Escola de Camins
Escola Tècnica Superior d'Enginyeria de Camins, Canals i Ports
UPC BARCELONATECH

TREBALL FINAL DE MÀSTER

Developement of a compressible solver for the simulation of explosions in electric transformers

Treball realitzat per:
Pau Márquez Martín

Dirigit per:
Dr. Riccardo Rossi
Dr. Rubén Zorrilla

Màster en:
Enginyeria en Mètodes Numèrics

Barcelona, 21 de juny del 2021

Departament d'Enginyeria Civil i Ambiental

Acknowledgements

I would most sincerely like to thank Professor Rosi and Doctor Zorrilla for the opportunity they provided me with. Working with CIMNE in collaboration with Siemens has been a remarkable professional experience. During the passing of the time I grew in appreciation for the support I have been given by them and the other members of the Kratos team. With their help I have studied thoroughly the topic of numerical simulation, and I will keep doing so.

Additional gratitude is due to Dr. Zorrilla for having provided me with his Master's Thesis document, template and .tex file, which saved me a lot of time and work.

Last but not least, I would also like to acknowledge the contributions of all those people that, at some point, have helped me disinterestedly with the progress of this work.

Abstract

The transformer manufacturing sector often suffers explosions originated in the oil-filled tanks due to arcing faults that generate flammable gas as the oil surrounding the electrical arc vaporizes. In a matter of milliseconds, the volume containing the gas is pressurized as the oil inertia prevents it from growing and reducing its pressure. This phenomena leads to the formation of pressure waves due to the strong pressure gradient at the oil-gas interface, which propagate and interact with the transformer structure. The tank walls may sustain the first pressure peak, of dynamic nature, which tends to be the highest in magnitude, but has a short period. On the contrary, the static pressure which builds up in the tank as the reflecting pressure waves interact with the incoming waves may generate the highest hazard and lead to explosions, tank rupture, pollution and expensive human and material costs.

In this work, a weakly-compressible multi-fluid flow formulation simulates with the Finite Element implementation of the Navier-Stokes equations the phenomena detailed above in order to have a proper understanding of the physical conditions in the tank and devise adequate depressurization strategies.

This thesis will show the capabilities of the code, developed by reusing existing code from the `FluidDynamicsApplication` of Kratos Multiphysics code, in order to ease the designing steps of the transformer geometries, and localize critical points which may contribute to amplify the effect of the pressurized bubbles.

An equation-of-state model, together with a constitutive law and a customized solver capable of dealing with the convection of the bubble interface will be developed. Moreover, the capability of simulating the elastic response of the walls by means of a partitioned fluid-structure interaction was also requested. As it will be seen, the nature of the problem will imply a strong coupling given the large displacements that may be produced in

the structure in a single time-step.

This thesis will be characterized by high computational cost simulations, due to the necessity to refine the mesh on the interface region. For that reason, a few significant problems have been chosen to be solved in order to assess the performance of the implemented strategies. However, exact reference problems in literature are not available given the uncommon quasi-incompressible methodology chosen to solve the problem.

Resum

El sector dels transformadors elèctrics està afectat per possibles explosions en l'interior dels tancs plens d'oli que aïllen els components elèctrics a causa de malfuncionaments elèctrics que generen un gas inflamable a mesure que l'oli que envolta l'arc elèctric es vaporitza. En questió de mili-segons, el volum de gas és pressuritzat donat que la inèrcia del fluid que l'envolta prevé la bombolla d'incrementar el seu volum i reduir-ne la pressió. Aquest fenòmen comporta la formació d'ones de pressió a causa del fort gradient de pressió a la interfície entre el gas i el líquid, que es propaguen i interaccionen amb l'estructura del transformador. Les parets del tanc poden suportar el primer pic de pressió, de caràcter dinàmic, que tendeix a ser el més gran en magnitud però de curt període. Tot i això, la pressió estàtica que es genera en el tanc a mesura que les ones de pressió interaccionen amb les ones emergents de la bombolla és la més perillosa i la que pot generar explosions, trencament del tanc, contaminació i costs tant materials com humans.

En aquesta tesis, una formulació multi-fluid i quasi-incompressible simula a través de la implementació de les equacions de Navier-Stokes amb el mètode d'Elements Finits el fenòmen detallat per tal de tenir coneixement i entendre les condicions físiques dins del tanc i implementar estratègies de depressurització adequades.

Aquesta tesis mostrarà les capacitats del codi desenvolupat a partir de la reutilització d'altres fragments de codi ja existents a l'aplicació `FluidDynamicsApplication` de Kratos Multiphysics, amb l'objectiu de facilitar les etapes de disseny de la geometria dels transformadors elèctrics, i localitzar els punts crítics que puguin contribuir a amplificar l'efecte de les bombolles de gas.

Un model d'equació d'estat, juntament amb una llei constitutiva i un *solver* adaptat amb la possibilitat d'incloure la convecció de la interfície de la bombolla seran desenvolupats.

A més a més, la possibilitat de simular la resposta elàstica de les parets del tanc a partir de la interacció fluid-estructura també es va incloure entre els objectius del treball. Com es veurà, la naturalesa del problema imposarà fortes restriccions a l'acoblament donats els grans desplaçaments que es poden produir en l'estructura en espais molt curts de temps.

Les simulacions desenvolupades estan caracteritzades per un elevat cost computacional, donada la necessitat de refinar la malla en la zona de grans gradients de pressió i en l'entorn de la bombolla. Per aquest motiu, una sèrie de problemes significatius s'han seleccionat per a ser resolts i mostrats per tal d'avaluar l'eficiència de les estratègies implementades. Tot i això, no es mostraran comparacions exactes amb problemes ja resolts ja que aquests no estan disponibles donada la metodologia escollida per resoldre el problema.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this thesis is on simulating pressure waves in a compressible fluid flow characterized by the effect of a gas-liquid interface in a transformer tank. This interface separates a liquid from a highly pressurized gas product of the evaporation of a small volume of liquid. Yet, it is not the goal of this investigation to model phase changes, but rather how the multi-phase flow may interact with the surrounding structures by the generation of pressure waves.

Next, a motivation for carrying out this work is stated that remarks the importance of studying transformer tank explosions and their numerical treatment in research and industrial applications.

Then, a general overview over the prerequisites of numerical simulations is given. The different steps of a numerical simulation are clarified in general and for the simulation of multi-phase flows in particular. Afterwards, an introduction to the numerical simulation of two-phase flows is given, including references to already existing work in the literature. Finally, the objectives of this work are listed and an outlook on the structure of this thesis is given.

1.1 Motivation

It is well known that electrical power systems have been deployed all over the world as a natural consequence of the increasing demand on electrical energy but also as a success product of the development of electrical devices such as transformers. As long as renewable solar energy does not constitute the source of electrical energy generation

as a key action against the impact of climate change, transformers will be needed to enable transmission lines carry electricity at high AC voltages from nuclear or thermal power plants to city substations. The voltage conversion process requires of the use of insulating fluids to absorb the heat from the electrical resistance [16]. This fluid will be referred to as an insulant or dielectric liquid, and in most modern electrical transformers, mineral oil derived from petroleum crude oil is used. Therefore, the oil has the purpose of both absorbing heat and insulating the transformer tank from sparks between the high voltage windings. An image of a reference transformer is given in Fig. 1.1.

Figure 1.1: Real transformer and tank. Courtesy of Siemens Energy.

Oil-filled transformers are regarded as being safely operational devices and explosion events or other catastrophic failures are highly uncommon [16]. However, a series of events may trigger an increase the temperature in the tank, its most usual causes being overcharging or lightning strikes. When this happens, an excessive electrical current is transmitted into the transformer, which generates enough heat to increase the oil's temperature. In such circumstances, when the temperature oscillates between 150 and 300^{o}C the hydrocarbon compounds found in the oil are decomposed into hydrogen and methane, which generate a flammable vapour. These vapour tends to dissolve partially or entirely in the mineral oil [20], but if sufficient amount of gas is generated, the overpressure inside the sealed tank may lead to its rupture, which would not only pose a severe environmental problem due to the toxicity of the leaked oil-gas mixture but also a hazardous situation since the contact with any high-voltage component could ignite the gas and generate an explosive event.

Although the chances of such events to occur are very low, cases have been reported of leaking oil due to gasketing, cracked insulation or loose manhole covers [19], which may lead to environmental pollution. However it has been reported that 70% to 80% of transformer failures are due to internal winding insulation failure [18], which may trigger the creation of sparks from the electrical coil and the subsequent formation of gas bubbles and overcharge of the transformer tank. The latter occurs for two reasons, the first being unsuitable dielectric properties of the insulant, which favour the generation of electric currents through the oil, and the second the liquid inertia preventing the expansion of the generated gas, which leads to its pressurization and the subsequent propagation of the dynamic pressure peak and its interaction with the tank walls [6].

This problematic does not only compromise the structural integrity of the tank but also increases the possibility of an explosion. As mentioned, cases have been reported of explosions affecting neighbouring transformers, with oil fires that can last up to 28 hours and thermal radiation with oil temperatures from 960 to 1200^{o}C (Fig. 1.2). Unfortunately, these cases also jeopardize human security, as probably the worst transformer accident occurred in a a coalmine in western Turkey in 2014 certifies [21]. An electrical fault resulted in a transformer explosion and a fire. More than 200 people were killed in the disaster, and 80 were injured.

Figure 1.2: Projected chimney due to an internal fault. A chimney in its normal configuration can be seen on the right in the background. Extracted from [12].

This information underlines the consequences of the tank explosions, and emphasize the necessity to study the behavior of the fluids when over-pressure occurs, with the ultimate analysis of transformer protection systems and and the advantages of their operation.

In front of this scenario, investigations on the response of transformer tanks and their

design in the presence of a potential gas bubble appearing in certain points of the geometry need to be undertaken. Some authors have conducted experimental results to validate their own models [27], but this is dangerous, cumbersome and expensive at the same time. Moreover, another reason that advises against such an approach is the fact that it is complicated to measure the pressure distribution inside gas bubbles and it is also hard to determine an internal velocity distribution there [5]. The natural alternative are the numerical methods, which introduce a physical model to simulate the real phenomena.

As will be seen, the gas-liquid treatment will play an important role when constructing the numerical model. Many physical phenomena require of the use of multi-phase flows and its use has become extensive in applications of diverse kinds. Just to mention some gas-liquid examples, the motion undergone by rain drops in air is studied in meteorology, whereas cavitation is a common circumstance when studying aerodynamic behavior of hydro turbine airfoils. This phenomena occurs when a liquid flowing over a surface generates vapor bubbles due to a reduction of its pressure, and will grow as long as they remain in the low pressure region. The generation of such vapor bubbles and, by extension, its collapse at some point on the surface might indeed cause structural damage and even failure of the materials, apart from the fact that the resulting pressure distribution is such as to cause an increase in drag and a lift loss in airfoils.

There is clearly a strong motivation to employing numerical approaches to resolve the pressure gradients around the bubble interface, since this particular problem would be incapable of solution without them, at least in terms of level of detail. Its usage has recently been intensified in industry and research with the development of the computer science, which allows reducing experimental investigations to longer and often unnecessary processes except if motivated by the model validation. This thesis will be devoted to fully numerical modelling, since no experimental tests will be done. However, work by others will serve as comparison.

The long and short of it is that some companies have been required to invest resources in order to prevent transformer explosions. Specifically, transformer manufacturers have focused on improving performance of their designs, and a tool that enables them to quickly detect possible conflicting parts in the geometry that could give rise to an overpressure and possible rupture of the tank would be helpful. This tool would also facilitate the process of setting up a complete simulation in any CFD program, since this is a cumbersome process.

1.2 Numerical simulation

The workflow characterizing a numerical simulation consists of several steps that allow the transition from an observable phenomena to the visualization of the results that model its nature. Each step introduces simplifications and errors in some degree and its assessment will be key to validating the results.

The first step would be choosing an appropriate mathematical model and the corresponding set of equations. The present case requires equations to be posed in up to three-dimensional domains and capable of capturing discontinuities across an interface. For this reason, a set of nonlinear equations such as the viscous Navier-Stokes or the Euler equations would be a suitable choice, with the obvious inconvenient that they require a numerical method for their resolution, thus lacking analytical solutions. In fact, only simplified sets of Partial Differential Equations within geometrically trivial boundaries offer analytical solutions.

The second step is thus clear, that is, the need to discretize the domain on a computational mesh. The object of the discretization will be the Finite Element, instead of other popular choices such as Finite Differences or Finite Volumes. This forces the appearance of discrete values for the main quantities, which will be defined on the nodes, and by means of a linear interpolation the result is mapped onto the element. The continuity in time will also be replaced by a discretization, given the clear unsteady nature of the problem.

The last step of the workflow is the running of the simulation and post-processing the results, where the results are assessed according to the simplifications and assumptions made in the last steps.

1.3 Objectives

The main objective of this work is the development of a compressible multi-phase solver in Kratos Multiphysics framework that is able to accurately simulate the propagation and interaction of pressure waves in domains up to three-dimensional. The next topics will be studied:

- The development of a weakly-compressible Navier-Stokes formulation starting from an existing quasi-incompressible element. The symbolic and automatic differentiation and stabilization will be carried out by considering density and sound

velocity as nodal variables, thus allowing these quantities to change as a function of space and time.

- The implementation of a shock capturing method and the corresponding development of a new constitutive law that allows artificial viscosities.
- The creation of an equation-of-state process that allows selecting the Tait equation for the density and sound velocity calculations at every time step.
- The derivation of a new solver starting from the Two-Fluid solver that incorporates the particularities of the new element and addresses the computation of a pressure-based method with a level-set approach for two-phase flows.
- The coupling of the new features in the Fluid Dynamics Applications with the Structural Mechanics Application, so that the deformation and stresses on the tank walls may be computed.

These objectives imply dedication into each of the phases of the numerical workflow, starting from the mathematical model derived for the compressible approach, to the numerical methods implemented for the resolution of the interface problem. Eventually, the overall method is assessed with 2D and 3D numerical simulations with respect to single and two-phase problems.

1.4 Contents

In the following lines, the contents of the present document are briefly depicted.

Chapter 2 contains a review on the State of the Art regarding the use of numerical methods for compressible multi-phase flows and for tracking the evolution of a moving interface. It also explains the literature trends when approaching Fluid-Structure interaction and also the techniques required to update the mesh.

Chapter 3 serves as an introduction to the mathematical treatment of the problem, a general overview is given on the boundary conditions and the flow field particularities. The used frames of references and governing equations are presented in a general way. Eventually, the mathematical development of the phenomena that will appear in the methodology is presented, so that the reader can be used to the formulation and derivations employed.

Chapter 4 collects the methodology employed to solve the proposed objectives. In it, a detailed derivation of the stabilized Finite Element formulation is given, together with the chosen techniques to track the interface and capture the pressure shocks.

Eventually, chapter 5 summarizes the results of all the tests performed, including validation tests.

Finally, chapter 6 states the conclusions and the future work lines.

Chapter 2

Literature review

This chapter is aimed to be a review on the state of the art of the propagation of pressure waves in a multi-phase flow environment. The chapter has been divided into four parts. The first and second sections approach the two main phenomena associated with the formation of a pressurized gas bubble, meaning pressure wave propagation and bubble motion. While the first one describes the compressible treatment of multi-phase flows, and the difficulties arising from using numerical methods, the second studies the interface between two compressible immiscible fluids. A description will be given in the third section on fluid-structure interaction methods found in literature to work out this simulation, and in the fourth a summary of mesh-updating techniques.

2.1 Numerical methods for compressible multi-phase flows

The study of a pressure wave propagating through a fluid proceeds by assuming a physical and mathematical model able to describe the flow. This is not straightforward, since each model will present assumptions and limitations that need to be considered when interpreting the results.

Many studies have been carried out regarding the topic of numerical models for fluid dynamics, encompassing different approaches to enhancing the understanding of complex flow problems. Many research and industrial areas, such as aerospace or automotive, have placed interest on the development of techniques for computing the propagation of shock waves. This implies the introduction of a density variation in the spatial domain

and compressibility effects so that jumps in the material properties can be obtained through the solution of the motion equations, in contrary to what would be obtained by an incompressible approach.

In the general case, it is considered a wave propagating in a fluid which is both viscous and heat-conducting. This is clearly a complex flow problem in which there is a strong transient phenomena. This problem can be undertaken by using the Navier-Stokes or the Euler equations, where the latter typically exclude viscous effects and heat conduction since shock wave dynamics are assumed to have a predominant effect over viscosity and heat conduction. The fact that it is heat-conducting would imply that the pressure is no longer a function of the density alone, but of the temperature as well. This naturally arises if compressibility is assumed, where the thermodynamics description of the fluid is included through coupling the energy equation to the mass and momentum, and closing the system by adding an equation of state, relating the pressure to the temperature and the density. The incompressible approach allows decoupling the energy equation from the others and thus there is no need to use an equation of state, since the pressure does not have a thermodynamic meaning. In fact, by using an incompressible approach the pressure waves are forced to propagate at infinite speed and cannot be described in this manner. In addition, the incompressibility condition would add another source of numerical difficulty since the pressure takes the role of fulfilling the divergence-free condition.

Considering these aspects, it is clear that all the models investigated need to be derived from a compressible theory. The following subsections define the main models found in literature, which, in some extent, explore the capabilities of the Navier-Stokes, Euler and acoustic theory models to physically describe how to compute the propagation of pressure waves from the oil-gas interface.

2.1.1 The seven-equation model

The most complete method found in literature, in which both gas and liquid phases are considered compressible and share equal values for pressure and velocity only at a single point defined at the interface, is extensive in literature and was first devised by Baer and Nunziato in the field of detonation. The model is developed as a combination of any set of seven independent equations from the mass, momentum and energy balance for each phase. This model accurately computes the pressure wave propagation within fluid-gas flows considering gravitational, viscous and thermal effects. Some of the equations are

concerned with one of the phases and others refer to the mixture subject to the volume fractions of each phase, so that for instance, the mixture quantities are

$$\begin{cases} \rho = \rho_f \alpha_f + \rho_g \alpha_g \\ \mathbf{u} = \alpha_f \frac{\rho_f}{\rho} \mathbf{u}_f + \alpha_g \frac{\rho_g}{\rho} \mathbf{u}_g \\ p = p_f \alpha_f + p_g \alpha_g \end{cases} \tag{2.1}$$

Being α_f and α_g are the volume fractions of fluid and gas, respectively, subject to the saturation constraint $\alpha_f + \alpha_g = 1$.

The seven set of PDEs chosen, for instance, in [7] (financed by an advanced company solving complex 3D transformer geometries using finite volumes on unstructured meshes) are:

$$\frac{\partial \alpha_g}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla} \alpha_g = 0 \tag{2.2a}$$

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u}) = 0 \tag{2.2b}$$

$$\frac{\partial (\alpha_g \rho_g)}{\partial t} + \boldsymbol{\nabla} \cdot (\alpha_g \rho_g \mathbf{u}) = 0 \tag{2.2c}$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u} : \mathbf{u} + p\mathbf{I}) = \Phi_g^u + \Phi_\mu^u \tag{2.2d}$$

$$\frac{\partial E}{\partial t} + \boldsymbol{\nabla} \cdot (E + p)\mathbf{u} = \Phi_g^E + \Phi_\mu^E + \Phi_T^E \tag{2.2e}$$

These equations describe numerically and theoretically the global phenomena. g refers to gravitational effects, μ to viscous and T to thermal. Equations of state are also required for each phase to give consistency to the thermodynamic closure.

2.1.2 Compressible non-viscous method

The field of underwater explosions is a common engineering problem in which the gas bubble formed generates an initial shock wave propagating at fast speeds (approximately 1 m within 500 microseconds) [9]. For this reason, the Euler equations may be used without loss of accuracy to describe the fluid as non-viscous, non-heating and compressible. The specific form of the Euler equations, extracted from [9] is

$$\mathbf{U}_t + \mathbf{F}_x + \mathbf{G}_y + \mathbf{H}_z = 0 \tag{2.3}$$

in which

$$\mathbf{U} = (\rho \quad \rho u \quad \rho v \quad \rho w \quad E)^T \tag{2.4a}$$

$$\mathbf{F} = (\rho u \quad \rho u^2 + p \quad \rho uv \quad \rho uw \quad u(E+p))^T \tag{2.4b}$$

$$\mathbf{G} = (\rho v \quad \rho vu \quad \rho v^2 + p \quad \rho vw \quad v(E+p))^T \tag{2.4c}$$

$$\mathbf{H} = (\rho w \quad \rho wu \quad \rho wv \quad \rho w^2 + p \quad w(E+p))^T \tag{2.4d}$$

$$E = \rho e + \frac{1}{2}\rho(u^2 + v^2 + w^2) \tag{2.4e}$$

Where $\mathbf{u} = (u, v, w)$ are the velocity components, $\mathbf{U}$ is the matrix form of the state variables (varying along time), $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{H}$ are the matrices of the numerical fluxes along the indicated axis directions. E represents the total energy of a unit volume and e is the specific total energy. Then, as usual, an equation of state closes the system of equations for both faces. The choice of equation is important, as if the pressure and internal energy are only a function of density (and not temperature) in the fluid, the energy equation is not necessary to be solved for the liquid phase. The discretization of the system (2.4) is done in [9] by using the Total Variation Diminishing scheme.

It is important to consider that this method is devised for the underwater explosions' field, which presents large pressure ratios between gas/oil (order of 1000 or more). The tank explosion problem does not typically present such large ratios. In the same paper, the level-set method is used to track the interface and the Real Ghost Fluid Method is chosen to compute the variables at the interface by means of ghost points, thus allowing such high ratios between gas and oil without leading to divergence.

2.1.3 Lagrangian weakly-compressible model

In [33], a monolithic Lagrangian approach is used for the fluid domain with the addition of a slight fluid compressibility, which relates the continuity equation with the mechanical pressure. A local volume variation is assumed instead of null velocity-field diverge in such a way that thermal effects are not assumed to have an important influence, therefore the energy conservation equation remains uncoupled. A complete development of this approach is shown in Appendix A, where the eventual matrix system resulting from this model is shown. Moreover, the same resulting equation are derived analytically.

2.1.4 Transient pressure acoustic model

Some authors also face the problem of studying the transient behavior of tank overpressure by a single acoustic equation, which may then be coupled with a mechanical code. This approach considers viscous attenuation due to the insulating oil, which prevents the gas bubble from expanding due to its inertia. Therefore, the equation governing the propagation and superposition of pressure waves described in [38] is

$$\frac{1}{\rho c^2}\frac{\partial p_t}{\partial t} - \boldsymbol{\nabla}\left[\frac{\boldsymbol{\nabla} p_t}{\rho} - \frac{1}{\rho c^2}\left(\frac{4\mu}{3} + \mu_B\right)\frac{\partial \boldsymbol{\nabla} p_t}{\partial t}\right] = 0 \tag{2.5}$$

where ρ refers to the density, c denotes the sound speed, p_t is the pressure field, μ the dynamic viscosity and the bulk viscosity μ_B. In [38] this equation is discretized with 3D FEM and coupled with a tank wall mechanical model.

2.2 Numerical methods to describe the bubble motion

The generation of an electrical arc in a tank full of oil causes a complex sequence of physical events to occur. An internal fault electric model has to be combined with a multi-fluid flow solver in order to track the interface of the bubble and compute the pressure inside it based on the energy released by the arc.

2.2.1 Internal bubble pressure

According to [38], the arc energy may be computed as a function of the voltage drop across the arc and the current on the assumption of a low impedance turn-to-turn fault. With this energy, a dynamic heating process at constant volume takes place and the internal pressure on the bubble may be computed using the perfect gas equation

$$P_{gas} dV_{gas} = m_{gas} R dT_{gas} \tag{2.6}$$

Once the energy is computed, either by the electrical model or known beforehand, the volume of gas is needed to compute the pressure in (2.6). A logarithmic expression was proposed by SERGI [27]. In the experimental investigation they conducted, it was shown that the gas volume increased slower with the accumulation of arc energy through

the expression

$$V_{gas} = 0.44ln(W_{arc} + 5474.3) - 3.8 \tag{2.7}$$

Equation (2.7) relates the energy and generated gas of volume. Then, with the gas mass flow rate, which will be assumed to be $5Kg/s$, and the volume of gas which is going to be computed at each time step through the update of the bubble surface, it is obtained the density of the gas ρ_{gas}. On the other hand, the energy injected to the liquid is needed in order to know the gas temperature. The energy will be used to heat the surrounding liquid to vaporization temperature, to change its state and eventually heat the gas. The following workflow summarizes the process.

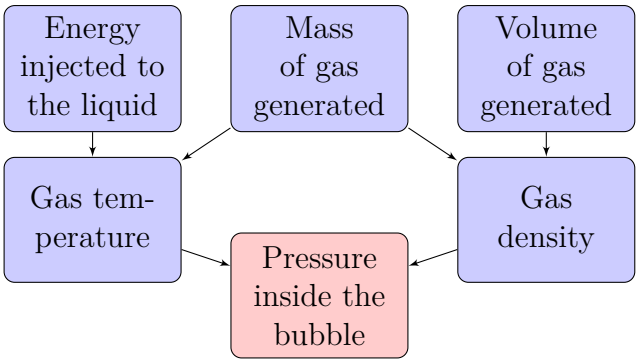


Figure 2.1: Steps to compute the gas pressure.

If the energy is not known, then the temperature must be assumed to be close to the temperature of vaporization of the liquid, around 200 ºC.

Another common approach in literature regarding this workflow is to assume that the gas bubble has already been created by the arc and the gas is already under pressure. Eventually, (2.6) turns into

$$p = \rho(\gamma - 1)e \tag{2.8}$$

Where ρ is the gas density, $\gamma = C_p/C_v$ is a constant relating specific heat capacity and $e = C_pT/\gamma$ is the internal specific internal energy.

After the arcing occurs, the increasing volume of gas generates an approximately spherical or circular (if 2D) shape at high pressure, which interacts with the surrounding oil and generates pressure waves due to the high pressure gradient.

Although the shape of the bubble will tend to be symmetrical, the interaction of the

pressure waves with the interface will distort its shape and render it irregular. For this reason, a method will be needed to track the interface position at each time step, so that the bubble volume may be computed with precision. The main Lagrangian and Eulerian multi-fluid flow techniques are reviewed next.

2.2.2 Multi-fluid flow methods

Particularities of dealing with multi-fluid flows are extensive and common in many problems in engineering, when analyzing problems with two immiscible fluids, such as simulating air and water in free surface problems. In the present case, both Eulerian and Lagrangian methods are available, as we aim to solve the Navier-Stokes problem in the gas and oil domains with the appropriate transmission conditions, that is, equal stresses and velocities at the interface. The P-FEM, VOF and Level-Set Methods are considered with a two-phase flow.

The Particle Finite Element Method (P-FEM)

The P-FEM employs an updated Lagrangian description to describe the motion of particles in the different sub-domains. As usual, using a Lagrangian framework allows getting rid of the non-linear and non-symmetric convective terms in the Navier-Stokes equations, while requiring an efficient technique to update the nodes' position [29].

The Lagrangian approach is not the usual choice but still it is employed in several applications. In a Lagrangian tracking of the interface, ALE techniques are used, and the nodes representing the interface are treated in a purely Lagrangian way, therefore matching the interface and generating distorted elements. If the flow is not too complex this is reasonable, as the advantages are sharp tracking and capturing the discontinuity of the velocity gradients, which are different since the fluids in contact have different viscosities. If the pressure space is also discontinuous, it also allows capturing pressure discontinuities, and this is specially beneficial for the present case. Nevertheless, it is due repeating that this only convenient if the problem is not too complex, otherwise the mesh is distorted and the elements fold.

The P-FEM, which considers each of the nodes of the mesh as particles, remeshes at each time step, while keeping the connectivities the same. In essence, once the boundary has been identified, the mesh is generated so that the problem may be solved by the standard FEM. In this way, the state variables are solved at time t^{n+1}, and the particles are moved afterwards to obtain the cloud of nodes at the new time. The complication in

this is that an algorithm is needed to identify boundary nodes, since the interface shape may be complex, and therefore computational time is required for these algorithmic steps. In this regard, the Alpha Shape method may be used for the boundary definition [29]. Figure 2.2 details the process, where the red nodes would be the gas nodes.

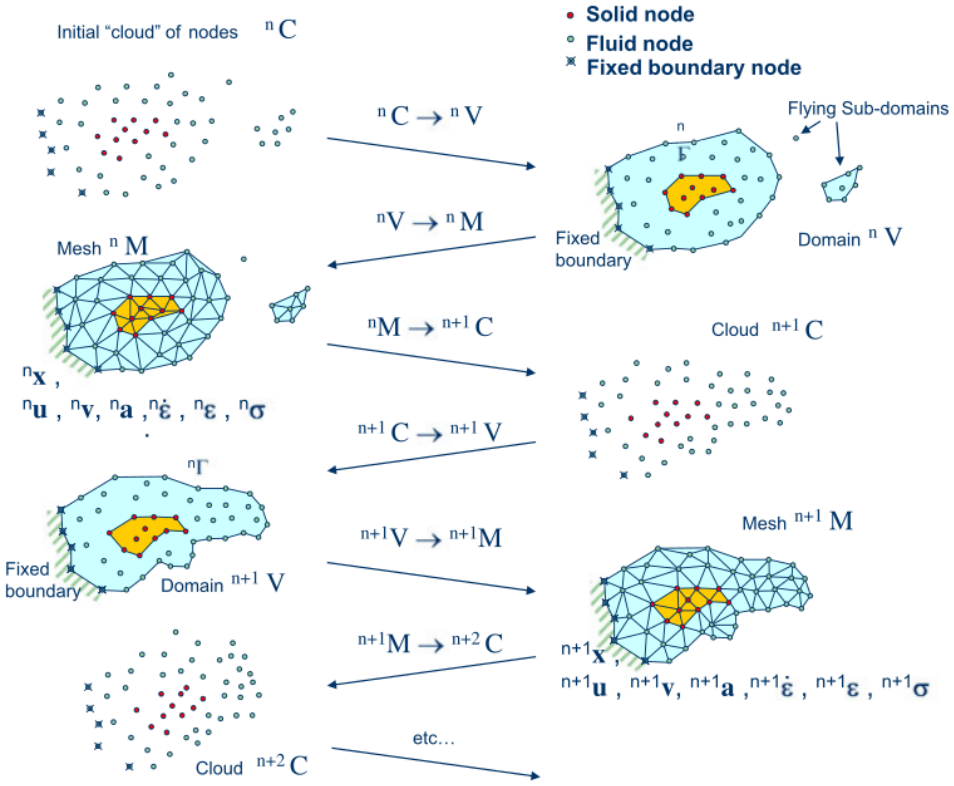


Figure 2.2: Sequence of steps to update the particles in a problem consisting of fluid and solid sub-domains.

Volume of Fuid Method

The VOF method is a numerical technique for tracking the position and shape of the interface with an Eulerian configuration. It is an advection scheme based on a scalar fraction function extending over the whole computational domain, so that when there is a fluid interface in the cell, the function C takes values between 0 and 1. For a cell full of gas, $C = 1$, and for a cell full of oil $C = 0$, as shown in Fig. 2.3. This function is then advected following the flow-field velocity, according to the transport equation

$$\frac{\partial C_m}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla} C_m = 0 \tag{2.9}$$

Where C_m refers to the fraction of the m-th fluid in the present cell in a system of n-th fluids. In this way, for each cell

$$\sum_{m=1}^{n} C_m = 1 \qquad \Phi = \sum_{m=1}^{n} \Phi_m C_m \tag{2.10}$$

Which means that, in each cell, the volume of fluid is constant, and the state variables are computed as a weighted sum. The disadvantage of this method is that the function is discontinuous, so that the interface is not sharp.

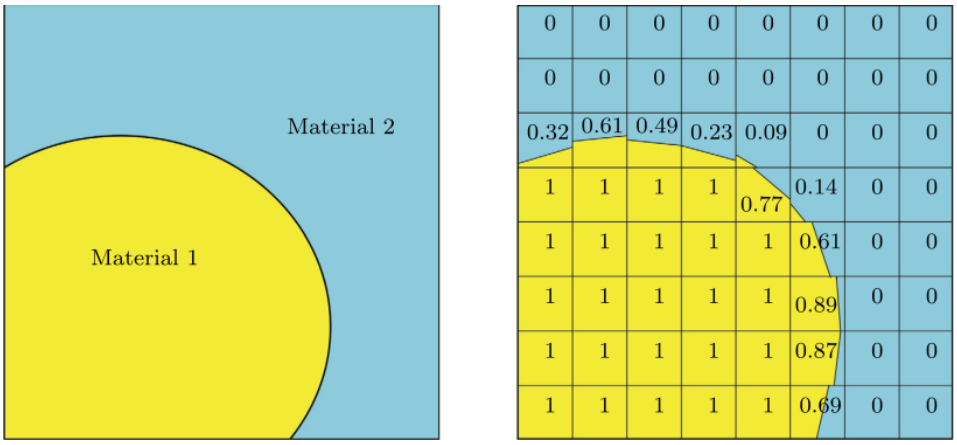


Figure 2.3: Discretized VOF function values. Extracted from [31].

The Level Set Method

The Level-Set method overcomes the difficulties present in the VOF method by allowing a sharp tracking of the interface, so that, instead of having a discontinuous fraction, a signed distance function is defined in each point of the flow field to the interface, so that it is exactly 0 at the interface (see Fig. 2.4). With that, the flow field at each time step is solved at both sides of the interface. Furthermore, as in the VOF method, the advancement of the location of the interface is performed according to the advection equation (2.9).

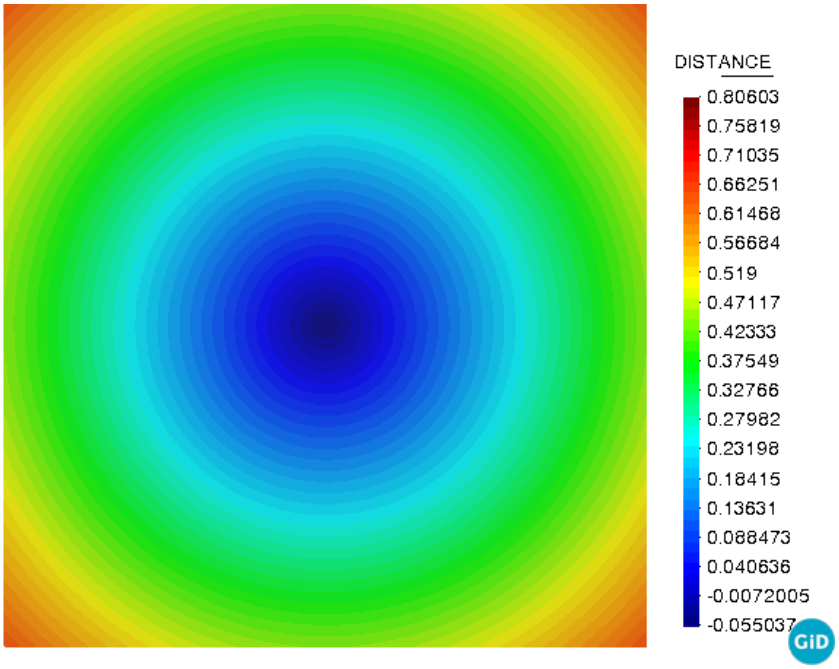


Figure 2.4: Distance function used for the definition of the gas bubble in a 2D domain.

According to [9], the level-set method successfully captures the interface in an underwater explosion simulation, as well as the interaction between the fluid-fluid interface and shock waves. As a drawback, the distance function tends to be distorted after a few steps, reason why it is commonly reinitialized every time step. This process will be automatically carried out by the solver.

2.3 Numerical methods on Fluid-Structure Interaction

The problematic inherent in potential transformer tank rupture is the sustained overpressures, which may damage the structure and result in a tank explosion. That is the reason why many companies are developing depressurization devices that may avoid static pressure increments over structural withstand limits. However, before the protection technology is activated, dynamic peaks of pressure interact with the tank walls for a short period of time. For this reason it has been necessary to simulate by means of computational tools its effect over the structure.

The solid mechanics problem, which computes the mechanical stress and displacements

on the wall, is governed by a three-equation model, containing the constitutive equation (stress-strain), the compatibility and the transient equilibrium equation (load-stress) [38]. The three relations are, respectively

$$\rho_t \frac{\partial^2 \mathbf{u}}{\partial t^2} - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}_t = \mathbf{F} \tag{2.11a}$$

$$\boldsymbol{\sigma}_t - \boldsymbol{\sigma}_0 = \mathbb{C} : (\boldsymbol{\epsilon}_t - \boldsymbol{\epsilon}_0) - \boldsymbol{\epsilon}_p \tag{2.11b}$$

$$\boldsymbol{\epsilon} = \frac{1}{2}\Big[(\nabla \mathbf{u}) + (\nabla \mathbf{u})^T\Big] \tag{2.11c}$$

Where ρ_t is the material density, $\mathbf{u}$ are the displacements, $\boldsymbol{\sigma}_t$ is the stress tensor, $\boldsymbol{F}$ is the total load, $\boldsymbol{\epsilon}_0$ and $\boldsymbol{\sigma}_0$ are the initial stress and strain, $\boldsymbol{\epsilon}_p$ is the plastic strain tensor and $\mathbb{C}$ is the fourth-order elasticity tensor.

When coupling the computation of the fluid with the stresses on the walls, the complexity increases substantially. Some of the reasons may be listed below [34]:

- Complex and dynamic boundaries immersed in the fluid domain.
- The structure needs to be modeled in more detail, including the effect of bolted and welded assemblies.
- Typically involving material and geometric non-linearity.

