Abstract

Fluid-structure interaction is a typical example of a coupled problem involving two different domains, governed by different equations. Such problems are a classical challenge for the finite element method, and could not be properly solved until relatively recent times, when the required computational power became widely available.

The first half of this project deals with the existing techniques for the solution of the fluid-structure interaction problem, reviewing some common methods to tackle the problem. First, the numerical techniques used to solve each individual domain are briefly introduced, including the fractional step scheme, which allows the uncoupling of the pressure and the velocity in the the fluid domain. Then, the interaction problem is explained as the combination of the two domains, and segregated Dirichlet-Neumann schemes are introduced as a method to solve each domain separately.

Unfortunately, the Dirichlet-Neumann approach to the interaction problem has convergence problems in cases that involve a very flexible structure or a heavy fluid. It will be explained that the lack of convergence in those cases is a consequence of the uncoupling, which does not model the shared boundary between the two domains properly. To solve this problem, a correction will be introduced, simulating the effect of the structure over the fluid by approximating some terms that where dropped in the uncoupling process.

To conclude the first part of the document, a simple fluid-structure interface problem is solved and used as a benchmark to compare different numeric approaches to the problem.

The second half of the project has a more practical subject, as it deals with the pre-process of finite element problems in general and the fluid-structure interaction in particular. It analyzes the use of GiD as a pre-process module for solver applications developed within the Kratos framework. Both programs were developed to be as generic as possible, and therefore be able to tackle a greater number of different physical problems. Unfortunately, this means that the use of the first as a support tool for the other is inefficient and, as a consequence of the different design principles followed when writing each of them, somewhat counterintuitive. This process could be simplified by modifying the default GiD behaviour, but this has to be done separately for each different physical problem for which a Kratos-based application is developed. One of the objectives of this project is to design and implement a tool that can automate the process of writing the required configuration files and streamline the process of creating Kratos models using GiD, which has the double advantage of reducing the amount of work required to develop new Kratos applications and making the use of GiD as a pre-process tool for Kratos easier to learn.

Resum

El problema d'interacció entre fluid i estructura és un expemple típic de problema acoblat que involucra dos dominis diferents, regits per diferents equacions. Aquest tipus de problemes representen un repte clàssic per al mètode dels elements finits, i no han pogut ser resolts fins fa relativament poc temps, quan la potència computacional necessària ha estat disponible.

La primera meitat d'aquesta tesina introdueix les tècniques més habituals per tractar el problema d'interacció fluid-estructura. En primer lloc, es descriuen els plantejaments més coumns per resoldre cada un dels dominis per separat, incloent-hi els esquemes de *fractional step*, que permeten desacoblar la velocitat de la pressió en el domini fluid. A continuació, s'explica el problema d'interacció com la combinació dels dos dominis, i els esquemes segregregats Dirichlet-Neumann s'introdueixen com un mètode que permet resoldre els dos dominis de forma independent.

Desafortunadament, els mètodes basats en el desacoblament Dirichlet-Neumann presenten problemes de convergència en els casos que involucren una estructura molt flexible o un fluid pesant. Es demostrarà que, en aquests casos, la falta de convergència és el resultat de no haver tractat de forma adequada la interfase entre els dos dominis. Per resoldre aquest problema, es pot corregir el mètode habitual simulant una part de l'efecte de l'estructura soble el fluid que originalment es modelava amb uns termes menyspreats durant el procés de desacoblament.

Com a conclusió de la primera part de la tesina, s'ha resolt un problema senzill d'interacció fluid-estructura i s'ha utilitzat per comparar els resultats obtinguts mitjançant diferents mètodes.

La segona part de la tesina té un plantejament molt més pràctic, ja que tracta el preprocés dels problemes d'elements finits en general i dels d'interacció fluid-estructura en particular. Analitza l'ús de GiD com a mòdul de preprocés per a aplicacións desenvolupades en l'entorn Kratos. Tots dos programes han estat desenvolupats de la forma més genèrica possible, per tal de poder abordar un nombre més gran de problemes diferents. Desafortunadament, això fa que l'ús del primer com a eina de suport per al segon sigui poc eficient i fins i tot contraintuitiu per a l'usuari. Això es pot simplificar modificant el comportament del GiD, però cal fer-ho per separat per cada problema físic diferent que es vulgui tractar amb una aplicació basada en Kratos. Un dels objectius d'aquesta tesina és desenvolupar i implementar una eina que automatitzi el procés de configurar el GiD i agilitzi la creació de nous models automatitzant les tasques més pesades o menys intuitives. Això té una utilitat doble: redueix l'esforç que cal invertir en desenvolupar noves aplicacions basades en Kratos i fa més senzill l'ús del GiD com a mòdul de preprocés per Kratos.

A New Tool for the Practical Simulation of Fluid-Structure Interaction Problems

Jordi Cotela Dalmau

June 23, 2009

Contents

Introduction and objectives

Fluid-structure interaction defines a broad category of physical problems where the aim is to model the response of a flexible structure to the actions exerted by a surrounding fluid. It is a relatively complex problem, as it involves two entities controlled by different equations and, as such, it could only be tackled in relatively recent times, once the finite element techniques for each of the individual problems had reached a certain degree of maturity and the required computational power was easily available.

This project deals with one of the tools available to solve the fluid-structure interaction problem, the multiphysics finite element code Kratos (see [1]). As a platform to solve finite element problems in any field of physics, it is a natural tool to use in interaction problems, those that (typically) involve two or more different fields. A solver for fluid-structure interaction problems has already been implemented in Kratos, taking advantage of the existing code for both the structural dynamics and the incompressible fluid problems.

This last point introduces an important practical aspect of interaction problems. There is a background of research in each of the individual fields, treated as an independent problem, and the coupled problem should be able to benefit from those developments. To achieve this, the preferred approach when developing software that can solve interaction problems is to treat each one of the different domains as a black box, where a certain action is introduced and a response is returned. In this way, each domain's existing solving strategies can be used in the interaction problem, avoiding the need to develop a specific solution strategy for the coupled problem. In addition, the modular approach is interesting as the solving strategy for each domain can be adjusted or modified as needed, taking advantage of any advances in its field.

