalculateLocalSystemAux(MatrixType& rLeftHandSideMatrix,VectorType& rRightHandSideVector,
        MatrixType& MassMatrix, const int flag, ProcessInfo& rCurrentProcessInfo,
        const unsigned int ndivisions, const double Area, VectorType& volumes,
        array_1d<double,TNumNodes> N,
        boost::numeric::ublas::bounded_matrix<double, TNumNodes, TDim> DN_DX,
        MatrixType& Ngauss,const VectorType& signs)
        {

        const unsigned int LocalSize = (TDim + 1) * TNumNodes;
         //**************************************************
        //Get Vector of BDF coefficients
        const Vector& BDFVector = rCurrentProcessInfo[BDF_COEFFICIENTS];

         array_1d<double,3> bf(3,0.0);

        double positive_volume = 0.0;
        double negative_volume = 0.0;

        if(ndivisions == 1) //compute gauss points for exact integration of a tetra element
        {
            const GeometryType::IntegrationPointsArrayType& IntegrationPoints =
                this->GetGeometry().IntegrationPoints(GeometryData::GI_GAUSS_2);

            Ngauss = this->GetGeometry().ShapeFunctionsValues(GeometryData::GI_GAUSS_2);

            volumes.resize(IntegrationPoints.size(),false);

            for (unsigned int g = 0; g <
                this->GetGeometry().IntegrationPointsNumber(GeometryData::GI_GAUSS_2); g++)
                volumes[g] = 6.0*Area * IntegrationPoints[g].Weight();

        }

        for (unsigned int igauss = 0; igauss < Ngauss.size1(); igauss++)
        {
            double wGauss = volumes[igauss];


            if(signs[igauss] > 0) //check positive and negative volume
                positive_volume += wGauss;
            else
```

```
        negative_volume += wGauss;
}


const double min_area_ratio = 1e-6;

//***************************************************
//compute LHS and RHS + first part of mass computation
for (unsigned int igauss = 0; igauss < Ngauss.size1(); igauss++)
{

        if(proceder(flag,signs,igauss)){
                //assigning the gauss data
                for (unsigned int k = 0; k < TNumNodes; k++)
                        N[k] = Ngauss(igauss, k);
                double wGauss = volumes[igauss];

                //***************************************************
                // Calculate this element's fluid properties
                double Density, KinViscosity;
                this->EvaluateInPoint(Density, DENSITY, N);
                this->EvaluateInPoint(KinViscosity, VISCOSITY, N);
                this->EvaluateInPoint(bf,BODY_FORCE,N);

                double Viscosity;
                this->GetEffectiveViscosity(Density, KinViscosity, N, DN_DX,
                                                Viscosity, rCurrentProcessInfo);

                this->AddMomentumRHS(rRightHandSideVector, Density, N, wGauss);

                // Get Advective velocity
                array_1d<double, 3 > AdvVel;
                this->GetAdvectiveVel(AdvVel, N);
                // Calculate stabilization parameters
                double TauOne, TauTwo;

                //compute stabilization parameters
                this->CalculateTau(TauOne, TauTwo, AdvVel, Area, Density, Viscosity,
                                                rCurrentProcessInfo);

                this->AddIntegrationPointVelocityContribution(rLeftHandSideMatrix,
                                        rRightHandSideVector, Density, Viscosity, AdvVel,
                                        TauOne, TauTwo, N, DN_DX, wGauss);

                // //compute mass matrix - terms related to real mass
                this->AddConsistentMassMatrixContribution(MassMatrix, N, Density,
                                                        wGauss);

        }

}

//lump mass matrix
this->LumpMassMatrix(MassMatrix);

//add mass matrix stabilization contributions
for (unsigned int igauss = 0; igauss < Ngauss.size1(); igauss++)
{
        if(proceder(flag,signs,igauss)){
        //assigning the gauss data
        for (unsigned int k = 0; k < TNumNodes; k++)
                N[k] = Ngauss(igauss, k);
                double wGauss = volumes[igauss];
                // Calculate this element's fluid properties
                double Density;
                this->EvaluateInPoint(Density, DENSITY, N);

                double KinViscosity;
                this->EvaluateInPoint(KinViscosity, VISCOSITY, N);
                double Viscosity;
                this->GetEffectiveViscosity(Density, KinViscosity, N, DN_DX,
                                                Viscosity, rCurrentProcessInfo);
                // Get Advective velocity
                array_1d<double, 3 > AdvVel;
```

```
                this->GetAdvectiveVel(AdvVel, N);
                // Calculate stabilization parameters
                double TauOne, TauTwo;
                this->CalculateTau(TauOne, TauTwo, AdvVel, Area, Density, Viscosity,
                                                        rCurrentProcessInfo);

                this->AddMassStabTerms(MassMatrix, Density, AdvVel, TauOne, N, DN_DX,
                                                        wGauss);
        }