For these reasons, advanced strategies are devised for computing the inner pressure waves on the wall [23].

The most usual approach for dealing with the tank explosion is to use the shell theory, since the thickness of the walls are negligible compared to the tank height and length. The temporal and spatial evolution of the mechanical stresses and deformations are also computed through transient time-dependent simulations. Other common techniques are to use Lagrangian and Arbitrary Lagrangian-Eulerian frameworks for the Solid and Fluid Mechanics problems, respectively. Another important ingredient is the consideration of non-overlapping domains, so both sub domains' meshes are connected through an interface. As for the transmission conditions, Dirichlet-Neumann technique is typically chosen, where the Dirichlet case enforces the continuity of either velocities or displacements and the Neumann case enforces the continuity of tractions.

2.3.1 Coupling the flow and structure solvers

The coupling in time, however, plays the most important part. The two common choices are the following:

Monolithic, in which the degrees of freedom for the solid and fluid domains are solved in a single discrete equation system, thus the possibility to use different software packages for each problem is rejected. If the meshes are matching, it is common to solve the whole system in terms of displacements. Then, while considered to be more robust, since no partitioning-domain errors are introduced, are indeed more expensive to solve and the system matrix is commonly poorly conditioned, since a single matrix contains variables of different nature and magnitude [30].

Partitioned, or staggered approach exploit the fact that individual solvers for the fluid and solid parts are usually available to split the coupling and solve one problem at a time. Now the effort is put in the coupling algorithm, that exchanges the information between boundaries to enforce continuity of variables. As a drawback, the adaptability of the method incurs in the convergence rate, as both solvers advance in time iteratively.

Then, when considering partitioned methods, largely used for the problem at hand, there are important concepts regarding the coupling of multi-physics transient physics. To introduce those, it is important to state the flow control procedure first, which consists in solving the pressure field with the hydrodynamics code to later compute the mechanical stresses and deformations with the structural mechanics code. The number of computations done at each time step for either solver defines the coupling as weak or as strong.

In [23], both coupling techniques are presented for a description of the mechanical behavior of 3D complex geometries by coupling an already developed multi-phase flow hydrodynamic compressible (HYCTEP) code with an structural analysis code (ASTER) [1]. Fig. 2.5 shows a comparison between both procedures.

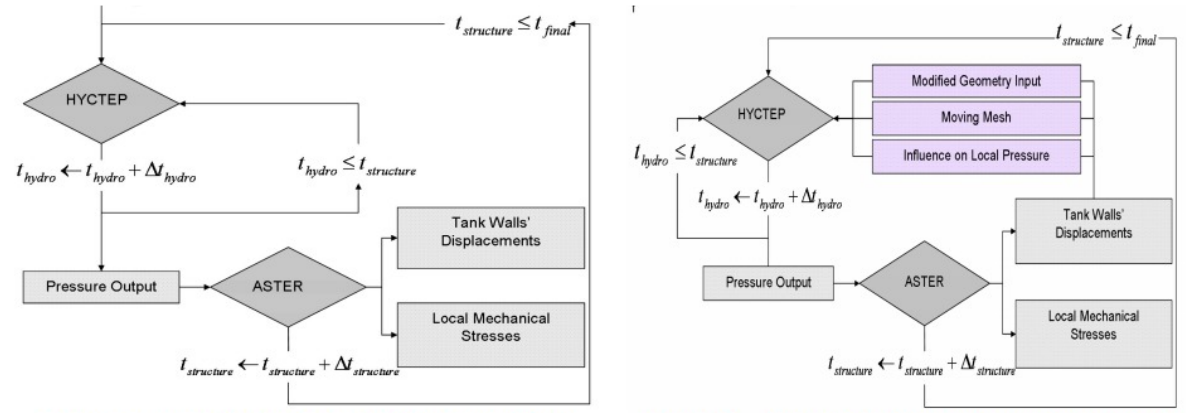


Figure 2.5: Workflow for strong and weak coupling for the tank deformation problem. Extracted from [23]

In the weak coupling, several iterations are done to compute the pressure on the walls, until the structural time-step is reached. Then, the structural code receives as input the loads on the walls from the pressure field, so that the displacements and stresses are extracted. However, the tank-deformation problem must necessarily consider the effect of such deformations onto the pressure field, since a sustained and controlled deformation of the walls could lead to a sufficient depressurization of the fluid volume and avoid using depressurization devices. For this reason, the strong coupling seems a more suitable approach, where the displacements are accounted for in the hydrodynamics code after the structural computations. As several interactions between solvers are done at each time step, contrary to the weak coupling (where the exchange of information is produced once) it is easier to achieve synchronization between solutions [36]. In fact, the number of interactions depends on the convergence criteria, which is satisfied when enough level of coupling is reached.

The latter already introduced some of the advantages of strong coupling, which are better stability of the coupled algorithm, second order time accuracy and the unique manner (for Partitioned schemes) to satisfy energy conservation at the fluid-structure interface. The computational cost, however, is larger, since more iterations are performed. It is important to underline that the iterations are a necessary condition, as advancing in time both solvers successively is not enough, but rather explicit time-marching schemes or implicit methods may be used if information is shared frequently [36].

The fluid-structure density ratio ρ_s/ρ_f also plays an important role. While weak coupling may be subject to numerical stability constraints based on this ratio, the strong

coupling does not introduce stability constraints and is well suited for small ratios [39]. For large-scale deformation problems, to ensure numerical stability of the FSI solver, strong coupling is typically used [3]. This, combined with the fact that partitioned approaches require a computational time of the same order as monolithic approaches, gives the preferred choice: strong coupling with a partitioned scheme (see Fig. 2.6).

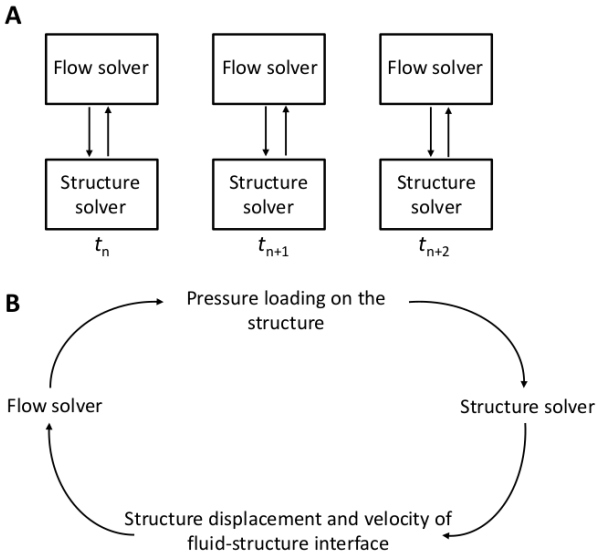


Figure 2.6: (A) shows the partitioned approach, whereas (B) details the exchange of information between solvers at each time step. Extracted from [3].

2.3.2 Main FSI algorithms

Several algorithms have been developed for the numerical computation of the fluid-structure interaction that also include wave propagation phenomena. All focus primarily on satisfying the continuity conditions at the interface, which is the main concerning topic in this kind of problems.

Arbitrary Lagrangian–Eulerian (ALE)

Already introduced, this method was first devised in (Donea, 1982) [13]. A conforming mesh fits the structural interface, and the Solid Mechanics problem is then solved by a Lagrangian formulation, with the Eulerian formulation for the Navier-Stokes equations for the flow. By enforcing continuity of velocities and surface tractions on the interface, the problem is solved for the fluid and the structure at the interface [34]. Eventually, the

mesh is mapped to the deformed domain by means of a remeshing algorithm at each time step. An example of applying ALE methods for pressure-wave propagation problems is given in [22]. This method is widely used when discretizing spatially problems in fluid and structural dynamics in which large deformations in both domains are expected.

Smoothed particle hydrodynamics (SPH)

The SPH method is another technique commonly used in dealing with fluid-solid interfaces, with the difference that it is a mesh-free method, in which the physical quantities of the state variables in every particle are computed by weighting the contribution of the surrounding particles inside a basin of attraction [30]. The fluid and solid equations are solved in a Lagrangian framework and the continuity conditions at the interface are satisfied based on appropriate SPH kernel functions [34]. The choice of these interpolation functions (which are typically Gaussian or cubic splines) signify the principal difference of these methods over standard FE methods. Further information on SPH may be found in [24].

Ghost Fluid Method (GFM)

The GFM method for tracking fluid/structure interfaces was created by Liu et al. (2003) as an enhanced version of a fluid/fluid interface problem for shock wave impacts, in which a number of ghost points are created at the other side of the flow interface to satisfy continuity requirements [34]. While it is commonly used to track multi-fluid problems, it is also suited to describe solid deformations. Once the ghost media is created, the characteristic quantities are approximated at the fluid/solid interface by solving a Riemann problem. The difference of this method with respect to the others, in terms of mesh procedures, is that it does not require remeshing, as in the ALE, since an implicit level-set representation is adopted. This is an advantage, since frequent remeshing can affect the solution accuracy at points next to the interface [34].

2.4 Mesh-updating procedures

The principal drawback when dealing with ALE techniques is the need to re-mesh, which may be complex and be computationally expensive. *Mesh-regularization* and *mesh-adaption* are two strategies that may influence the success of the algorithm. In any case, the problem at hand will be characterized by not knowing a priori the mesh movement.

Then, as explained, the procedure is described by a Lagrangian frame at the boundaries (solid nodes), while after a suitable transition zone, an Eulerian formulation is employed [14] for the fluid. The nodes on the transition zone are interpolated. The method can be used with 2D and 3D triangular, quadrilateral, tetrahedral and hexahedral elements, among others.

The **mesh-regularization** method updates the mesh at each time-step so that the pattern is regular and element distortion is reduced. This way, the error associated to entangled computing zones is reduced.

On the other hand, the **mesh-adaption** method focuses on improving accuracy through a better management of the current mesh, by concentrating nodes on high-gradient regions while keeping the overall number of elements the same. An error indicator is then provided which is used to maintain an equally distribution on the domain.

This problem needs to satisfy the following boundary conditions

$$\mathbf{u}(\mathbf{x},t) = \mathbf{u}_D,\ \mathbf{x} \in \Gamma_D \tag{2.12a}$$

$$\mathbf{n}\cdot\boldsymbol{\sigma} = \mathbf{t},\ \mathbf{x} \in \Gamma_N \tag{2.12b}$$

$$\tag{2.12c}$$

which are kinematic and dynamic conditions at the boundary of the domain. More specifically, at the fluid-structure interface, the particles are linked to the flexible (or rigid) walls, therefore the particles cannot cross them and there must be a continuous stress across the surface. Whilst the first condition simply translates into $\mathbf{n}\cdot\mathbf{u}_{mesh} = \mathbf{n}\cdot\mathbf{u}$, the second is accounted for in the weak formulation of the conservation equation. For the mesh, the boundary conditions are

$$\mathbf{u}_{mesh}(\mathbf{x},t) = \mathbf{u}(\mathbf{x},t)\,,\ \mathbf{x} \in \Gamma_{int} \tag{2.13a}$$

$$\mathbf{u}_{mesh}(\mathbf{x},t) = 0\,,\ \mathbf{x} \in \Gamma\backslash\Gamma_{int} \tag{2.13b}$$

While the first equation states that the mesh is attached to the interface, the second prescribes null movement normal to the surfaces of the rest of the fluid boundary. Moreover, on the interface, equality of velocities and displacements between fluid and structure must hold.

Chapter 3

Problem formulation

It has been mentioned that the two main aspects concerning transformer tank explosions due to internal arcing faults are the motion of the gas bubble and the propagation of pressure waves. Here the mathematical basis for describing the fluid flow is explained, alongside with all the concepts needed to understand the methodology adopted to carry out the simulations. A description of the frames of reference, governing equations and the main features of weakly compressibility is given.

The Navier-Stokes equations are employed to describe many problems in fluid mechanics, as they account for the transient equilibrium between internal and external forces on a fluid particle. The problem to be modeled is complex and will require of the use of other methods, though the Navier-Stokes equations will be the essential building block of the development.

3.1 The description of the flow field

The domain in which the physical description of the problem will be formulated is referred to as Ω, characterized by being a nonempty connected subset of the space $\mathbb{R}^n$, with n being the number of space dimensions. The bounded domain Ω is assumed to be filled up with some fluid, which in the present case will be water, gas or an oil with thermodynamic properties similar to standard transformer oils.

The flow region will be delimited by a boundary $\partial\Omega$, in which are assumed certain smoothness conditions. If the domain is particularized into a transformer tank, $\partial\Omega$ is associated with the solid wall. Then, if the gas bubble is included, its surface S will

also belong to the boundary (i.e. $S \in \partial\Omega$). As for the normal vectors, the convection is followed to define them positive to the exterior of the surface, and with that it is important to notice that the normal vector of the gas bubble surface will point inwards.

The bubble surface S may undergo a motion in the flow domain and its surface expected to vary in time. When this occurs, the shape of the flow domain will evolve, meaning that its deformation will have to be considered when writing the conservation laws (see Fig. 3.1 for clarification). In this case, the domain will be referred to as Ω_t to emphasize its dependence with the time variable t, the time interval being $[0, T)$ and $t \in [0, T)$.

Then, the state of the fluid will be characterized by the state variables: $T(\mathbf{x}, t)$, $p(\mathbf{x}, t)$ and $\mathbf{u}(x, t)$, which are, respectively, the temperature, pressure and velocity of the fluid at $(t, \mathbf{x}) = (t, x_1, \ldots, x_n)$, with $t \in [0, T)$ and $x \in \Omega$. The chosen physical model to describe the motion of the fluid will allow establishing relations among the state variables. As it will be seen, assumptions on the fluid behavior will discard the temperature as a state variable to be solved in the domain.

Eventually, although it is clear that a solution of the flow field is to be found both in the fluid and in the gas parts of the domain, the nature of the problem will require considering the bubble as a moving boundary condition, exerting a pressure force on the fluid. In particular, the pressure will be given by the arc energy, but the velocity will have to be solved within the bubble. For this reason, when formulating the conservation laws in the domain, the control volume will be considered to be the liquid part, with oil or water.

3.1.1 Frame of reference

The numerical simulation of the fluid dynamics problem at hand will have to deal with distortion of the flow region when the gas bubble propagates while keeping a smooth track of the fluid-gas and fluid-structure interfaces. For this reason, it is not a trivial decision weather to use a Lagrangian or an Eulerian viewpoint when working with fluid-structure interaction or even fluid mechanics problems, although the latter is mainly formulated with the Eulerian formulation. Since the kinematical description of the domain relates the continuum region with the nodal discretization, choosing one or the other will condition the ability of the numerical code to deal with large mesh distortion.

The basic concepts related to the kintematical description of the flow field are assumed to be known. However, the reader is referred to [15] for a mathematical description of the continuum mechanics' concepts of the description of motion.

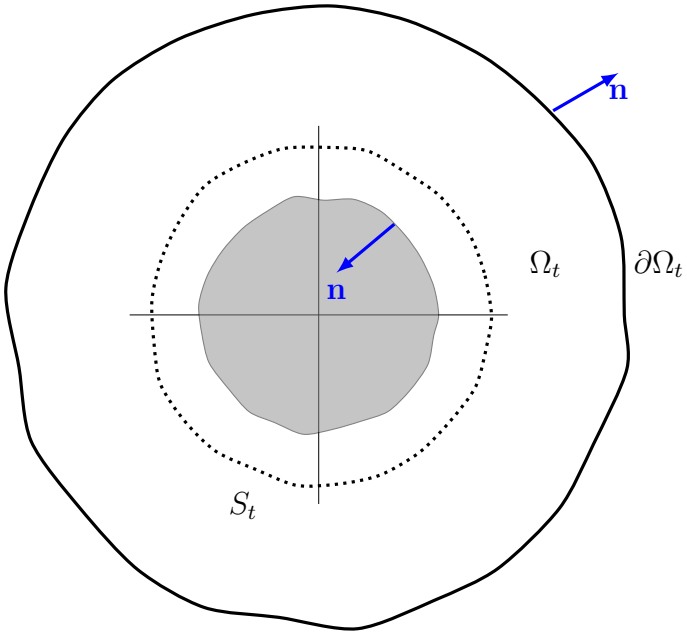


Figure 3.1: Schematic representation of the domain.

In a Lagrangian context, the material points are coincident with the mesh nodes at all times, therefore the local derivative replaces the material derivative and no convective velocities are needed. For this reason, the fact that the nodes contain the same particles during the simulation renders this framework suitable for tracking structural deformations, in which there is a clear history-dependent behavior. Free-surfaces and interface between two or more materials are easily dealt with under a Lagrangian perspective, and as it will be seen, the multi-fluid flow methods considered for tracking the gas-oil interface will contemplate Lagrangian choices. However, large deformations may fold excessively the mesh or force a re-meshing, otherwise the algorithm may undergo a loss in accuracy or finish abruptly the computation [15].

By considering an Eulerian framework, the mesh nodes are fixed and the material particles pass through. Therefore, the variation of the quantities at the nodes as the simulation evolves will imply a local and a convective treatment, thus the gradient of the quantities will be advected with the velocity of the material particles at the nodes. The advantages found in the Lagrangian framework, namely the easy tracking of moving interfaces, will become drawbacks, whereas the disadvantages will be easily handled in the Eulerian framweork, where the continuum can be deformed to a larger extent without compromising the accuracy or stability of the simulation. Another issue emerging from

the addition of convective terms is the non-symmetry character the formulation evolves into, compensated through the introduction of stabilization techniques. However, the stabilized finite element formulation is a price to be paid if reliable numerical results are sought [14].

These two approaches are widely used in Structural Mechanics or Fluid Mechanics problems when they are not combined. However, in Fluid-Structure Interaction problems, the Arbitrary Lagrangian-Eulerian (ALE) framework will encompass the benefits of both in the same problem, so their advantages are kept. It has been seen the spatial configuration $\mathbf{x}$, used for the Eulerian problems, in which the nodes of the mesh are fixed in space, and then the material configuration $\mathbf{X}$, in which the mesh nodes follow the particles. The new ALE frame of reference $\boldsymbol{\mathcal{X}}$ will have a movement of its own, where the coordinates identify the grid points, not spatial points nor material particles. For the problem at hand, the Eulerian approach will be comprised in the fluid part, whereas the boundaries (the tank walls) will be solver through a Lagrangian approach, and the governing equations written in the ALE method will allow a smooth connection between both parts. In this manner, the convective velocity used for the governing equations will be

$$\mathbf{c} := \mathbf{v} - \hat{\mathbf{v}} = \frac{\partial \mathbf{x}}{\partial \boldsymbol{\mathcal{X}}} \cdot \mathbf{w} \tag{3.1}$$

Where the following definitions are employed

$$\mathbf{v} = \left.\frac{\partial \mathbf{x}}{\partial t}\right|_{\mathbf{X}} \qquad \hat{\mathbf{v}} = \left.\frac{\partial \mathbf{x}}{\partial t}\right|_{\boldsymbol{\mathcal{X}}} \qquad \mathbf{w} = \left.\frac{\partial \boldsymbol{\mathcal{X}}}{\partial t}\right|_{\mathbf{X}} \tag{3.2}$$

In this way, $\mathbf{v}$ is the material velocity and $\hat{\mathbf{v}}$ is the mesh velocity. The physical meaning of $\mathbf{w}$ is the variation of the nodal coordinate associated with a fixed material particle with time. The latter shows that, if $\boldsymbol{\mathcal{X}}$ is coincident with $\mathbf{x}$, the mesh velocity is zero and the ALE frame evolves into the Eulerian approach, whereas if $\boldsymbol{\mathcal{X}}$ coincides with $\mathbf{X}$, the mesh velocity and material velocity are the same and the ALE recovers the Lagrangian algorithm.

In this thesis, the three mentioned descriptions of motion will be used. The Eulerian viewpoint, is, in agreement with the general trend, the one used for cases in which there is no structure interaction, as it is contained in the ALE description only by setting $\mathbf{u_m} = 0$. For the cases in which a structural coupling is present, the ALE description is automatically activated with the grid motion, and for this reason it is convenient to

write the equations in the ALE framework. On the other hand, the Lagrangian framework is used only as a complementary part to propose an alternative physical model in which a single equation for the pressure is obtained. This approach intended to solve the in-homogeneous acoustic wave equation in time domain with a finite-element discretization, and a single equation is obtained after adopting the Lagrangian framework. The development of it is present in Appendix 6.2.

A clear one-dimensional example of the three cases is shown in Fig. 3.2, and in Fig. 3.3 the ALE improvements over the Lagrangian framework by handling distortion of the continuum without folding the elements or failing to reproduce the interface. This case is especially meaningful for the Thesis since it has been extracted from the modeling of an explosive in a water vessel. This shows the difficulties in approaching such a problem from a Lagrangian context.

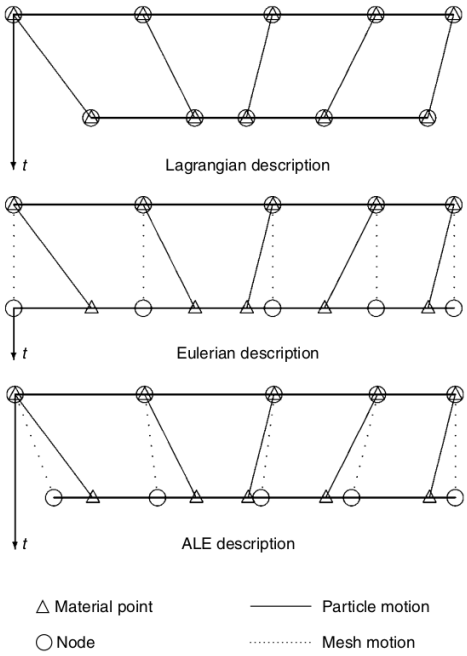


Figure 3.2: Comparison between frames of reference. Extracted from [15].

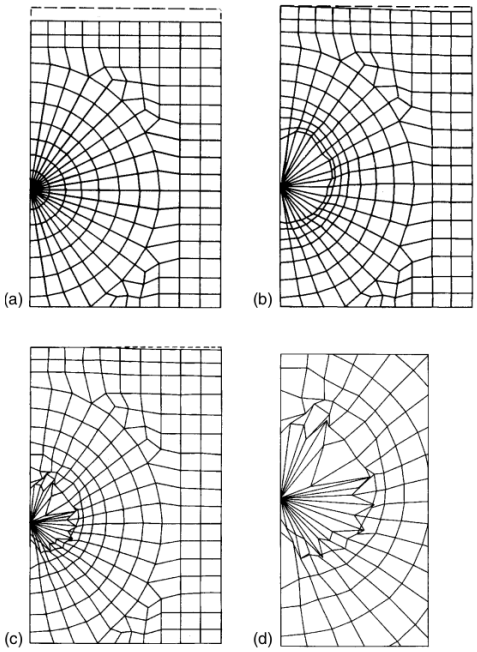


Figure 3.3: Lagrangian versus ALE descriptions: (a) initial FE mesh; (b) ALE mesh at t = 1 ms; (c) Lagrangian mesh at t = 1 ms; (d) details of interface in Lagrangian description. Extracted from [15].

3.2 Governing equations

The governing equations for the problem are extracted from the integral form of the conservation laws. The analysis of a finite region in space, by balancing the incoming and outgoing flux and its effects will allow formulating a description of the fluid movement from the differential point of view, in which a detailed description of the fluid in each point of the flow field is conducted. When establishing integral relations in the flow region, it is necessary to define a control volume. The election of this control volume is important since it will condition the relative velocities between the fluid and the control surfaces. The current election of the control volume is that which coincides with the material volume at all time steps, hence the conservation laws will have to be derived considering a deforming control volume. To do this, the rate of change of the scalar and vector quantities will have to be taken outside the integral, hence arising the need of using the common Reynolds transport theorem for a smooth function $f(\mathbf{x}, t)$

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega_t} f(\mathbf{x},t)\,d\Omega = \int_{V_c\equiv\Omega_t} \frac{\partial f(\mathbf{x},t)}{\partial t}\,dV + \int_{S_c\equiv\partial\Omega_t} f(\mathbf{x},t)\mathbf{u}\cdot\mathbf{n}\,dS \tag{3.3}$$

Where V_c is a control volume coinciding with the material volume Ω_t at the considered time t. Similarly, the integral over the control surface S_c represents the flux of the quantity $f(\mathbf{x}, t)$ across a fixed curve coinciding at time t with the boundary of the material volume. Since $S_t \in \partial\Omega_t$ is the only moving boundary of the domain, the integral will be over $S_c \equiv S_t$. Then, $\mathbf{u} = \mathbf{u}(\mathbf{x}, t)$ represents the material velocity of the particles which at time t occupy a spatial position such that $\mathbf{x} \in S_c$. A similar form holds for the conservation of a vector quantity.

By further noting that

$$\int_{S_c} f(\mathbf{x}, t)\mathbf{u} \cdot \mathbf{n}\, dS = \int_{V_c} \boldsymbol{\nabla} \cdot (f\mathbf{u})\, dV \tag{3.4}$$

it is possible to obtain the Eulerian differential forms of the conservation equations for mass, momentum and energy.

The mass conservation equation ($f(\mathbf{x}, t) = \rho$) reads

$$\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla}\rho = -\rho \boldsymbol{\nabla} \cdot \mathbf{u} \tag{3.5}$$

with ρ being the density.

The momentum conservation equation ($\mathbf{f}(\mathbf{x}, t) = \rho\mathbf{u}$) reads

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u}\right) = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho\mathbf{u} \tag{3.6}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor and $\mathbf{b}$ is the specific body force vector.

Eventually, the energy conservation equation ($\mathbf{f}(\mathbf{x}, t) = e$) reads

$$\rho\left(\frac{\partial e}{\partial t} + \mathbf{u} \cdot \nabla e\right) = \boldsymbol{\nabla} \cdot (\boldsymbol{\sigma} \cdot \mathbf{u}) + \mathbf{u} \cdot \rho\mathbf{u} \tag{3.7}$$

where e is the specific total energy.

Once the Eulerian form is obtained, the same equations written in ALE form are obtained by modifying the convective velocities in the left-hand side, since this is the term accounting for the relative velocities between the grid points and the material particles. Since the convective velocity for the Eulerian form is the material velocity $\mathbf{u}$ itself, the ALE form will employ another convective velocity, namely $\mathbf{a} = \mathbf{u} - \mathbf{u_m}$, where $\mathbf{u_m}$ is the

mesh velocity.

$$\begin{cases} \frac{\partial \rho}{\partial t} + -\mathbf{a} \cdot \boldsymbol{\nabla} \rho = -\rho \boldsymbol{\nabla} \cdot \mathbf{u} \\ \rho\Big(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{a} \cdot \nabla)\mathbf{u}\Big) = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho \mathbf{u} \\ \rho\Big(\frac{\partial e}{\partial t} + \mathbf{u} \cdot \nabla e\Big) = \boldsymbol{\nabla} \cdot (\boldsymbol{\sigma} \cdot \mathbf{u}) + \mathbf{a} \cdot \rho \mathbf{u} \end{cases} \tag{3.8}$$

It is seen the similarity with the original Eulerian form, the only changes being the convective velocities and the local derivatives, which now will be taken in another frame of reference. It is therefore understood why it is convenient to work with this form, since it is a hybrid approach that allows recovering the fully Lagrangian ($\mathbf{a} = 0$) or Eulerian ($\mathbf{a} = \mathbf{u}$) forms by modifying the convective term.

The ALE differential forms of the conservation laws in (3.8) will be used as the strong form of the stabilized finite-element formulation to discretize the problem. It is important no notice that these equations have been formulated only under the premise of continuity of density, velocity and energy. That means that the flow may still be incompressible or compressible, stationary or unstationary, viscous or inviscid, and may also be discretely used in other reference systems different than cartesian.

Moreover, the physical description of the physical field will have to considerate the a compressible fluid, since the propagation of shock waves.

To complete the system of equations in (3.8), an equation of state has to be added, relating the density to the temperature (or entropy) and the pressure, i.e. $\rho = \rho(p, T)$. The relations between the main thermodynamic properties will depend on the nature of the fluid and will have to be able to capture the complete physics of the problem also when the pressure range tends to the compressible region.

The model thus obtained will be similar to the complete flow models used in compressible frameworks in terms of complexity, since the mechanical and thermodynamic aspects of the fluid are taken into account (viscosity, thermal, gravity, etc.). As mentioned, in this approach the energy equation is coupled to the rest of the governing equations as the density is not a function of the pressure only. The resulting system of equations is therefore strongly coupled and highly non-linear, and requires high computational cost. In the next subsection some assumptions will be made that will allow removing part of the system's complexity by uncoupling the energy equation.

3.2.1 Pressure-density relations

So far, the general case of a wave propagating in a fluid which is both viscous and heat-conducting has been subject to study. The fact that it is heat-conducting implies that the pressure is not a function of the density alone, but of the temperature as well. The speed of sound will be a function of $c(p, T)$.

If the Mach number of the problem is introduced as $M = U/c_0$, where U is the speed of propagation of the pressure wave and c_0 is the sound speed of the undisturbed fluid ahead of the shock, it is noted that, under the assumption of a weak shock, the Mach number is close to one and entropy changes may be neglected [4]. This is a common approach in the field of underwater explosions, where the expected pressure shocks are expected to be higher. However, the fact that the shock is weak being justified by a small Mach number does not deprive the model from allowing high pressure gradients, since the latter will be greatly influenced by how the compressibility of the fluid is dealt with.

Thus, according to the latter, the local sound speed is defined as

$$c = \sqrt{\left(\frac{\partial p}{\partial \rho}\right)_S} \tag{3.9}$$

the subscript S denoting the dependence of pressure on density and entropy. Now, if in the shock is weak in the vicinity of the bubble, the entropy changes throughout the fluid may be neglected and the pressure considered as a function of density alone [4]. To model it as a compressible liquid requires relations which connect the main thermodynamic properties: pressure, density and temperature. These relations have to be able to capture the complete physics of the problem also when the pressure range tends to the compressible region. For this reason, an equation of state describing the thermodynamic properties of the fluid in the form $p = p(\rho)$ is sought. The Tait equation of state (EOS) is proposed in [10] alongside with the Stiffened Gas EOS and the Nobel-Abel Stiffened Gas(NASG) EOS for the prediction of thermodynamic properties of water over a wide range of pressures (1×10^5 Pa to 1×10^9 Pa) and temperatures (280 K to 370 K).

The Tait equation relates the pressure and the density as follows,

$$p = K_0 \left[\left(\frac{\rho}{\rho_0} \right)^{\theta} - 1 \right] + p_0 \tag{3.10}$$

Where p_0 and ρ_0 are respectively the pressure and the density of water at the reference temperature. Parameters $K_0 = 3 \times 10^8$ Pa and $\theta = 7$ are weak functions of temperature and pressure and are held constant.

The Stiffened Gas EOS relates pressure, density and temperature as follows:

$$p = \rho(\gamma - 1)c_v T - p_\infty \tag{3.11}$$

where p_∞ is a constant parameter representing the molecular attraction between water molecules, c_v is the specific heat at constant volume and γ is the ratio of specific heats.

The Tait EOS is able to produce accurate results within the stipulated pressure range and does not require the prior knowledge on temperature. Although [10] disserts on the superiority of the modified NASG EOS over the Tait EOS in modelling non-isothermal flow problems with high accuracy, the former is more complex and is primarily meant for the prediction of the saturation properties of certain liquids, which is not the purpose of the sought EOS.

Eventually, if the Tait equation is chosen as the EOS to model the compressibility of the fluid, the sound speed can be computed according to (3.9).

$$c = \sqrt{\left(\frac{\partial p}{\partial \rho}\right)_S} = \sqrt{\frac{K_0 \theta \left(\frac{\rho}{\rho_0}\right)^{\theta-1}}{\rho_0}} \tag{3.12}$$

With (3.12), density and sound speed can be computed at each time step only as a function of pressure.

At this point, with the chosen EOS, it is time to look closely at the relation between temperature and entropy. The net heat added per unit time to a fluid particle is given by

$$\rho T \frac{\mathrm{d}s}{\mathrm{d}t} = \kappa \nabla^2 T \tag{3.13}$$

where κ is the coefficient of thermal conduction (here idealized as a constant). This relation tells that, in order to assume $ds/dt \approx 0$, the process must be either isothermal or adiabatic. Neither is exactly true, but according to [32], for freely propagating acoustic waves with typical frequencies of interest, the numerical implications of assuming isothermal flow are equal to $ds/dt \approx 0$, for which the fluid flow of the present problem will be considered adiabatic. This assumption, as mentioned, will allow suppressing

the temperature as a state variable and thus uncoupling the energy equation from the general system (3.8), so that only the continuity and momentum equations will have to be solved for the velocity and pressure.

3.2.2 Energy conservation in fluids

The fact that the energy equation is uncoupled from the system does not mean that it can not be solved, specially when it may serve as a tool to check that there is no energy loss in the system. This will help validating the model once the simulations are set up, and will contribute to asserting that the numerical treatment for the fluid-gas interface has been tackled correctly, or at lest up to a point in which the energy transmitted by the pressure force is conserved in the domain. For that, it is necessary to recover the energy equation in its most general form,

$$\frac{\mathrm{d}Q}{\mathrm{d}t} - \frac{\mathrm{d}W}{\mathrm{d}t} = \frac{\mathrm{d}E}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\left(\int_{V_c} e\rho \, dV\right) + \int_{S_c} e\rho(\mathbf{u}\cdot\mathbf{n}) \, dS \tag{3.14}$$

Where E is the total energy and e is the specific total energy. The convention is followed such that Q positive means heat transmitted to the system and W positive means work done by the system. In equation (3.14), it is important to note that the material derivative is taken for the energy variation in the control volume and therefore it has to be computed outside the integral. As for the surface integral, since there is no energy flux to the control volume crossing through its boundaries, this term will be zero.

We neglect any source of heat Q to the system and assume that the only contribution to the work term $\dot{W}$ is the work of the pressure forces $\dot{W}_p$, where the dot indicates temporal derivative. $\dot{W}_p$ is only produced on the surface, and its total value, per unit time, is the integral over the control surface of the elemental force per the normal component of the velocity towards the control volume, and accounts for the rate of work done per unit area and by the fluid on the control surface.

$$\dot{W}_p = \int_{S_c} p(\mathbf{u}\cdot\mathbf{n}) \, dS \tag{3.15}$$

Since the normal vector has been defined towards the bubble, and the work is done on

the system, (3.14) transforms into

$$\frac{\mathrm{d}}{\mathrm{d}t}\Big(\int_{V_c} e\rho\, dV\Big) = -\int_{S_c} p(\mathbf{u}\cdot\mathbf{n})\, dS \tag{3.16}$$

Now the total energy per unit volume $E = e\rho$ can be expressed as

$$E = \frac{1}{2}\rho|u|^2 + \rho U_p \qquad U_p = \int_{1/\rho}^{1/\rho_0} p d\frac{1}{\rho} \tag{3.17}$$

Here p is total pressure and U_p is the specific internal energy relative to the ambient state. Since the Tait equation in (3.10) states the relation between the pressure and density, the integral is straightforwardly computed to obtain

$$U_p = \frac{K_0\rho^{\theta-1}}{(\theta-1)\rho_0^{\theta}} + \frac{K_0 - p_0}{\rho} \tag{3.18}$$

Eventually, by integrating eq. (3.16) in time from t^n to t^{n+1} renders a discrete conservation law for the energy, in which $\mathbb{E}$ is the total energy

$$|\mathbb{E}^{n+1}| - |\mathbb{E}^n| = \int_{t^n}^{t^{n+1}} \int_{S_c} p(\mathbf{u}\cdot\mathbf{n})\, dS \tag{3.19}$$

3.2.3 Energy conservation for a perfect fluid

During the development of the thesis, a new formulation was derived for the energy conservation in a fluid that would allow checking with a lower programming effort if the interface was conserving energy in the domain. For that, the starting point was the same as in the previous subsection, i.e. eq. (3.16),

$$\frac{\mathrm{d}}{\mathrm{d}t}\Big(\int_{V_c} e\rho\, dV\Big) = -\int_{S_c} p(\mathbf{u}\cdot\mathbf{n})\, dS \tag{3.20}$$

The purpose was to simplify further the left-hand side of (3.20), by making use of the following simplifications,

- The fluid is perfect, i.e. it is inviscid and compressible.
- There are no gravitational effects.
- The control volume is not deformable, thus the temporal derivative can be taken inside the integral.

Considering the latter, the momentum equation transforms into following, where the

mechanical constitutive equation is replaced by the isotropic pressure tensor so that

$$\rho\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = -\boldsymbol{\nabla}p \tag{3.21}$$

As for eq. (3.20) it will be

$$\frac{\mathrm{d}}{\mathrm{d}t}\Big(\int_{V_c} e\rho\, dV\Big) = \int_{V_c}\frac{\mathrm{d}}{\mathrm{d}t}(e\rho)\, dV) = \int_{V_c}\rho\frac{\mathrm{d}}{\mathrm{d}t}\Big(\frac{1}{2}\mathbf{u}\cdot\mathbf{u} + U_p\Big)\, dV \tag{3.22}$$

By analyzing separately the terms in (3.22), and using (3.17)

$$\rho\frac{\mathrm{d}}{\mathrm{d}t}\Big(\frac{1}{2}\mathbf{u}\cdot\mathbf{u}\Big) = \rho\mathbf{u}\cdot\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = -\mathbf{u}\cdot\boldsymbol{\nabla}p = -\boldsymbol{\nabla}\cdot(p\mathbf{u}) + p\boldsymbol{\nabla}\cdot\mathbf{u} \tag{3.23a}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}(\rho U_p) = \frac{\mathrm{d}}{\mathrm{d}\rho}(\rho U_p)\frac{\mathrm{d}\rho}{\mathrm{d}t} = \frac{p}{\rho}\frac{\mathrm{d}\rho}{\mathrm{d}t} \tag{3.23b}$$

Now, by replacing the continuity equation in (3.23a), and substituting back in (3.22) it is obtained that

$$\frac{\mathrm{d}}{\mathrm{d}t}\Big(\int_{V_c} e\rho\, dV\Big) = -\int_{V_c}\boldsymbol{\nabla}\cdot(p\mathbf{u})\, dV = -\int_{S_c} p(\mathbf{u}\cdot\mathbf{n})\, dS \tag{3.24}$$

When evaluating the balance between energy influx and energy variation in the fluid domain, both equations (3.16) and (3.24) will be programmed to see whether the assumed simplifications (specially the fact that the control volume is not deforming) introduce a tolerable error. In fact, (3.24) reaches the divergence theorem applied to the integral of the pressure forces on the boundary, so the computation of both terms should be equal.