For large practical problems, the process of generating all the required data for the solver typically becomes a challenge, as it involves generating a finite element mesh writing all the required information for each individual node and element in a format that the solver can understand. For this reason, specific applications where developed to automate the so called pre-process of the problem.

One of such pre-process applications is GiD, which is the program of choice for the input of data for Kratos-based solvers. The need for this project appeared when it became clear that the interaction between GiD and Kratos was not as natural enough, as it required the end user to perform unintuitive tasks that could be easily automated. To do so, GiD has to be configured by providing it a series of files that contain the require information about Kratos, the format used to write its input, the physical problem that will be modeled (because GiD needs to know, for example, which boundary conditions can be applied to a specific

problem) and the tasks that should be automated. The aim of this project is not write the required configuration files for GiD, which is a mechanical task in itself, but to develop a tool that can automatically generate the required files to configure GiD to work together with any existing Kratos application and, hopefully, any new applications developed in the future.

Such application is desirable because the process of configuring GiD requires specific knowledge of some of its internal aspects, which is not required for a typicial GiD user. In addition, the configuration files have to be modified every time that new features are implemented for the associated Kratos solvers, which is a cumbersome process.

In the preceding lines, the two main point of interest of this project have been introduced: the fluid-structure interaction problem and the need of a relatively user-friendly method to define the large finite element models required for practical problems, in the fluid-structure interaction problem in particular, but also from a more generic point of view.

The project has two clearly differentiated parts, that deal with each of those points. The first one reviews the fluid-structure interaction problem and some techniques that can be used to solve it. It approaches the more theoretical aspects of the problem, which means that it involves mainly bibliographical research. The objectives set for this part of the project are:

- To review the finite element techniques used to solve each one of the domains involved in the coupled problem separately.
- To study how the single-field techniques can be combined in a coupled solver.
- To develop a simple application that can solve some example problems.

On the other hand, the second part of the project is eminently practical. To design a support tool for the solution of fluid-structure interaction problems, the following steps have been followed:

- To learn the basic aspects of Kratos, Python — the language Kratos uses to control the solution of each problem — and GiD.
- To analyze the process of using GiD to generate input for Kratos and find ways to simplify it, specially in the aspects where it is less intuitive.
- To design a program that can configure GiD automatically and, in addition, provide it the instructions required to implement the simplifications introduced in the last point.
- To debug the program and use it to solve real problems.

In addition of developing and testing the program, there is also the need to document it so that new users can use and expand it without the author's supervision. For this reason, a manual, which is included in this document as the appendix A, has been written.

Chapter 1

The fluid-structure interaction problem

Fluid-structure interaction problems describe the dynamic behaviour of systems that involve a flexible structural element and a fluid. This constitutes a coupled problem, as it is impossible to model the response of either of the two domains without knowing the behaviour of the other, and a considerable challenge for the available numerical techniques. This chapter introduces a basic theoretical background that will allow the solution of the problem, both in a monolithic fashion and by uncoupling the problem by means of a segregated Dirichlet-Neumann scheme.

1.1 State of the art

Over the last 40 years, the finite element method experimented an exponential growth which led to the definition of a set of reliable computational techniques for many different problems. This maturity, together with the availability of increasingly powerful computational resources gave rise in relatively recent times to an increasing interest in coupled problems.

One of such coupled problems is the simulation of the interaction between flexible structures and fluids, which represents one of the most active areas of research in the field of numerical methods. The problem was tackled over the years by a large number of authors working in many different fields of engineering. Due to its inherent importance for the aerospace sector, fluid-structure interaction technologies where developed for the simulation of aeroelastic problems [2], which where mostly addressed using compressible solvers. Different *loose coupling* algorithms where devised and their stability studied in application to both simple test problems and complex engineering structures. The importance of the so called geometric-conservation law was also pointed out and related to the accuracy of the energy conservation for the overall coupling [3], [4].

The application of such coupling methods to problems involving the interaction of structures with incompressible fluids was attempted by many different authors. It was soon recognized that *loose coupling* algorithms were highly unstable and this highlighted the need for *tightly coupled* solution schemes. The first results in this direction were obtained by simple iteration between the fields,

while convergence was guaranteed by introducing an underrrelaxation factor. Acceleration techniques were devised for the choice of "optimal" relaxation factors. The most widely known and successful of them is the Aitken accelerator, first proposed in [5] and published later in [6]. Alternative techniques to achieve the same results were proposed in [7]. One of the most important features for such coupling methods is its great modularity, which implied that different solver technologies were used as black-boxes. Standard projection schemes [8] were often used to exploit their great computational efficiency. As an example, the application of a fractional step scheme for the solution of coupled problems was presented in [9] both in a *loose coupling* fashion and within an iterative scheme to go towards a *strong coupling* approach.

It is only in very recent times that the fluid-structure interaction problem has started to be considered as a monolithic problem in which the fluid and the structure are presented as parts of a unique discretization. this approach presents clear advantages over previous presentations, as in particular it allows performing a mathematical analysis of the overall coupled process.

This concept gave rise to both monolithic techniques for the solution of the fluid-structure interaction problem and to semi-explicit schemes which are based on the application of the pressure splitting concept starting from the monolithic problem. Such tightly coupled semi-implicit schemes are first addressed in [10]. A further development of the idea is presented in [11]. Exploiting the unified formulation, some mathematical explanation for the so called mass effect was presented in [12]. The role played by the stabilization in the field was highlighted in a very clear way in [3]. The same thesis also presents a nice discussion on the importance of the Geometric Conservation Law depending on the choice of the conservative or non-conservative form in the implementation of the Navier-Stokes problem.

An alternative coupling paradigm based on the Robin-Robin condition is presented in [13], while a different formulation based on the use of a modified projection at the interface is presented in [14].

1.2 Coupled systems

Fluid-structure interaction constitutes a coupled problem. According to [15], coupled problems can be broadly defined as those that involve multiple domains and dependent variables which usually (but not always) describe different physical phenomena and in which

- neither domain can be solved while separated from the other;
- neither set of dependent variables can be eliminated at the differential equation level.