    }


    //************************************************
    //consider contributions of mass to LHS and RHS
    //add Mass Matrix to the LHS with the correct coefficient
    noalias(rLeftHandSideMatrix) += BDFVector[0]*MassMatrix;
    //do RHS -= MassMatrix*(BDFVector[1]*un + BDFVector[2]*u_(n-1))
    //note that the term related to BDFVector[0] is included in the LHS
    array_1d<double,LocalSize> aaa = ZeroVector(LocalSize);
    for(unsigned int k = 0; k<TNumNodes; k++)
    {
        unsigned int base=k*(TDim+1);
        for(unsigned int step=1; step<BDFVector.size(); step++)
        {
            const array_1d<double,3>& u =
                            this->GetGeometry()[k].FastGetSolutionStepValue(VELOCITY,step);
            const double& bdf_coeff = BDFVector[step];
            aaa[base]   += bdf_coeff*u[0];
            aaa[base+1] += bdf_coeff*u[1];
            aaa[base+2] += bdf_coeff*u[2];
        }
    }
    noalias(rRightHandSideVector) -= prod(MassMatrix,aaa);


    //************************************************
    //finalize computation of the residual
    // Now calculate an additional contribution to the residual: r -= rLeftHandSideMatrix * (u,p)
    VectorType U = ZeroVector(LocalSize);
    int LocalIndex = 0;
    for (unsigned int iNode = 0; iNode < TNumNodes; ++iNode)
    {
        array_1d< double, 3 > & rVel =
                    this->GetGeometry()[iNode].FastGetSolutionStepValue(VELOCITY);
        for (unsigned int d = 0; d < TDim; ++d) // Velocity Dofs
        {
            U[LocalIndex] = rVel[d];
            ++LocalIndex;
        }
        U[LocalIndex] = this->GetGeometry()[iNode].FastGetSolutionStepValue(PRESSURE);
       ++LocalIndex;
    }
    noalias(rRightHandSideVector) -= prod(rLeftHandSideMatrix, U);

    }

    //CONDENSATION FUNCTION
    void condensate(MatrixType& LHSMatrixT, MatrixType& LHSMatrix, VectorType& RHSVectorT,
            VectorType& RHSVector, VectorType& pon, int ponflag, VectorType& eltaldistances,
            array_1d<double,6>& edge_areas)
{

    //Determining the nodes to use
    int count=0;
    int count2=0;
    Vector in(TNumNodes);
    Vector out(TNumNodes);
    for(unsigned int i=0; i<TNumNodes;i++)
            if(pon[i]==ponflag){
                    in[count]=i;
                    count++;
            }else{
                    out[count2]=i;
```

```
                count2++;
        }

//Determining Ns
Matrix N(count2,TNumNodes);
noalias(N)=ZeroMatrix(count2,TNumNodes);
//loop over edges
double NI,NJ;
unsigned int edge_counter;

for(unsigned int lamb = 0; lamb < count2; lamb++){
        NI=0;
        NJ=0;
        edge_counter = 0;
        for(unsigned int i=0; i<TNumNodes; i++)
        {
                for(unsigned int j=i+1; j<TNumNodes; j++){
                        if(eltaldistances[i]*eltaldistances[j] < 0){ //edge is cut
                                if(i == out[lamb] || j == out[lamb]){
                                        NI = fabs(eltaldistances[j]) /
                                                        fabs(eltaldistances[i]) +
                                                        fabs(eltaldistances[j]));
                                        NJ = 1 -NI;
                                        N(lamb,i) += NI * edge_areas[edge_counter];
                                        N(lamb,j) += NJ * edge_areas[edge_counter];
                                }
                        }
                        edge_counter++;
                }
        }
}

//Preparing containers
unsigned int k=0; unsigned int l=0;
Matrix reducedMatrix(count*4,count*4);
noalias(reducedMatrix)=ZeroMatrix(count*4,count*4);
unsigned int m=0; unsigned int n=0;
Matrix residualMatrix((TNumNodes-count)*4+3*count2,(TNumNodes-count)*4+3*count2);
noalias(residualMatrix)=ZeroMatrix((TNumNodes-count)*4+3*count2,(TNumNodes-count)*4+3*count2);
Matrix residualMatrixInv((TNumNodes-count)*4+3*count2,(TNumNodes-count)*4+3*count2);
unsigned int o=0; unsigned int p=0;
Matrix additionalMatrixI(count*4,(TNumNodes-count)*4+3*count2);
noalias(additionalMatrixI)=ZeroMatrix(count*4,(TNumNodes-count)*4+3*count2);
unsigned int q=0; unsigned int r=0;
Matrix additionalMatrixII((TNumNodes-count)*4+3*count2,count*4);
noalias(additionalMatrixII)=ZeroMatrix((TNumNodes-count)*4+3*count2,count*4);