3.2.4 Constitutive relations

The analysis of the governing equations is here expanded by interpreting the term $\boldsymbol{\sigma}$ in eq. (3.8). In order to specify the nature of the considered fluid in Ω, let us decompose the momentum equation in the chosen system

$$\frac{\partial\rho}{\partial t} + \mathbf{a}\cdot\boldsymbol{\nabla}\rho = -\rho\boldsymbol{\nabla}\cdot\mathbf{u} \tag{3.25a}$$

$$\rho\left(\frac{\partial\mathbf{u}}{\partial t} + (\mathbf{a}\cdot\nabla)\mathbf{u}\right) = \boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \rho\mathbf{u} \tag{3.25b}$$

Assuming the oil inside the transformer tank as a Newtonian fluid, eq. (3.25) automatically translates into the Navier-Stokes equations, and the mechanical constitutive equation is computed as

$$\boldsymbol{\sigma} = -p\mathbf{I} + \mathbf{s} = -p\mathbf{I} + \mathbb{C} : \boldsymbol{\nabla}^S \mathbf{u} \tag{3.26}$$

where $-p\mathbf{I}$ is the isotropic pressure tensor and $\mathbf{s}$ is the viscous stress tensor. In its most general form, the viscous tress tensor is assumed to depend linearly on the rate of change of the strains of an elementary volume of fluid, determined by the symmetric part of $\boldsymbol{\nabla} u$. Therefore it is expressed as the product of a fourth-order constant constitutive tensor $\mathbb{C}$ and the strain rate tensor $\boldsymbol{\nabla}^S \mathbf{u}$. In isotropic media, $\mathbb{C}$ has the form

$$\mathbb{C}_{ijkl} = \lambda \delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}) \quad i,j,k,l \in 1,2,3 \tag{3.27}$$

Here, μ is the dynamic viscosity coefficient and λ is the second viscosity coefficient. Both depend, slightly, on the density and the temperature of the fluid. The double contraction of a fourth-order tensor $\mathbb{C}$ with a second-order tensor $\boldsymbol{\nabla}^S \mathbf{u}$ is defined as $s_{ij} = \mathbb{C}_{ijkl} d_{kl}$ where s_{ij} are the components of $\mathbf{s} = \mathbb{C} : \boldsymbol{\nabla}^S \mathbf{u}$ and d_{kl} are the components $\mathbf{d} = \boldsymbol{\nabla}^S \mathbf{u}$. Now, in order to separate the Cauchy stress tensor into the isotropic and deviatoric parts, the same is done with $\mathbf{d}$.

$$d_{ij} = \frac{1}{3}\frac{\partial u_k}{\partial x_k} + \left(\frac{\partial u_i}{\partial x_j} - \frac{1}{3}\frac{\partial u_k}{\partial x_k}\right) \tag{3.28}$$

Replacing (3.27) and (3.28) in the constitutive equation (3.26) yields

$$\sigma_{ij} = \left(-p + \kappa \frac{\partial u_k}{\partial x_k}\right)\delta_{ij} + \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}\right) \tag{3.29}$$

The parameter $\kappa = \left(\lambda + \frac{2}{3}\mu\right)$ is the bulk viscosity, and as it is seen in (3.29), κ influences the addition of a viscous term into the isotropic part of the Cauchy stress tensor. As noted in [8], the assumption that κ is negligible (i.e. Stokes's hypothesis) is a common approach in the analysis of compressible fluids since it simplifies the isotropic part of the tensor. In those cases, however, it is not being neglected by assuming $\kappa \approx 0$ (as it tends to be of the same order of magnitude than μ or λ) but rather by the fact that $p >> \kappa \frac{\partial u_k}{\partial x_k}\delta_{ij}$. Although it is true that the pressure term will be higher in magnitude than the divergence term in the vast majority of cases, using this assumption would

underestimate the temporal variations of density in a problem in which pressure waves are expected. Indeed, the acoustic waves are associated with an oscillatory isotropic change in volume between opposite values [8] and therefore the Stokes' assumption is not considered in this case.

3.3 Extension to the weakly compressible regime

Eventually, the problem formulation for the fluid is concluded by outlining the strategy to tackle the fluid compressibility, or rather to overcome the issues emerging from the numerical implications of an incompressible regime. The difficulties in constructing a numerical method characterized by a divergence-free constraint on the velocity field are detailed in [14] and are characterized by the fact that the pressure role is to adjust itself on the domain to satisfy the incompressibility condition and therefore cannot be regarded as a state variable associated to a constitutive equation. Another difficulty, more concerting taking into account the nature of the problem at hand, is that the fluid flow would not sustain large pressure gradients, nor changes in the density due to the compressibility effects. The latter forces the propagation speed of the pressure waves to rise to large values (infinite if $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$) and accentuated in the low Mach number regime. Since this physical interpretation of the problem has detrimental effects on the validity of the results, the numerical simulation has to be extended to the weakly compressible regime, in which the hyperbolic-elliptic equations are kept in contrast to the usual compressible hyperbolic system.

Regarding time stepping schemes in numerical simulation of unsteady compressible problems, the explicit approach is rather common [28], although the tight constraint on the time step Δt such techniques pose make them highly inefficient for low Mach numbers. The CFL condition for such a case is

$$\frac{\Delta t}{h}\max(c + |\mathbf{u}|) \leq 1 \tag{3.30}$$

which means that the time step is subject to the bulk modulus, very high when dealing with quasi incompressible flows. In accordance, very low time steps would have to be employed to satisfy the condition. On the other hand, if the convection terms are approximated in an implicit manner, then the ideal time step is limited by [28]

$$\frac{\Delta t}{h}\max(|\mathbf{u}|) \leq 1 \tag{3.31}$$

By treating the convective terms implicitly, it is meant that in treating the nonlinear convective term $(\mathbf{a}\cdot\nabla)\mathbf{u}$, $\mathbf{a}$ and $\mathbf{u}$ are evalauted at t^{n+1}. The interesting features of an implicit approach are the the unconditional stability of the scheme and relaxation of the time-stepping constraint, although they increase the computational cost associated with the nonlinearity of the convective term [14].

Such implicit treatments in compressible solvers are formulated via the so-called density-based methods or pressure-based methods. While in density-based methods the pressure is computed through the equation of state from the conservative variables, pressure-based methods seem to be more robust [28]. This is a consequence of the fact that density becomes a constant in incompressible flows, and pressure adapts to satisfy this constraint, thus stating the need to avoid density as a primary variable. Pressure-based methods, in which pressure is the primary variable and density is obtained from the equation of state, do not have any such limitations and may be used regardless of the compressibility of the problem, also when shocks are present [2]. In using pressure-based methods, the mentioned divergence-free constraint will be replaced by the continuity equation of a compressible fluid, while keeping the momentum equation as originally. For this reason it will be referred to as the weakly-compressible approach, and is obtained after replacing the density partial time derivative by

$$\frac{\partial\rho}{\partial t} = \frac{1}{\rho c^2}\frac{\partial p}{\partial t} \tag{3.32}$$

In conclusion, the final form of the weakly-compressible Navier-Stokes equations is

$$\frac{1}{\rho c^2}\frac{\partial p}{\partial t} + \mathbf{a}\cdot\boldsymbol{\nabla}\rho + \rho\boldsymbol{\nabla}\cdot\mathbf{u} = 0 \tag{3.33a}$$

$$\rho\left(\frac{\partial\mathbf{u}}{\partial t} + (\mathbf{a}\cdot\nabla)\mathbf{u}\right) - \boldsymbol{\nabla}\cdot(\mathbb{C}:\nabla^s\mathbf{u}) + \boldsymbol{\nabla}p = \rho\mathbf{b} \tag{3.33b}$$

Then, to complete the system, the EOS will be used to compute the density. Several commentaries may be made regarding eq. (3.33):

- The term ρc^2, also called bulk modulus, indicates the amount of compressibility present. For neraly incompressible flows, the pressure signals propagate fast so that $\boldsymbol{\nabla}\cdot\mathbf{u}\approx 0$, with the flow conditions approaching the incompressible regime, without posing the mentioned problems related to it being exactly 0.
- The pressure and velocity fields are related, as in the momentum equation.

- The term $\boldsymbol{\nabla}\rho$ is not being neglected.
- Since the EOS does not depend on the temperature, the weakly-compressibility allows uncoupling the energy equation.
- For correct prediction of the physical phenomenon the velocities encountered in the modeled problem should be several orders of magnitude smaller than the finite sound speed introduced by the compressibility [30]. As shall be seen, this will be the case for the simulation.

This closes the problem formulation chapter. Next, it will be seen how the numerical approximation of this system is done by means of the Galerkin finite element discretization, and how and why the method will be stabilized.

Chapter 4

Methodology

4.1 Numerical approximation

4.1.1 The variational problem

In this section, the notation that will be used in order to describe the variational form of the problem in the domain is introduced. Let us define for $k = 1, 2, ...$

$$H^k(\Omega) = \{v \in L^2(\Omega) : D^\alpha v \in L^2(\Omega) \ |\alpha| \leq k\} \tag{4.1}$$

The Sobolev space $H^k(\Omega)$ consists of all functions v on Ω, that, together with its partial derivatives of order α, belong to the Hilbert space $L^2(\Omega)$. Let us define this space of square integrable functions as

$$L^2(\Omega) = \{v : v \quad \text{is defined on} \quad \Omega \quad \text{and} \int_\Omega v^2 dx < \infty\} \tag{4.2}$$

In order to reduce the continuity constraint on the derivatives of the unknowns in the Navier-Stokes equations, the weak form requires the introduction of classes of functions for the velocity field and the pressure field. With respect to the velocity $\mathbf{u}$ the space of admissible solutions is denoted by $\boldsymbol{\mathcal{S}}$. Since the velocity must satisfy Dirichlet boundary conditions on the no-slip walls of the domain, the trial solution space $\boldsymbol{\mathcal{S}}$ containing the approximating functions for the velocity will be:

$$\boldsymbol{\mathcal{S}} := \{\mathbf{u} \in H^1(\Omega) \,|\, \mathbf{u} = \mathbf{u_D} \,\text{on}\, \Gamma_D\} \tag{4.3}$$

The weighting functions of the velocity, $\mathbf{w}$, belong to the space $\boldsymbol{\mathcal{V}}$, which has the same properties as $\boldsymbol{\mathcal{S}}$ except that $\mathbf{w}$ need to vanish on the boundary in which velocity is prescribed, so that

$$\boldsymbol{\mathcal{V}} := \{\mathbf{w} \in H^1(\Omega) \,|\, \mathbf{w} = 0 \,\text{on}\, \Gamma_D\} \tag{4.4}$$

Eventually, the space of functions for the pressure is defined as $\mathcal{Q}$. As will be seen, spatial derivatives of pressure will not appear in the weak form of the problem, therefore the trial solutions for pressure are only required to be square-integrable. So far the procedure has been identical to what would be expected in a typical incompressible Navier-Stokes scenario. However, for the present case, two circumstances alter the common approach taken. First, pressure may be explicitly prescribed in the boundary, therefore the space for trial and weighting functions will not be the same. Second, an initial condition for the pressure field will have to be specified. This is a consequence of the fact that the pressure-based method for weakly-compressible flows introduces a temporal variation on the pressure, and thus it will have to be specified everywhere in the domain. In other words, the pressure will not be defined in a single point as in the case of purely Dirichlet boundary conditions for the velocity and incompressible flow but on the entire domain.

The trial solution space for pressure functional approximations will be

$$\mathcal{Q} := \{q \in H^0(\Omega) \,|\, p = p_D \,\text{on}\, \Gamma_D\} \tag{4.5}$$

To conclude with the notation, the symbol $\langle \cdot, \cdot \rangle$, is used to denote the integral of the product of two functions on the boundary Γ. The $L^2(\Omega)$ inner product in Ω is denoted by $(\cdot, \cdot)$. With that, the weak form of the problem is obtained by testing (3.33) against arbitrary test functions $\mathbf{w}$ and q . The weak form can be written as: find $\mathbf{u}$ and p belonging to the space of unknowns, such that

$$\left(q, \frac{1}{\rho c^2}\frac{\partial p}{\partial t}\right) + (q, \boldsymbol{\nabla} \cdot \mathbf{u}) = 0 \quad \forall q \in \mathcal{Q} \tag{4.6a}$$

$$\left(\mathbf{w}, \rho\frac{\partial \mathbf{u}}{\partial t}\right) + (\mathbf{w}, \rho(\mathbf{a} \cdot \nabla)\mathbf{u}) - (\mathbf{w}, \boldsymbol{\nabla} \cdot (\mathbb{C} : \nabla^s \mathbf{u})) = (\mathbf{w}, \rho\mathbf{b}) \quad \forall \mathbf{w} \in \boldsymbol{\mathcal{V}} \tag{4.6b}$$

together with appropriate sets of boundary and initial conditions. As for the initial

conditions, it will be setting the pressure field equal to the atmospheric pressure, and the boundary conditions will be the no-slip conditions and the bubble pressure fixed on some nodes of the domain.

4.1.2 The Galerkin finite element discretization

The Finite Element Method (FEM) approximation of the continuous variational problem in (4.6) is done by discretizing the domain Ω into element domains Ω_e, characterized by a mesh size h so that $\text{diam}(\Omega_e) \leq h$. The following is to define the interpolating spaces of continuous picewise polynomial test functions, so it is defined $\boldsymbol{\mathcal{V}}_h \subset \boldsymbol{\mathcal{V}}, \mathcal{Q}_h \subset \mathcal{Q}$ and their associated Galerkin projection of the solution by

$$\boldsymbol{\mathcal{V}}_h \equiv \text{span}\{N_1, ..., N_{Nu}\}; \quad \mathcal{Q}_h \equiv \text{span}\{\hat{N}_1, ..., \hat{N}_{Np}\}; \tag{4.7}$$

with N_j and $\hat{N}_j$ being the shape functions associated with node j of the element. The Galerkin FEM problem will thus consist of finding a finite element solution $\mathbf{u}_h, p_h$ belonging to the interpolating space, and satisfying equation (4.6). The interpolation of the different variables will be

$$\begin{aligned} \mathbf{u}(\mathbf{x}, t) \approx \mathbf{u}_h(\mathbf{x}, t) &= \sum_{j=1}^{N_u} \mathbf{u}_j(t) N_j(\mathbf{x}) \\ p(\mathbf{x}, t) \approx p_h(\mathbf{x}, t) &= \sum_{j=1}^{N_p} p_j(t) \hat{N}_j(\mathbf{x}) \\ \partial_t \mathbf{u}(\mathbf{x}, t) \approx \partial_t \mathbf{u}_h(\mathbf{x}, t) &= \sum_{j=1}^{N_u} \partial_t \mathbf{u}_j(t) N_j(\mathbf{x}) \end{aligned} \tag{4.8}$$

The velocity and pressure interpolations for the whole set of simulations will be made on a simplex element, in which both velocity and pressure are continuous linear, making olny use of the base element nodes, as shown in Fig. 4.1.

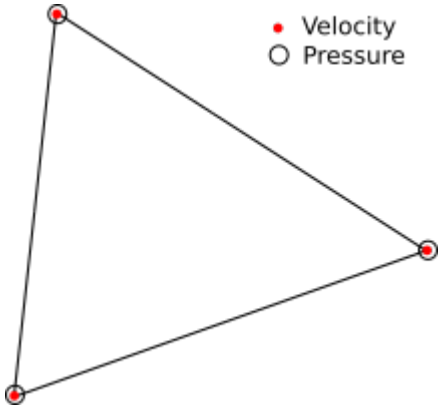


Figure 4.1: Simplex element used.

The choice of this element, clearly being convenient in terms of computational cost, does not satisfy the LBB condition on coupling the velocity and pressure FE spaces. This condition states that velocity and pressure spaces cannot be chosen arbitrarily, and in order to obtain a stable finite element approximate solution $\mathbf{u}_h$ and p_h, a suitable pair of spaces, $\boldsymbol{\mathcal{V}}_h$ and $\mathcal{Q}_h$, must be chosen [14]. As this is not the case, the FE Galerkin formulation will be unstable and a stabilization technique will be necessary to circumvent this drawback. This technique will also be of use to overcome the unstable nature of the non-symmetric operators, when the convection is dominant.

4.1.3 The space discrete variational multi-scale stabilized finite element formulation

As an alternative to the classic SUPG or GLS stabilization techniques, the variational multi-scale method (VMS) is used to construct what will later evolve into the sub-grid scale (SGS) method. The VMS exploits the idea that the discretisation of the domain into a FE mesh can only provide a coarse-scale part of the solution $(\mathbf{u}_h, p_h)$, while there is another fine-scale component that completes the solution and cannot be computed by means of a FE mesh $(\mathbf{u}_s, p_s)$. Instead, it is resolved analytically.

The additive decomposition of the solution and the corresponding test functions,

$$\mathbf{u} = \mathbf{u}_h + \mathbf{u}_s, \quad \mathbf{w} = \mathbf{w}_h + \mathbf{w}_s \tag{4.9a}$$

$$p = p_h + p_s, \quad q = q_h + q_s \tag{4.9b}$$

needs to be accompanied by the corresponding splitting of the functional spaces $\boldsymbol{\mathcal{V}} = \boldsymbol{\mathcal{V}}_h \oplus \boldsymbol{\mathcal{V}}_s$ and $\mathcal{Q} = \mathcal{Q}_h \oplus \mathcal{Q}_s$.

At this point it is introduced the residual vector $\mathbf{R}(\mathbf{u}, p) = (R_\rho(\mathbf{u}, p), \mathbf{R_m}(\mathbf{u}, p))$, which is composed by the residuals of the two equations in (3.33),

$$R_\rho(\mathbf{u}, p) = -\left(\frac{1}{\rho c^2}\frac{\partial p}{\partial t} + \frac{1}{\rho}\mathbf{a}\cdot\boldsymbol{\nabla}\rho + \boldsymbol{\nabla}\cdot\mathbf{u}\right) \tag{4.10a}$$

$$\mathbf{R_m}(\mathbf{u}, p) = \rho\mathbf{b} - \rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{a}\cdot\nabla)\mathbf{u}\right) + \boldsymbol{\nabla}\cdot(\mathbb{C} : \nabla^s\mathbf{u}) - \boldsymbol{\nabla}p \tag{4.10b}$$

Now, as an attempt to minimize the weighted sum of the residuals, the following FE functional is proposed

$$\Psi(\mathbf{w}, q, \mathbf{u}, p) = (q, R_\rho(\mathbf{u}, p)) + (\mathbf{w}, \mathbf{R_m}(\mathbf{u}, p)) \tag{4.11}$$

Because of the linearity of the problem considered so far, the weak form of the problem now becomes: find $\mathbf{u_h}$ and p_h such that

$$\begin{aligned} &-\left(q_h, \frac{1}{\rho c^2}\frac{\partial p_h + p_s}{\partial t}\right) + \left(q_h, \frac{1}{\rho}\mathbf{a}\cdot\boldsymbol{\nabla}\rho\right) + (q_h, \boldsymbol{\nabla}\cdot(\mathbf{u}_h + \mathbf{u}_s)) + (\mathbf{w}_h, \rho\mathbf{b}) \\ &-\left(\mathbf{w}_h, \rho\frac{\partial(\mathbf{u}_h + \mathbf{u}_s)}{\partial t}\right) - (\mathbf{w}_h, \rho\mathbf{a}\cdot\boldsymbol{\nabla}(\mathbf{u}_h + \mathbf{u}_s)) + (\mathbf{w}_h, \boldsymbol{\nabla}\cdot(\mathbb{C} : \nabla^s(\mathbf{u}_h + \mathbf{u}_s))) \\ &-(\mathbf{w}_h, (\boldsymbol{\nabla}(p_h + p_s))) = 0 \quad \forall q_h \in \mathcal{Q}_h,\ \mathbf{w}_h \in \boldsymbol{\mathcal{V}}_h \end{aligned} \tag{4.12a}$$

$$\begin{aligned} &-\left(q_s, \frac{1}{\rho c^2}\frac{\partial p_h + p_s}{\partial t}\right) + \left(q_s, \frac{1}{\rho}\mathbf{a}\cdot\boldsymbol{\nabla}\rho\right) + (q_s, \boldsymbol{\nabla}\cdot(\mathbf{u}_h + \mathbf{u}_s)) + (\mathbf{w}_s, \rho\mathbf{b}) \\ &-\left(\mathbf{w}_s, \rho\frac{\partial(\mathbf{u}_h + \mathbf{u}_s)}{\partial t}\right) - (\mathbf{w}_s, \rho\mathbf{a}\cdot\boldsymbol{\nabla}(\mathbf{u}_h + \mathbf{u}_s)) + (\mathbf{w}_s, \boldsymbol{\nabla}\cdot(\mathbb{C} : \nabla^s(\mathbf{u}_h + \mathbf{u}_s))) \\ &-(\mathbf{w}_s, (\boldsymbol{\nabla}(p_h + p_s))) = 0 \quad \forall q_s \in \mathcal{Q}_h,\ \mathbf{w}_s \in \boldsymbol{\mathcal{V}}_s \end{aligned} \tag{4.12b}$$

Eq. (4.12a) determines the resolved scales, whereas (4.12b) governs the problem for the unresolved scales. The goal is to solve (4.12b) analytically as a function of the coarse scale solution and substitute it back to (4.12a) to get rid of the fine scale solution.

So far, the solution for the fine scale problem has not been computed, but it is clear that in determining the coarse solution, the fine scale problem must first be translated into

a problem that can be solved. Given the particularities of the fine scale finite element space, an analytical solution is not available and an approximation will have to be taken [11]. The algebraic equation that results of approximating the exact solution of the fine scale problem is

$$p_s = \tau_1 R_\rho(\mathbf{u}_h, p_h) \tag{4.13a}$$

$$\mathbf{u}_s = \tau_2 \mathbf{R}_\mathbf{m}(\mathbf{u}_h, p_h) \tag{4.13b}$$

Where τ_1, τ_2 are stabilization parameters and are sought to be an approximation of the inverse of the differential operator in eq. (4.12b). Specifically

$$\tau_1 = \left(\frac{c_1 \mu}{h^2} + \frac{c_2 \rho ||\mathbf{a}||}{h} \right)^{-1} \quad \tau_2 = \frac{h^2}{c_1 \tau_1} = \mu + \frac{c_2 \rho h ||\mathbf{a}||}{c_1} \tag{4.14}$$

where $c_1 = 4$, $c_2 = 2$ and h is the characteristic element size.

Before substituting this result into the coarse scale problem, another commonly used, and computationally efficient assumption is taken regarding the partial time derivative of the subscales. The subscales are termed dynamic or quasi-static if they are considered to be time-dependent or not, respectively. As shown in (Codina, Principe, Guasch, and Badia, 2007), the dynamic treatment leads to a correct behavior of time integration schemes and better accuracy, as could be expected [11]. However, this is at the expense of the computational cost, therefore the quasi-static treatment of partial time derivatives is selected. Therefore, $\frac{\partial \mathbf{u}_s}{\partial t} \approx 0$ and $\frac{\partial p_s}{\partial t} \approx 0$. It is also standard to impose $\mathbf{u}_s, p_s = 0$ along the finite element edges in order to localize the fine-scale problem in the interior of each finite element [14].

Once this is done, there is already a space discrete variational multi-scale stabilized finite element formulation. However, in order to adapt the formulation to the linear element and reduce the order on the pressure and subscale terms, integration by parts is required. In detail, the terms that will undergo a modification are the following:

$$(q_h, \boldsymbol{\nabla} \cdot (\mathbf{u}_h + \mathbf{u}_s)) = (q_h, \boldsymbol{\nabla} \cdot \mathbf{u}_h) - (\boldsymbol{\nabla} q_h, \mathbf{u}_s) + \underbrace{\langle q_h, \mathbf{u}_s \cdot \mathbf{n} \rangle}_{\approx 0} \tag{4.15a}$$

$$(\mathbf{w}_h, \rho \mathbf{a} \cdot \boldsymbol{\nabla}(\mathbf{u}_h + \mathbf{u}_s)) = (\mathbf{w}_h, \rho \mathbf{a} \cdot \boldsymbol{\nabla} \mathbf{u}_h) - (\boldsymbol{\nabla} \mathbf{w}_h, \rho \mathbf{a} \mathbf{u}_s) - (\mathbf{w}_h, \rho \mathbf{u}_s \cdot \boldsymbol{\nabla} \mathbf{a}) + \underbrace{\langle \mathbf{w}_h, \rho \mathbf{a} \mathbf{u}_s \cdot \mathbf{n} \rangle}_{\approx 0} \tag{4.15b}$$

$$(\mathbf{w}_h, \boldsymbol{\nabla} \cdot (\mathbb{C} : \nabla^s (\mathbf{u}_h + \mathbf{u}_s))) = -(\boldsymbol{\nabla} \mathbf{w}_h, \mathbb{C} : \nabla^s (\mathbf{u}_h + \mathbf{u}_s)) + \langle \mathbf{w}_h, (\mathbb{C} : \nabla^s (\mathbf{u}_h + \mathbf{u}_s)) \rangle \tag{4.15c}$$

$$(\mathbf{w}_h, \boldsymbol{\nabla}(p_h + p_s)) = -(\boldsymbol{\nabla} \cdot \mathbf{w}_h, p_h) - (\boldsymbol{\nabla} \cdot \mathbf{w}_h, p_s) + \langle p_h, \mathbf{w}_h \cdot \mathbf{n} \rangle + \underbrace{\langle p_s, \mathbf{w}_h \cdot \mathbf{n} \rangle}_{\approx 0} \tag{4.15d}$$

It is interesting to note that the integration of the sub-scale terms on the boundary are approximated to zero, so that the only contribution to the boundary integral comes from the Cauchy traction vector, i.e. $\langle \mathbf{w}_h, \mathbf{t} \rangle = \langle \mathbf{w}_h, (\mathbb{C} : \nabla^s \mathbf{u}_h - p_h \mathbf{I}) \cdot \mathbf{n} \rangle$. Eventually, by inserting eqs. (4.15) into (4.12a) and adopting the quasi-static consideration, it is obtained the final Galerkin functional to be implemented.

$$\begin{aligned} &-\left(q_h, \frac{1}{\rho c^2} \frac{\partial p_h}{\partial t}\right) - \left(q_h, \frac{1}{\rho} \mathbf{a} \cdot \boldsymbol{\nabla} \rho\right) - (q_h, \boldsymbol{\nabla} \cdot \mathbf{u}_h) + (\boldsymbol{\nabla} q_h, \mathbf{u}_s) + (\mathbf{w}_h, \rho \mathbf{b}) \\ &\quad -\left(\mathbf{w}_h, \rho \frac{\partial \mathbf{u}_h}{\partial t}\right) - (\mathbf{w}_h, \rho \mathbf{a} \cdot \boldsymbol{\nabla} \mathbf{u}_h) + (\boldsymbol{\nabla} \mathbf{w}_h, \rho \mathbf{a} \mathbf{u}_s) + (\mathbf{w}_h, \rho \mathbf{u}_s \cdot \boldsymbol{\nabla} \mathbf{a}) \\ &-(\boldsymbol{\nabla} \mathbf{w}_h, \mathbb{C} : \nabla^s (\mathbf{u}_h + \mathbf{u}_s)) + \langle \mathbf{w}_h, \mathbf{t} \rangle + (\boldsymbol{\nabla} \cdot \mathbf{w}_h, p_h) + (\boldsymbol{\nabla} \cdot \mathbf{w}_h, p_s) = 0 \end{aligned} \tag{4.16}$$

It will be implemented symbolically with Python, and a template in C++ will also be created so that the new element, `WeaklyCompressibleNavierStokes`, may be compiled inside the `Custom Elements` folder of the Kratos Multiphysics repository. The template contains the LHS and RHS derivation of the element, include the nodal data variables in their array and matrices form. It also contains the stabilization parameters.

4.2 Shock-capturing methods

During the analysis of the first simulations, the presence of overshoots and undershoots around the pressure waves and the bubble interface is early observed. While some

simulations may seek to reduce the numerical diffusion to achieve a proper reproduction of the differential equations, it will be of interest in the present case to add artificial diffusion to smooth out the pressure sharp features. In resolving these high-gradient features, also referred to as unstable subgrid-scale (SGS), the difficulties presented in the numerical code to maintain accurate, stable and physical results are reduced.

The goal sought when applying such shock capturing methods, a particular branch of SGS that deals with the numerical oscillations and nonlinear stability associated with large pressure differences from node to node [17], is to alleviate the resolution constraint when capturing such sharp features, as the artificial diffusion added helps increasing the discretization distance occupied by the singularity, thus giving the numerical code a higher scope of action to stabilize and avoid non-physical results such as negative pressures around the bubble interface.

The stabilization approach selected for the problem is to increase the physical values of the bulk viscosity (κ) and shear viscosity (μ) over the smallest distance allowed by the discretization [17]. In particular,

$$\kappa = \kappa_f + \kappa^*, \qquad \mu = \mu_f + \mu^* \tag{4.17}$$

Here, the subindex f and $*$ indicate physical and artificial values, respectively. The pressure waves are stabilized through κ^*, whereas μ^* serves to stabilize the shear gradient. The detailed procedures to compute the artificial values are complex and are detailed in [17]. As a summary, the unstable SGS features are detected through the use of shock and shear sensors, and the artificial values are applied in this zones accordingly. The goal of such sensors is thus to identify pressure waves and high-gradient shear layers.

In order to take into account the new values for κ and μ, the constitutive laws will have to be subect to modifications. In particular, the Newtonian Constitutive law implemented in Kratos has taken into consideration the Stokes' hypothesis, hence κ is not considered. For that reason, the constitutive equation in (3.29) will be re-implemented inside the following files:

- `fluid_constitutive_law`: it will compute the fourth-order constitutive order $\mathbb{C}$, written in Voigt notation.
- `newtonian_compressible_{2,3}d_law`: computes the viscous stress tensor $\mathbf{s}$ in equation (3.26).

First, let us focus on the constant fourth-order (viscosity) constitutive tensor $\mathbb{C}$. It

was stated in eq. (3.27), as a function of the dynamic viscosity μ and second viscosity coefficient λ. Now, by replacing $\lambda = \kappa - \frac{2}{3}\mu$, it is obtained

$$\mathbb{C}_{ijkl} = \kappa\delta_{ij}\delta_{kl} + \mu\Big(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk} - \frac{2}{3}\delta_{ij}\delta_{kl}\Big) \tag{4.18}$$

Writing it in Voigt form, the general 3D form of the tensor is obtained

$$\mathbb{C} = \begin{bmatrix} \kappa + \frac{4\mu}{3} & \kappa - \frac{2\mu}{3} & \kappa - \frac{2\mu}{3} & 0 & 0 & 0 \\ \kappa - \frac{2\mu}{3} & \kappa + \frac{4\mu}{3} & \kappa - \frac{2\mu}{3} & 0 & 0 & 0 \\ \kappa - \frac{2\mu}{3} & \kappa - \frac{2\mu}{3} & \kappa + \frac{4\mu}{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix} \tag{4.19}$$

For a 2D case, the tensor will be

$$\mathbb{C} = \begin{bmatrix} \kappa + \frac{4\mu}{3} & \kappa - \frac{2\mu}{3} & 0 \\ \kappa - \frac{2\mu}{3} & \kappa + \frac{4\mu}{3} & \kappa - \frac{2\mu}{3} & 0 \\ 0 & 0 & \mu \end{bmatrix} \tag{4.20}$$

As for the viscous stress tensor, let us retrieve eq. 3.29 and separate its terms into the pressure and viscous parts. From this equation, the stress tensor in index notation is given by

$$s_{ij} = \mu\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3}\frac{\partial v_k}{\partial x_k}\delta_{ij}\right) + \kappa\frac{\partial v_k}{\partial x_k}\delta_{ij} \tag{4.21}$$

The viscous stress vector will then be, in Voigt form

$$\mathbf{s} = \begin{cases} 2\mu\Big(\Big[\boldsymbol{\nabla}^S\mathbf{u}\Big]_i - \frac{Tr\left(\boldsymbol{\nabla}^S\mathbf{u}\right)}{3}\Big) + \kappa Tr\Big(\boldsymbol{\nabla}^S\mathbf{u}\Big) & i \in (0,1,2) \\ \mu\Big[\boldsymbol{\nabla}^S\mathbf{u}\Big]_i & i \in (3,4,5) \end{cases} \tag{4.22}$$

Whereas, for a 2D case, it will have the form

$$\mathbf{s} = \begin{cases} 2\mu\Big(\Big[\boldsymbol{\nabla}^S\mathbf{u}\Big]_i - \frac{Tr\left(\boldsymbol{\nabla}^S\mathbf{u}\right)}{3}\Big) + \kappa Tr\Big(\boldsymbol{\nabla}^S\mathbf{u}\Big) & i \in (0,1) \\ \mu\Big[\boldsymbol{\nabla}^S\mathbf{u}\Big]_i & i \in (2) \end{cases} \tag{4.23}$$

The general form of the equations for the constitutive tensor and stress vector will allow

us to compute them either in case shock capturing techniques are applied or not. The latter will only require substituting the physical values, without presence of articifial additions, whereas the former will require substituting κ and μ by their new expressions in eq. 4.17. When doing so, it is assumed that $\kappa_f = 0$.

4.3 Numerical treatment of the interface

The multi-fluid technique which has been selected to track the position and shape of the interface is the Level-Set method, for the clear benefits it presents. Moreover, and particularly for the present problem, the initialization of the distance function is trivial and does not require an iterative algorithm to find the closest distance to the interface of each node. Rather, it computes the distance to the interface as the difference of radius. Once the position of the interface is obtained, the volume of gas may be updated and the gas pressure (considered homogeneous in the gas domain) may be computed with the perfect gas law according to algorithm 1.

Algorithm 1 Bubble pressure computation.

$t = 0$
Initial bubble radius R_0
Initial bubble center (x_c, y_c, z_c)
Initial mass $m = m_0$
Initialize Level-Set function $d_i = \sqrt{(x_i - x_c)^2 + (y_i - y_c)^2 + (z_i - z_c)^2} - R_0^2$

while $t \leq t_{end}$ **do**:
    Compute bubble volume$\rightarrow$ Get V_{gas}
    Check time:
    if $t < 50 \times 10^{-3}$ **then**:
        $m+ = 5\Delta t$
    end if
    $\rho_{gas} = m/V_{gas}$
    $e = c_p T/\gamma$
    $p_{gas} = \rho_{gas} e(\gamma - 1)$
end while

The previous steps allow computing the pressure value in one part of the domain. For this reason, the nodes belonging to this part of the domain are assigned this pressure for the step computations, as of a Dirichlet boundary condition. The density ρ_{gas} is also fixed in the nodes of the bubble. The way it is done requires of a procedure to prepare these boundary conditions before solving for the state variables, in the `ApplyBoundaryConditions` step · There are two approaches to apply these conditions, related to the treatment of the nodes of the elements cut by the interface.

- The first consists of fixing the pressure on all the nodes of the cut element, as shown in Fig. 4.2. This means that, at each time step, all nodes are imposed free degrees of freedom for the pressure and velocity (except those at the wall, which have prescribed velocity from the no-slip condition). Then, the nodes with a red and blue dots are imposed fixed pressure degrees of freedom and assigned the gas pressure p_{gas}.
- The second approach consists of the same procedure but instead of fixing the pressure on the blue dotted node, it is rather fixed on the red dotted nodes only, that is, the gas pressure p_{gas} is imposed only on the nodes which are inside the interface, or, in other words, which have negative distance values.

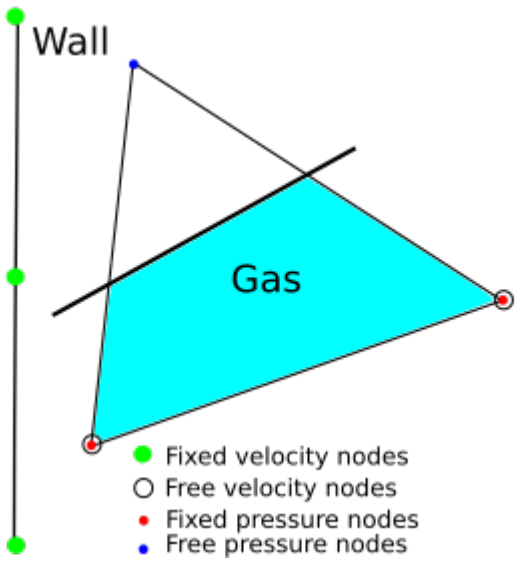


Figure 4.2: Interface element.