Two field coupled problems can be mathematically expressed through a ODE system in the form

$$\begin{pmatrix} \mathbf{M}_{xx} & \mathbf{M}_{xy} \\ \mathbf{M}_{yx} & \mathbf{M}_{yy} \end{pmatrix} \begin{pmatrix} \ddot{\mathbf{x}} \\ \ddot{\mathbf{y}} \end{pmatrix} + \begin{pmatrix} \mathbf{C}_{xx} & \mathbf{C}_{xy} \\ \mathbf{C}_{yx} & \mathbf{C}_{yy} \end{pmatrix} \begin{pmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{y}} \end{pmatrix} + \begin{pmatrix} \mathbf{K}_{xx} & \mathbf{K}_{xy} \\ \mathbf{K}_{yx} & \mathbf{K}_{yy} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_x \\ \mathbf{f}_y \end{pmatrix} \quad (1.1)$$

where $\mathbf{x}$ and $\mathbf{y}$ represent the state variables for each different field and the matrix terms appear as a consequence of a spatial discretization.

The terms of the form $\mathbf{A}_{xy}$ and $\mathbf{A}_{yx}$ in equation (1.1) express the interaction or *coupling* between the two fields. In this sense, that equation represents a two-way coupling, in which each field depends on the other. The alternative, one-way coupling, is found when only one of the fields depends on the other. Mathematically, this happens when either all of the $\mathbf{A}_{xy}$ or all of the $\mathbf{A}_{yx}$ terms are zero. Finally, if both the $\mathbf{A}_{xy}$ and $\mathbf{A}_{yx}$ terms are zero, the system is uncoupled.

Assuming that all terms in equation (1.1) are explicitly known, the ODE system that it describes can be directly discretized in time, leading to the so called *monolithic* formulation of the coupled problem. Note that, in doing so, different time discretization schemes can be used for each domain if it proves convenient. The disadvantage of the monolithic solution, however, is that the resulting system is generally very large, as all the variables of the system must be solved at the same time, and the system is often badly conditioned due to the existence of terms derived from completely different physical problems, whose values can be very different.

Given this observation, it is often appealing to split the coupled system and solve it by advancing in time separately each of the fields, which are made to interact at given steps of the procedure. This splitting can be done either starting from the discrete monolithic system, after the time discretization is performed (algebraic partitioning) or by splitting the differential system before the time discretization is introduced (differential splitting).

In general, different splitting techniques do not lead to equivalent formulations and can have a significant impact on the properties of the overall partitioned method. In any case, they will ultimately result in an alternative formulation of the monolithic problem of the type

$$\mathbf{H}^1\mathbf{u}_{n+1} = \mathbf{g}_{n+1} - \mathbf{H}^2\mathbf{u}^P_{n+1} \quad ; \quad \mathbf{u}_{n+1} = \begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} \tag{1.2}$$

where $\mathbf{H}^1 + \mathbf{H}^2 = \mathbf{H}^{monolithic}$ and $\mathbf{u}^P_{n+1}$ is a prediction.

The feasibility of such an approach, however, relies on the assumption that the system is linear and all its coefficients are known, which is not true in many cases and, in particular in the interaction between a flexible structure and an incompressible fluid (modeled using the Navier-Stokes equations).

The assumption of linearity simply does not hold because of the mathematical structure of the problem. The later point, however, deserves a more detailed explanation. As a non-linear system, the fluid-structure interaction generally involves solving a system of equations such as

$$\mathbf{f}^s(\mathbf{x}^\mathbf{s}, \mathbf{x}^\mathbf{f}) = \mathbf{0}$$
$$\mathbf{f}^f(\mathbf{x}^\mathbf{s}, \mathbf{x}^\mathbf{f}) = \mathbf{0}$$

which are linearized using Newton-Raphson as

$$\begin{pmatrix} \frac{\partial \mathbf{f}^s}{\partial \mathbf{x}^s} & \frac{\partial \mathbf{f}^s}{\partial \mathbf{x}^f} \\ \frac{\partial \mathbf{f}^f}{\partial \mathbf{x}^s} & \frac{\partial \mathbf{f}^f}{\partial \mathbf{x}^f} \end{pmatrix} \begin{pmatrix} \mathbf{x}^s \\ \mathbf{x}^f \end{pmatrix} = - \begin{pmatrix} \mathbf{f}^s(\mathbf{x}^\mathbf{s}, \mathbf{x}^\mathbf{f}) \\ \mathbf{f}^f(\mathbf{x}^\mathbf{s}, \mathbf{x}^\mathbf{f}) \end{pmatrix}$$

where, as expected, the off-diagonal terms of the tangent matrix express the dependency of one field on the other.

The solution of the fluid domain is typically not done in an explicit way, which means that the fluid terms of the tangent matrix, and $\mathbf{f}^f(\mathbf{x}^\mathbf{s}, \mathbf{x}^\mathbf{f})$ in particular, are unknown. The present approach to the solution of the coupled system would involve reformulating the solution of the fluid domain, which is not desirable. It is generally preferred to solve the problem combining existing techniques for each domain, which avoids the need to develop new solution methods specific to the coupled problem.

The following pages will introduce a way to combine standard solution methods for both the structure and the fluid domains into an algorithm that can be used to solve the fluid-structure interaction problem.

1.3 Basic equations

The starting point for both the fluid domain and the structural domain is the dynamic equilibrium equation, expressed in differential form as

$$\rho \frac{D\mathbf{v}}{Dt} - \nabla \cdot \sigma = \mathbf{f}_{ext} \tag{1.3}$$

In flow problems, this equation can be rewritten to take into account the deformation of the finite element mesh used to discretize the domain. This can be done using an Arbitrary Lagrangian-Eulerian (ALE) approach (see [4]), where the mesh can move independently of the fluid. This results in the equation

$$\rho \frac{\partial \mathbf{v}}{\partial t} + \rho \left(\mathbf{v} - \mathbf{v}_M\right) \cdot \nabla \mathbf{v} - \nabla \cdot \sigma = \mathbf{f}_{ext} \tag{1.4}$$

where $\mathbf{v}_M$ represents the velocity of the nodes of the mesh, as opposed to the velocity of the fluid particles that occupy the same point of space in any given time step. Note that, while taking $\mathbf{v}_M = \mathbf{v}$ or $\mathbf{v}_M = \mathbf{0}$ respectively return the Lagrangian and Eulerian descriptions of the fluid, other values can be uses for intermediate cases.