Vector rhr(4*count);
Vector rhres(4*count2+3*count2);
rhr=ZeroVector(4*count);
rhres=ZeroVector(4*count2+3*count2);


//Creating the sections

for(unsigned int i=0; i < TNumNodes; i++)
        for(unsigned int j=0; j < TNumNodes; j++){
                if(pon[i]==ponflag && pon[j]==ponflag){
                        for(unsigned int a=0; a<4 ; a++)
                                for(unsigned int b=0; b<4 ; b++)

                                        reducedMatrix(4*k+a,4*l+b)=LHSMatrix(4*i+a,4*j+b);
                                        l++;
                                        if(!(l<count)){l=0; k++;}
                        }
                        if(pon[i]!=ponflag && pon[j]!=ponflag){
                                for(unsigned int a=0; a<4 ; a++)
                                        for(unsigned int b=0; b<4 ; b++)

                                        residualMatrix(4*m+a,4*n+b)=LHSMatrix(4*i+a,4*j+b);
                                        n++;
                                        if(!(n<(TNumNodes-count))){n=0; m++;}
                        }
                        if(pon[i]==ponflag && pon[j]!=ponflag){
```

```
                        for(unsigned int a=0; a<4 ; a++)
                                for(unsigned int b=0; b<4 ; b++)
                                                        7
                                        additionalMatrixI(4*o+a,4*p+b)=LHSMatrix(4*i+a,4*j+b);
                                        p++;
                                        if(!(p<(TNumNodes-count))){p=0; o++;}
                        }
                        if(pon[i]!=ponflag && pon[j]==ponflag){
                                for(unsigned int a=0; a<4 ; a++)
                                        for(unsigned int b=0; b<4 ; b++)

                                        additionalMatrixII(4*q+a,4*r+b)=LHSMatrix(4*i+a,4*j+b);
                                        r++;
                                        if(!(r<count)){r=0; q++;}
                        }
                }

        for(unsigned int i=0; i<count; i++){
                int index=in[i];
                rhr[4*i]=RHSVector[4*index];
                rhr[4*i+1]=RHSVector[4*index+1];
                rhr[4*i+2]=RHSVector[4*index+2];
                rhr[4*i+3]=RHSVector[4*index+3];
        }
        for(unsigned int i=0; i<count2; i++){
                int index=out[i];
                rhres[4*i]=RHSVector[4*index];
                rhres[4*i+1]=RHSVector[4*index+1];
                rhres[4*i+2]=RHSVector[4*index+2];
                rhres[4*i+3]=RHSVector[4*index+3];
        }

        //Adding the stabilization sections
        Matrix Mb(TNumNodes,TNumNodes+1);
        Matrix MbT(TNumNodes+1,TNumNodes);
        noalias(Mb)=ZeroMatrix(TNumNodes,TNumNodes+1);
        for(unsigned int i=0; i<TNumNodes; i++)
                for(unsigned int j=0; j<TNumNodes+1; j++)
                        if(i==j) Mb(i,j)=1;
        noalias(MbT)=trans(Mb);

        for(unsigned int lamb=0; lamb < count2; lamb++){
                for(unsigned int i=0; i<count; i++){
                        int j2=0,index=0;
                        for(unsigned int j=(TNumNodes-count)*4+3*lamb;
                                        j< (TNumNodes-count)*4+3+3*lamb; j++){
                                j2=j-(TNumNodes-count)*4-3*lamb; index=in[i];
                                additionalMatrixI(4*i,j)=N(lamb,index)*MbT(0,j2);
                                additionalMatrixI(4*i+1,j)=N(lamb,index)*MbT(1,j2);
                                additionalMatrixI(4*i+2,j)=N(lamb,index)*MbT(2,j2);
                                additionalMatrixI(4*i+3,j)=N(lamb,index)*MbT(3,j2);
                        }
                }
                for(unsigned int j=0; j<count; j++){
                        int i2=0,index=0;
                        for(unsigned int i=(TNumNodes-count)*4+3*lamb;
                                        i< (TNumNodes-count)*4+3+3*lamb; i++){
                                i2=i-(TNumNodes-count)*4-3*lamb; index=in[j];
                                additionalMatrixII(i,4*j)=N(lamb,index)*Mb(i2,0);
                                additionalMatrixII(i,4*j+1)=N(lamb,index)*Mb(i2,1);
                                additionalMatrixII(i,4*j+2)=N(lamb,index)*Mb(i2,2);
                                additionalMatrixII(i,4*j+3)=N(lamb,index)*Mb(i2,3);
                        }
                }
                for(unsigned int a=0; a<count2; a++){
                        int b2=0,index=0;
                        for(unsigned int b=(TNumNodes-count)*4+3*lamb;
                                                b< (TNumNodes-count)*4+3+3*lamb; b++){
                                b2=b-(TNumNodes-count)*4-3*lamb; index=out[a];
                                residualMatrix(4*a,b)=N(lamb,index)*MbT(0,b2);
                                residualMatrix(4*a+1,b)=N(lamb,index)*MbT(1,b2);
                                residualMatrix(4*a+2,b)=N(lamb,index)*MbT(2,b2);
                                residualMatrix(4*a+3,b)=N(lamb,index)*MbT(3,b2);
```