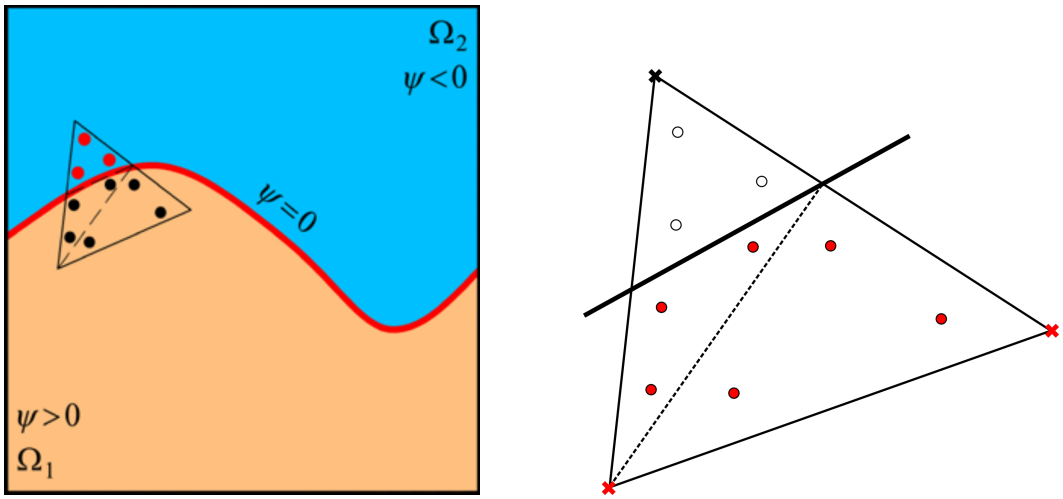
The second approach has been observed to give better results in terms of stability, apart from the fact that is less computationally expensive. The first approach, requires, on the other hand, a loop through all the nodes to compute its distance function and then

another loop through all the elements to assign p_{gas} to all its nodes.

Now let us focus on line 10 of the algorithm. It has been assumed that a constant mass inflow of 5 Kg/s is added to the bubble volume during the first 50 ms. This is in accordance with the data provided by Siemens Energy in order to model the gas bubble.

The fact that there are elements cut by the interface requires the use of Modified Shape functions that appear after generating new elements from the cut element. This can be seen clearly in Fig. 4.2, where the blue region represents the gas (Level-Set function $\psi < 0$), the yellow region represents the oil ($\psi > 0$) and the red line is the interface ($\psi = 0$). In that case, the element cut is rearranged into new triangular elements (2D in this case) and new Gauss points are generated for each new sub-element. This will later be of use to determine, for instance, the area or volume of the gas, the energy contained in the fluid or the normal vectors to the surface. In order to compute the normal vectors, the interface shape functions and gradients are also needed.

The latter is needed for better accuracy in the results, since not doing so would imply losing the contribution of the Gauss points belonging to the cut elements. Then, the weights and location of the new Gauss Points in Fig. 4.3b are used to integrate the density, pressure and velocity over the modified regions to compute its energy, for instance.



(a) Representation of a cut element on the domain. Extracted from [26].

(b) Detailed of the Modified Gauss Points for the cut element.

Figure 4.3: Modified integration rule for an element cut by the interface.

4.4 Modified python solver

A compressible solver which allows the resolution of this problem requires the customization of already existing python solvers in Kratos, with the goal of combining all the numerical particularities which the problem presents. Indeed, if comparing the new solver to the `navier_stokes_vms_monolithic.py`, the new solver presents the following modifications:

- Include the variables that are defined as being Nodal, so that the solver may access the value at each node. Examples of such variables are `DENSITY,SOUND_VELOCITY` and `DISTANCE`. In this case, the Distance variable is needed for tracking the interface. The other variables are those which were defined at the beginning when defining the new Element.
- Include the `ShockCapturingProcess`. This process, called at the `Initialize` step before constructing and initializing the solution strategy, is necessary for computing the values of `ARTIFICIAL_BULK_VISCOSITY` and `ARTIFICIAL_DYNAMIC_VISCOSITY`.
- Perform the level-set convection according to the previous step velocity.
- Recompute the distance field according to the new level-set position

The main functions regarding the level-set convection process are extracted from the Navier-Stokes two-fluid solver. During the `InitializeSolutionStep` process the level-set convection as well as the Distance re-initialization processes are performed. The re-initialization of the variable is necessary, as was explained in 2.2.2, as the level-set function tends to be distorted after a few computational steps. Eventually, in the `FinalizeSolutionStep`, the level-set convection process is performed again to complete the solution step.

Chapter 5

Tests and results

This chapter collects all the tests carried out to validate the implementation of the methodologies described in the previous chapters. First of all, the *Kratos Multiphysics* framework as well as the main aspects of the implementation are described. Secondly, the differents tests carried out are described together with the results.

Regarding the developed simulations, they are divided into 2D and 3D. First, a series of tests to validate the implementation are presented, such as convergence and energy conservation tests. Secondly, 2D cases are shown to evaluate the implemented code. Eventually, a series of 3D cases are included to assert the performance of the implementation in an FSI context as well as a rigid-wall context.

Unless otherwise stated, the parameters used to model the liquid in the tank are the following:

Parameter	Symbol	Value
Density	ρ	880 Kg/m^3.
Dynamic viscosity	μ	0.02 Ns/m^2.
Specific heat	c	1860 (J/Kg)/K
Heat capacity ratio	γ	1.4
Gas temperature	T	200 ^{o}C

Table 5.1: General simulation parameters.

Whilst the three last parameters are needed in the `FluidParameters.json`, the first one is needed for the Tait Equation of State.

5.1 Code implementation

5.1.1 Kratos Multiphysics and GiD framework

The code developed in this work has been implemented using *Kratos Multiphysics* (Kratos) open-source framework for the implementation of numerical methods in multi-disciplinary simulation software. The programming languages have been Python, for the running scripts and simple tasks, and C++, for the development of more advanced utilities and processes, which require to be run faster. The code has been created in the `FluidDynamicsApplication`, and some of it has already been merged into the Master branch of the repository.

Moreover, the professional version of the *GiD* commercial software has been used as pre and post-processor. GiD includes a friendly user interface that allows generating the Model Part files, as well as the simulation files, thanks to a specific *problemtype*. In this way, the problem-type allows to introduce all the problem settings such as materials, boundary conditions or solution strategies needed to perform the simulation, which is then executed with Kratos through the `MainKratos.py` file.

5.2 Validation tests

5.2.1 Energy conservation in the domain with MMS

The purpose of this test is to ensure that the power emitted from the bubble interface through pressure forces is in accordance with the rate of change of the total energy for the material volume described in Fig. 3.1. That is, the rate of change of the fluid particles' energy at each time step has to be equal to the power of the pressure forces at the interface nodes, as indicated by eq. (3.16).

Problem description

In order to check the latter, the method of manufactured solutions will be used. That is, by defining analytically a solution field for the state variables (pressure and velocity) and assuming density and sound velocity to remain constant, the energy conservation computed analytically will be compared to the numerical procedure developed in a separate utility. For that we propose a circular domain, such as that in Fig. 5.1, with

a circular hole in the middle representing the bubble, whose radius varies in time. The fluid properties in the domain are also imposed to vary in time through the following relations

$$p(R,t) = 100t(1+R), \quad v(R) = \frac{1}{1+R^2} \tag{5.1}$$

Where

$$R(t) = 0.1 + \alpha ct, \quad \text{with} \quad \alpha = 7 \times 10^{-3},\, c = 1.5 \times 10^{3} \tag{5.2}$$

The velocity is assumed to be in the radial direction, so its vector already contains the normal direction. Hence, the power emitted from the bubble will be

$$P(R,t) = \int_{\theta=0}^{\theta=2\pi} u(R)p(R,t)R\,d\theta \tag{5.3}$$

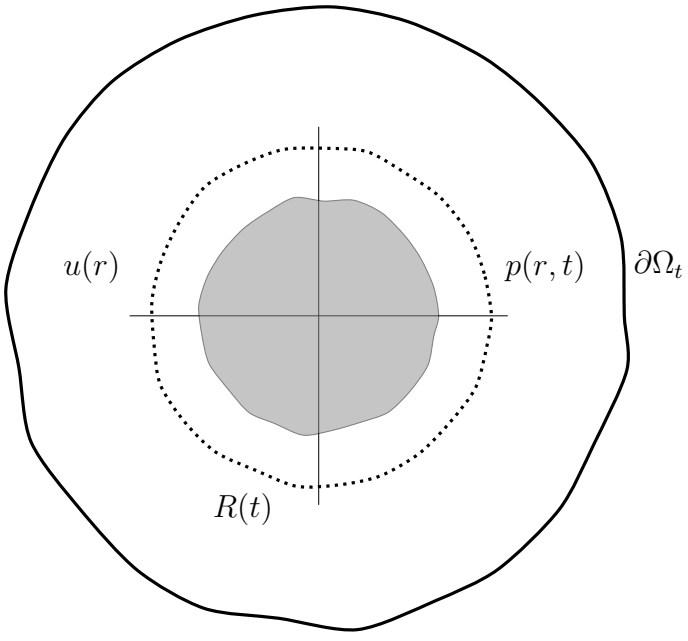


Figure 5.1: Schematic representation of the domain.

Equation (5.7) may be integrated symbolically to obtain an expression for the power at each time step, once the radius has been substituted. On the other hand, the analytical

value for the energy may be computed as

$$E(r,t) = \int_{r=R}^{r=R_d} \Big(p(r,t) + \frac{1}{2}\rho u(r)^2 \Big) 2\pi r \, dr \tag{5.4}$$

Where R_d is the radius of the entire domain. Once the analytical values are obtained, the numerical energy and power rate is obtained by imposing the manufactured velocity and pressure fields to the nodal degrees of freedom at each time step. For that, a substitution is again required and also a change of base, from cylindrical to Cartesian, since for the numerical utility both u_x and u_y are needed. Once the state variables are set, the utility is called and the numerical values for the energy and pressure rate are obtained. In order to compare both values, the power rate per unit time is modified as follows: $W^n = E^{n-1} + P\Delta t$, so that at each time step W^n has to be equal to the total energy in the domain E^n.

The total energy in the domain is composed by two parts: the contribution of the elements which are inside the fluid domain, and the contribution of the part of those which are cut by the interface, so that

$$E_{total} = \sum_{e=1}^{N_e} E_e, \quad E_e = \sum_{p=1}^{n_p} W_p \Big(\frac{1}{2}\rho_p \mathbf{u}_p \cdot \mathbf{u}_p + p_p \Big) \tag{5.5}$$

Being n_p the number of Gauss points on the element, E_e the elemental energy and N_e the number of elements. Depending on whether the elements are fully inside the fluid or cut by the interface, W_p will be the standard or the positive-side integration points' weight, referring to the positive side the part of the elements belonging to the fluid part (outwards of the interface). In order to compute the value of the variables in the Gauss points, a similar approach is followed

$$\Phi_p = \sum_{i=1}^{N_n} N_i^{(p)} \Phi_i \tag{5.6}$$

Where Φ_i is the variable evaluated at node i and $N_i^{(p)}$ contains the shape function for node i, evaluated at the Gauss point p. Again, depending on the element, the standard or the positive side shape functions will be used. Then, the work rate is computed as follows

$$W_{total} = \sum_{e=1}^{N_e} W_e, \quad W_e = \sum_{p=1}^{\hat{n}_p} \hat{W}_p \left(\hat{\mathbf{u}}_p \cdot \frac{\mathbf{n}_p}{|\mathbf{n}_p|} \hat{p}_p \right) \tag{5.7}$$

Here $\hat{n}_p$ is the number of interface Gauss points and $\mathbf{n}_p$ are the positive-side area normal vectors at Gauss point p. $\hat{\mathbf{u}}_p$ and $\hat{p}_p$ are computed in the same way as in the elements with the difference that now the shape functions are evaluated at the Gauss points of the interface. This is achieved through the `ComputeInterfacePositiveSideShapeFunctions` function.

The results are assessed by comparing the numerical and analytical values of the total energy rate of change and the work of pressure forces per unit time. For that, the relative error is computed using the analytical values as reference. Fig. 5.2 shows the error during 10 ms, when the radius bubble has increased from 0.1 to 0.205. The simplified energy rate explained in 3.2.3 is also shown, and it is concluded that it is equal to the work of pressure forces per unit time, as it was shown to be the application of the divergence theorem. The error shows the same decreasing trend for both values.

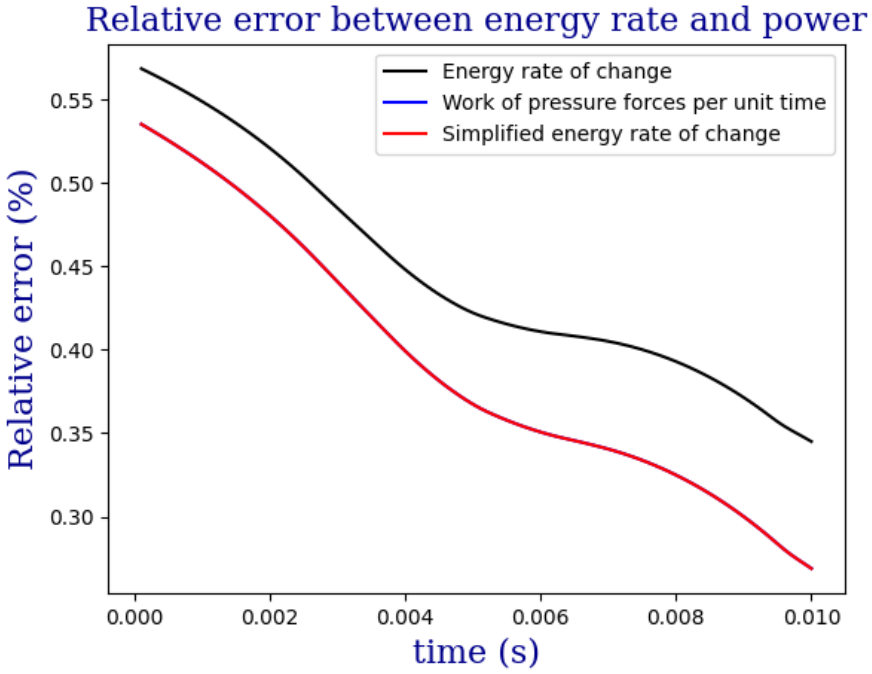


Figure 5.2: Comparison between analytical and numerical computations.

5.2.2 Energy conservation in a non-symmetric 3D domain

Problem description

Once the utility generated to asses the energy conservation has been proven to be valid with the MMS method, now it is applied to the simulation of a 3D domain in which a pressurized bubble changes its shape and transmits energy to the domain. In this case, the same goal is devised, as the rate of change of the energy in the fluid domain will be compared to the power of the pressure forces.

The main parameters used for this test are

Parameter	Symbol	Value
Initial radius	R_0	0.03 m ,
Initial mass	m_0	5.6×10^{-3} Kg
Initial bubble center		[6, 1, 3] m
Tank dimensions		$10 \times 4 \times 5$ m

Table 5.2: Simulation parameters for 3D energy conservation.

The results are shown in Fig. 5.3, where it can be concluded that, although the relative errors are negligible, the simplified version of the divergence theorem still provides better accuracy. The oscillations may be attributed to the sudden shape change of the bubble, which is no longer spherical once the simulation starts. The point is that, with two different methods, the energy is conserved if using the new formulation.

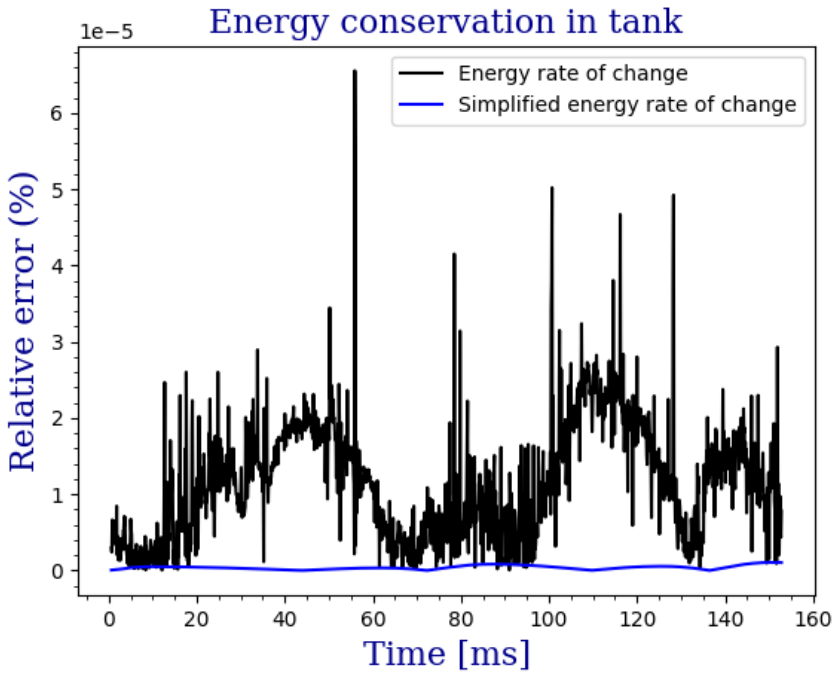


Figure 5.3: Comparison between energy numerical computations.

5.2.3 Convergence assessment in a non-symmetric 3D domain

The purpose of this test is to evaluate the error magnitude in a convergence graph when refining the mesh on a three-dimensional domain with a pressurized bubble located at a third of the diagonal of a unitary cube. All meshes are run up to 2 ms and the results at that time are used for comparison.

It is important to note that the solution fields are approximated using linear shape functions and that the domain is discretized using tetrahedral elements. This implies that the error of the state variables is quadratic with respect to the mesh size, so that, for instance, for the pressure field it holds that $p = p_h + O(h^2)$. Then, considering that the error estimates for the L2 space is expected to be of the form

$$||u - u_h||_{L_2} \leq Ch^{p+1} \tag{5.8}$$

The theoretical convergence rate will be assessed not by analytical results, since they are not available, but by a refined enough mesh ($h \longrightarrow 0$). This way, different mesh sizes will be tested against the finest mesh the computer allows. The finest mesh with

which the case could be run was $h = 0.015$ m, so that it will be considered the reference value.

The error estimate will be of the form

$$||u - u_h||_{L_2}^{(e)} = \sum_{p=1}^{n_p} \left(\sqrt{\frac{(u - u_h)^2 W_p}{u^2 W_p}} \right) \tag{5.9}$$

Where u_h is the approximated nodal solution and u is the solution of the finest mesh.

The process to compute the errors requires the mapping from the finest to the coarse mesh, so that in each node of the coarser meshes the numerical and *exact* values are available. This requires the use of the mapping application and the setting up of the case accordingly. The mapped solutions are written into a text file, which is then loaded by the created utility that computed the errors, `compute_errors.h`. This utility reads the reference values on the nodes and compares them to the approximated values by integrating the error over the element. For the elements which are not cut by the interface, `rGeom.ShapeFunctionsValues` is used, whereas for the nodes cut, `positive_side_sh_func` and `negative_side_sh_func` are used.

The parameters used to model the initial gas bubble are the following:

Parameter	Symbol	Value
Initial radius	R_0	0.1 m ,
Initial pressure	p_0	7.43 bar
Initial bubble center		[2, 2, 2]/3 m
Tank dimensions		$1 \times 1 \times 1$ m

Table 5.3: Simulation parameters for convergence analysis.

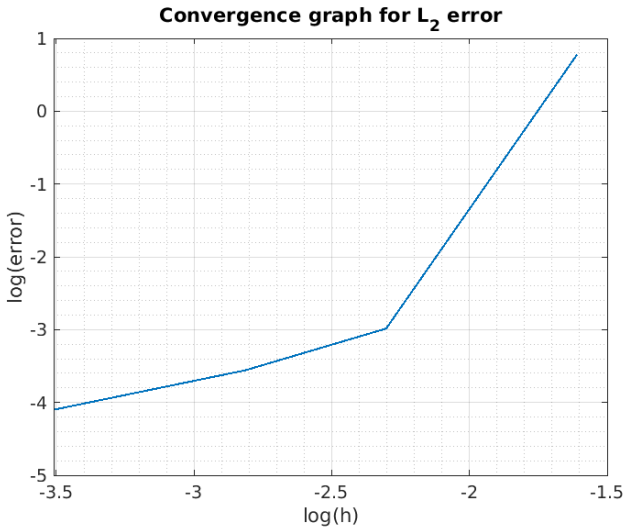
The mesh sizes chosen to perform the analysis have been, from finer to coarser: 0.03, 0.06, 0.1 and 0.2. The results for the error magnitude for the density, velocity norm and pressure and are presented in Fig. 5.4. As may be observed, the error decreases with the refinement of the mesh. Only for the sake of judging the mapping process, the errors obtained when mapping the solution field to the finest mesh have been included, and as can be seen, they are 0 at machine precision. The convergence graphs have been included in Fig. 5.4. Only for the velocity a straight line is obtained, and if a liner interpolation is performed, a slope of 1.925 is obtained, very close to the theoretical 2 for linear elements. For the pressure and density, the coarsest mesh gives wrong results, so it should not be used when interpolating linearly the results. All the same, the error

is decreased with the mesh refinement. If an interpolation is performed only for the linear range, slopes of 0.924 and 0.695 are obtained for density and pressure, in Fig. 5.4b and 5.4a, respectively.

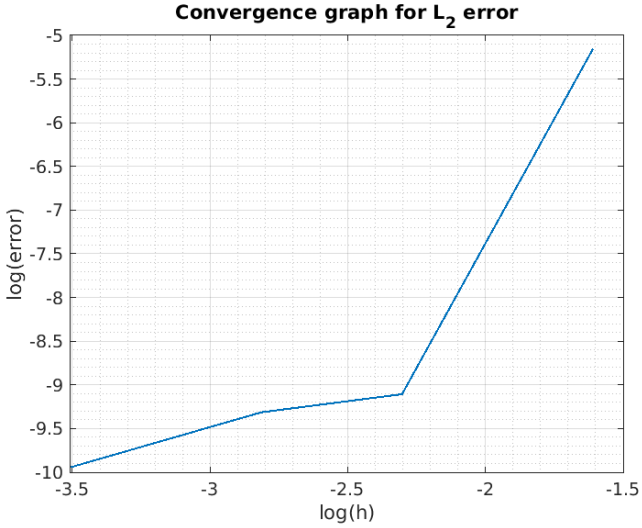
To complete the information, the resulting pressure field at a plane cutting through the bubble of the 3D domain is shown for all the meshes in Fig. 5.5. As seen, the results start looking similar for meshes of size $h = 0.06$ and higher. For coarser meshes, there are not enough nodes inside the bubble and its interface cannot be tracked consistently. In any case, this gives an indication on which mesh size to use, the correct one being $h = 0.0015$.

Convergence table				
Mesh size h	Nº nodes	$\|\|u-u_h\|\|_{L_2}$	$\|\|\rho-\rho_h\|\|_{L_2}$	$\|\|p-p_h\|\|_{L_2}$
0.2	298	3.5937	5.780e−3	2.1753
0.1	1997	0.6718	1.109e−4	0.0505
0.06	8759	0.2534	9.017e−5	0.0285
0.03	66469	0.0931	4.804e−5	0.0166
0.015	518722	2.389e−13	2.420e−16	2.738e−15

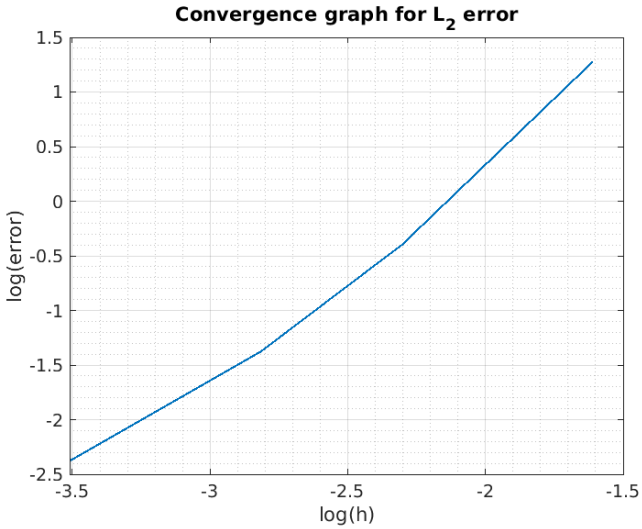
Table 5.4: Error magnitudes for the convergence analysis.



(a) Pressure convergence.

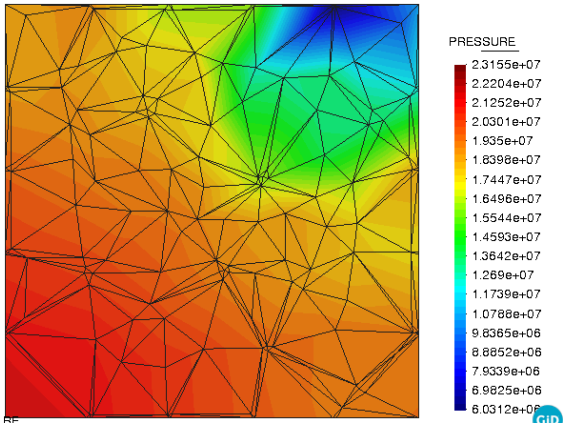


(b) Density convergence.

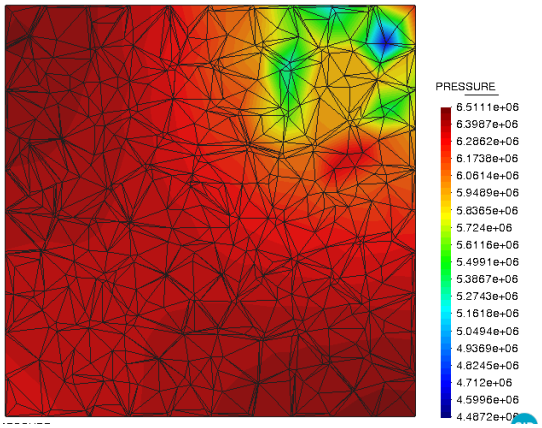


(c) Velocity convergence.

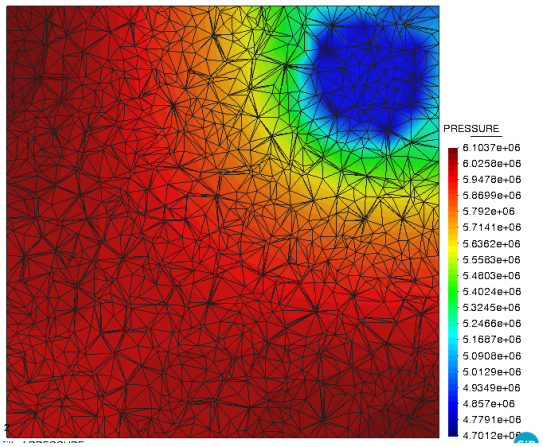
Figure 5.4: Convergence analysis.



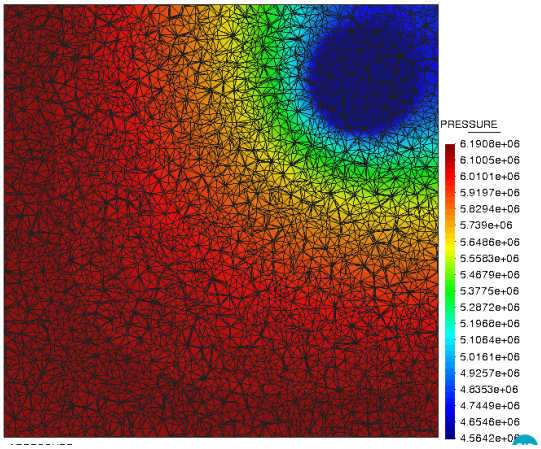
(a) Pressure field with $h = 0.2$ m.



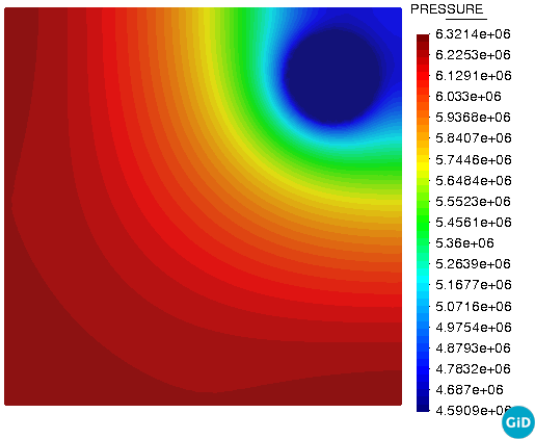
(b) Pressure field with $h = 0.1$ m.



(c) Pressure field with $h = 0.06$ m.



(d) Pressure field with $h = 0.03$ m.



(e) Pressure field with $h = 0.0015$ m.

Figure 5.5: Comparison between pressure field cut of the domain.

5.2.4 Evolution of a pressurized fluid region

The purpose of this test was to observe the evolution of a circular region with twice the atmospheric pressure than the rest of the domain. In this case, there is no interface, but the same fluid which is initialized with two different pressure values. This means that an initial pulse will be propagated through the domain until an steady state is obtained, since no energy will be added to the domain. Moreover, in this case, since there are no pressure forces acting on the domain boundary, the energy will remain constant throughout the simulation.

The initial pressure is set to be 2×10^5 Pa in a circular region of radius 0.2 m at the center of a 10×4 domain. Once the simulation starts, an initial pressure wave is generated and propagates outwards in a symmetrical manner given the circular shape of the pressurized region and absence of gravity. The expansion of the compressed region introduces energy in the rest of the domain, and as a result the medium is compressed, as opposed to the central region, which reduces its density and pressure (Fig. 5.6). Then, when the wave interacts with the solid walls, the pressure is reflected and superimposed to the incoming wave.

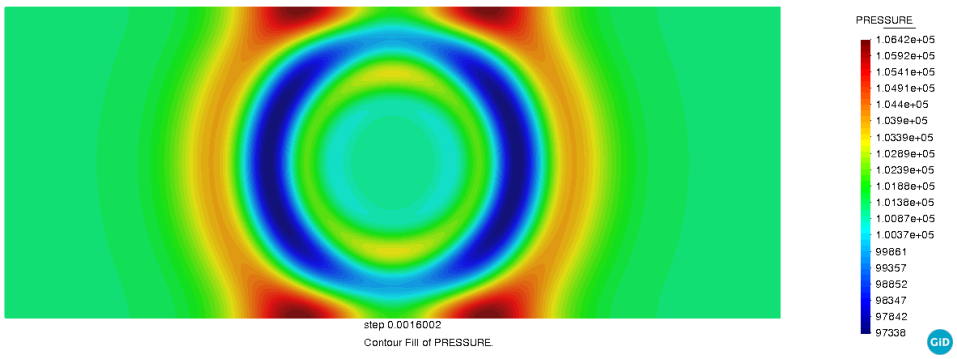


Figure 5.6: Evolution of the pressure ring.

The energy in this case was perfectly conserved, since there was no interface and the whole domain could be considered as a single fluid (see Fig. 5.7). Interestingly, this configuration only allowed setting up a pressure of 2×10^5 Pa in the circular region at the center, since higher pressures would lead to divergence problems. When modeling the bubble with the interface, the pressure in the bubble can be set to be much higher.

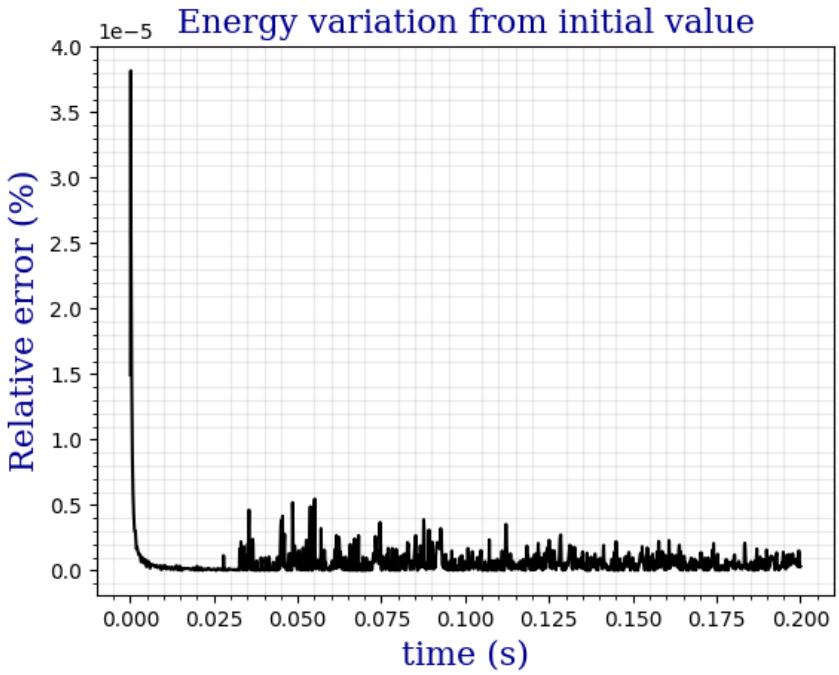


Figure 5.7: Assessment of energy conservation in the sealed tank.

5.2.5 Two-dimensional simulation of a double underwater explosion

This test is extracted from [37] as a tool to evaluate the level-set representation of a more complicated distance function. In this case, two underwater explosions are simulated by generating two highly pressurized gas bubbles of the same radius, symmetrically positioned with respect to the domain, which has dimensions 4×4. Regarding the bubbles, their position is $(1.4, 2)$ and $(2.6, 2)$ with 0.3 radius each. A representation of the case can be seen in Fig. 5.7. The initial pressure and density for the bubbles is $p = 59$ bars, $\rho = 17.7$ Kg/m^3, whereas the initial pressure and density for the water part is 1 bar and 1000 Kg/m^3. These are not the parameters of the original problem, as it contains a pressure ratio of 1000, which is not a feasible value for the developed code. As mentioned in the state of the art, the ghost fluid method used in this paper allows for higher pressure ratios.

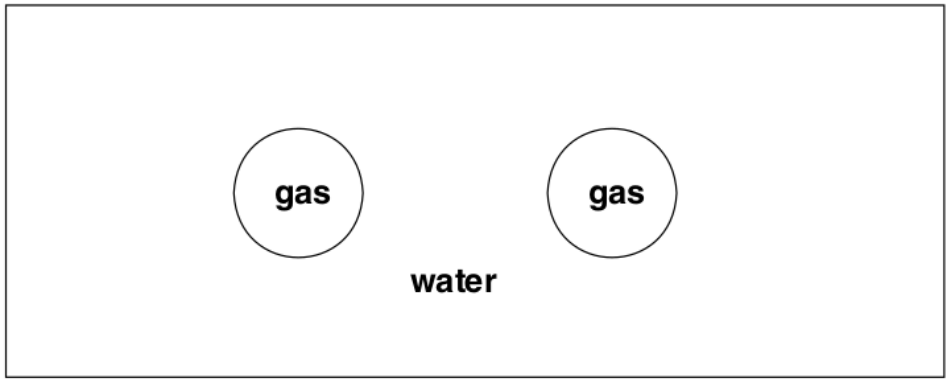


Figure 5.8: Schematics of the problem setup.

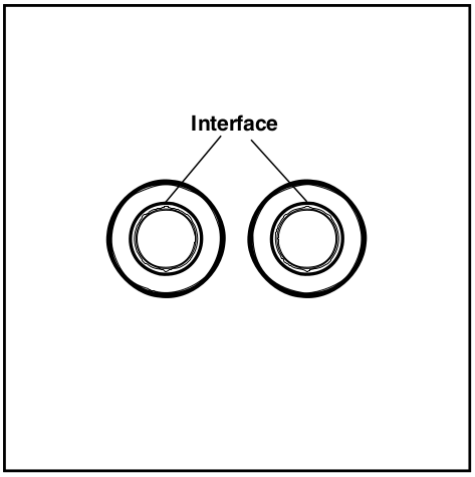
In any case, the values set for this problem will allow a fair comparison in terms of contour pressure lines during the first time steps of the simulation, since its purpose is not to compute the bubble pressures but the efficiency of the level-set technique in generating a symmetric and uniform solution.

At the initial time steps, two pressure waves are generated from the bubble surfaces and expand to the walls. At time $t = 1.8 \times 10^{-4}$, shortly after the situation depicted in 5.9, the contact of the waves produces two reflected pressure waves traveling in the opposite direction, and the pressure increases in the contact region, see Fig. 5.10. In Fig. 5.11, the reflected waves reach the gaseous interface and originate further waves inside the bubbles, which will change its shape shortly afterwards. Fig. 5.12 shows the level-set contours at the same time step.

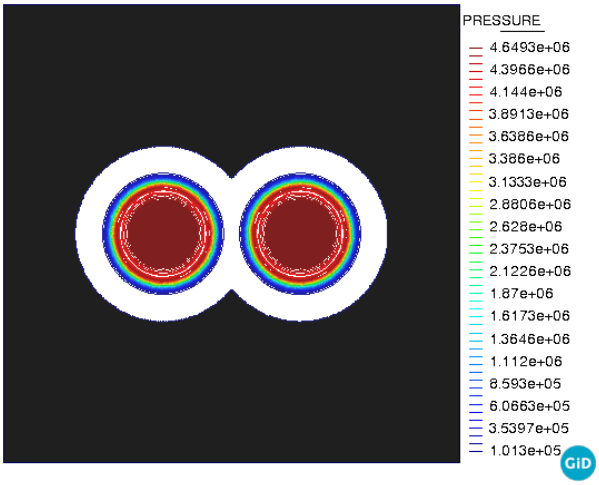
By observing the similarity between the results, it can be concluded that the weakly compressible approach is capable, first, of describing the propagation of pressure waves at the correct sound speed, and second, of modeling correctly the interaction between equal waves and their reflection. As mentioned, the results were not expected to be completely the same since the reference pressure ratio has not been selected and the modeling approach is not fully compressible but quasi-compressible. On the other hand, the efficiency of the level-set when tracking the position of the interfaces and the contours after shock collision has been assessed positively.