A different approach is normally used to describe the structure and the fluid. The Lagrangian description is used for the structural domain, as the displacement of the particles is relatively small and a mesh that moves with them won't be greatly deformed. For the fluid domain the Eulerian description is preferred, as the fluid particles tend to move considerably from their initial position and a mesh that follows them would end up greatly deformed.

Note that equation (1.4) holds for both the fluid and structural domains, as no constitutive equation has been introduced yet.

For most fluid-structure interaction problems, the fluid can be assumed Newtonian and incompressible. In this case, the stress tensor σ can be split into its isochoric and deviatoric components

$$\sigma = \sigma_\tau - p\mathbf{I} \tag{1.5}$$

where

$$p = \frac{1}{3} \operatorname{Tr}(\sigma) \tag{1.6}$$

$$\sigma_\tau = 2\mu \nabla^s \mathbf{v} \tag{1.7}$$

with

$$\nabla^s \mathbf{v} = \frac{1}{2}\left(\nabla \mathbf{v} + \nabla \mathbf{v}^T\right)$$

The set of equations to be solved for the incompressible domain becomes

$$\rho \frac{D\mathbf{v}}{Dt} - \nabla \cdot \sigma_\tau + \nabla p = \mathbf{f}_{ext} \tag{1.8}$$

$$\nabla \cdot \mathbf{v} = 0 \tag{1.9}$$

Where equation (1.9), derived from the mass conservation equation, ensures the incompressibility of the fluid. This system of equations is well posed in the continuum but leads to difficulties once its discretized unless great care is taken in choosing the pressure and velocity spaces (see for example [4]). This difficulty can be circumvented by modifying equation (1.9) to include a pressure stabilization term. The idea is to slightly modify the incompressibility condition just enough (given a time step and element size) to allow for a numerical solution. There are different ways to implement this stabilization, which will be addressed later.

Equations (1.8) and (1.9) can be discretized, resulting in

$$\rho^f \mathbf{M}^f \frac{\partial \mathbf{v}}{\partial t} + \mathbf{K}^* \left(\mathbf{v} - \mathbf{v}_M\right) \mathbf{v} + \mathbf{G}\mathbf{p} = \mathbf{f}^f_{ext} \tag{1.10}$$

$$\mathbf{D}\mathbf{v} + \frac{\tau}{\rho^f}\mathbf{S}\mathbf{p} = \mathbf{0} \tag{1.11}$$

where the term $\frac{\tau}{\rho}\mathbf{S}$ represents the chosen stabilization scheme, τ being a scalar stabilization factor. $\mathbf{v}$ and $\mathbf{p}$ represent the velocity and pressure values at a given time step. The various matrices that appear in equations (1.10) and (1.11) follow a standard notation used, for example, in [4] or [16]. They are obtained by the Galerkin discretization of the involved differential operators

$$\begin{gathered}
\nabla \rightarrow \mathbf{G} \quad \text{Gradient operator} \\
(\nabla \cdot) \rightarrow \mathbf{D} = -\mathbf{G}^T \quad \text{Divergence operator} \\
(\mathbf{v} - \mathbf{v}_M) \cdot \nabla \rightarrow \mathbf{K}\left(\mathbf{v} - \mathbf{v}_M\right) \\
(-\nabla \cdot \sigma_\tau) \rightarrow \mathbf{K}_\mu \cdot \mathbf{v} \quad \text{viscous term}
\end{gathered}$$

The convection part requires an additional stabilization term, which can be grouped with the last two using

$$\mathbf{K}^* := \mathbf{K}\left(\mathbf{v} - \mathbf{v}_M\right) + \mathbf{K}_\mu + \mathbf{S}_{conv}$$

The expression of these matrices depends on the type of spatial discretization used. In chapter 2, they will be particularised for a specific unidimensional example.

On the other hand, given a constitutive law and a spatial discretization, the discrete equation that controls the behaviour of the structural domain can be obtained by the Galerkin discretization of equation (1.3) as

$$\rho^s \mathbf{M}^s \mathbf{a} + \mathbf{C}^s \mathbf{v} + \mathbf{K}^s\left(\mathbf{u}\right)\mathbf{u} = \mathbf{f}^s_{ext} \tag{1.12}$$

where $\mathbf{u}$ is the vector of nodal displacements, $\mathbf{K}^s\left(\mathbf{u}\right)\mathbf{u}$ is the discrete form of the term $\nabla \cdot \sigma$ and $\mathbf{C}^s$ is introduced to account for viscous dampening.

The equations obtained until now allow us to define the fluid-structure interaction problem as a set of non-linear equations, using the velocity and pressure of the nodes as variables. In order to solve this system, it will prove convenient to rewrite the involved equations in residual form, yielding

$$\mathbf{r}^s(\mathbf{v}^s) = \mathbf{f}^s_{ext} - \rho^s \mathbf{M}^s \frac{\partial \mathbf{v}^s}{\partial t} - \mathbf{C}^s \mathbf{v}^s - \mathbf{K}^s\left(\mathbf{u}^s\right)\mathbf{u}^s \tag{1.13}$$

for the structural domain and

$$\mathbf{r}^f(\mathbf{v}^f, \mathbf{p}^f) = \mathbf{f}^f_{ext} - \rho^f \mathbf{M}^f \frac{\partial \mathbf{v}}{\partial t} - \mathbf{K}^*\left(\mathbf{v}^f - \mathbf{v}_M\right)\mathbf{v}^f - \mathbf{G}\mathbf{p}^f \tag{1.14}$$

$$\psi^f(\mathbf{v}^f, \mathbf{p}^f) = \mathbf{D}\mathbf{v}^f + \frac{\tau}{\rho}\mathbf{S}\mathbf{p}^f \tag{1.15}$$

for the fluid domain. Note that this can be done because both the displacement and the acceleration can be expressed as functions of the velocity.