```
                        residualMatrix(b,4*a)=N(lamb,index)*Mb(b2,0);
                        residualMatrix(b,4*a+1)=N(lamb,index)*Mb(b2,1);
                        residualMatrix(b,4*a+2)=N(lamb,index)*Mb(b2,2);
                        residualMatrix(b,4*a+3)=N(lamb,index)*Mb(b2,3);
                }
        }
}


//Substracting "constraint residual" from the RHSVector
VectorType U = ZeroVector(4*TNumNodes);
VectorType UUP = ZeroVector(4*TNumNodes);
int LocalIndex = 0;
for (unsigned int iNode = 0; iNode < TNumNodes; ++iNode)
{
        array_1d< double, 3 > & rVel =
        this->GetGeometry()[iNode].FastGetSolutionStepValue(VELOCITY);
        for (unsigned int d = 0; d < 3; ++d) // Velocity Dofs
        {
                U[LocalIndex] = rVel[d];
                ++LocalIndex;
        }
       U[LocalIndex] = this->GetGeometry()[iNode].FastGetSolutionStepValue(PRESSURE);
       ++LocalIndex;
}
for(unsigned int i = 0; i < count; i++){
        unsigned int index = in[i];
        UUP[4*i] = U[4*index]; UUP[4*i+1] = U[4*index+1];
        UUP[4*i+2] = U[4*index+2]; UUP[4*i+3] = U[4*index+3];
}

Matrix H(3*count2,4*TNumNodes);
Vector newR(3*count2);
for(unsigned int i = 4*count2; i < 4*count2 + 3*count2; i++)
        for(unsigned int j = 0; j < 4*count; j++)
                H(i-4*count2,j) = additionalMatrixII(i,j);
for(unsigned int i = 4*count2; i < 4*count2 + 3*count2; i++)
        for(unsigned int j = 0; j < 4*count2; j++)
                H(i-4*count2,j+4*count) = residualMatrix(i,j);

noalias(newR) = prod(H,UUP);

for(unsigned int i = 4*count2; i < 4*count2 + 3*count2; i++)
        rhres[i] -= newR[i - 4*count2];



//Operating
InvertMatrix(residualMatrix,residualMatrixInv);
Matrix tmp,othertmp;

tmp = prod(residualMatrixInv,additionalMatrixII);
othertmp = prod(additionalMatrixI,tmp);
noalias(reducedMatrix)-= othertmp;
Vector tmp2;
tmp2 = prod(residualMatrixInv,rhres);
noalias(rhr) -= prod(additionalMatrixI,tmp2);

//Storing results
k=0; l=0;
for(unsigned int i=0; i < TNumNodes; i++)
        for(unsigned int j=0; j < TNumNodes; j++){
                if(pon[i]==ponflag && pon[j]==ponflag){
                        for(unsigned int a=0; a<4 ; a++)
                                for(unsigned int b=0; b<4 ; b++)

                                LHSMatrixT(4*i+a,4*j+b)+=reducedMatrix(4*k+a,4*l+b);
                                l++;
                                if(!(l<count)){l=0; k++;}
                }
        }

int index=0;
for(unsigned int i=0; i< TNumNodes; i++)
```

```
                if(pon[i]==ponflag){
                        RHSVectorT[4*i]+=rhr[4*index];
                        RHSVectorT[4*i+1]+=rhr[4*index+1];
                        RHSVectorT[4*i+2]+=rhr[4*index+2];
                        RHSVectorT[4*i+3]+=rhr[4*index+3];
                        index++;
                }
}


bool proceder(const int flag,const Vector& signs,const int igauss){
        if(flag==0) return true;
        if(flag==1 && signs[igauss]>0) return true;
        if(flag==2 && signs[igauss]<0) return true;
        return false;
}
```