5.3 Two-dimensional tests

These tests constitute the fist approximation towards computing the pressures on the walls of a tank, assuming it is 2D. The tank is modeled as a 10×4 rectangle and the



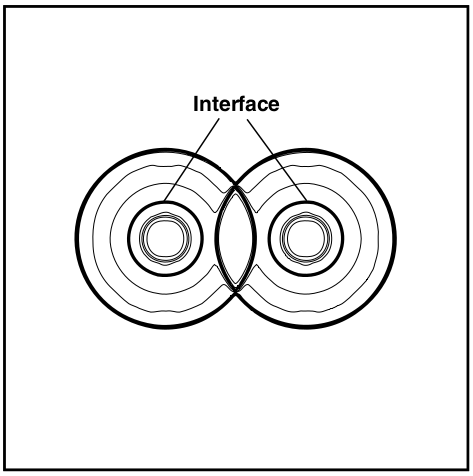
(a) Reference pressure contours [37]



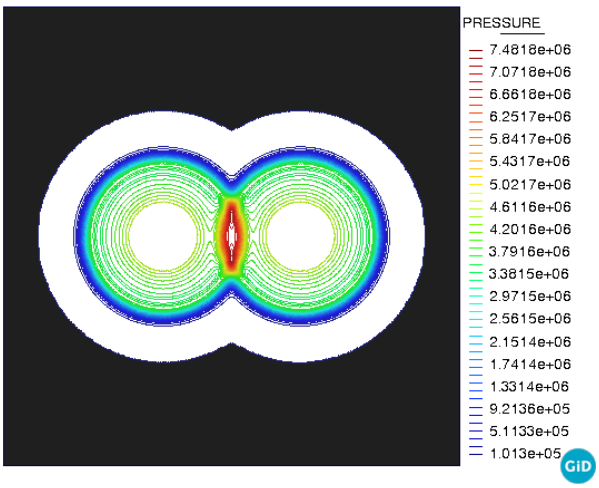
(b) Pressure contours with Kratos.

.

Figure 5.9: Comparison at $t = 1.2 \times 10^{-4}$.



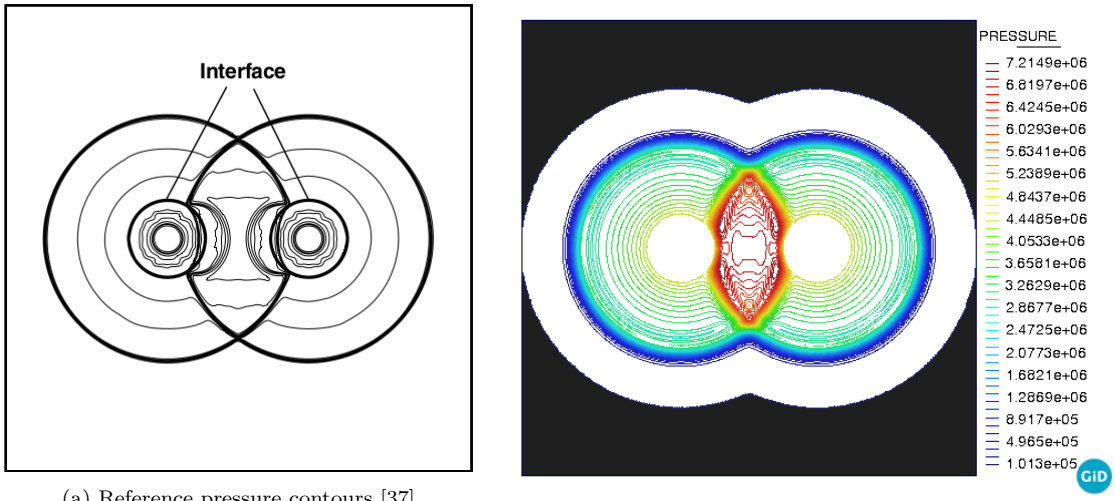
(a) Reference pressure contours [37]



(b) Pressure contours with Kratos.

.

Figure 5.10: Comparison at $t = 2.9 \times 10^{-4}$.

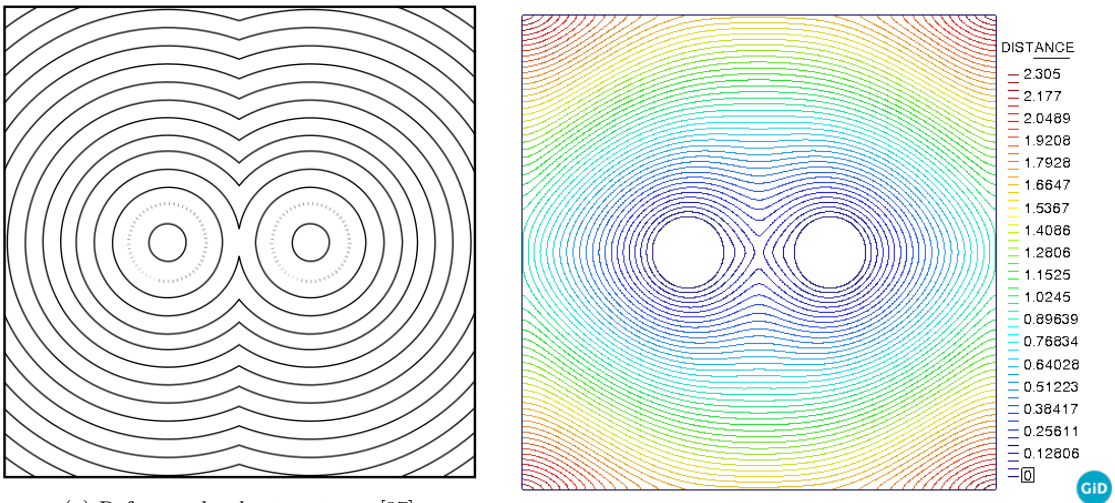


(a) Reference pressure contours [37]

(b) Pressure contours with Kratos.

.

Figure 5.11: Comparison at $t = 4.6 \times 10^{-4}$.



(a) Reference level-set contours [37]

(b) Level-set contours with Kratos.

.

Figure 5.12: Comparison at $t = 4.6 \times 10^{-4}$.

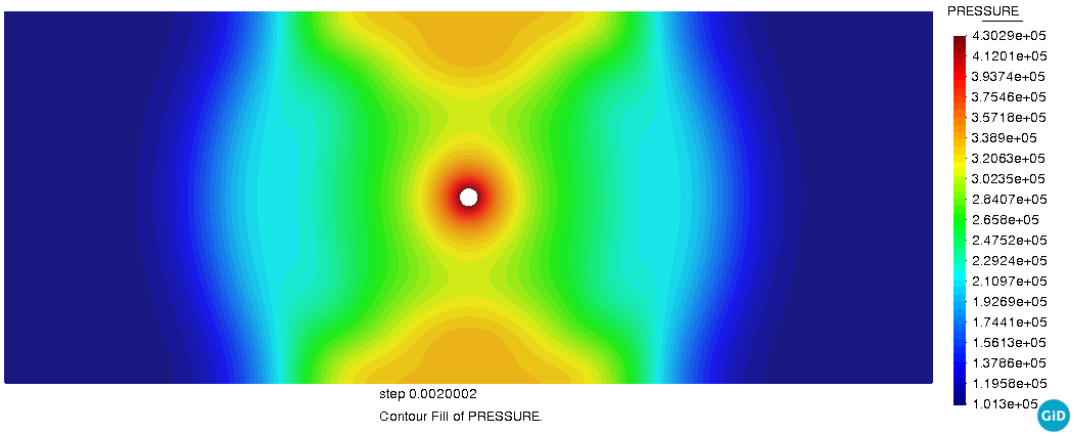
explosion is located at the center. The results are presented in the next subsections.

5.3.1 Bubble modeled as external pressure in surface

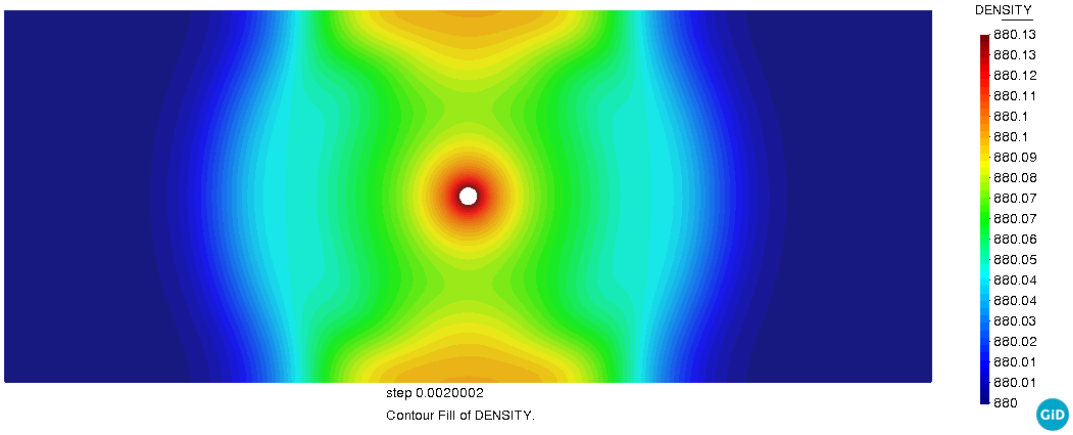
The fist contact with a 2D simulation was intended to evaluate the response of the new developed Weakly-compressible element. For that reason, the constitutive law and the Level-Set modeling of the bubble (with the corresponding customized solver) were not added in the simulation. Instead, the 2D Newtonian constitutive law and the Monolithic one-fluid solver were selected. As a consequence, the bubble had to be modeled in an alternative way, which was to think of it as an static circular contour with no fluid inside. The pressure was held fixed throughout the simulation and it was applied through the `EXTERNAL_PRESSURE` variable. Optionally, the `PRESSURE` variable might be constrained to be the value of the `EXTERNAL_PRESSURE` in order to help the solver reach the desired boundary condition coming from the circular source.

The fact that the circular source is in direct contact with the fluid nodes, that is, that there is no multi-fluid interface that separates both regions difficult the capability to rise the external pressure applied, so that a maximum of 100 bars was seen to be the computational limit. Another difficulty is that there are no shock capturing methods applied, so that there is no artificial viscosity added.

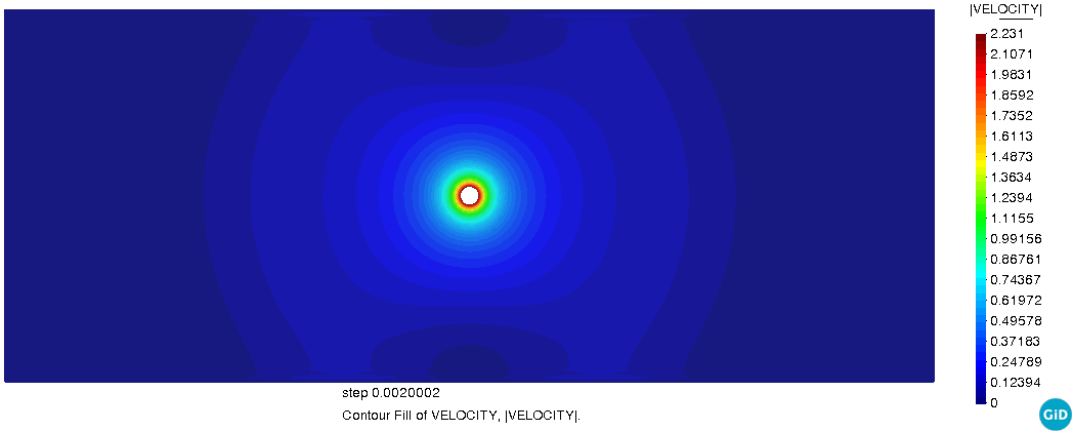
The results shown for the pressure, velocity and density in Fig. 5.13 are in accordance with what has been explained. The pressurized region compresses the fluid, with higher density around the ring. As for the velocity (not the sound velocity), the fluid particles move at higher speed also around the ring.



(a) Pressure field.



(b) Density field.



(c) Velocity field.

Figure 5.13: Simulation with external pressure applied at $t = 2$ ms.

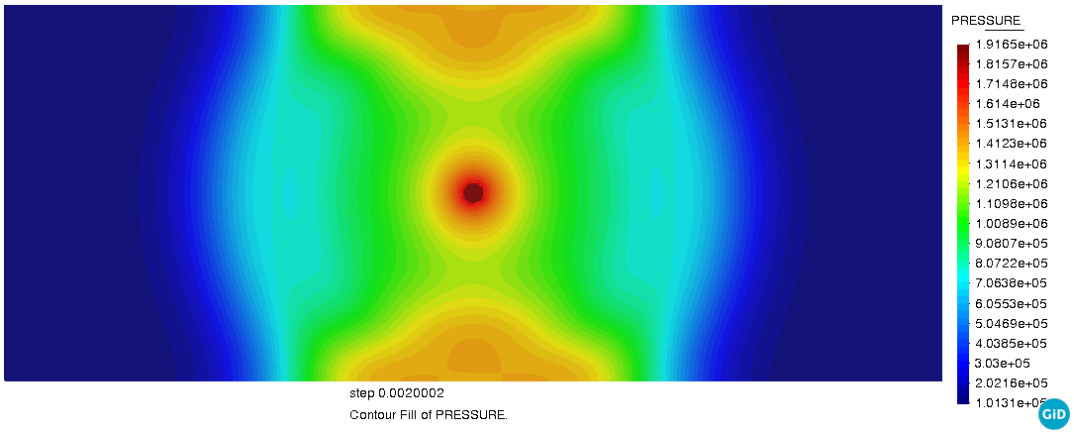
5.3.2 Bubble modeled with level-set technique

This section presents the results for the same problem, with the difference that all the developed features are put to use. In addition to the last case, the new constitutive law and the customized solver are chosen, which means that now the bubble is modeled with the Perfect Gas equation, and its interface tracked with the level-set method. As can be expected, the emission of pressure waves will now be of variable nature, both in shape and magnitude, since the bubble will be changing its area. Moreover, a mass inflow ratio of 5 Kg/s is added to the bubble, to simulate the arcing phenomena, which typically lasts around 50 to 100 ms. This requires the implementation of the algorithm detailed in 4.3 in the `MainKratos.py` file. Other necessary modifications are the addition of the `CompressibleNewtonian2DLaw` as the new constitutive law in `FluidMaterials.json` and the use of the customized Monolithic solver explained in 4.4.

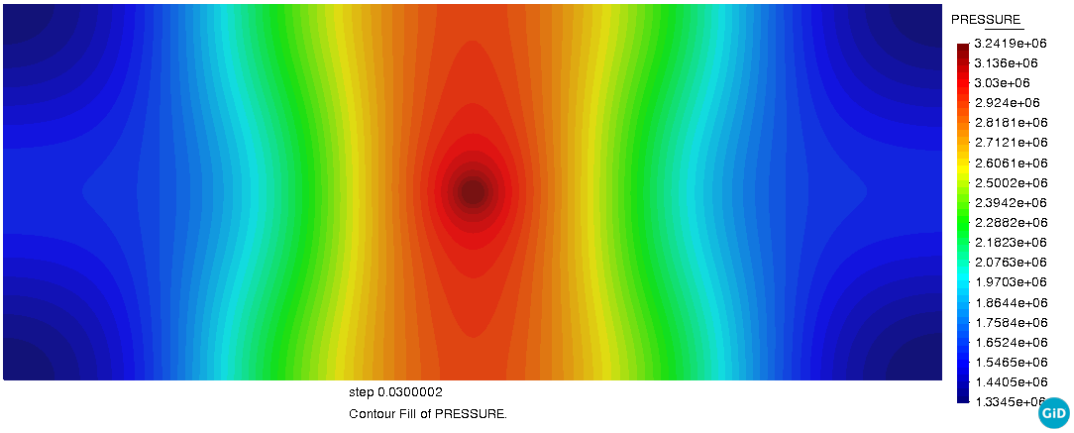
The parameters used to model the initial gas bubble are the following:

- Initial mass $m_0 = 0.5$ Kg.
- Initial radius $R_0 = 0.1$ m.

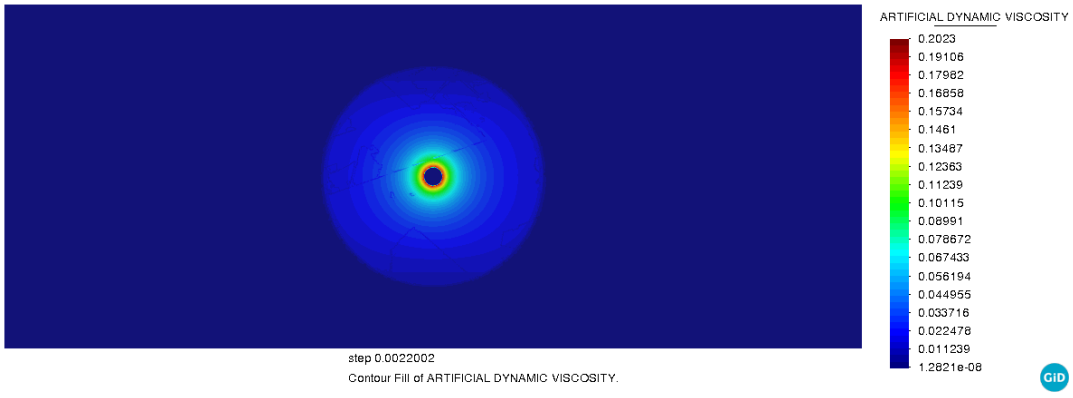
The results of the simulation are briefly depicted in Fig. 5.14, where the evolution of the pressure field is given for two different time-steps. In Fig. 5.14a the pressure field is shown at the same time instant as in Fig. 5.13, where the external pressure was used, and the results are very similar, except as in pressure magnitude, as different parameters have been used. The pressure profiles across a horizontal line spanning the bubble are shown in Fig. 5.15 for both cases. The effect of the artificial viscosity is seen in the pressure over-shots. Its activation can also be seen in Fig. 5.14c, where, as could be expected, the maximum values are concentrated around the bubble, where the sharp pressure gradients form.



(a) Pressure field at $t = 2$ ms.

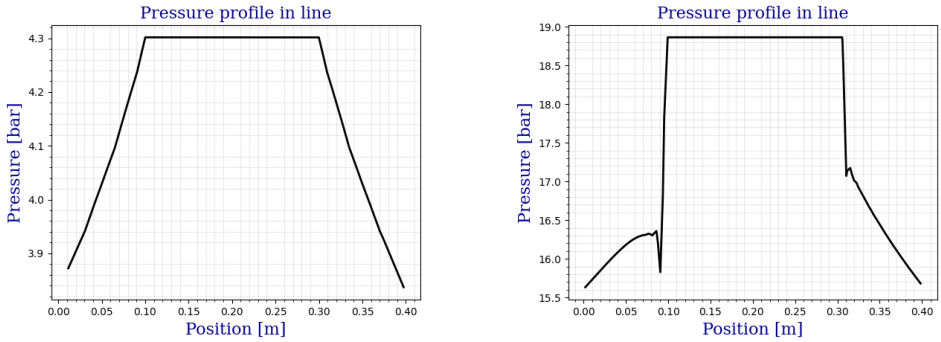


(b) Pressure field at $t = 30$ ms.



(c) Artificial dynamic viscosity field at $t = 30$ ms.

Figure 5.14: Simulation with full level-set modeling.



(a) Reference profile with external pressure.

(b) Pressure profile with multi-fluid simulation.

Figure 5.15: Comparison between horizontal pressure line graphs.

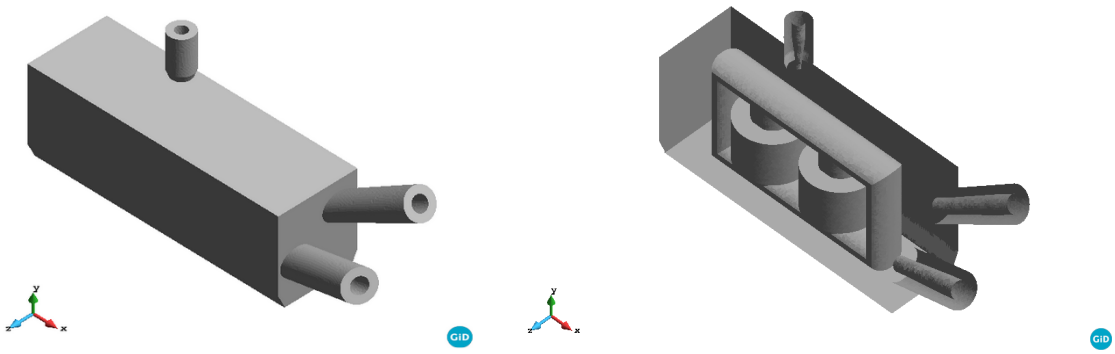
Now it is necessary to see which is the effect of this pressure bubble on the tank walls, i.e. how the pressure field evolves in the domain faces and how it is influenced by the arc energy. Also, how does the bubble radius change in time.

5.4 Three-dimensional tests

5.4.1 Bubble modeled with level-set technique

So far, all the presented tests were 2D cases. In this section a 3D case is performed to show the further capabilities of the developed code. Due to the large computational cost of solving a 3D case, the accuracy when meshing the domain is reduced compared to the 2D cases, and as a consequence, more convergence problems appear. The same element and solver are used, and the only difference is that the `CompressibleNewtonian3Dlaw` is here used instead.

The problem here presented consists of a model of a real transformer tank provided by Siemens Energy. This model had to be edited with GiD as the model was exported to Parasolid and difficulties were found when trying to mesh it directly. For that reason, a simplified model of the real tank was created by removing the unnecessary parts and keeping the main parts as simpler shapes. The resulting transformer can be seen in Fig. 5.16.



(a) Exterior view.

(b) Interior parts.

Figure 5.16: Simplified model of the transformer.

It has to be pointed out that the purpose of this problem is not to study the deformation of the tank but to further test the code in Kratos in a 3D more complicated geometry, and to place the bubble in complicated zones such as in the turrets where there is not space for the pressure waves to extend radially.

Regarding the geometry, it consists in a transformer tank of dimensions $10 \times 4 \times 5$, not including the turrets. The fluid parameters are those mentioned in the beginning. The other parameters are those in the next table.

Parameter	Symbol	Value
Initial radius	R_0	0.06 m ,
Initial mass	m_0	5×10^{-3} Kg
Initial bubble density	ρ_0	5.68 Kg/m ^{3}
Initial bubble pressure	p_0	8.69 bar

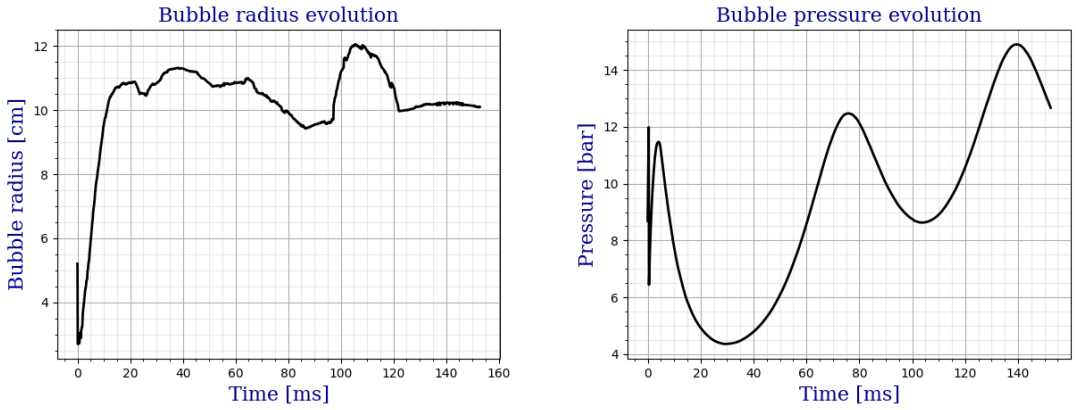
Table 5.5: Simulation parameters for 3D test.

Then, an unstructured mesh of 565553 nodes and 3345471 elements has been used in the fluid.

Last but not least, the bdf2 scheme in Kratos has been used for time discretization. The total time of the simulation is $t = 150$ ms while the time increment is set as an automatic value between 1×10^{-4} and 1×10^{-7} to satisfy 1 CFL condition.

5.4.2 Results assessment

The results are shown in terms of pressure bubble and radius evolution. During the first time steps, the bubble adjusts to the surrounding conditions, and thus a sharp variation is observed in Fig. 5.17. After that, a smooth variation is observed for the pressure, whose oscillating character is not seen in the radius evolution, which holds between 10 and 12 for the majority of the time. There is not an observable difference either when the gas inflow stops after 50 ms.

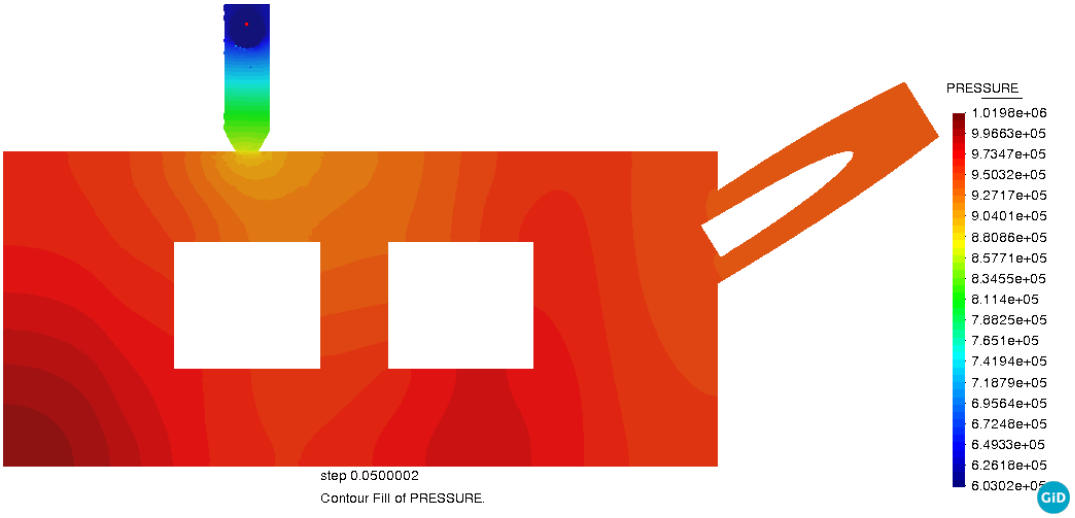


(a) Radius evolution.

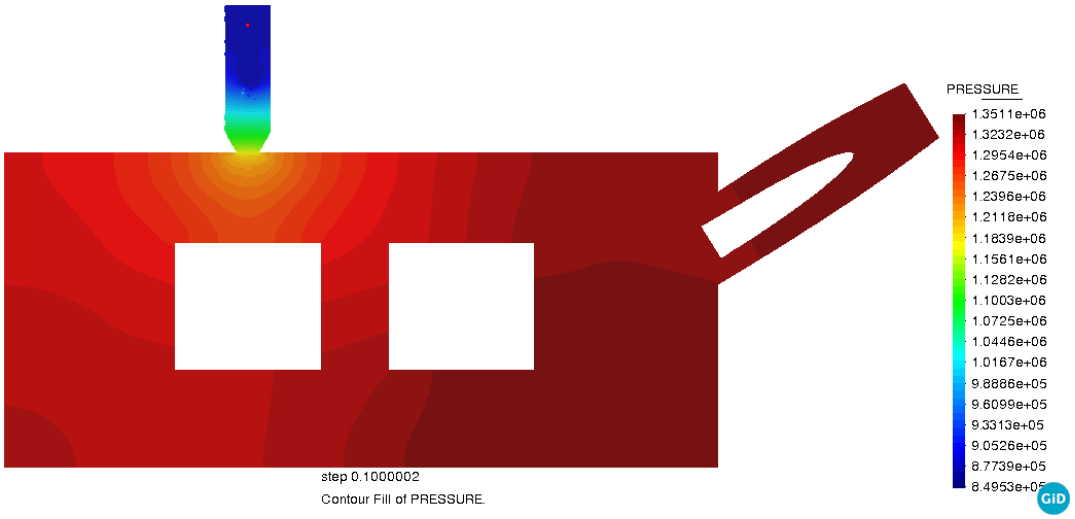
(b) Pressure evolution.

Figure 5.17: Evolution on the bubble center.

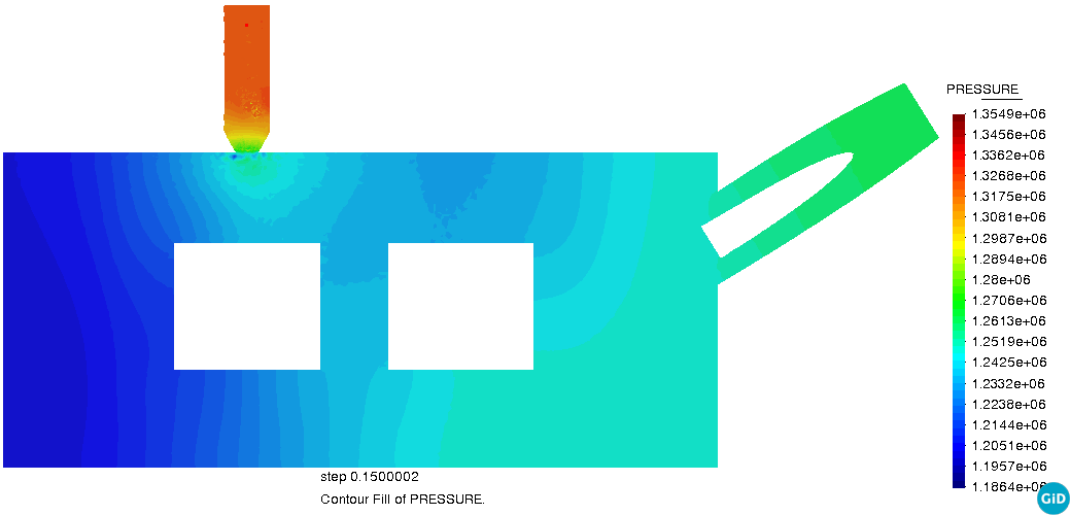
Then, a plane cutting through the bubble at different time steps is observed in Fig. 5.18. Only at 150 ms the pressure around the bubble is higher that the tank pressure, which indicated that the pressure walls evolve correctly and are transmitted through all the domain. However, the most critical point is in the vertical turret where the bubble is located, as the high values of pressure are constant in time.



(a) $t = 50$ ms.



(b) $t = 100$ ms.



(c) $t = 150$ ms.

Figure 5.18: Evolution of the pressure field.

5.4.3 FSI with cubic structure

Problem description

In this section the coupling approaches explained in chapter 2 are used to perform an analysis on a unitary cubic domain with a pressurized bubble located at a third of the diagonal. As now the walls will be elastic, this test will be appropriate to assess the deformations on the walls and weather the pressures on the walls are significantly reduced when compared to a case with the same parameters but non-deforming walls.

The problem will consist on solving the Navier-Stokes equations considering that the solution field will be coupled at the wall interface for the solid mechanics problem. The boundary conditions for the problem will consist on zero displacements and rotations for the edges of the cube, plus a zero mesh-displacement on the lower surface, where the tank does not present any deformation. Regarding the mesh, an unstructured mesh of 13071 nodes and 69532 elements has been set. The time as well as the material parameters of the simulation are listed below.

Parameter	Symbol	Value
Wall thickness	t	7 cm
Steel density	ρ	7850 Kg/m ^{3}
Young Modulus	E	200 GPa
Poisson ratio	ν	0.3
Initial bubble density	ρ_0	2.6 Kg/m ^{3}
Initial bubble pressure	p_0	4.6 bar
Total time	t	30 ms
Time-step	Δt	$1 \times 10^{-4} sec.$

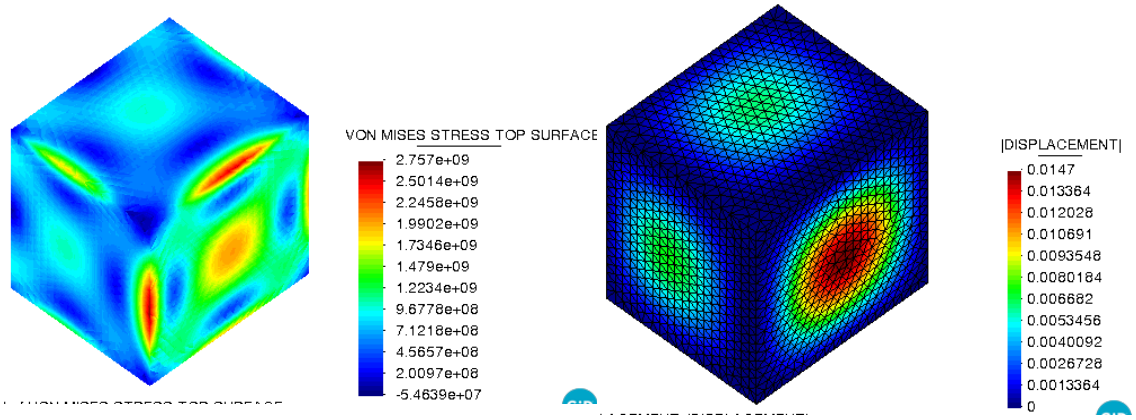
Table 5.6: Simulation parameters for FSI test.

Finally, the analysis has been set to be of implicit non-linear dynamic type, whit a residual criterion for the convergence criterion, so that $\|\mathbf{r}\| = \|\mathbf{v}_{\Gamma,1} - \mathbf{v}_{\Gamma,2}\| \leq 1 \times 10^{-5}$. As for the coupling strategy settings, the MVQN solver type with has been selected with $w_0 = 0.825$.

Last but not least, the reader may notice the high thickness on the walls. This is the minimum value that could be set without leading to a simulation failure, whereas the desired values would have been 1 or 2 cm. This will be left as future work. However, the purpose of the simulation was to demonstrate that the wall deformation could help on reducing the pressure on the walls, while on the elastic region.

Results assessment

First, the stresses and displacements are shown in Fig. 5.19, which denote the correct application of boundary conditions. The asymmetry of the results in both cases goes in accordance with the position of the bubble, which means that some faces will be more affected. The results show, after 20 ms of simulation, that the maximum displacement is of the order of 14 mm, whereas the maximum stress is 3 GPa, a stress much larger that the yielding tensile strength, which can be assumed to be 300 MPa.



(a) Von Mises stress on top surface.

(b) Displacement modulus.

Figure 5.19: Structural deformation for $t = 20$ ms.

Now, with the increment of volume produced by the deformation of the walls, it could be expected a global reduction of the pressure values on the faces. For that, the faces are numbered according to Fig. 5.20, where the bubble is initially located at $[2, 2, 2]/3$, which is the corner of faces 2,3 and 6.

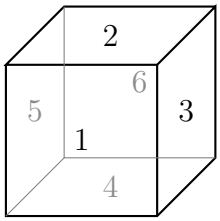
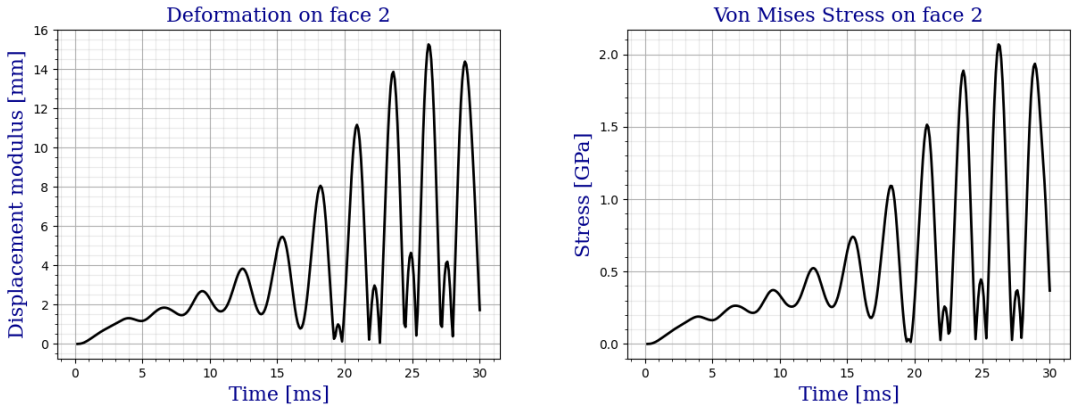


Figure 5.20: Faces numbering.

Once the faces are numbered, the pressure evolution on the faces is studied by comparing the FSI case with the rigid-wall case. The purpose is to show that the deformation on the walls relaxes the pressure levels on the walls. In Fig. 5.22 faces 1 to 3 are shown, and in Fig. 5.23 faces 4 to 6. The highest pressure is found on the top wall, i.e. face 2, which is a result that could be expected given that the top wall is where the depressurization devices are typically located.

For the rigid-wall case, the simulation was run up to 200 ms, whereas for the FSI case, only 30 ms, as that was decided to be the critical point. For the rigid case, it is interesting to see how the pressure reaches an steady value once the gas flow stops. On the other hand, it can be seen that although the thickness of the walls is 7 cm, the pressure levels on the walls drop to near 60% if compared to the rigid-wall case.