1.4 Numerical tools

Before discussing the combination of the structure and the fluid domains in a single problem, the numerical tools required to solve it will be briefly discussed. The two main points of interest that will be explained here are the solution of non-linear systems of equations and the time stepping scheme used.

1.4.1 Time integration

A Newmark method is used to integrate the solution over time. The Newmark family comprises some of the most popular time integration schemes in structural dynamics (see for example [17]). Newmark methods are based on the finite difference formulas

$$x_{n+1} = x_n + \Delta t\, v_n + \frac{\Delta t^2}{2} a_n + \beta\, \Delta t^3 \left(\frac{a_{n+1} - a_n}{\Delta t}\right) \tag{1.16}$$

$$v_{n+1} = v_n + \Delta t\, a_n + \gamma\, \Delta t^2 \left(\frac{a_{n+1} - a_n}{\Delta t}\right) \tag{1.17}$$

In most structural dynamics problems, the main variable is the displacement of the structure nodes, and the two equations are rewritten as

$$v_{n+1} = (x_{n+1} - x_n) - \left(\frac{\gamma}{\beta} - 1\right) v_n - \frac{\Delta t}{2}\left(\frac{\gamma}{\beta} - 2\right) a_n$$

$$a_{n+1} = \frac{1}{\beta\,(\Delta t)^2}\,(x_{n+1} - x_n) - \left(\frac{1}{\beta\,\Delta t}\right) v_n - \left(\frac{1}{2\beta} - 1\right) a_n$$

On the other hand, in the fluid domain the velocity is the main variable. As it is convenient to formulate the equations for both domains using the same variables, the last two equations will be rewritten as

$$x_{n+1} = x_n + \Delta t \left(\frac{\beta}{\gamma} v_{n+1} + \left(1 - \frac{\beta}{\gamma}\right) v_n\right) + \frac{\Delta t^2}{2}\left(1 - \frac{2\beta}{\gamma}\right) a_n \tag{1.18}$$

$$a_{n+1} = \frac{1}{\gamma\,\Delta t}\,(v_{n+1} - v_n) + \frac{\gamma - 1}{\gamma} a_n \tag{1.19}$$

Newmark methods are unconditionally stable if β and γ are chosen to verify the inequality

$$2\beta \geq \gamma \geq \frac{1}{2}$$

and will behave as a second order method if $\gamma = 1/2$.

For the examples implemented in this project, the the average acceleration method or trapezoidal rule has been used. It is one of the most popular methods of the Newmark family, characterized by taking $\beta = 1/4$ and $\gamma = 1/2$.

1.4.2 The Newton-Raphson method

The discrete form of the continuity of linear momentum, for both the structural and the fluid formulations (equations (1.13) and (1.14), respectively), constitutes a system of non-linear equations. To solve them, the Newton-Raphson method can be used.

Given a non-linear system of equations with the general form $\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}(\mathbf{x})$, expressed in residual form as $\mathbf{f}(\mathbf{x}) := \mathbf{A}(\mathbf{x})\mathbf{x} - \mathbf{b}(\mathbf{x}) = \mathbf{0}$, a solution can be found by correcting successive approximations until convergence is reached

$$\mathbf{f}(\mathbf{x}^k) \neq \mathbf{0} \quad \rightarrow \quad \mathbf{x}^{k+1} = \mathbf{x}^k + \Delta\mathbf{x}^{k+1}$$

For the Newton-Raphson method, the value of the correction term, $\Delta\mathbf{x}^{k+1}$, is determined by means of a first-order Taylor's series expansion:

$$\mathbf{0} = \mathbf{f}(\mathbf{x}^{k+1}) \simeq \mathbf{x}^{\mathbf{k}} + \frac{\partial \mathbf{f}(\mathbf{x}^k)}{\partial \mathbf{x}^k}\,\Delta\mathbf{x}^{k+1}$$

This expression involves the Jacobian matrix:

$$\mathbf{J}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}) := \begin{pmatrix} \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1}(\mathbf{x}) & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_2}(\mathbf{x}) & \cdots & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_n}(\mathbf{x}) \\ \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_1}(\mathbf{x}) & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_2}(\mathbf{x}) & \cdots & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_n}(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{f}_n}{\partial \mathbf{x}_1}(\mathbf{x}) & \frac{\partial \mathbf{f}_n}{\partial \mathbf{x}_2}(\mathbf{x}) & \cdots & \frac{\partial \mathbf{f}_n}{\partial \mathbf{x}_n}(\mathbf{x}) \end{pmatrix}$$

Using this definition, the iteration function of the Newton-Raphson method can be expressed as

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \mathbf{J}(\mathbf{x}^k)^{-1}\,\mathbf{f}(\mathbf{x}^k)$$

Obviously, the inverse of the Jacobian matrix is never computed. In practice, the Newton-Raphson method is implemented as:

1. Choose an initial approximation $\mathbf{x}^0$.

2. Solve the linear system

$$\mathbf{J}(\mathbf{x}^k)\Delta\mathbf{x}^{k+1} = -\mathbf{f}(\mathbf{x}^k)$$

3. Update the value of the unknown variables with $\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta\mathbf{x}^{k+1}$.

4. Check convergence. If it was not achieved, go back to step 2.

By using this method, the zero of the non-linear system can be determined by iteratively solving a linearized system.

The main advantage of Newton-Raphson when compared to other available methods is that it has quadratic convergence. Unfortunately, it also has some drawbacks. The most important of them are related to the use of the Jacobian matrix as it requires the evaluation of the derivatives of $\mathbf{f}(\mathbf{x})$ at each iteration, which increases the computational cost. In addition, there is no guarantee that the Jacobian matrix is not singular for some values of $\mathbf{x}$. Finally, for non-linear systems in the form of $\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}(\mathbf{x})$, the structure of $\mathbf{A}(\mathbf{x})$, which could allow the use of a faster specific linear solving method in step 2, is typically lost when computing $\mathbf{J}(\mathbf{x})$. It should be noted, however, that for the fluid-structure interaction problem, the structure of $\mathbf{A}(\mathbf{x})$ can be preserved by making certain approximations as long as the external forces remain constant.

There are some important remarks to be made about the application of the Newton-Raphson method to the fluid-structure interaction problem. For now, the fluid and the structure will be treated independently, as the combination of the two domains will be discussed in the following section.

In the structural domain, the only variables are the nodal velocities. The Jacobian matrix can be obtained from equation (1.13) and, for a given time step, it will take the form

$$\left(\frac{\partial \mathbf{r}^k}{\partial \mathbf{v}^k}\right) = \frac{\partial}{\partial \mathbf{v}^k}\left(\mathbf{f}^{\,k} - \rho^s \mathbf{M}^s \mathbf{a}^k - \mathbf{C}^s \mathbf{v}^k - \mathbf{K}^s \mathbf{u}^k\right)$$

At this point, it is commonly assumed that the matrices are approximately constant, resulting in

$$\left(\frac{\partial \mathbf{r}^k}{\partial \mathbf{v}^k}\right) = \frac{\partial \mathbf{f}^{\,k}}{\partial \mathbf{v}^k} - \rho^s \mathbf{M}^s \frac{\partial \mathbf{a}^{\,k}}{\partial \mathbf{v}^k} - \mathbf{C}^s \frac{\partial \mathbf{v}^{\,k}}{\partial \mathbf{v}^k} - \mathbf{K}^s \frac{\partial \mathbf{u}^{\,k}}{\partial \mathbf{v}^k}$$

Besides the term $\frac{\partial \mathbf{f}^{\,k}}{\partial \mathbf{v}^k}$, which depends on the actions in each problem, the three remaining derivatives are dependent on the time stepping method used. For the Newmark method, they take the values

$$\frac{\partial \mathbf{a}^{\,k}}{\partial \mathbf{v}^k} = \frac{1}{\gamma\,\Delta t}\,\mathbf{I} \tag{1.20}$$

$$\frac{\partial \mathbf{v}^{\,k}}{\partial \mathbf{v}^k} = \mathbf{I} \tag{1.21}$$

$$\frac{\partial \mathbf{x}^{\,k}}{\partial \mathbf{v}^k} = \frac{\beta\,\Delta t}{\gamma}\,\mathbf{I} \tag{1.22}$$

where $\mathbf{I}$ is the identity matrix. A similar reasoning can be applied for the fluid domain, where equations (1.20–1.22) still hold, seeing how they are obtained directly from the time stepping method.

1.5 Fractional step schemes

An important remark about equations (1.14) and (1.15) is that the pressure and the velocity values are coupled in the fluid domain. This is not a problem in itself as, by choosing a suitable linearization, both variables could be solved at

the same time by recursively solving a linear system that could be symbolically expressed as

$$\begin{pmatrix} -\frac{\partial \mathbf{r}^f}{\partial \mathbf{vp}} \\ -\frac{\partial \psi^f}{\partial \mathbf{vp}} \end{pmatrix} \begin{pmatrix} d\mathbf{v} \\ d\mathbf{p} \end{pmatrix} = \begin{pmatrix} \mathbf{r}^f(\mathbf{v}, \mathbf{p}) \\ \psi^f(\mathbf{v}, \mathbf{p}) \end{pmatrix}$$

Unfortunately, this approach generally results in numerical problems once a solution is attempted. This is easy to understand, considering that such a system of equations involves variables that are generally scaled in a very different way. In practice, it is convenient to uncouple the velocity and the pressure.

Fractional step schemes, analyzed in [16], are widely used in the solution of the incompressible Navier-Stokes equation, and will prove useful to solve the fluid domain of the fluid-structure interaction problem. The acceleration in equation (1.8) is approximated as a function of the velocity for each time step, using equation 1.19 if the Newmark scheme is used or an equivalent for other time integration methods. For the remainder of this section, the Newmark method will be assumed, although the basic reasoning ia equally aplicable to any other scheme. This allows us to write equations (1.10) and (1.11) for a given time step as

$$\rho^f \mathbf{M}^f \left(\frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\gamma\, \Delta t} + \frac{\gamma - 1}{\gamma}\, a^n \right) + \mathbf{K}^* \left(\mathbf{v}^{n+\theta} \right) \mathbf{v}^{n+\theta} + \mathbf{G}\mathbf{p}^{n+1} = \mathbf{f}^{\,n+\theta} \tag{1.23}$$

$$\mathbf{D}\mathbf{v}^{n+1} + \frac{\tau}{\rho^f} \mathbf{S}\mathbf{p}^{n+1} = \mathbf{0} \tag{1.24}$$

Where θ is chosen according to the time stepping method. Typical values are $\theta = 1$ for Backwards Euler and $\theta = 1/2$ for the second-order Crank-Nicholson scheme.

Fractional step methods are based on introducing at this point an auxiliary variable $\hat{\mathbf{v}}^{n+1}$ that represents an intermediate value of the velocity:

$$\rho^f \mathbf{M}^f \left(\frac{\hat{\mathbf{v}}^{n+1} - \mathbf{v}^n}{\gamma\, \Delta t} + \frac{\gamma - 1}{\gamma}\, a^n \right) + \mathbf{K}^* \left(\mathbf{v}^{n+\theta} \right) \mathbf{v}^{n+\theta} + \mathbf{G}\mathbf{p}^{n} = \mathbf{f}^{\,n+\theta} \tag{1.25}$$

$$\rho^f \mathbf{M}^f \left(\frac{\mathbf{v}^{n+1} - \hat{\mathbf{v}}^{n+1}}{\gamma\, \Delta t} \right) + \mathbf{G} \left(\mathbf{p}^{n+1} - \mathbf{p}^n \right) = \mathbf{0} \tag{1.26}$$

$$\mathbf{D}\mathbf{v}^{n+1} + \frac{\tau}{\rho^f} \mathbf{S}\mathbf{p}^{n+1} = \mathbf{0} \tag{1.27}$$