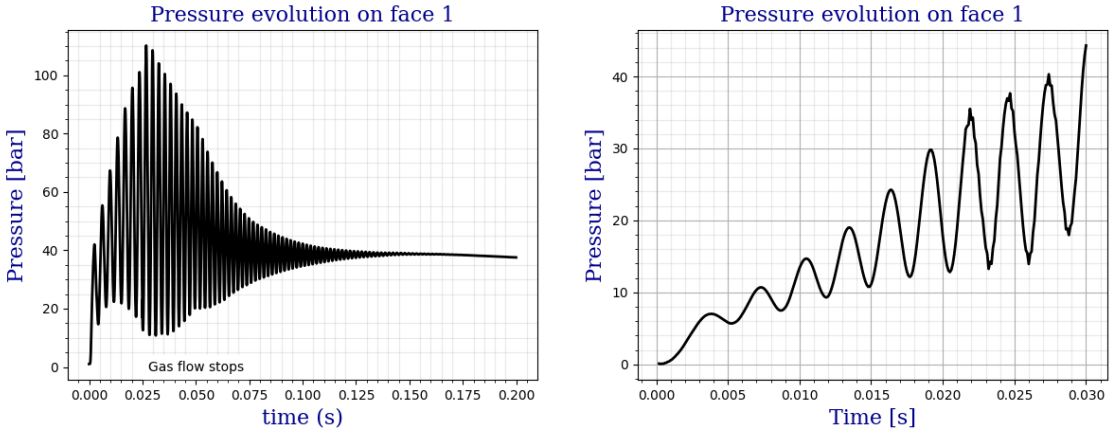
Eventually, the critical point on the surface of top face is studied through the Von Mises stress and displacements. This is useful to have an idea of the amount of energy released from the explosion and how it can deform the walls and rupture a cubic domain of 1m side even if its thickness is 7 cm. The deformations are large and the stresses are far over the yielding point.



(a) Displacement evolution.

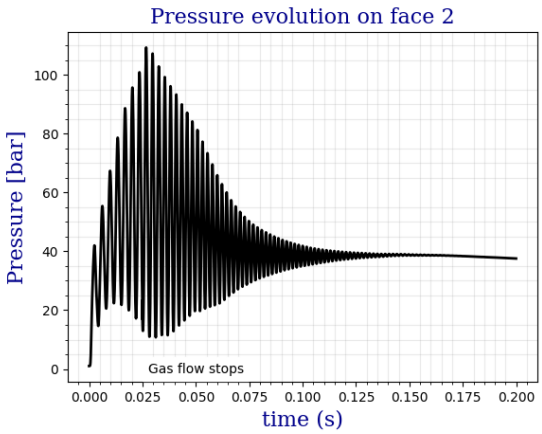
(b) Stress evolution.

Figure 5.21: Critical point analysis.

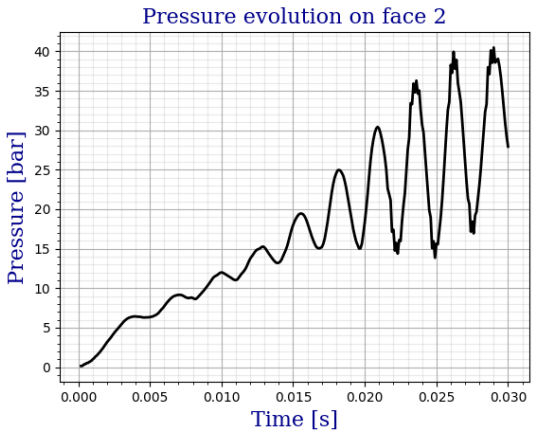


(a) Rigid wall.

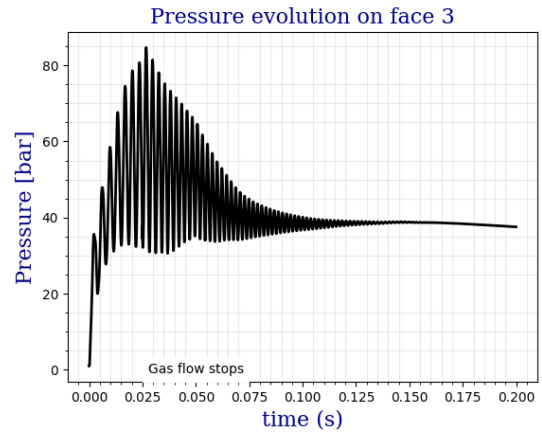
(b) Elastic wall.



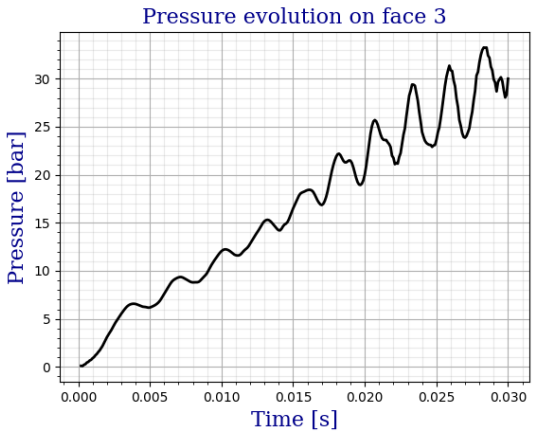
(c) Rigid wall.



(d) Elastic wall.

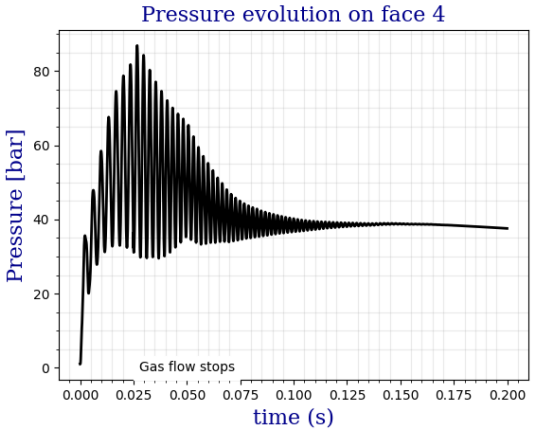


(e) Rigid wall.

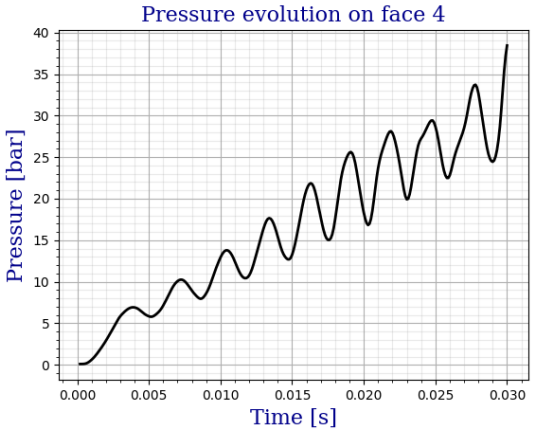


(f) Elastic wall.

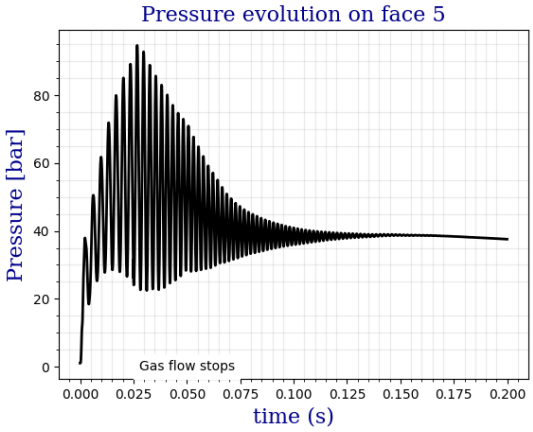
Figure 5.22: Pressure comparison at the wall's center.



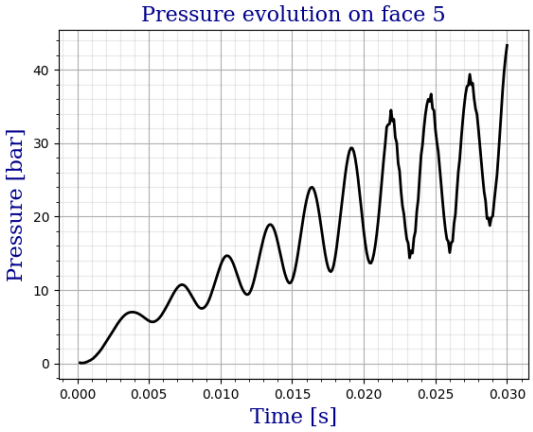
(a) Rigid wall.



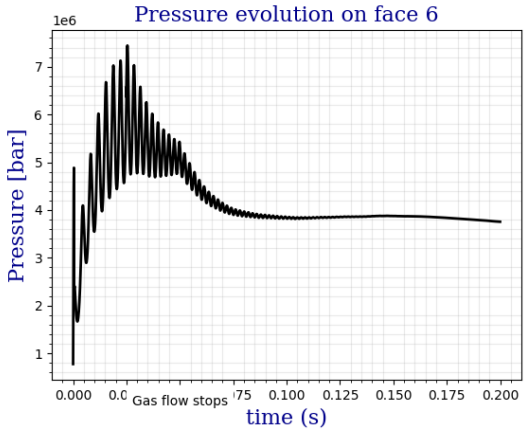
(b) Elastic wall.



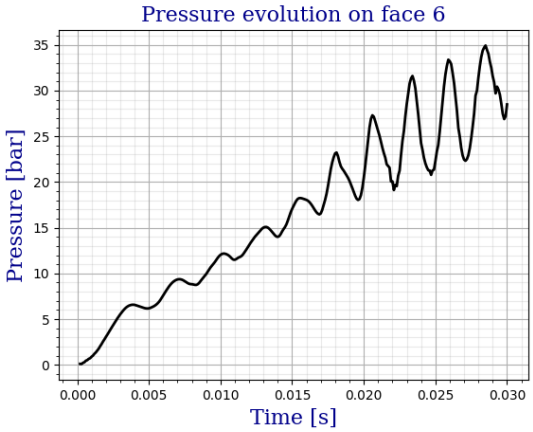
(c) Rigid wall.



(d) Elastic wall.



(e) Rigid wall.



(f) Elastic wall.

Figure 5.23: Pressure comparison at the wall's center.

Chapter 6

Conclusions

The aim of this thesis was to develop a compressible solver capable of dealing with multi-fluid flow at high pressures. To do that, the `NavierStokes` element in Kratos Multiphysics was used as a starting point to develop the particularities of the new element. All the aspects of creating a new element and its respective tests have been covered. In particular, the following topics have been covered when working with the `FluidDynamicsApplication`.

- Derivation of the `weakly_compressible_navier_stokes_element` in the symbolic generation context, together with its template. The corresponding tests have also been created and its execution checks the correctness of the new element.
- In the `custom_constitutive` folder, a new constitutive law has been generated that allows taking into consideration the artificial values for the dynamic and bulk viscosity, which results from the `ShockCapturingProcess`. The new law is `newtonian_compressible_3d_law` and `newtonian_compressible_2d_law`. The respective tests have also been created and checked. The fluid constitutive law `fluid_constitutive_law` has also been modified in order to use these values.
- An `apply_equation_of_state_process.py` which lets the user choose the equation for the fluid. The `tait_equation_process` implements the Tait equation for the fluid, with positive results.
- A customized solver, `navier_stokes_solver_levelset`, that encompasses the Monolithic solver with the convection process of the Two Fluids solver to move the level-set function.
- Customized utilities that served to the project at some point, although they are

not going to be implemented into the repository. They are `calculate_cut_area`, which computes the area or volume occupied by the bubble,
`calculate_normal_vector`, which computes the energy rate and power emitted from the interface and `calculate_errors`, which computes the L2 norm errors of the simulations.

Important parameters to take into account are:

- Mesh resolution: The mesh has to be fine inside and around the bubble, so that 20 elements may fit along its diameter. The mesh on the fluid does not require to be as fine although values of the order of 0.03m are needed in order to get accurate results, as shown in the convergence tests.
- Time discretization: An automatic time increment is to be set as a value between 1×10^{-4} and 1×10^{-7} to satisfy 1 CFL condition.
- The convergence criteria is to be more accurate, with 10 maximum iterations for the fluid solver, and relative and absolute tolerances of 1×10^{-5} and 1×10^{-7}, respectively.
- Eulerian error compensation must be set to True.
- The use of Restart Files may come in useful when dealing with large simulations.

6.1 Achievements

In this work a weakly-compressible element has been generated, together with all the required additions mentioned above. The element has been successfully tested in a wide range of tests and validated by convergence and energy conservation tests and proved capable of simulating different geometries, from a cube to a full tank. All the tests have proven the new element and processes to be adequate to predict the values of pressure on the tank walls with more celerity than if using a full compressible approach. Thus, the developed method has been shown to be an efficient strategy to analyze the transformer tank explosions, which has not been developed before.

It is a robust formulation, and all the coded processes have also been tested and checked. The result is a level-set tracking of a fluid-gas interface which can sustain very large pressure gradients thanks to the addition of artificial viscosity and a correct tuning of the parameters and settings.

Knowledge has been gained on all the aspects of the process, from programming in Python and C++ to meshing procedures in Gid. Understanding of a finite element

formulation has also been extensively improved and studied throughout this time.

Finally, it is interesting to point out that some of the developed code of this work will remain in Kratos Multiphysics repository, which is open source and available to download.

6.2 Future work-lines

The work to be done next is to manage to perform a full simulation with the correct thickness on the walls, that is, 1 or 2 cm. If sufficient computational resources are provided, the 3D test that was developed on the full tank could also be modeled with FSI. With that, it could be possible to have an idea of which regions are more vulnerable of rupture if a bubble is formed nearby.

The other point of study, which was developed also during this thesis but without success, was the inclusion of a rupture disk capable of opening to the atmosphere if enough pressure levels are reached. Then, the goal would be to set the nodes pertaining to the surface of the pressure relief device a fixed atmospheric pressure. The nodes at the center would open first and progressively towards the contour of the disk. This could be very helpful when assessing if the elastic limit on the tanks is reached and compare the FSI cases whenever relief devices are used or not.

Appendix A

Derivation of a Lagrangian approach for pressure-wave propagation

This appendix shows the numerical techniques followed to derive the Lagrangian description of motion for the pressure-waves formulation. Both the derivation in the continua and in the numerical fields are done.

Recalling the insentropic condition, which allows uncoupling the Energy equation, the Navier-Stokes system in the Lagrangian form is obtained as

$$\begin{cases} \frac{\partial p}{\partial t} + k\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \\ \rho \mathbf{u_t} - \mu \nabla^2 \mathbf{u} + \boldsymbol{\nabla} p = \rho \mathbf{f} \end{cases} \tag{A.1}$$

It has been introduced here the bulk modulus, $k = \rho c^2$. Again, the continuity equation is the condition for quasi-incompressibility, where the divergence of the velocity is no longer zero. For a clear understanding of this term, the reader is referred to Section 3.3.

Now it is introduced a FE discretization. The time dependent variational form of the problem is

$$\begin{cases} (q, \partial_t p) + (q, k\boldsymbol{\nabla} \cdot \mathbf{u}) = 0 \quad \forall q \in Q \\ (\mathbf{v}, \rho \partial_t \mathbf{u}) + \mu(\boldsymbol{\nabla}\mathbf{v}, \boldsymbol{\nabla}\mathbf{u}) - \langle \mathbf{v}, \mathbf{n} \cdot \boldsymbol{\nabla}\mathbf{u} \rangle_\Gamma = \langle \mathbf{v}, \rho \mathbf{f} \rangle \quad \forall \mathbf{v} \in \mathbf{V} \end{cases} \tag{A.2}$$

so it is defined $V_h \subset V, Q_h \subset Q$ and their associated Galerkin projection of the solution

by

$$V_h \equiv \text{span}\{N_1, ..., N_{Nu}\}; \quad Q_h \equiv \text{span}\{\hat{N}_1, ..., \hat{N}_{Np}\}; \tag{A.3}$$

$$\begin{aligned} \mathbf{u}(\mathbf{x},t) \approx \mathbf{u}^h(\mathbf{x},t) &= \sum_{j=1}^{N_u} \mathbf{u}_j(t) N_j(\mathbf{x}) \\ p(\mathbf{x},t) \approx p^h(\mathbf{x},t) &= \sum_{j=1}^{N_p} p_j(t) \hat{N}_j(\mathbf{x}) \\ \partial_t \mathbf{u}(\mathbf{x},t) \approx \partial_t \mathbf{u}^h(\mathbf{x},t) &= \sum_{j=1}^{N_u} \partial_t \mathbf{u}_j(t) N_j(\mathbf{x}) \end{aligned} \tag{A.4}$$

and considering the weight functions equal to the element shape functions, will lead to obtaining the following matrices that will fill the algebraic system defined above. The matricial notation is the following, considering the standard system-reduction as the method to solve it.

$$\begin{aligned} L_{ij} = \int_\Omega \boldsymbol{\nabla} N_i \cdot \boldsymbol{\nabla} N_j; \quad G_{ij} = \int_\Omega \boldsymbol{\nabla} \cdot (N_i \mathbf{e}_k)\, \hat{N}_j; \quad D_{ij} = G_{ji} \\ M_{ij} = \int_\Omega N_i\, N_j; \quad Mp_{ij} = \int_\Omega \hat{N}_j\, \hat{N}_j; \end{aligned} \tag{A.5}$$

The semi-discrete form of these equations once discretized read

$$\begin{cases} \mathbf{M_p} \frac{\partial p^h}{\partial t} + k\mathbf{D}\mathbf{u^h} = 0 \\ \rho \mathbf{M} \frac{\partial \mathbf{u^h}}{\partial t} - \mu \mathbf{L}\mathbf{u^h} + \mathbf{G}p^h = \mathbf{F} \end{cases} \tag{A.6}$$

A monolithic scheme is used to perform the time integration, with a second order approximation

$$\begin{cases} \mathbf{M_p} \frac{p_h^{n+1} - p_h^n}{\Delta t} + k\mathbf{D}\mathbf{u}_\mathbf{h}^{\mathbf{n}+1} = 0 \\ \rho \mathbf{M} \frac{\mathbf{u}_h^{n+1} - \mathbf{u}_h^n}{\Delta t} - \mu \mathbf{L}\mathbf{u}_h^{n+1/2} + \mathbf{G}p_h^{n+1} = \mathbf{F}^{n+1/2} \end{cases} \tag{A.7}$$

The matricial system is then,

$$\begin{bmatrix} \frac{\rho}{\Delta t}\mathbf{M} - \frac{\mu}{2}\mathbf{L} & \mathbf{G} \\ k\mathbf{D} & \frac{1}{\Delta t}\mathbf{M_p} \end{bmatrix} \begin{bmatrix} \mathbf{u^{n+1}} \\ p^{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{F} + (\frac{\rho}{\Delta t}\mathbf{M} + \frac{\mu}{2}\mathbf{L})\mathbf{u}^n \\ \frac{p^n}{\Delta t}\mathbf{M_p} \end{bmatrix} \tag{A.8}$$

Now it is possible to write $\mathbf{u^{n+1}}$ as

$$\mathbf{u^{n+1}} = \frac{\Delta t}{\rho}\mathbf{M}^{-1}(\mathbf{F} - \mathbf{G}p^{n+1}) - \mathbf{u^n} \tag{A.9}$$

If this expression is introduced into the continuity equation it is obtained

$$\frac{1}{\Delta t^2}\mathbf{M_p}(p^{n+1} - p^n) + \frac{k}{\rho}\mathbf{D}\mathbf{M}^{-1}(\mathbf{F} - \mathbf{G}p^{n+1}) - \frac{k}{\Delta t}\mathbf{D}\mathbf{u^n} = 0 \tag{A.10}$$

Where $\mathbf{M}$ is the mass matrix, $\mathbf{M_p}$ the pressure mass matrix both used in the lumped format, $\mathbf{L}$ the Laplacian, $\mathbf{G}$ the gradient and $\mathbf{D}$ the divergence matrix.

A.1 The discrete laplacian operator

Now let us focus on the matrices that appeared in equation (A.10). The operator $\mathbf{D}\mathbf{M}^{-1}\mathbf{G}p^h$ is a discrete operator acting on the pressure. On the present section it will be shown how it is possible to approximate this operator as a continuous Laplacian acting on the nodal level, or, in other words, as the divergence of the projection of the gradient of the pressure into the nodes. That is, $\mathbf{D}\mathbf{M}^{-1}\mathbf{G} \approx \mathbf{L}$.

$$G_{ij} = -\int_\Omega \boldsymbol{\nabla}\cdot(N_i\mathbf{e}_k)\,\hat{N}_j \longrightarrow G_{ij}p_j = -\int_\Omega \boldsymbol{\nabla}\cdot(N_i\mathbf{e}_k)\,\hat{N}_j p_j = \int_\Omega N_i\,\boldsymbol{\nabla}\hat{N}_j p_j - \int_\Gamma N_i\,\hat{N}_j\cdot\mathbf{n}p_j \tag{A.11}$$

Where $\boldsymbol{\nabla}\cdot(N_i\mathbf{e}_k) = \frac{\partial N_i}{\partial x_k}$ and the term $\boldsymbol{\nabla}(\hat{N}_j p_j)$ can be seen as the components of the gradient of the pressure, which is computed inside the element. It will be referred to as $\boldsymbol{\nabla}p_{el}$.

Now let us think of a continuous vector $\boldsymbol{\pi}$ that belongs to the finite element space V_h. Now let us define a functional $\Psi(\boldsymbol{\pi})$ which has the form

$$\Psi(\boldsymbol{\pi}) = \int_\Omega (\boldsymbol{\pi} - \boldsymbol{\nabla}p_{el})^2 \tag{A.12}$$

$$\boldsymbol{\pi}^h = \sum_{k=1}^{N_p} \boldsymbol{\pi}_k \hat{N}_k; \tag{A.13}$$

Now it is clear that if $\Psi(\boldsymbol{\pi})$ is minimized with respect to $\boldsymbol{\pi}$, $\boldsymbol{\pi}$ will be the best continuous approximation to the elemental gradient of pressure, which is discontinuous.

$$\frac{\partial \Psi(\boldsymbol{\pi})}{\partial \pi_i} = \int_\Omega 2N_i(\hat{N}_j \boldsymbol{\pi_j} - \boldsymbol{\nabla}\hat{N}_j p_j) = 0 \longrightarrow \int_\Omega N_i \hat{N}_j \boldsymbol{\pi_j} = \int_\Omega N_i \boldsymbol{\nabla}\hat{N}_j p_j \longrightarrow \mathbf{M}\boldsymbol{\pi} = \mathbf{G}p \tag{A.14}$$

Therefore it has been found that $\boldsymbol{\pi} = \mathbf{M}^{-1}\mathbf{G}p$ is the best nodal approximation to the gradient of pressure. Now returning to eq. (A.10),

$$\mathbf{D}\mathbf{M}^{-1}\mathbf{G}p^h \equiv \mathbf{D}\boldsymbol{\pi} \equiv \boldsymbol{\nabla}\cdot\boldsymbol{\pi} \longrightarrow \mathbf{D}\mathbf{M}^{-1}\mathbf{G}p^h \approx \mathbf{L}p^h \tag{A.15}$$

With this it is stated that the divergence operator applied on the continuous vector (which has meaning of a gradient) has the meaning of a continuous Laplacian applied to the pressure field. Therefore it is evolving from the discontinuous level at the elements to the discrete level at nodes. The equation that has to be solved for each time step for the pressure variable is:

$$\frac{1}{\Delta t^2}\mathbf{M_p}(p^{n+1} - p^n) + \frac{k}{\rho}(\mathbf{D}\mathbf{M}^{-1}\mathbf{F} - \mathbf{L}p^{n+1}) - \frac{k}{\Delta t}\mathbf{D}\mathbf{u^n} = 0 \tag{A.16}$$

This equation could also be obtained from the system

$$\begin{cases} \mathbf{M_p}\frac{\partial p^h}{\partial t} + k\mathbf{D}\mathbf{u^h} = 0 \\ \rho\mathbf{M}\frac{\partial \mathbf{u^h}}{\partial t} - \mu\mathbf{L}\mathbf{u^h} + \mathbf{G}p^h = \mathbf{F} \end{cases} \tag{A.17}$$

by simply deriving the continuity equation with respect to time and substitute it in the momentum equation. This is obtained by considering that both the bulk modulus and the discrete operators do not depend on time, which is a wrong assumption, except under the premise of small deformations

$$\frac{\partial \mathbf{u}^h}{\partial t} = -\frac{1}{\rho}\mathbf{M}^{-1}\mathbf{G}p^h \longrightarrow \mathbf{M_p}\,\mathrm{d}p^h\,t - \frac{k}{\rho}\mathbf{D}\mathbf{M}^{-1}\mathbf{G}p^h = 0 \longrightarrow \mathbf{M_p}\,\mathrm{d}p^h\,t - \frac{k}{\rho}\mathbf{L}p^h \approx 0 \tag{A.18}$$

A.2 Derivation of the same expression in the continuum

The same equation that was used to derive the matricial form of the problem is now stated:

$$\begin{cases} \frac{\partial p}{\partial t} + k\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \\ \rho \mathbf{u_t} - \mu \nabla^2 \mathbf{u} + \boldsymbol{\nabla} p = \rho \mathbf{f} \end{cases} \tag{A.19}$$

So far no assumptions have been made, so eq. (A.19) refers to the motion of a fluid in the presence of gravity. For most applications in acoustics, the effects of gravity can be neglected, as they account for spatial variations of mean values of pressure, density and temperature [35]. It will therefore be neglected.

The derivation will be made by disturbing an equilibrium state of the fluid by means of an isentropic compression. For this reason, the properties of the fluid will oscillate about equilibrium in a small period of time, so that

$$\rho = \rho_0(\mathbf{x}) + \rho'(\mathbf{x}, t); \quad p = p_0(\mathbf{x}) + p'(\mathbf{x}, t) \tag{A.20}$$

the deviation from equilibrium is such that $|\rho'| << \rho_0$, $|p'| << p_0$. The oscillatory velocity of the fluid $\mathbf{u}$ can also be expected to be small. With these considerations, when substituting (A.20) into (A.19), only the first-order terms will be retained. Also the sound velocity, $c^2 = \frac{\partial p}{\partial \rho}$, can be expanded about the equilibrium value of the density. First the continuity equation:

$$\frac{\partial (p_0(\mathbf{x}) + p'(\mathbf{x}, t))}{\partial t} + (\rho_0(\mathbf{x}) + \rho'(\mathbf{x}, t))(c_0^2 + O(\rho^2))\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \tag{A.21}$$

Simplifying,

$$\frac{\partial p'(\mathbf{x}, t)}{\partial t} + \rho_0(\mathbf{x}) c_0^2(\mathbf{x}) \boldsymbol{\nabla} \cdot \mathbf{u} = 0 \longrightarrow \frac{1}{\rho_0(\mathbf{x}) c_0^2(\mathbf{x})} \frac{\partial p'(\mathbf{x}, t)}{\partial t} + \boldsymbol{\nabla} \cdot \mathbf{u} = 0 \tag{A.22}$$

If the time derivative is applied,

$$\frac{1}{\rho_0(\mathbf{x}) c_0^2(\mathbf{x})} \frac{\partial^2 p'(\mathbf{x}, t)}{\partial t^2} + \boldsymbol{\nabla} \cdot \mathbf{u_t} = 0 \tag{A.23}$$

As for the momentum equation,

$$(\rho_0(\mathbf{x}) + \rho'(\mathbf{x},t))\mathbf{u_t} - \mu\nabla^2\mathbf{u} + \boldsymbol{\nabla}(p_0(\mathbf{x}) + p'(\mathbf{x},t)) = (\rho_0(\mathbf{x}) + \rho'(\mathbf{x},t))\mathbf{g} \tag{A.24}$$

Retaining only first order terms

$$\rho_0(\mathbf{x})\mathbf{u_t} + \boldsymbol{\nabla}(p_0(\mathbf{x}) + p'(\mathbf{x},t)) = (\rho_0(\mathbf{x}) + \rho'(\mathbf{x},t))\mathbf{g} \tag{A.25}$$

Now, by acknowledging that the equilibrium pressure and density satisfy the condition $\boldsymbol{\nabla} p_0 = \rho_0 \mathbf{g}$

$$\rho_0(\mathbf{x})\mathbf{u_t} + \boldsymbol{\nabla} p'(\mathbf{x},t) = \frac{\rho'}{\rho_0}\boldsymbol{\nabla} p_0 \approx 0 \longrightarrow \mathbf{u_t} + \frac{1}{\rho_0(\mathbf{x})}\boldsymbol{\nabla} p'(\mathbf{x},t) \approx 0 \tag{A.26}$$

Now, if the divergence operator is applied,

$$\boldsymbol{\nabla}\cdot\mathbf{u_t} + \frac{1}{\rho_0(\mathbf{x})}\boldsymbol{\nabla}\cdot\boldsymbol{\nabla} p'(\mathbf{x},t) \approx 0 \tag{A.27}$$

Eventually, by combining equations (A.27) and (A.23),

$$\frac{\partial p'(\mathbf{x},t)}{\partial t} = c_0^2(\mathbf{x})\boldsymbol{\nabla}^2 p'(\mathbf{x},t) \tag{A.28}$$

Our purpose is to solve the in-homogeneous acoustic wave equation in time domain. The ordinary differential system is

$$\mathbf{M_p}\frac{d^2\mathbf{p}}{dt^2}(t) + \mathbf{L}\mathbf{p}(t) = 0 \tag{A.29}$$

A classical approximation is obtained by a centered second-order finite difference scheme as follows

$$\mathbf{M_p}\frac{\mathbf{p}^{n+1} - 2\mathbf{p}^n + \mathbf{p}^{n-1}}{\Delta t^2} + \mathbf{L}\mathbf{p}^n = 0 \tag{A.30}$$

If using a mass-lumping technique it would avoid the computation of the inverse of the mass matrix.

Appendix B

Bubble pressure through acoustic theory

This option was explored as a tool to compute the bubble pressure by means of a theoretical approach, instead of using the perfect gas equation. This option relies on the acoustic theory to estimate the gas bubble expansion, which is of slow nature compared to the relative sound-speed of the surrounding medium.

The wave dynamics of the pressure pulse are solved with the acoustic wave equation, which has solution for the velocity potential of outward facing waves

$$\begin{aligned} \phi &= \frac{f(r-ct)}{r} \\ u = \frac{\partial \phi}{\partial r} &= \frac{1}{r} f'(r-ct) - \frac{1}{r^2} f(r-ct) \\ p &= -\rho \frac{\partial \phi}{\partial t} = \rho c \frac{f'(r-ct)}{r} \end{aligned} \tag{B.1}$$

Where ϕ is the potential and c is the sound velocity. Particularizing the expression for the velocity at the value of $r = R$, where R is the radius of the gas-bubble

$$\frac{dR}{dt} = -\frac{f(R-ct)}{R^2} + \frac{f'(R-ct)}{R} \tag{B.2}$$

The time-evolution of the bubble radius can be used to obtain an expression for the

pressure field as a function of position and time. Making a change of variable $w = R - ct$

$$\frac{dR}{dt} = -\frac{f(w)}{w^2}\left(1 - \frac{ct}{R}\right)^2 + \frac{f'(w)}{w}\left(1 - \frac{ct}{R}\right); \tag{B.3}$$

Now it is possible to solve the ODE for the condition $f(w = 0) = 0$. For that we need an expression for the evolution of the radius of the sphere with respect to time. To options have been found in literature,

$$R(t) = A(1 - e^{-t/\tau}); \qquad R(t) = \alpha ct \tag{B.4}$$

One is exponential and the other is linear. They have been extracted from [25], which compare in an interesting study the evolution of the pressure waves emitted from a bubble generated in an aqueous environment by means of experimental tests and the acoustic theory. The exponential fit is very close to the actual evolution. In both cases A, τ, α are fitting parameters. With the time-evolution of the bubble radius, the pressure field as a function of position and time is obtained, and a value for the pressure at the radial coordinate $r = R$, which corresponds to the bubble pressure. [25] gives the following pressure field for the exponential law of radius evolution.

$$p(r, t) = \frac{\rho}{4\pi r} Q'\left(t - \frac{r}{c}\right) \tag{B.5}$$

Where Q' is the volumetric expansion. However, this approach may only be useful for the first instants of the simulation, but as will be seen, the radius variation will depend greatly on the interaction with the reflected waves.

One is exponential and the other is linear. They have been extracted from [25], which compare in an interesting study the evolution of the pressure waves emitted from a bubble generated in an aqueous environment by means of experimental tests and the acoustic theory. The exponential fit is very close to the actual evolution. In both cases A, τ, α are fitting parameters. With the

Appendix C

Developed code

In this appendix the most important and representative Python and C++ files developed during this work are collected.