At this point, the approximation

$$\mathbf{K}^* \left(\hat{\mathbf{v}}^{n+\theta} \right) \hat{\mathbf{v}}^{n+\theta} \approx \mathbf{K}^* \left(\mathbf{v}^{n+\theta} \right) \mathbf{v}^{n+\theta} \tag{1.28}$$

where $\hat{\mathbf{v}}^{n+\theta} := \theta \hat{\mathbf{v}}^{n+1} + (1 - \theta)\mathbf{v}^n$, will be assumed. By reorganizing the terms in equation (1.26), $\mathbf{v}^{n+1}$ can be expressed as a function of $\hat{\mathbf{v}}^{n+1}$ and inserted in (1.27), obtaining the following set of equations

$$\rho^f \mathbf{M}^f \left(\frac{\hat{\mathbf{v}}^{n+1} - \mathbf{v}^n}{\gamma\, \Delta t} + \frac{\gamma - 1}{\gamma}\, a^n \right) + \mathbf{K}^* \left(\hat{\mathbf{v}}^{n+\theta} \right) \hat{\mathbf{v}}^{n+\theta} + \mathbf{G}\mathbf{p}^{n} = \mathbf{f}^{n+\theta} \tag{1.29}$$

$$\frac{\gamma\, dt}{\rho}\, \mathbf{D} \left(\mathbf{M}^f \right)^{-1} \mathbf{G} \Delta \mathbf{p}^{n+1} = \mathbf{D}\hat{\mathbf{v}}^{n+1} \tag{1.30}$$

$$\rho^f \mathbf{M}^f \left(\frac{\mathbf{v}^{n+1} - \hat{\mathbf{v}}^{n+1}}{\gamma\, \Delta t} \right) + \mathbf{G} \Delta \mathbf{p}^{n+1} = \mathbf{0} \tag{1.31}$$

where the notation $\Delta\mathbf{p}^{n+1} := \left(\mathbf{p}^{n+1} - \mathbf{p}^{n}\right)$ has been introduced.

This set of equations can be solved in succession to obtain, in order, $\hat{\mathbf{v}}^{\mathbf{n+1}}$, $\Delta\mathbf{p}^{n+1}$ and $\mathbf{v}^{n+1}$. Thanks to the introduction of the fractional step velocity $\hat{\mathbf{v}}^{n+1}$ and the approximation made in (1.28), the pressure and the velocity can be obtained while avoiding the problems associated to the coupled solution.

The inverse of the mass matrix $\mathbf{M}^f$ appears in equation (1.30), but it is never calculated in practice. A common approach to this situation is to approximate the term $\mathbf{D}\left(\mathbf{M}^f\right)^{-1}\mathbf{G}$ by the Laplacian matrix, $\mathbf{L}$, which is obtained from the Galerkin discretization of the Laplacian operator. Unfortunately, this approach has a serious drawback. While equation (1.30) constitutes a well posed linear system and can be solved as is, if the $\mathbf{DM}^{-1}\mathbf{G}$ product is approximated by the Laplacian matrix the system requires additional boundary conditions.

Usually, this can be solved because the pressure in the Neumann boundary is known a priori. If an external force is applied over a region of the fluid boundary, the pressure will equilibrate it. However, in cases where the fluid has no Neumann boundary, such as the second example analyzed in chapter 2, there is no way to solve the approximated system. A more robust alternative to $\mathbf{L}$ is to calculate the $\mathbf{D}\left(\mathbf{M}^f\right)^{-1}\mathbf{G}$ using a diagonal mass matrix, which can be inverted easily. With this alternative, the system matrix is considerably more expensive to compute than $\mathbf{L}$, but it allows the solution of the problem when no pressure values can be found a priori.

For now, no assumptions will be made about the method used to compute $\mathbf{D}\left(\mathbf{M}^f\right)^{-1}\mathbf{G}$. In chapter 2, both alternatives will be implemented and compared.

With this in mind, the fluid domain can be solved using a segregated scheme in the form

1. Solve equation (1.29) for $\hat{\mathbf{v}}$.
2. Use $\hat{\mathbf{v}}$ and equation (1.30) to obtain $\Delta\mathbf{p}$.
3. Calculate the end of step velocity $\mathbf{v}$ using equation (1.31)

1.6 Pressure stabilization

It is known that the solution of the fluid domain can lead to numerical difficulties as a consequence of the incompressibility constraint imposed in the mass conservation equation (1.9). To avoid this problem, an additional stabilization term was introduced in equation (1.11). A typical implementation of the stability term is

$$\frac{\tau}{\rho}\mathbf{S} := \frac{\tau}{\rho}\mathbf{DM}^{-1}\mathbf{G}\mathbf{p}_{kn} - \frac{\tau}{\rho}\mathbf{Lp} \tag{1.32}$$

where $\mathbf{p}$ is the nodal pressure vector for the current iteration and $\mathbf{p}_{kn}$ is the last known value for the pressure, which, depending on the time stepping scheme, will be the value obtained in the previous time step (for pure fractional step methods) or the value of the pressure obtained in the last iteration for the current time step (for predictor-corrector schemes).

The addition of a stabilization term and the choice of its implementation is discussed in detail in [18]. An intuitive way of understanding it is taking into account that the expression $\mathbf{Dv} = \mathbf{0}$ means that the fluid is incompressible.

This incompressibility means that finding a solution will be more difficult, so the equation is modified to allow for a certain degree of compressibility in the solution. This is done by adding two terms, $\mathbf{Lp}$ and $\mathbf{DM}^{-1}\mathbf{Gp}_{kn}$, that are approximately equal to each other (remember that $\mathbf{Lp}$ is frequently used as an approximation to $\mathbf{DM}^{-1}\mathbf{Gp}$), which is enough to find a solution while ensuring that the added compressibility is minimal.