C.1 Symbolic generation

```
from sympy import *
from KratosMultiphysics import *
from KratosMultiphysics.sympy_fe_utilities import *

## Settings explanation
# DIMENSION TO COMPUTE:
# This symbolic generator is valid for both 2D and 3D cases. Since the element has been programed with a dimension template in Kratos,
# it is advised to set the dim_to_compute flag as "Both". In this case the generated .cpp file will contain both 2D and 3D implementations.
# LINEARISATION SETTINGS:
# FullNR considers the convective velocity as "v-vmesh", hence v is taken into account in the derivation of the LHS and RHS.
# Picard (a.k.a. QuasiNR) considers the convective velocity as "a", thus it is considered as a constant in the derivation of the LHS and RHS.
# DIVIDE BY RHO:
# If set to true, divides the mass conservation equation by rho in order to have a better conditioned matrix. Otherwise the original form is kept.
# ARTIFICIAL COMPRESSIBILITY:
# If set to true, the time derivative of the density is introduced in the mass conservation equation together with the state equation
# dp/drho=c^2 (being c the sound velocity). Besides, the velocity divergence is not considered to be 0. These assumptions add some extra terms
# to the usual Navier-Stokes equations that act as a weak compressibility controlled by the value of "c".
# CONVECTIVE TERM:
# If set to true, the convective term is taken into account in the calculation of the variational form. This allows generating both
# Navier-Stokes and Stokes elements.

## Symbolic generation settings
mode = "c"
do_simplifications = False
dim_to_compute = "Both"              # Spatial dimensions to compute. Options:  "2D","3D","Both"
divide_by_rho = True                 # Divide the mass conservation equation by rho
```

```
ASGS_stabilization = True           # Consider ASGS stabilization terms
formulation = "WeaklyCompressibleNavierStokes" # Element type. Options: "WeaklyCompressibleNavierStokes", "Stokes"

if formulation == "WeaklyCompressibleNavierStokes":
    convective_term = True
    artificial_compressibility = True
    linearisation = "Picard" # Convective term linearisation type. Options: "Picard", "FullNR"
    output_filename = "weakly_compressible_navier_stokes.cpp"
    template_filename = "weakly_compressible_navier_stokes_cpp_template.cpp"
elif formulation == "Stokes":
    convective_term = False
    artificial_compressibility = False
    output_filename = "symbolic_stokes.cpp"
    template_filename = "symbolic_stokes_cpp_template.cpp"
else:
    err_msg = "Wrong formulation. Given \'" + formulation + "\'. Available options are \'WeaklyCompressibleNavierStokes\' and \'Stokes\'."
    raise Exception(err_msg)

info_msg = "\n"
info_msg += "Element generator settings:\n"
info_msg += "\t - Element type: " + formulation + "\n"
info_msg += "\t - Dimension: " + dim_to_compute + "\n"
info_msg += "\t - ASGS stabilization: " + str(ASGS_stabilization) + "\n"
info_msg += "\t - Pseudo-compressibility: " + str(artificial_compressibility) + "\n"
info_msg += "\t - Divide mass conservation by rho: " + str(divide_by_rho) + "\n"
print(info_msg)

#TODO: DO ALL ELEMENT TYPES FOR N-S TOO
if formulation == "NavierStokes" or formulation == "WeaklyCompressibleNavierStokes":
    if (dim_to_compute == "2D"):
        dim_vector = [2]
        nnodes_vector = [3] # tria
    elif (dim_to_compute == "3D"):
        dim_vector = [3]
        nnodes_vector = [4] # tet
    elif (dim_to_compute == "Both"):
        dim_vector = [2, 3]
        nnodes_vector = [3, 4] # tria, tet
elif formulation == "Stokes":
    # all linear elements
    if (dim_to_compute == "2D"):
        dim_vector = [2, 2]
        nnodes_vector = [3, 4] # tria, quad
    elif (dim_to_compute == "3D"):
        dim_vector = [3, 3, 3]
        nnodes_vector = [4, 6, 8] # tet, prism, hex
    elif (dim_to_compute == "Both"):
        dim_vector = [2, 2, 3, 3, 3]
        nnodes_vector = [3, 4, 4, 6, 8] # tria, quad, tet, prism, hex

## Initialize the outstring to be filled with the template .cpp file
print("Reading template file \'"+ template_filename + "\'\n")
templatefile = open(template_filename)
outstring = templatefile.read()

for dim, nnodes in zip(dim_vector, nnodes_vector):

    if(dim == 2):
        strain_size = 3
    elif(dim == 3):
        strain_size = 6

    impose_partion_of_unity = False
    N,DN = DefineShapeFunctions(nnodes, dim, impose_partion_of_unity)

    ## Unknown fields definition
    v = DefineMatrix('v',nnodes,dim)            # Current step velocity (v(i,j) refers to velocity of node i component j)
    vn = DefineMatrix('vn',nnodes,dim)          # Previous step velocity
```

```
95      vnn = DefineMatrix('vnn',nnodes,dim)        # 2 previous step velocity
96      p = DefineVector('p',nnodes)                # Pressure
97      pn = DefineVector('pn',nnodes)              # Previous step pressure
98      pnn = DefineVector('pnn',nnodes)            # 2 previous step pressure
99
100     ## Fluid properties
101     if artificial_compressibility:
102         # If weak-compressibility is on, the density (rho) and speed of sound (c) become nodal
        variables
103         rho_nodes = DefineVector('rho',nnodes) # Nodal density
104         c_nodes = DefineVector('c',nnodes)     # Nodal sound speed
105         rho = rho_nodes.transpose()*N          # Density Gauss pt. interpolation
106         c = c_nodes.transpose()*N              # Sound speed Gauss pt. interpolation
107         rho = rho[0]
108         c = c[0]
109     else:
110         # With no weak-compressibility, the density (rho) is retrieved from the element properties
        and there is no speed of sound need
111         rho = Symbol('rho', positive = True)     # Density
112
113     ## Test functions definition
114     w = DefineMatrix('w',nnodes,dim)            # Velocity field test function
115     q = DefineVector('q',nnodes)                # Pressure field test function
116
117     ## Other data definitions
118     f = DefineMatrix('f',nnodes,dim)            # Forcing term
119
120     ## Constitutive matrix definition
121     C = DefineSymmetricMatrix('C',strain_size,strain_size)
122
123     ## Stress vector definition
124     stress = DefineVector('stress',strain_size)
125
126     ## Other simbols definition
127     dt  = Symbol('dt', positive = True)         # Time increment
128     nu  = Symbol('nu', positive = True)         # Kinematic viscosity (mu/rho)
129     mu  = Symbol('mu', positive = True)         # Dynamic viscosity
130     h = Symbol('h', positive = True)
131     dyn_tau = Symbol('dyn_tau', positive = True)
132     stab_c1 = Symbol('stab_c1', positive = True)
133     stab_c2 = Symbol('stab_c2', positive = True)
134
135     ## Backward differences coefficients
136     bdf0 = Symbol('bdf0')
137     bdf1 = Symbol('bdf1')
138     bdf2 = Symbol('bdf2')
139
140     ## Data interpolation to the Gauss points
141     f_gauss = f.transpose()*N
142     v_gauss = v.transpose()*N
143
144     ## Convective velocity definition
145     if convective_term:
146         if (linearisation == "Picard"):
147             vconv = DefineMatrix('vconv',nnodes,dim)    # Convective velocity defined a symbol
148         elif (linearisation == "FullNR"):
149             vmesh = DefineMatrix('vmesh',nnodes,dim)    # Mesh velocity
150             vconv = v - vmesh                           # Convective velocity defined as a velocity
        dependent variable
151         else:
152             raise Exception("Wrong linearisation \'" + linearisation + "\' selected. Available
        options are \'Picard\' and \'FullNR\'.")
153         vconv_gauss = vconv.transpose()*N
154
155     ## Compute the stabilization parameters
156     if convective_term:
157         stab_norm_a = 0.0
158         for i in range(0, dim):
159             stab_norm_a += vconv_gauss[i]**2
160         stab_norm_a = sqrt(stab_norm_a)
```

```
161         tau1 = 1.0/((rho*dyn_tau)/dt + (stab_c2*rho*stab_norm_a)/h + (stab_c1*mu)/(h*h)) #
        Stabilization parameter 1
162         tau2 = mu + (stab_c2*rho*stab_norm_a*h)/stab_c1                                  #
        Stabilization parameter 2
163     else:
164         tau1 = 1.0/((rho*dyn_tau)/dt + (stab_c1*mu)/(h*h)) # Stabilization parameter 1
165         tau2 = (h*h) / (stab_c1 * tau1)                    # Stabilization parameter 2
166 
167     ## Compute the rest of magnitudes at the Gauss points
168     accel_gauss = (bdf0*v + bdf1*vn + bdf2*vnn).transpose()*N
169 
170     p_gauss = p.transpose()*N
171     if artificial_compressibility:
172         pder_gauss = (bdf0*p + bdf1*pn + bdf2*pnn).transpose()*N
173 
174     w_gauss = w.transpose()*N
175     q_gauss = q.transpose()*N
176 
177     ## Gradients computation (fluid dynamics gradient)
178     grad_w = DfjDxi(DN,w)
179     grad_q = DfjDxi(DN,q)
180     grad_p = DfjDxi(DN,p)
181     grad_v = DfjDxi(DN,v)
182     if artificial_compressibility:
183         grad_rho = DfjDxi(DN,rho_nodes)
184 
185     div_w = div(DN,w)
186     div_v = div(DN,v)
187     if convective_term:
188         div_vconv = div(DN,vconv)
189 
190     grad_sym_v = grad_sym_voigtform(DN,v)       # Symmetric gradient of v in Voigt notation
191     grad_w_voigt = grad_sym_voigtform(DN,w)     # Symmetric gradient of w in Voigt notation
192     # Recall that the grad(w):sigma contraction equals grad_sym(w)*sigma in Voigt notation since
        sigma is a symmetric tensor.
193 
194     # Convective term definition
195     if convective_term:
196         convective_term_gauss = (vconv_gauss.transpose()*grad_v)
197         rho_convective_term_gauss = vconv_gauss.transpose()*grad_rho
198 
199     ## Compute galerkin functional
200     # Navier-Stokes functional
201     if (divide_by_rho):
202         rv_galerkin = rho*w_gauss.transpose()*f_gauss - rho*w_gauss.transpose()*accel_gauss -
        grad_w_voigt.transpose()*stress + div_w*p_gauss - q_gauss*div_v
203         if artificial_compressibility:
204             rv_galerkin -= (1/(rho*c*c))*q_gauss*pder_gauss
205             if convective_term:
206                 rv_galerkin -= (1/rho)*q_gauss*rho_convective_term_gauss
207         if convective_term:
208             rv_galerkin -= rho*w_gauss.transpose()*convective_term_gauss.transpose()
209     else:
210         rv_galerkin = rho*w_gauss.transpose()*f_gauss - rho*w_gauss.transpose()*accel_gauss  -
        grad_w_voigt.transpose()*stress + div_w*p_gauss - rho*q_gauss*div_v
211         if artificial_compressibility:
212             rv_galerkin -= (1/(c*c))*q_gauss*pder_gauss
213             if convective_term:
214                 rv_galerkin -= q_gauss*rho_convective_term_gauss
215         if convective_term:
216             rv_galerkin -= rho*w_gauss.transpose()*convective_term_gauss.transpose()
217 
218     ##  Stabilization functional terms
219     # Momentum conservation residual
220     # Note that the viscous stress term is dropped since linear elements are used
221     vel_residual = rho*f_gauss - rho*accel_gauss - grad_p
222     if convective_term:
223         vel_residual -= rho*convective_term_gauss.transpose()
224 
225     # Mass conservation residual
226     if (divide_by_rho):
```

```
227             mas_residual = -div_v
228             if artificial_compressibility:
229                 mas_residual -= (1/(rho*c*c))*pder_gauss
230                 if convective_term:
231                     mas_residual -= (1/rho)*rho_convective_term_gauss
232         else:
233             mas_residual = -rho*div_v
234             if artificial_compressibility:
235                 mas_residual -= (1/(c*c))*pder_gauss
236                 if convective_term:
237                     mas_residual -= rho_convective_term_gauss
238 
239         vel_subscale = tau1*vel_residual
240         mas_subscale = tau2*mas_residual
241 
242         # Compute the ASGS stabilization terms using the momentum and mass conservation residuals above
243         if (divide_by_rho):
244             rv_stab = grad_q.transpose()*vel_subscale
245         else:
246             rv_stab = rho*grad_q.transpose()*vel_subscale
247         if convective_term:
248             rv_stab += rho*vconv_gauss.transpose()*grad_w*vel_subscale
249             rv_stab += rho*div_vconv*w_gauss.transpose()*vel_subscale
250         rv_stab += div_w*mas_subscale
251 
252         ## Add the stabilization terms to the original residual terms
253         if (ASGS_stabilization):
254             rv = rv_galerkin + rv_stab
255         else:
256             rv = rv_galerkin
257 
258         ## Define DOFs and test function vectors
259         dofs = Matrix( zeros(nnodes*(dim+1), 1) )
260         testfunc = Matrix( zeros(nnodes*(dim+1), 1) )
261 
262         for i in range(0,nnodes):
263 
264             # Velocity DOFs and test functions
265             for k in range(0,dim):
266                 dofs[i*(dim+1)+k] = v[i,k]
267                 testfunc[i*(dim+1)+k] = w[i,k]
268 
269             # Pressure DOFs and test functions
270             dofs[i*(dim+1)+dim] = p[i,0]
271             testfunc[i*(dim+1)+dim] = q[i,0]
272 
273         ## Compute LHS and RHS
274         # For the RHS computation one wants the residual of the previous iteration (residual based
            formulation). By this reason the stress is
275         # included as a symbolic variable, which is assumed to be passed as an argument from the previous
              iteration database.
276         print("Computing " + str(dim) + "D RHS Gauss point contribution\n")
277         rhs = Compute_RHS(rv.copy(), testfunc, do_simplifications)
278         rhs_out = OutputVector_CollectingFactors(rhs, "rhs", mode)
279 
280         # Compute LHS (RHS(residual) differenctiation w.r.t. the DOFs)
281         # Note that the 'stress' (symbolic variable) is substituted by 'C*grad_sym_v' for the LHS
            differenctiation. Otherwise the velocity terms
282         # within the velocity symmetryc gradient would not be considered in the differenctiation, meaning
              that the stress would be considered as
283         # a velocity independent constant in the LHS.
284         print("Computing " + str(dim) + "D LHS Gauss point contribution\n")
285         SubstituteMatrixValue(rhs, stress, C*grad_sym_v)
286         lhs = Compute_LHS(rhs, testfunc, dofs, do_simplifications) # Compute the LHS (considering stress
            as C*(B*v) to derive w.r.t. v)
287         lhs_out = OutputMatrix_CollectingFactors(lhs, "lhs", mode)
288 
289         ## Replace the computed RHS and LHS in the template outstring
290         outstring = outstring.replace("//substitute_lhs_" + str(dim) + 'D' + str(nnodes) + 'N', lhs_out)
291         outstring = outstring.replace("//substitute_rhs_" + str(dim) + 'D' + str(nnodes) + 'N', rhs_out)
292 
```

```
293 ## Write the modified template
294 print("Writing output file \'" + output_filename + "\'")
295 out = open(output_filename,'w')
296 out.write(outstring)
297 out.close()
```

C.2 Example MainKratos.py file

```
1  from __future__ import print_function, absolute_import, division #makes KratosMultiphysics backward
       compatible with python 2.6 and 2.7
2
3  import KratosMultiphysics
4  from KratosMultiphysics.FluidDynamicsApplication.fluid_dynamics_analysis import FluidDynamicsAnalysis
5  from KratosMultiphysics.FluidDynamicsApplication import CalculateCutArea
6  from KratosMultiphysics.FluidDynamicsApplication import CalculateNormalVector
7
8  import sys
9  import time
10 import numpy as np
11
12 class FluidDynamicsAnalysisWithFlush(FluidDynamicsAnalysis):
13
14     def __init__(self,model,project_parameters,flush_frequency=10.0):
15         super(FluidDynamicsAnalysisWithFlush,self).__init__(model,project_parameters)
16         self.flush_frequency = flush_frequency
17         self.last_flush = time.time()
18         self.time_vec = np.array([])
19         self.energy_vec = np.array([])
20         self.work_vec = np.array([])
21         self.bubble_radius = 0.03
22         self.bubble_center = np.array([1, 1, 2])/3
23
24     def Initialize(self):
25         super(FluidDynamicsAnalysisWithFlush,self).Initialize()
26         self.wall = self.model['FluidModelPart.NoSlip3D_Wall']
27         KratosMultiphysics.CalculateNodalAreaProcess(self._GetSolver().main_model_part,2).Execute()
28         for node in self._GetSolver().GetComputingModelPart().Nodes:
29             distance = ((node.X - self.bubble_center[0])**2 + (node.Y - self.bubble_center[1])**2 + (
       node.Z - self.bubble_center[2])**2)**0.5 - self.bubble_radius
30             node.SetSolutionStepValue(KratosMultiphysics.DISTANCE, distance)
31         self.mass = 3e-4
32         self.E0 = 0
33
34     def ApplyBoundaryConditions(self):
35         super(FluidDynamicsAnalysisWithFlush,self).ApplyBoundaryConditions()
36         gamma = 1.47
37         mu = 5.68e6
38         cp = 1860
39         T = 200 + 273
40
41         calc_total_area=CalculateCutArea(self._GetSolver().GetComputingModelPart())
42         total_area = calc_total_area.Execute()
43         if (self._GetSolver().GetComputingModelPart().ProcessInfo[KratosMultiphysics.TIME] < 50e-3):
44           self.mass += 5*self._GetSolver().GetComputingModelPart().ProcessInfo[KratosMultiphysics.
       DELTA_TIME]
45         rho_gas = self.mass/total_area
46         e1 = cp/gamma*T
47         p1 = rho_gas*(gamma - 1)*e1
48
49
50         for node in self._GetSolver().GetComputingModelPart().Nodes:
51             d = node.GetSolutionStepValue(KratosMultiphysics.DISTANCE)
52             node.Free(KratosMultiphysics.PRESSURE)
53             node.Free(KratosMultiphysics.DENSITY)
54             if (d <= 0):
55                 node.Fix(KratosMultiphysics.PRESSURE)
56                 node.SetSolutionStepValue(KratosMultiphysics.PRESSURE,p1)
57                 node.Fix(KratosMultiphysics.PRESSURE)
```

```
58                  node.SetSolutionStepValue(KratosMultiphysics.DENSITY,rho_gas)
59
60
61      def FinalizeSolutionStep(self):
62          super(FluidDynamicsAnalysisWithFlush,self).FinalizeSolutionStep()
63
64          if self.parallel_type == "OpenMP":
65              now = time.time()
66              if now - self.last_flush > self.flush_frequency:
67                  sys.stdout.flush()
68                  self.last_flush = now
69
70
71
72  if __name__ == "__main__":
73
74      with open("ProjectParameters.json",'r') as parameter_file:
75          parameters = KratosMultiphysics.Parameters(parameter_file.read())
76
77      model = KratosMultiphysics.Model()
78      simulation = FluidDynamicsAnalysisWithFlush(model,parameters)
79      simulation.Run()
```

C.3 Equation of state

C.3.1 Tait equation

```
1  #include "tait_equation_process.h"
2
3  namespace Kratos
4  {
5      /* Public functions ******************************************************/
6      TaitEquationProcess::TaitEquationProcess(
7          ModelPart& rModelPart,
8          Parameters& rParameters):
9      mrModelPart(rModelPart)
10     {
11         // Read settings from parameters
12         KRATOS_ERROR_IF_NOT( rParameters.Has("rho_0") ) <<
13         "In Tait Equation Process: \'rho_0\' not found in parameters." << std::endl;
14
15         Parameters rhoParam = rParameters.GetValue("rho_0");
16         mrho_0 = rhoParam.GetDouble();
17
18         KRATOS_ERROR_IF( mrho_0 <= 0.0 ) <<
19             "In Tait Equation Process: Incorrect value for \'rho_0\' parameter:" << std::endl <<
20             "Expected a positive double, got " << mrho_0 << std::endl;
21
22         KRATOS_ERROR_IF_NOT( rParameters.Has("p_0") ) <<
23         "In Tait Equation Process: \'p_0\' not found in parameters." << std::endl;
24
25         Parameters PressureParam = rParameters.GetValue("p_0");
26         mp_0 = PressureParam.GetDouble();
27
28         KRATOS_ERROR_IF( mp_0 <= 0.0 ) <<
29             "In Tait Equation Process: Incorrect value for \'p_0\' parameter:" << std::endl <<
30             "Expected a positive double, got " << mp_0 << std::endl;
31
32         KRATOS_ERROR_IF_NOT( rParameters.Has("k_0") ) <<
33         "In Tait Equation Process: \'k_0\' not found in parameters." << std::endl;
34
35         Parameters kParam = rParameters.GetValue("k_0");
36         mk_0 = kParam.GetDouble();
37
38         KRATOS_ERROR_IF_NOT( rParameters.Has("theta") ) <<
39         "In Tait Equation Process: \'theta\' not found in parameters." << std::endl;
```

```
40
41          Parameters ThetaParam = rParameters.GetValue("theta");
42          mtheta = ThetaParam.GetDouble();
43      }
44
45      TaitEquationProcess::~TaitEquationProcess()
46      {
47
48      }
49
50      void TaitEquationProcess::Execute()
51      {
52          this->AssignTaitEquation();
53      }
54
55      void TaitEquationProcess::ExecuteInitialize()
56      {
57          this->ValidateModelPart();
58      }
59
60      void TaitEquationProcess::ExecuteInitializeSolutionStep()
61      {
62          this->AssignTaitEquation();
63      }
64
65      std::string TaitEquationProcess::Info() const
66      {
67          return "TaitEquationProcess";
68      }
69
70      void TaitEquationProcess::PrintInfo(std::ostream& rOStream) const
71      {
72          rOStream << "TaitEquationProcess";
73      }
74
75      void TaitEquationProcess::PrintData(std::ostream& rOStream) const
76      {
77      }
78
79      /* Protected functions ****************************************************/
80
81      void TaitEquationProcess::ValidateModelPart()
82      {
83          // Nodal variables
84          KRATOS_ERROR_IF_NOT( mrModelPart.GetNodalSolutionStepVariablesList().Has(DENSITY) ) <<
85          "\'DENSITY\' variable is not added to the ModelPart nodal data." << std::endl;
86
87          KRATOS_ERROR_IF_NOT( mrModelPart.GetNodalSolutionStepVariablesList().Has(SOUND_VELOCITY) ) <<
88          "\'SOUND_VELOCITY\' variable is not added to the ModelPart nodal data." << std::endl;
89
90          KRATOS_ERROR_IF_NOT( mrModelPart.GetNodalSolutionStepVariablesList().Has(PRESSURE) ) <<
91          "\'PRESSURE\' variable is not added to the ModelPart nodal data." << std::endl;
92      }
93
94      void TaitEquationProcess::AssignTaitEquation()
95      {
96          int num_nodes = mrModelPart.NumberOfNodes();
97          // #pragma omp parallel for firstprivate(num_nodes,ambient_temperature)
98          for (int i = 0; i < num_nodes; ++i)
99          {
100             ModelPart::NodeIterator iNode = mrModelPart.NodesBegin() + i;
101             double pressure = iNode->FastGetSolutionStepValue(PRESSURE);
102             double mod_rho = mrho_0*std::pow((pressure - mp_0)/mk_0 + 1,1/mtheta);
103             double modified_c = std::pow(mk_0*mtheta*std::pow(mod_rho/mrho_0,mtheta - 1)/mrho_0,0.5);
104             iNode->FastGetSolutionStepValue(DENSITY) = mod_rho;
105             iNode->SetValue(SOUND_VELOCITY,modified_c);
106         }
107
108     }
109
110     /* External functions ****************************************************/
```

```
111
112     /// output stream function
113     inline std::ostream& operator << (
114         std::ostream& rOStream,
115         const TaitEquationProcess& rThis)
116     {
117         rThis.PrintData(rOStream);
118         return rOStream;
119     }
120
121 }
```

C.4 Constitutive laws

C.4.1 Fluid Constitutive law

```
1  //     |  /           |
2  //     ' /   __| _` | __|  _ \   __|
3  //     . \   |    (   | |   (   |\__ `
4  //    _|\_\_|  \__,_|\__|\___/ ____/
5  //                   Multi-Physics
6  //
7  //  License:         BSD License
8  //                   Kratos default license: kratos/license.txt
9  //
10 //  Main authors:    Jordi Cotela
11 //
12
13 #include "fluid_constitutive_law.h"
14
15 namespace Kratos {
16
17 // Life cycle ////////////////////////////////////////////////////////////////
18
19 FluidConstitutiveLaw::FluidConstitutiveLaw():
20     ConstitutiveLaw() {}
21
22 FluidConstitutiveLaw::FluidConstitutiveLaw(const FluidConstitutiveLaw& rOther):
23     ConstitutiveLaw(rOther) {}
24
25 FluidConstitutiveLaw::~FluidConstitutiveLaw() {}
26
27 // Public operations /////////////////////////////////////////////////////////
28
29 ConstitutiveLaw::Pointer FluidConstitutiveLaw::Clone() const {
30     KRATOS_ERROR << "Calling base FluidConstitutiveLaw::Clone method. This "
31                     "class should not be instantiated. Please check your "
32                     "constitutive law."
33                  << std::endl;
34     return Kratos::make_shared<FluidConstitutiveLaw>(*this);
35 }
36
37 void FluidConstitutiveLaw::CalculateMaterialResponseCauchy(Parameters& rValues) {
38     KRATOS_ERROR << "Calling base "
39                     "FluidConstitutiveLaw::CalculateMaterialResponseCauchy "
40                     "method. This class should not be instantiated. Please "
41                     "check your constitutive law."
42                  << std::endl;
43 }
44
45 int FluidConstitutiveLaw::Check(const Properties& rMaterialProperties,
46     const GeometryType& rElementGeometry,
47     const ProcessInfo& rCurrentProcessInfo) {
48     KRATOS_ERROR << "Calling base "
49                     "FluidConstitutiveLaw::Check "
50                     "method. This class should not be instantiated. Please "
```

```
51                     "check your constitutive law."
52                 << std::endl;
53     return 999;
54 }
55
56 // Access ////////////////////////////////////////////////////////////////////
57
58 int& FluidConstitutiveLaw::CalculateValue(ConstitutiveLaw::Parameters& rParameters, const Variable<
       int>& rThisVariable,int& rValue) {
59     return ConstitutiveLaw::CalculateValue(rParameters,rThisVariable,rValue);
60 }
61
62 double& FluidConstitutiveLaw::CalculateValue(ConstitutiveLaw::Parameters& rParameters, const Variable
       <double>& rThisVariable, double& rValue) {
63     rValue = this->GetEffectiveViscosity(rParameters);
64     return rValue;
65 }
66
67 Vector& FluidConstitutiveLaw::CalculateValue(ConstitutiveLaw::Parameters& rParameters, const Variable
       <Vector>& rThisVariable, Vector& rValue) {
68     return ConstitutiveLaw::CalculateValue(rParameters,rThisVariable,rValue);
69 }
70
71 Matrix& FluidConstitutiveLaw::CalculateValue(ConstitutiveLaw::Parameters& rParameters, const Variable
       <Matrix>& rThisVariable, Matrix& rValue) {
72     return ConstitutiveLaw::CalculateValue(rParameters,rThisVariable,rValue);
73 }
74
75 array_1d<double, 3 > & FluidConstitutiveLaw::CalculateValue(ConstitutiveLaw::Parameters& rParameters,
        const Variable<array_1d<double, 3 > >& rThisVariable,array_1d<double, 3 > & rValue) {
76     return ConstitutiveLaw::CalculateValue(rParameters,rThisVariable,rValue);
77 }
78
79 array_1d<double, 6 > & FluidConstitutiveLaw::CalculateValue(ConstitutiveLaw::Parameters& rParameters,
        const Variable<array_1d<double, 6 > >& rThisVariable, array_1d<double, 6 > & rValue) {
80     return ConstitutiveLaw::CalculateValue(rParameters,rThisVariable,rValue);
81 }
82
83 // Inquiry ///////////////////////////////////////////////////////////////////
84
85 ConstitutiveLaw::SizeType FluidConstitutiveLaw::WorkingSpaceDimension() {
86     KRATOS_ERROR << "Calling base "
87                     "FluidConstitutiveLaw::WorkingSpaceDimension "
88                     "method. This class should not be instantiated. Please "
89                     "check your constitutive law."
90                 << std::endl;
91     return 0;
92 }
93
94 ConstitutiveLaw::SizeType FluidConstitutiveLaw::GetStrainSize() {
95     KRATOS_ERROR << "Calling base "
96                     "FluidConstitutiveLaw::GetStrainSize "
97                     "method. This class should not be instantiated. Please "
98                     "check your constitutive law."
99                 << std::endl;
100    return 0;
101 }
102
103 // Info //////////////////////////////////////////////////////////////////////
104
105
106 std::string FluidConstitutiveLaw::Info() const {
107     return "FluidConstitutiveLaw";
108 }
109
110 void FluidConstitutiveLaw::PrintInfo(std::ostream& rOStream) const {
111     rOStream << this->Info();
112 }
113
114 void FluidConstitutiveLaw::PrintData(std::ostream& rOStream) const {
115     rOStream << this->Info();
```

```
116 }
117
118 // Protected operations ////////////////////////////////////////////////////
119
120 void FluidConstitutiveLaw::NewtonianConstitutiveMatrix2D(
121     double EffectiveViscosity,
122     Matrix& rC) {
123
124     constexpr double two_thirds = 2./3.;
125     constexpr double four_thirds = 4./3.;
126
127     rC(0,0) = EffectiveViscosity * four_thirds;
128     rC(0,1) = -EffectiveViscosity * two_thirds;
129     rC(0,2) = 0.0;
130     rC(1,0) = -EffectiveViscosity * two_thirds;
131     rC(1,1) = EffectiveViscosity * four_thirds;
132     rC(1,2) = 0.0;
133     rC(2,0) = 0.0;
134     rC(2,1) = 0.0;
135     rC(2,2) = EffectiveViscosity;
136 }
137
138 void FluidConstitutiveLaw::NewtonianConstitutiveMatrix3D(
139     double EffectiveViscosity,
140     Matrix& rC) {
141
142     rC.clear();
143
144     constexpr double two_thirds = 2./3.;
145     constexpr double four_thirds = 4./3.;
146
147     rC(0,0) = EffectiveViscosity * four_thirds;
148     rC(0,1) = -EffectiveViscosity * two_thirds;
149     rC(0,2) = -EffectiveViscosity * two_thirds;
150
151     rC(1,0) = -EffectiveViscosity * two_thirds;
152     rC(1,1) = EffectiveViscosity * four_thirds;
153     rC(1,2) = -EffectiveViscosity * two_thirds;
154
155     rC(2,0) = -EffectiveViscosity * two_thirds;
156     rC(2,1) = -EffectiveViscosity * two_thirds;
157     rC(2,2) = EffectiveViscosity * four_thirds;
158
159     rC(3,3) = EffectiveViscosity;
160     rC(4,4) = EffectiveViscosity;
161     rC(5,5) = EffectiveViscosity;
162 }
163
164 void FluidConstitutiveLaw::NewtonianConstitutiveMatrix2D(
165     double DynamicViscosity, double BulkViscosity, Matrix& rC) {
166
167     constexpr double two_thirds = 2./3.;
168     constexpr double four_thirds = 4./3.;
169
170     rC(0,0) = DynamicViscosity * four_thirds + BulkViscosity;
171     rC(0,1) = -DynamicViscosity * two_thirds + BulkViscosity;
172     rC(0,2) = 0.0;
173     rC(1,0) = -DynamicViscosity * two_thirds + BulkViscosity;
174     rC(1,1) = DynamicViscosity * four_thirds + BulkViscosity;
175     rC(1,2) = 0.0;
176     rC(2,0) = 0.0;
177     rC(2,1) = 0.0;
178     rC(2,2) = DynamicViscosity;
179 }
180
181 void FluidConstitutiveLaw::NewtonianConstitutiveMatrix3D(
182     double DynamicViscosity, double BulkViscosity, Matrix& rC) {
183
184     rC.clear();
185
186     constexpr double two_thirds = 2./3.;
```

```
187     constexpr double four_thirds = 4./3.;
188
189     rC(0,0) = DynamicViscosity * four_thirds +  BulkViscosity;
190     rC(0,1) = -DynamicViscosity * two_thirds +  BulkViscosity;
191     rC(0,2) = -DynamicViscosity * two_thirds +  BulkViscosity;
192
193     rC(1,0) = -DynamicViscosity * two_thirds + BulkViscosity;
194     rC(1,1) = DynamicViscosity * four_thirds + BulkViscosity;
195     rC(1,2) = -DynamicViscosity * two_thirds + BulkViscosity;
196
197     rC(2,0) = -DynamicViscosity * two_thirds + BulkViscosity;
198     rC(2,1) = -DynamicViscosity * two_thirds + BulkViscosity;
199     rC(2,2) = DynamicViscosity* four_thirds   + BulkViscosity;
200
201     rC(3,3) = DynamicViscosity;
202     rC(4,4) = DynamicViscosity;
203     rC(5,5) = DynamicViscosity;
204 }
205
206 // Protected access ////////////////////////////////////////////////////////
207
208 double FluidConstitutiveLaw::GetEffectiveViscosity(ConstitutiveLaw::Parameters& rParameters) const {
209     KRATOS_ERROR << "Accessing base class FluidConstitutiveLaw::GetEffectiveViscosity." << std::endl;
210     return 0.0;
211 }
212
213 double FluidConstitutiveLaw::GetValueFromTable(
214     const Variable<double> &rVariableInput,
215     const Variable<double> &rVariableOutput,
216     ConstitutiveLaw::Parameters &rParameters) const
217 {
218     // Get material properties from constitutive law parameters
219     const Properties &r_properties = rParameters.GetMaterialProperties();
220
221     double gauss_output;
222     if (r_properties.HasTable(rVariableInput,rVariableOutput)) {
223         // Get geometry and Gauss pt. data
224         const auto &r_geom = rParameters.GetElementGeometry();
225         const auto &r_N = rParameters.GetShapeFunctionsValues();
226
227         // Compute the input variable Gauss pt. value
228         double gauss_input = 0.0;
229         for (unsigned int i_node = 0; i_node < r_N.size(); ++i_node) {
230             const double &r_val = r_geom[i_node].FastGetSolutionStepValue(rVariableInput);
231             gauss_input += r_val * r_N[i_node];
232         }
233
234         // Retrieve the output variable from the table
235         const auto &r_table = r_properties.GetTable(rVariableInput, rVariableOutput);
236         gauss_output = r_table.GetValue(gauss_input);
237     } else {
238         KRATOS_ERROR << "FluidConstitutiveLaw " << this->Info() << " has no table with variables " <<
      rVariableInput << " " << rVariableOutput << std::endl;
239     }
240
241     return gauss_output;
242 }
243
244 // Serialization ///////////////////////////////////////////////////////////
245
246 void FluidConstitutiveLaw::save(Serializer& rSerializer) const {
247     KRATOS_SERIALIZE_SAVE_BASE_CLASS(rSerializer, ConstitutiveLaw );
248 }
249
250 void FluidConstitutiveLaw::load(Serializer& rSerializer) {
251     KRATOS_SERIALIZE_LOAD_BASE_CLASS(rSerializer, ConstitutiveLaw );
252 }
253
254 }
```

C.4.2 2D weakly-compressible law

```
1  //     |  /           |
2  //     ' /   __| _` | __|  _ \   __|
3  //     . \   |   (   | |   (   |\__ `
4  //    _|\_\_|  \__,_|\__|\___/ ____/
5  //                   Multi-Physics
6  //
7  //  License:         BSD License
8  //                   Kratos default license: kratos/license.txt
9  //
10 //  Main authors:    Ruben Zorrilla
11 //
12
13 // System includes
14 #include <iostream>
15
16 // External includes
17
18 // Project includes
19 #include "includes/cfd_variables.h"
20 #include "includes/checks.h"
21 #include "custom_constitutive/newtonian_compressible_2d_law.h"
22 #include "fluid_dynamics_application_variables.h"
23
24 namespace Kratos
25 {
26
27 //***************************CONSTRUCTOR*****************************************
28 //*******************************************************************************
29
30 CompressibleNewtonian2DLaw::CompressibleNewtonian2DLaw()
31     : FluidConstitutiveLaw()
32 {}
33
34 //***************************COPY CONSTRUCTOR*****************************************
35 //*******************************************************************************
36
37 CompressibleNewtonian2DLaw::CompressibleNewtonian2DLaw(const CompressibleNewtonian2DLaw& rOther)
38     : FluidConstitutiveLaw(rOther)
39 {}
40
41 //***************************CLONE*****************************************
42 //*******************************************************************************
43
44 ConstitutiveLaw::Pointer CompressibleNewtonian2DLaw::Clone() const {
45     return Kratos::make_shared<CompressibleNewtonian2DLaw>(*this);
46 }
47
48 //***************************DESTRUCTOR*****************************************
49 //*******************************************************************************
50
51 CompressibleNewtonian2DLaw::~CompressibleNewtonian2DLaw() {}
52
53 ConstitutiveLaw::SizeType CompressibleNewtonian2DLaw::WorkingSpaceDimension() {
54     return 2;
55 }
56
57 ConstitutiveLaw::SizeType CompressibleNewtonian2DLaw::GetStrainSize() {
58     return 3;
59 }
60
61 void  CompressibleNewtonian2DLaw::CalculateMaterialResponseCauchy(Parameters& rValues)
62 {
63     const Flags& options = rValues.GetOptions();
64     const Vector& r_strain_rate = rValues.GetStrainVector();
65     Vector& r_viscous_stress = rValues.GetStressVector();
66     double mu_star = 0.0;
67     double beta_star = 0.0;
68
```

```
69      const double mu = this->GetEffectiveViscosity(rValues);
70      GetArtificialViscosities(beta_star, mu_star, rValues);
71
72      const double trace = r_strain_rate[0] + r_strain_rate[1];
73      const double volumetric_part = trace/3.0; // Note: this should be small for an incompressible
        fluid (it is basically the incompressibility error)
74
75      r_viscous_stress[0] = 2.0*(mu + mu_star)*(r_strain_rate[0] - volumetric_part) + beta_star*trace;
76      r_viscous_stress[1] = 2.0*(mu + mu_star)*(r_strain_rate[1] - volumetric_part) + beta_star*trace;
77      r_viscous_stress[2] = (mu + mu_star)*r_strain_rate[2];
78
79      if( options.Is( ConstitutiveLaw::COMPUTE_CONSTITUTIVE_TENSOR ) )
80      {
81          this->NewtonianConstitutiveMatrix2D(mu + mu_star,beta_star,rValues.GetConstitutiveMatrix());
82      }
83  }
84
85  int CompressibleNewtonian2DLaw::Check(
86      const Properties& rMaterialProperties,
87      const GeometryType& rElementGeometry,
88      const ProcessInfo& rCurrentProcessInfo)
89  {
90      // Check viscosity value
91      KRATOS_ERROR_IF(rMaterialProperties[DYNAMIC_VISCOSITY] <= 0.0)
92          << "Incorrect or missing DYNAMIC_VISCOSITY provided in process info for Newtonian2DLaw: " <<
        rMaterialProperties[DYNAMIC_VISCOSITY] << std::endl;
93
94      return 0;
95  }
96
97  std::string CompressibleNewtonian2DLaw::Info() const {
98      return "CompressibleNewtonian2DLaw";
99  }
100
101 double CompressibleNewtonian2DLaw::GetEffectiveViscosity(ConstitutiveLaw::Parameters& rParameters)
        const
102 {
103     const Properties &r_prop = rParameters.GetMaterialProperties();
104     const double effective_viscosity = r_prop[DYNAMIC_VISCOSITY];
105     return effective_viscosity;
106 }
107
108 void CompressibleNewtonian2DLaw::GetArtificialViscosities(double& rbeta_star, double& rmu_star,
        ConstitutiveLaw::Parameters& rParameters)
109 {
110     const SizeType n_nodes = 3;
111     const GeometryType& r_geom = rParameters.GetElementGeometry();
112     const array_1d<double,n_nodes>& rN = rParameters.GetShapeFunctionsValues();
113
114     // Compute Gauss pt. interpolation value
115     for (unsigned int i = 0; i < n_nodes; ++i){
116         rbeta_star += rN[i] * r_geom[i].GetValue(ARTIFICIAL_BULK_VISCOSITY);
117         rmu_star   += rN[i] * r_geom[i].GetValue(ARTIFICIAL_DYNAMIC_VISCOSITY);
118     }
119 }
120
121 void CompressibleNewtonian2DLaw::save(Serializer& rSerializer) const {
122     KRATOS_SERIALIZE_SAVE_BASE_CLASS( rSerializer, FluidConstitutiveLaw )
123 }
124
125 void CompressibleNewtonian2DLaw::load(Serializer& rSerializer) {
126     KRATOS_SERIALIZE_LOAD_BASE_CLASS( rSerializer, FluidConstitutiveLaw )
127 }
128
129 } // Namespace Kratos
```

C.4.3 3D weakly-compressible law

```
1 //    |  /           |
```

```
2  //     ' /   __| _` | __|  _ \   __|
3  //     . \   |   (   | |   (   |\__ `
4  //    _|\_\_|  \__,_|\__|\___/ ____/
5  //                   Multi-Physics
6  //
7  //  License:         BSD License
8  //                   Kratos default license: kratos/license.txt
9  //
10 //  Main authors:    Riccardo Rossi
11 //
12
13 // System includes
14 #include <iostream>
15
16 // External includes
17
18 // Project includes
19 #include "includes/cfd_variables.h"
20 #include "includes/checks.h"
21 #include "custom_constitutive/newtonian_compressible_3d_law.h"
22 #include "fluid_dynamics_application_variables.h"
23
24 namespace Kratos
25 {
26
27 //*****************************CONSTRUCTOR*****************************************
28 //*********************************************************************************
29
30 CompressibleNewtonian3DLaw::CompressibleNewtonian3DLaw()
31     : FluidConstitutiveLaw()
32 {}
33
34 //*****************************COPY CONSTRUCTOR************************************
35 //*********************************************************************************
36
37 CompressibleNewtonian3DLaw::CompressibleNewtonian3DLaw(const CompressibleNewtonian3DLaw& rOther)
38     : FluidConstitutiveLaw(rOther)
39 {}
40
41 //*******************************CLONE*********************************************
42 //*********************************************************************************
43
44 ConstitutiveLaw::Pointer CompressibleNewtonian3DLaw::Clone() const {
45     return Kratos::make_shared<CompressibleNewtonian3DLaw>(*this);
46 }
47
48 //*******************************DESTRUCTOR****************************************
49 //*********************************************************************************
50
51 CompressibleNewtonian3DLaw::~CompressibleNewtonian3DLaw() {}
52
53 ConstitutiveLaw::SizeType CompressibleNewtonian3DLaw::WorkingSpaceDimension() {
54     return 3;
55 }
56
57 ConstitutiveLaw::SizeType CompressibleNewtonian3DLaw::GetStrainSize() {
58     return 6;
59 }
60
61 void  CompressibleNewtonian3DLaw::CalculateMaterialResponseCauchy (Parameters& rValues)
62 {
63     const Flags& options = rValues.GetOptions();
64     const Vector& r_strain_rate = rValues.GetStrainVector();
65     Vector& r_viscous_stress = rValues.GetStressVector();
66     double mu_star = 0.0;
67     double beta_star = 0.0;
68
69     const double mu = this->GetEffectiveViscosity(rValues);
70     GetArtificialViscosities(beta_star, mu_star, rValues);
71
72     const double trace = r_strain_rate[0] + r_strain_rate[1] + r_strain_rate[2];
```

```
73      const double volumetric_part = trace/3.0; // Note: this should be small for an incompressible
        fluid (it is basically the incompressibility error)
74
75      //computation of stress
76      r_viscous_stress[0] = 2.0*(mu + mu_star)*(r_strain_rate[0] - volumetric_part) + beta_star*trace;
77      r_viscous_stress[1] = 2.0*(mu + mu_star)*(r_strain_rate[1] - volumetric_part) + beta_star*trace;
78      r_viscous_stress[2] = 2.0*(mu + mu_star)*(r_strain_rate[2] - volumetric_part) + beta_star*trace;
79      r_viscous_stress[3] = (mu + mu_star)*r_strain_rate[3];
80      r_viscous_stress[4] = (mu + mu_star)*r_strain_rate[4];
81      r_viscous_stress[5] = (mu + mu_star)*r_strain_rate[5];
82
83      if( options.Is( ConstitutiveLaw::COMPUTE_CONSTITUTIVE_TENSOR ) )
84      {
85          this->NewtonianConstitutiveMatrix3D(mu + mu_star,beta_star,rValues.GetConstitutiveMatrix());
86          //basetype::
87      }
88  }
89
90  int CompressibleNewtonian3DLaw::Check(
91      const Properties& rMaterialProperties,
92      const GeometryType& rElementGeometry,
93      const ProcessInfo& rCurrentProcessInfo)
94  {
95
96      // Check viscosity value
97      KRATOS_ERROR_IF(rMaterialProperties[DYNAMIC_VISCOSITY] <= 0.0)
98          << "Incorrect or missing DYNAMIC_VISCOSITY provided in process info for Newtonian2DLaw: " <<
        rMaterialProperties[DYNAMIC_VISCOSITY] << std::endl;
99
100     for (unsigned int i = 0; i < rElementGeometry.size(); i++) {
101         const Node<3>& rNode = rElementGeometry[i];
102         KRATOS_ERROR_IF(rNode.GetValue(ARTIFICIAL_DYNAMIC_VISCOSITY) < 0.0)
103             << "ARTIFICIAL_DYNAMIC_VISCOSITY was not correctly assigned to nodes for Constitutive Law
        .\n";
104         KRATOS_ERROR_IF(rNode.GetValue(ARTIFICIAL_BULK_VISCOSITY) < 0.0)
105             << "ARTIFICIAL_BULK_VISCOSITY was not correctly assigned to nodes for Constitutive Law.\n
        ";
106     }
107     return 0;
108 }
109
110 std::string CompressibleNewtonian3DLaw::Info() const {
111     return "CompressibleNewtonian3DLaw";
112 }
113
114 double CompressibleNewtonian3DLaw::GetEffectiveViscosity(ConstitutiveLaw::Parameters& rParameters)
        const
115 {
116     const Properties &r_prop = rParameters.GetMaterialProperties();
117     const double effective_viscosity = r_prop[DYNAMIC_VISCOSITY];
118     return effective_viscosity;
119 }
120
121 void CompressibleNewtonian3DLaw::GetArtificialViscosities(double& rbeta_star, double& rmu_star,
        ConstitutiveLaw::Parameters& rParameters)
122 {
123     const GeometryType& r_geom = rParameters.GetElementGeometry();
124     const SizeType n_nodes = r_geom.size();
125     const auto& rN = rParameters.GetShapeFunctionsValues();
126
127     // Compute Gauss pt. interpolation value
128     for (unsigned int i = 0; i < n_nodes; ++i){
129
130         rbeta_star += rN[i] * r_geom[i].GetValue(ARTIFICIAL_BULK_VISCOSITY);
131         rmu_star   += rN[i] * r_geom[i].GetValue(ARTIFICIAL_DYNAMIC_VISCOSITY);
132     }
133 }
134
135 void CompressibleNewtonian3DLaw::save(Serializer& rSerializer) const {
136     KRATOS_SERIALIZE_SAVE_BASE_CLASS( rSerializer, FluidConstitutiveLaw )
137 }
```

```
138
139 void CompressibleNewtonian3DLaw::load(Serializer& rSerializer) {
140     KRATOS_SERIALIZE_LOAD_BASE_CLASS( rSerializer, FluidConstitutiveLaw )
141 }
142
143 } // Namespace Kratos
```

C.5 Custom processes

C.5.1 Bubble area

```
1 #if !defined(KRATOS_CALCULATE_CUT_AREA_H_INCLUDED)
2 #define  KRATOS_CALCULATE_CUT_AREA_H_INCLUDED
3
4
5 // System includes
6 #include <string>
7 #include <iostream>
8
9
10 // External includes
11
12
13 // Project includes
14 #include "includes/define.h"
15 #include "includes/kratos_parameters.h"
16 #include "includes/model_part.h"
17 #include "processes/process.h"
18 #include "modified_shape_functions/tetrahedra_3d_4_modified_shape_functions.h"
19 #include "modified_shape_functions/triangle_2d_3_modified_shape_functions.h"
20
21
22 namespace Kratos
23 {
24   class CalculateCutArea
25   {
26   public:
27
28     KRATOS_CLASS_POINTER_DEFINITION(CalculateCutArea);
29
30         typedef Node<3> NodeType;
31         typedef Geometry<NodeType> GeometryType;
32
33     CalculateCutArea(ModelPart& model_part)
34       : mr_model_part(model_part)               //mr_model_part is saved as private variable (declared
        at the end of the file)
35     {
36       KRATOS_TRY
37       KRATOS_CATCH("")
38     }
39
40
41     ~CalculateCutArea()
42     {}
43
44
45     double Calculate()
46     {
47       KRATOS_TRY
48
49       double neg_vol = 0.0;
50       #pragma omp parallel for reduction(+: neg_vol)
51
52       //getting data for the given geometry
53             for (int i_elem = 0; i_elem < static_cast<int>(mr_model_part.NumberOfElements()); ++
       i_elem){ // iteration over all elements
```

```
54
55                  const auto ielem = mr_model_part.ElementsBegin() + i_elem;
56
57          Matrix shape_functions;
58                  GeometryType::ShapeFunctionsGradientsType shape_derivatives;
59                  Geometry<Node<3>> & rGeom = ielem->GetGeometry();
60                  unsigned int pt_count_neg = 0;
61
62                  // instead of using data.isCut()
63                  for (unsigned int pt = 0; pt < rGeom.Points().size(); pt++){
64                      if ( rGeom[pt].FastGetSolutionStepValue(DISTANCE) <= 0.0 ){
65                          pt_count_neg++;
66                      }
67                  }
68
69                  if ( pt_count_neg == rGeom.PointsNumber() ){
70                      // all nodes are negative (pointer is necessary to maintain polymorphism of
      DomainSize())
71                      neg_vol += ielem->pGetGeometry()->DomainSize();
72                  }
73                  else if ( 0 < pt_count_neg ){
74                      // element is cut by the surface (splitting)
75                      Kratos::unique_ptr<ModifiedShapeFunctions> p_modified_sh_func = nullptr;
76                      Vector w_gauss_neg_side(3, 0.0);
77
78                      Vector Distance( rGeom.PointsNumber(), 0.0 );
79                      for (unsigned int i = 0; i < rGeom.PointsNumber(); i++){
80                          // Control mechanism to avoid 0.0 ( is necessary because "
      distance_modification" possibly not yet executed )
81                          double& r_dist = rGeom[i].FastGetSolutionStepValue(DISTANCE);
82                          if (std::abs(r_dist) < 1.0e-12) {
83                              const double aux_dist = 1.0e-6* rGeom[i].GetValue(NODAL_H);
84                              if (r_dist > 0.0) {
85                                  #pragma omp critical
86                                  r_dist = aux_dist;
87                              } else {
88                                  #pragma omp critical
89                                  r_dist = -aux_dist;
90                              }
91                          }
92                          Distance[i] = rGeom[i].FastGetSolutionStepValue(DISTANCE);
93                      }
94
95
96                      if ( rGeom.PointsNumber() == 3 ){ p_modified_sh_func = Kratos::make_unique<
      Triangle2D3ModifiedShapeFunctions>(ielem->pGetGeometry(), Distance); }
97                      else if ( rGeom.PointsNumber() == 4 ){ p_modified_sh_func = Kratos::make_unique<
      Tetrahedra3D4ModifiedShapeFunctions>(ielem->pGetGeometry(), Distance); }
98                      else { KRATOS_ERROR << "The process can not be applied on this kind of element"
      << std::endl; }
99
100                     // Call the negative side modified shape functions calculator
101                     // Object p_modified_sh_func has full knowledge of slit geometry
102                     p_modified_sh_func->ComputeNegativeSideShapeFunctionsAndGradientsValues(
103                             shape_functions,                        // N
104                             shape_derivatives,                      // DN
105                             w_gauss_neg_side,                       // includes the weights of the GAUSS
      points (!!!)
106                             GeometryData::GI_GAUSS_1);              // first order Gauss integration
107
108                     for ( unsigned int i = 0; i < w_gauss_neg_side.size(); i++){
109                         neg_vol += w_gauss_neg_side[i];
110                     }
111                 }
112             }
113             return neg_vol;
114       KRATOS_CATCH("")
115     }
116
117
118   protected:
```

```
119
120
121     private:
122
123           ModelPart& mr_model_part;
124
125     };
126 }  // namespace Kratos.
127
128 #endif // KRATOS_CALCULATE_CUT_AREA_H_INCLUDED  defined
```

C.5.2 Energy rate and work

```
1  #if !defined(KRATOS_CALCULATE_NORMAL_VECTOR_H_INCLUDED)
2  #define  KRATOS_CALCULATE_NORMAL_VECTOR_H_INCLUDED
3
4
5  // System includes
6  #include <string>
7  #include <iostream>
8  #include <vector>
9  #include <array>
10 #include <tuple>
11
12 // External includes
13
14
15 // Project includes
16 #include "includes/define.h"
17 #include "includes/kratos_parameters.h"
18 #include "includes/model_part.h"
19 #include "processes/process.h"
20 #include "utilities/math_utils.h"
21 #include "utilities/divide_triangle_2d_3.h"
22 #include "utilities/divide_tetrahedra_3d_4.h"
23 #include "modified_shape_functions/triangle_2d_3_modified_shape_functions.h"
24 #include "modified_shape_functions/tetrahedra_3d_4_modified_shape_functions.h"
25
26 namespace Kratos
27 {
28   class CalculateNormalVector
29   {
30   public:
31
32     KRATOS_CLASS_POINTER_DEFINITION(CalculateNormalVector);
33
34         typedef Node<3> NodeType;
35         typedef Geometry<NodeType> GeometryType;
36
37     CalculateNormalVector(ModelPart& model_part)
38       : mr_model_part(model_part)              //mr_model_part is saved as private variable (declared
        at the end of the file)
39     {
40       KRATOS_TRY
41       KRATOS_CATCH("")
42     }
43
44
45     ~CalculateNormalVector()
46     {}
47
48
49     std::tuple<double, double, double> Calculate()
50     {
51       KRATOS_TRY
52
53             double work_rate {0}, total_energy{0}, total_energy_simplified {0};
54             const double A {1e5}, B {3e8}, gamma {7.15}, rho_0 {880};
55
```

```
56        //getting data for the given geometry
57        for(ModelPart::ElementsContainerType::iterator ielem = mr_model_part.ElementsBegin(); //looping
        the elements
58          ielem!=mr_model_part.ElementsEnd(); ielem++)
59        {
60          Matrix shape_functions;
61                  GeometryType::ShapeFunctionsGradientsType shape_derivatives;
62                  const GeometryType & rGeom = ielem->GetGeometry();
63                  GeometryType::Pointer p_geom = ielem->pGetGeometry();
64                  unsigned int pt_count_neg = 0;
65                  double elemental_energy {0};
66                  double elemental_energy_simplified {0};
67
68                  const unsigned int NumNodes = 4;
69
70                  Vector distances_vector( NumNodes, 0.0 );
71                  Vector pressure_vector( NumNodes, 0.0 );
72                  Vector density_vector( NumNodes, 0.0 );
73                  std::vector<array_1d<double, NumNodes>> velocity_vector;
74
75                  for (unsigned int i = 0; i < NumNodes; i++){
76                      distances_vector[i] = rGeom[i].FastGetSolutionStepValue(DISTANCE);
77                      pressure_vector[i] = rGeom[i].FastGetSolutionStepValue(PRESSURE);
78                      density_vector[i] = rGeom[i].FastGetSolutionStepValue(DENSITY);
79                      velocity_vector.push_back(rGeom[i].FastGetSolutionStepValue(VELOCITY));
80                      if ( distances_vector[i] <= 0.0 ){
81                          pt_count_neg++;
82                      }
83                  }
84
85                  if ( pt_count_neg > 0 && pt_count_neg <  NumNodes){
86                      // element is cut by the surface (splitting)
87
88                      // Construct the modified shape fucntions utility
89                      ModifiedShapeFunctions::Pointer p_modified_sh_func = nullptr;
90                      if (NumNodes == 4) {
91                          p_modified_sh_func = Kratos::make_shared<Tetrahedra3D4ModifiedShapeFunctions
        >(p_geom, distances_vector);
92                      } else if (NumNodes == 3) {
93                          p_modified_sh_func = Kratos::make_shared<Triangle2D3ModifiedShapeFunctions>(
        p_geom, distances_vector);
94                      } else { KRATOS_ERROR << "The process can not be applied on this kind of element"
         << std::endl; }
95
96                      ModifiedShapeFunctions::ShapeFunctionsGradientsType
        positive_side_sh_func_gradients, int_positive_side_sh_func_gradients;
97                      Vector positive_side_weights, int_positive_side_weights;
98                      Matrix positive_side_sh_func, int_positive_side_sh_func;
99
100                     // Call the interface outwards normal area vector calculator
101             std::vector<Vector> positive_side_area_normals;
102
103             p_modified_sh_func->ComputePositiveSideInterfaceAreaNormals(
104                 positive_side_area_normals,
105                 GeometryData::GI_GAUSS_2);
106
107                     p_modified_sh_func->ComputeInterfacePositiveSideShapeFunctionsAndGradientsValues(
108                         int_positive_side_sh_func,
109                         int_positive_side_sh_func_gradients,
110                         int_positive_side_weights,
111                         GeometryData::GI_GAUSS_2);
112
113                     unsigned int NGauss = int_positive_side_sh_func.size1();
114
115                     for (unsigned int g = 0; g < NGauss; g++)
116                     {
117                         array_1d<double,NumNodes> v_gauss_i = 0*rGeom[0].FastGetSolutionStepValue(
        VELOCITY);
118                         double p_gauss_i {0}, proj_i {0};
119                         for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
120                         {
```

```
121                             v_gauss_i   += int_positive_side_sh_func(g, iNode)*rGeom[iNode].
    FastGetSolutionStepValue(VELOCITY);
122                             p_gauss_i   += int_positive_side_sh_func(g, iNode)*rGeom[iNode].
    FastGetSolutionStepValue(PRESSURE);
123                         }
124                         proj_i = MathUtils<double>::Dot(v_gauss_i, positive_side_area_normals.at(g))
    / norm_2(positive_side_area_normals.at(g));
125                         work_rate += proj_i*p_gauss_i*int_positive_side_weights(g);
126                     }
127
128                     //ENERGY OF THE POSITIVE SIDE
129                     p_modified_sh_func->ComputePositiveSideShapeFunctionsAndGradientsValues(
130                         positive_side_sh_func,
131                         positive_side_sh_func_gradients,
132                         positive_side_weights,
133                         GeometryData::GI_GAUSS_2);
134                     NGauss = positive_side_sh_func.size1();
135                     for (unsigned int g = 0; g < NGauss; g++)
136                     {
137                         array_1d<double,NumNodes - 1> v_gauss_p = ZeroVector(NumNodes); //(0*rGeom
    [0].FastGetSolutionStepValue(VELOCITY);
138                         array_1d<double,NumNodes - 1> p_grad_p = prod(trans(
    positive_side_sh_func_gradients[g]), pressure_vector);
139                         double p_gauss_p {0}, div_vp_p {0}, rho_gauss_p {0};
140                         for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
141                         {
142                             div_vp_p       += positive_side_sh_func_gradients[g](iNode,0)*
    velocity_vector.at(iNode)(0)+ positive_side_sh_func_gradients[g](iNode,1)*velocity_vector.at(
    iNode)(1);
143                             v_gauss_p      += positive_side_sh_func(g, iNode)*rGeom[iNode].
    FastGetSolutionStepValue(VELOCITY);
144                             p_gauss_p      += positive_side_sh_func(g, iNode)*rGeom[iNode].
    FastGetSolutionStepValue(PRESSURE);
145                             rho_gauss_p    += positive_side_sh_func(g, iNode)*rGeom[iNode].
    FastGetSolutionStepValue(DENSITY);
146                         }
147
148                         double e = B*std::pow(rho_gauss_p/rho_0,gamma)/(gamma - 1) + B - A;
149                         //double e = p_gauss_p;
150                         elemental_energy += positive_side_weights(g)*(0.5*MathUtils<double>::Dot(
    v_gauss_p, v_gauss_p)*rho_gauss_p + e);
151                         elemental_energy_simplified += positive_side_weights(g)*(MathUtils<double>::
    Dot(v_gauss_p, p_grad_p) + p_gauss_p*div_vp_p);
152                     }
153                 } else if (pt_count_neg == 0){
154                     const GeometryType::IntegrationPointsArrayType& IntegrationPoints = rGeom.
    IntegrationPoints(GeometryData::GI_GAUSS_2);
155                     const unsigned int NumGauss = IntegrationPoints.size();
156
157                     Matrix N = rGeom.ShapeFunctionsValues(GeometryData::GI_GAUSS_2);
158                     Vector DetJ;
159                     GeometryType::ShapeFunctionsGradientsType DN_DX;
160                     rGeom.ShapeFunctionsIntegrationPointsGradients(DN_DX, DetJ, GeometryData::
    GI_GAUSS_2);
161
162                     for (unsigned int g = 0; g < NumGauss; g++)
163                     {
164                         double Weight = DetJ(g) * IntegrationPoints[g].Weight(); //"Jacobian" is 2.0*
    A for triangles
165                         array_1d<double,NumNodes - 1> vgauss = 0*rGeom[0].FastGetSolutionStepValue(
    VELOCITY);
166                         array_1d<double,NumNodes - 1> p_grad = prod(trans(DN_DX[g]), pressure_vector)
    ;
167                         double pgauss {0}, div_vel {0}, rho_gauss {0};
168
169                         for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
170                         {
171                             div_vel      += DN_DX[g](iNode,0)*velocity_vector.at(iNode)(0)+ DN_DX[g](
    iNode,1)*velocity_vector.at(iNode)(1);
172                             vgauss       += N(iNode,g)*rGeom[iNode].FastGetSolutionStepValue(VELOCITY
    );
```

```
173                             pgauss        += N(iNode,g)*rGeom[iNode].FastGetSolutionStepValue(PRESSURE
    );
174                             rho_gauss     += N(iNode,g)*rGeom[iNode].FastGetSolutionStepValue(DENSITY)
    ;
175                         }
176
177                         double e = B*std::pow(rho_gauss/rho_0,gamma)/(gamma - 1) + B - A;
178                         //double e = pgauss;
179                         elemental_energy             += Weight*(0.5*MathUtils<double>::Dot(vgauss,
    vgauss)*rho_gauss + e);
180                         elemental_energy_simplified += Weight*(MathUtils<double>::Dot(vgauss, p_grad)
     + pgauss*div_vel);
181                     }
182                 }
183
184                 total_energy += elemental_energy;
185                 total_energy_simplified += elemental_energy_simplified;
186             }
187             return  std::make_tuple(total_energy, work_rate, total_energy_simplified);
188       KRATOS_CATCH("")
189     }
190
191
192   protected:
193
194
195   private:
196
197         ModelPart& mr_model_part;
198
199   };
200 }  // namespace Kratos.
201
202 #endif // KRATOS_CALCULATE_NORMAL_VECTOR_H_INCLUDED  defined
```

C.5.3 FE errors

```
1  #if !defined(KRATOS_CALCULATE_NORML_H_INCLUDED)
2  #define  KRATOS_CALCULATE_NORML_H_INCLUDED
3
4
5  // System includes
6  #include <string>
7  #include <iostream>
8  #include <vector>
9  #include <array>
10 #include <tuple>
11 #include <sstream>
12 #include <fstream>
13
14 // External includes
15
16
17 // Project includes
18 #include "includes/define.h"
19 #include "includes/kratos_parameters.h"
20 #include "includes/model_part.h"
21 #include "processes/process.h"
22 #include "utilities/math_utils.h"
23 #include "utilities/divide_triangle_2d_3.h"
24 #include "utilities/divide_tetrahedra_3d_4.h"
25 #include "modified_shape_functions/triangle_2d_3_modified_shape_functions.h"
26 #include "modified_shape_functions/tetrahedra_3d_4_modified_shape_functions.h"
27
28 namespace Kratos
29 {
30   class CalculateNormL
31   {
32   public:
```

```
33
34      KRATOS_CLASS_POINTER_DEFINITION(CalculateNormL);
35
36          typedef Node<3> NodeType;
37          typedef Geometry<NodeType> GeometryType;
38
39      CalculateNormL(ModelPart& model_part)
40        : mr_model_part(model_part)               //mr_model_part is saved as private variable (declared
          at the end of the file)
41      {
42        KRATOS_TRY
43        KRATOS_CATCH("")
44      }
45
46
47      ~CalculateNormL()
48      {}
49
50
51      double Calculate()
52      {
53        KRATOS_TRY
54              std::string inFileName = "/home/pau/Desktop/TESTS/3D_convergence/rho003.txt";
55              std::ifstream inFile;
56              inFile.open(inFileName.c_str());
57              std::vector<double> exact_pressure;
58              std::string line;
59              long double number;
60
61              if (inFile.is_open())
62              {
63                   while(!inFile.eof()){
64                        getline(inFile,line);
65                        std::istringstream iss( line );
66
67                        while( iss >> number ){
68                             exact_pressure.push_back(number);
69                        }
70                   }
71
72                   inFile.close(); // CLose input file
73              }
74              int n = exact_pressure.size();
75              std::cout << n;
76              std::cout << std::setprecision(15)  << exact_pressure.at(0);
77              KRATOS_WATCH(exact_pressure.at(0))
78
79        //getting data for the _given geometry
80              double err_L2 {0}, norm_L2 {0};
81        for(ModelPart::ElementsContainerType::iterator ielem = mr_model_part.ElementsBegin(); //looping
          the elements
82          ielem!=mr_model_part.ElementsEnd(); ielem++)
83        {
84          Matrix shape_functions;
85                  GeometryType::ShapeFunctionsGradientsType shape_derivatives;
86                  const GeometryType & rGeom = ielem->GetGeometry();
87                  GeometryType::Pointer p_geom = ielem->pGetGeometry();
88                  unsigned int pt_count_neg = 0;
89                  double errL2_e {0}, normL2_e {0};
90
91                  const unsigned int NumNodes = 4;
92
93                  Vector distances_vector( NumNodes, 0.0 );
94                  Vector pressure_vector( NumNodes, 0.0 );
95                  Vector density_vector( NumNodes, 0.0 );
96
97                  std::vector<array_1d<double, NumNodes>> velocity_vector;
98
99                  for (unsigned int i = 0; i < NumNodes; i++){
100                      distances_vector[i] = rGeom[i].FastGetSolutionStepValue(DISTANCE);
101                      pressure_vector[i] = rGeom[i].FastGetSolutionStepValue(PRESSURE);
```

```
102                     density_vector[i] = rGeom[i].FastGetSolutionStepValue(DENSITY);
103                     velocity_vector.push_back(rGeom[i].FastGetSolutionStepValue(VELOCITY));
104                     if ( distances_vector[i] <= 0.0 ){
105                         pt_count_neg++;
106                     }
107                 }
108 
109                 if ( pt_count_neg > 0 && pt_count_neg <  NumNodes){
110                     // element is cut by the surface (splitting)
111 
112                     // Construct the modified shape fucntions utility
113                     ModifiedShapeFunctions::Pointer p_modified_sh_func = nullptr;
114                     if (NumNodes == 4) {
115                         p_modified_sh_func = Kratos::make_shared<Tetrahedra3D4ModifiedShapeFunctions
    >(p_geom, distances_vector);
116                     } else if (NumNodes == 3) {
117                         p_modified_sh_func = Kratos::make_shared<Triangle2D3ModifiedShapeFunctions>(
    p_geom, distances_vector);
118                     } else { KRATOS_ERROR << "The process can not be applied on this kind of element"
     << std::endl; }
119 
120                     ModifiedShapeFunctions::ShapeFunctionsGradientsType
    positive_side_sh_func_gradients, int_positive_side_sh_func_gradients;
121                     Vector positive_side_weights, int_positive_side_weights;
122                     Matrix positive_side_sh_func, int_positive_side_sh_func;
123 
124                     //ENERGY OF THE POSITIVE SIDE
125                     p_modified_sh_func->ComputePositiveSideShapeFunctionsAndGradientsValues(
126                         positive_side_sh_func,
127                         positive_side_sh_func_gradients,
128                         positive_side_weights,
129                         GeometryData::GI_GAUSS_2);
130 
131                     unsigned int NGauss = positive_side_sh_func.size1();
132 
133                     for (unsigned int g = 0; g < NGauss; g++)
134                     {
135                         double cexact_gauss {0}, cnum_gauss {0};
136 
137                         for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
138                         {
139                             cnum_gauss  += positive_side_sh_func(g, iNode)*rGeom[iNode].
    FastGetSolutionStepValue(DENSITY);
140                             //cnum_gauss  += positive_side_sh_func(g, iNode)*norm_2(rGeom[iNode].
    FastGetSolutionStepValue(VELOCITY));
141                             cexact_gauss    += positive_side_sh_func(g, iNode)*exact_pressure[rGeom[
    iNode].Id() - 1];
142                         }
143 
144                         errL2_e  += (cnum_gauss - cexact_gauss)*(cnum_gauss - cexact_gauss)*
    positive_side_weights(g);
145                         normL2_e += (cexact_gauss)*(cexact_gauss)*positive_side_weights(g);
146                     }
147                 } else if (pt_count_neg == 0)
148                 {
149                     const GeometryType::IntegrationPointsArrayType& IntegrationPoints = rGeom.
    IntegrationPoints(GeometryData::GI_GAUSS_2);
150                     const unsigned int NumGauss = IntegrationPoints.size();
151 
152                     Matrix N = rGeom.ShapeFunctionsValues(GeometryData::GI_GAUSS_2);
153                     Vector DetJ;
154                     GeometryType::ShapeFunctionsGradientsType DN_DX;
155                     rGeom.ShapeFunctionsIntegrationPointsGradients(DN_DX, DetJ, GeometryData::
    GI_GAUSS_2);
156 
157                     for (unsigned int g = 0; g < NumGauss; g++)
158                     {
159                         double Weight = DetJ(g) * IntegrationPoints[g].Weight(); //"Jacobian" is 2.0*
    A for triangles
160                         double exact_gauss {0}, num_gauss {0};
161 
```

```
162                         for (unsigned int iNode = 0; iNode < NumNodes; ++iNode)
163                         {
164                             num_gauss    += N(iNode,g)*rGeom[iNode].FastGetSolutionStepValue(DENSITY);
165                             //num_gauss    += N(iNode,g)*norm_2(rGeom[iNode].FastGetSolutionStepValue(
       VELOCITY));
166                             exact_gauss += N(iNode,g)*exact_pressure[rGeom[iNode].Id() - 1];
167                         }
168
169                         errL2_e  += (num_gauss - exact_gauss)*(num_gauss - exact_gauss)*Weight;
170                         normL2_e += (exact_gauss)*(exact_gauss)*Weight;
171                     }
172                 }
173
174                 err_L2 += errL2_e;
175                 norm_L2 += normL2_e;
176             }
177             err_L2 = std::pow(err_L2, 0.5)/std::pow(norm_L2, 0.5);
178             return  err_L2;
179       KRATOS_CATCH("")
180     }
181
182
183   protected:
184
185
186   private:
187
188         ModelPart& mr_model_part;
189
190   };
191 }  // namespace Kratos.
192
193 #endif // KRATOS_CALCULATE_NORMAL_VECTOR_H_INCLUDED  defined
```

Bibliography

[1] Aster official website. www.code-aster.org. Accessed: 2021-05-13.

[2] S. Acharya, B. Baliga, Kailash Karki, Jagarlapudi Murthy, C. Prakash, and S. Vanka. Pressure-based finite-volume methods in computational fluid dynamics. *Journal of Heat Transfer-transactions of The Asme - J HEAT TRANSFER*, 129, 04 2007.

[3] Shantanu Bailoor, Aditya Annangi, Jung Hee Seo, and Rajneesh Bhardwaj. A fluid-structure interaction solver for compressible flows with applications in blast loading on thin elastic structures. *Applied Mathematical Modelling*, 52, 05 2017.

[4] J.P. Best. *The Dynamics of Underwater Explosions*. PhD. Thesis, Uni. Wollongong., 1991.

[5] Markus Boger. *Numerical modeling of compressible two-phase flows with a pressure-based method*. PhD thesis, 01 2014.

[6] Ryan Brady, Sebastien Muller, Margareta Petrovan-Boiarciuc, Guillaume Perigaud, and Ben Landis. Prevention of transformer tank explosion: Part 3—design of efficient protections using numerical simulations. volume 3, 01 2009.

[7] Ryan Brady, Sébastien Muller, Gaël Bressy, Philippe Magnier, and Guillaume Périgaud. Prevention of transformer tank explosion: Part 2 — development and application of a numerical simulation tool. volume 4, 01 2008.

[8] Guido Buresti. A note on stokes' hypothesis. *Acta Mechanica*, 226, 10 2015.

[9] Le Cao, Wenli Fei, Holger Grosshans, and Nianwen Cao. Simulation of underwater

explosions initiated by high-pressure gas bubbles of various initial shapes. *Applied Sciences*, 7:880, 08 2017.

[10] Jishnu Chandran R and A. Salih. A modified equation of state for water for a wide range of pressure and the concept of water shock tube. *Fluid Phase Equilibria*, 483, 11 2018.

[11] Ramon Codina, Santiago Badia, Joan Baiges, and Javier Principe. Variational multiscale methods in computational fluid dynamics. pages 1–28, 2017.

[12] Jean-Bernard Dastous, Jacques Lanteigne, and Marc Foata. Numerical method for the investigation of fault containment and tank rupture of power transformers. *Power Delivery, IEEE Transactions on*, 25:1657 – 1665, 08 2010.

[13] J. Donea, S. Giuliani, and J.P. Halleux. An arbitrary lagrangian-eulerian finite element method for transient dynamic fluid-structure interactions. *Computer Methods in Applied Mechanics and Engineering*, 33:689–723, 09 1982.

[14] Jean Donea and Antonio Huerta. *Finite Element Methods for Flow Problems*. 04 2003.

[15] Jean Donea, Antonio Huerta, Jean-Philippe Ponthot, and Antonio Rodriguez-Ferran. *Arbitrary Lagrangian-Eulerian Methods*, volume 1, pages 413–437. 11 2004.

[16] Mohanad El-Harbawi and Fahad Al-Mubaddel. Risk of fire and explosion in electrical substations due to the formation of flammable mixtures. *Scientific Reports*, 10, 04 2020.

[17] Pablo Fernandez, Cuong Nguyen, and J. Peraire. A physics-based shock capturing method for unsteady laminar and turbulent flows. 01 2018.

[18] Zoran Gajic, Ivo Brnčić, Birger Hillström, and Igor Ivankovic. Sensitive turn-to-turn fault protection for power transformers. 01 2005.

[19] P. Goyal, A.K. Chatterjee, and N. Sharma. Transformer oil analysis: An overview. 17:81–85, 01 2005.

[20] Jocelyn Jalbert, R Gilbert, and P Tétreault. Simultaneous determination of dissolved gases and moisture in mineral insulating oils by static headspace gas chromatography with helium photoionization pulsed discharge detection. *Analytical chemistry*, 73:3382–91, 08 2001.

[21] T. Kayali, G. Tuysuz, and M Martinez. Transformer explodes in turkish coal mine; 201 die in fire., 2014.

[22] Ellen Kuhl, S. Hulshoff, and René Borst. An arbitrary lagrangian eulerian finite-element approach for fluid–structure interaction phenomena. *International Journal for Numerical Methods in Engineering*, 57:117 – 142, 05 2003.

[23] Ben Landis, Sébastien Muller, Margareta Petrovan-Boiarciuc, Ryan Brady, and Guillaume Périgaud. Prevention of transformer tank explosion: Part 4—development of a fluid structure interaction numerical tool. volume 4, 01 2009.

[24] G.R. Liu and D. Karamanlidis. Mesh free methods: Moving beyond the finite element method. *Applied Mechanics Reviews - APPL MECH REV*, 56, 03 2003.

[25] Geoffrey Maines, Matei Radulescu, A. Bacciochini, B. Jodoin, and J Lee. Pressure waves generated by metastable intermolecular composites in an aqueous environment. *Journal of Physics: Conference Series*, 500:052028, 05 2014.

[26] E. Moreno, Narges Dialami, and Miguel Cervera. Modeling of spillage and debris floods as newtonian and viscoplastic bingham flows with free surface with mixed stabilized finite elements. *Journal of Non-Newtonian Fluid Mechanics*, 290, 02 2021.

[27] Sébastien Muller, Ryan Brady, Gaël Bressy, Philippe Magnier, and Guillaume Périgaud. Prevention of transformer tank explosion: Part 1 — experimental tests on large transformers. 01 2008.

[28] Claus-Dieter Munz, Sabine Roller, R. Klein, and K.J. Geratz. The extension of incompressible flow solvers to the weakly compressible regime. *Computers & Fluids*, 32:173–196, 02 2003.

[29] E. Oñate, S. R. Idelsohn, M. A. Celigueta, and B. Suárez. *The Particle Finite Element Method (PFEM). An Effective Numerical Technique for Solving Marine, Naval and Harbour Engineering Problems*, pages 65–81. Springer Netherlands, Dordrecht, 2013.

[30] R. Rossi P. Ryzhakov, E. Oñate and S. Idelsohn. *Lagrangian FE methods for coupled problems in fluid mechanics.* Monograph CIMNE, 2010.

[31] Ashish Pathak and Mehdi Raessi. A three-dimensional volume-of-fluid method

for reconstructing and advecting three-material interfaces forming contact lines. *Journal of Computational Physics*, 307, 12 2015.

[32] Allan D. Pierce. *Acoustics. An Introduction to Its Physical Principles and Applications.* Springer, 2018.

[33] Pavel Ryzhakov, Riccardo Rossi, Sergio Idelsohn, and Eugenio Oñate. A monolithic lagrangian approach for fluid-structure interaction problems. *Computational Mechanics*, 46:883–899, 11 2010.

[34] Ruchao Shi and Yegao Qu. Numerical simulation of underwater explosion wave propagation in water–solid–air/water system using ghost fluid/solid method. *Journal of Fluids and Structures*, 90:354–378, 10 2019.

[35] Samuel Temkin. Elements of acoustics. *The Journal of the Acoustical Society of America*, 71(2):522–522, 1982.

[36] J.-M Vaassen, Pascal De Vincenzo, Charles Hirsch, and Benoit Leonard. Strong coupling algorithm to solve fluid-structure-interaction problems with a staggered approach. pages 117–, 08 2011.

[37] Chunwu Wang, Tiegang Liu, and Boo Khoo. A real ghost fluid method for the simulation of multimedium compressible flow. *SIAM J. Scientific Computing*, 28:278–302, 01 2006.

[38] Chenguang Yan, ZhiGuo Hao, Song Zhang, Baohui Zhang, and Tao Zheng. Numerical methods for the analysis of power transformer tank deformation and rupture due to internal arcing faults. *PLoS ONE*, 10, 07 2015.

[39] Xudong Zheng, Qian Xue, Rajat Mittal, and S Beilamowicz. A coupled sharp-interface immersed boundary-finite-element method for flow-structure interaction with application to human phonation. *Journal of biomechanical engineering*, 132:111003, 11 2010.