The stabilization term can be reformulated in terms of $\Delta\mathbf{p}^{n+1}$, yielding

$$\frac{\tau}{\rho}\,\mathbf{S} := \frac{\tau}{\rho}\left(\mathbf{DM}^{-1}\mathbf{G} - \mathbf{L}\right)\mathbf{p}_{kn} - \frac{\tau}{\rho}\,\mathbf{L}\Delta\mathbf{p}^{n+1} \tag{1.33}$$

It can then be inserted into equation (1.30), which means that the segregated scheme introduced by equations (1.29–1.31) can now be rewritten as

$$\rho\mathbf{M}\left(\frac{\hat{\mathbf{v}}^{n+1} - \mathbf{v}^{n}}{\gamma\,\Delta t} + \frac{\gamma - 1}{\gamma}\,a^{n}\right) + \mathbf{K}^{*}\left(\hat{\mathbf{v}}^{n+\theta}\right)\hat{\mathbf{v}}^{n+\theta} + \mathbf{G}\mathbf{p}^{n} = \mathbf{f}^{\,n+\theta} \tag{1.34}$$

$$\rho\mathbf{M}\boldsymbol{\Pi}^{n+1} = \mathbf{G}\mathbf{p}_{kn} \tag{1.35}$$

$$\left(\frac{\gamma\,\Delta t}{\rho}\,\mathbf{DM}^{-1}\mathbf{G} + \frac{\tau}{\rho}\,\mathbf{L}\right)\Delta\mathbf{p}^{n+1} = \mathbf{D}\hat{\mathbf{v}}^{n+1} + \frac{\tau}{\rho}\left(\mathbf{D}\boldsymbol{\Pi}^{n+1} - \mathbf{L}\mathbf{p}_{kn}\right) \tag{1.36}$$

$$\rho\mathbf{M}\left(\frac{\mathbf{v}^{n+1} - \hat{\mathbf{v}}^{n+1}}{\gamma\,dt}\right) + \mathbf{G}\Delta\mathbf{p}^{n+1} = \mathbf{0} \tag{1.37}$$

where $\boldsymbol{\Pi}^{n+1}$ is an auxiliary vector used to avoid finding $\mathbf{M}^{-1}$.

1.7 Fluid-structure interaction

The theory introduced up to this point provides a description of the fluid and the structure. The next step is combining the two domains in order to solve them as a single problem. The critical element of this union is the interface between the two. There, two conditions must hold:

- equilibrium of forces
- continuity of displacements

The entire problem domain must be described using a single finite element discretization which includes the two coupled parts. As different equations hold over the fluid and the structure, the finite elements must be defined entirely inside one of them (no element can be half structure and half fluid), which means that there will be a series of nodes placed exactly in the interface.

On the structural domain, where the Lagrangian description is usually employed, the displacements of the mesh will coincide with the displacements of the structure particles. This is not the case in the fluid domain, where the mesh does not move (if an Eulerian description is used) or moves independently of the fluid particles (using an ALE approach). To guarantee the continuity of displacements, the displacements of the interface must be imposed to the fluid domain, and the velocity of both the particles and the mesh modified accordingly.

The equilibrium of forces, on the other hand, will hold automatically at the end of each time step, as a result of imposing the conservation of linear momentum (1.3) in the entire domain.

As mentioned, there are two sets of independent variables for this problem, the pressure and the velocity. The pressure is only relevant in the fluid domain, and it can be safely ignored for the structure nodes. The velocity, on the other hand, is relevant on both domains, but it is obtained from different formulations due to the different constitutive equations and material descriptions used in both domains . This is specially relevant for the nodes of the interface which, during the finite element assembly, will receive contributions from both the structural and the fluid domains. To identify the nodes where the different equations apply, the following notation will be used

$$\mathbf{r} := \begin{pmatrix} \mathbf{r}^s \\ \mathbf{r}^{fs} \\ \mathbf{r}^f \end{pmatrix}$$

where $\mathbf{r}^s$ is obtained from equation (1.13), while $\mathbf{r}^f$ is obtained from equation (1.14). The interface term, $\mathbf{r}^{fs}$, receives contributions from both domains. This can be expressed as

$$\mathbf{r} := \begin{pmatrix} \mathbf{r}^s \\ \mathbf{r}^{fs}_s \\ \mathbf{0} \end{pmatrix} + \begin{pmatrix} \mathbf{0} \\ \mathbf{r}^{fs}_f \\ \mathbf{r}^f \end{pmatrix}$$

1.7.1 Monolithic linear momentum equation

This section introduces the monolithic linear momentum equation as an intermediate step between the monolithic problem and the uncoupled solution. To do it, the segregated scheme introduced in section 1.5 is extended to include the structural domain.

At this point, it is useful to rewrite all the involved equations in residual form. For the fluid part, assuming Newmark time stepping and $\theta = 1$, equations (1.34–1.37) become

$$\mathbf{r}^{n+1}_{f1} = \mathbf{f}^{n+1} - \rho^f \mathbf{M}^f \hat{\mathbf{a}}^{n+1} - \mathbf{K}^* \hat{\mathbf{v}}^{n+1} - \mathbf{G}\mathbf{p}^n \tag{1.38}$$

$$\rho^f \mathbf{M}^f \mathbf{\Pi}^{n+1} = \mathbf{G}\mathbf{p}_{kn} \tag{1.39}$$

$$\mathbf{r}^{n+1}_{f2} = \mathbf{D}\hat{\mathbf{v}}^{n+1} - \left(\frac{\gamma\,\Delta t}{\rho^f} \mathbf{D}\mathbf{M}^{f\,-1}\mathbf{G} + \frac{\tau}{\rho^f}\mathbf{L} \right) \Delta\mathbf{p}^{n+1} + \frac{\tau}{\rho^f}\left(\mathbf{D}\mathbf{\Pi}^{n+1} - \mathbf{L}\mathbf{p}_{kn} \right) \tag{1.40}$$

$$\mathbf{r}^{n+1}_{f3} = -\rho^f \mathbf{M}^f \left(\frac{\mathbf{v}^{n+1} - \hat{\mathbf{v}}^{n+1}}{\gamma\,\Delta t} \right) - \mathbf{G}\Delta\mathbf{p}^{n+1} \tag{1.41}$$

An interesting point is that, as the pressure is not defined for the structure, it can be assumed that $\Delta\mathbf{p}^{n+1}_s = \mathbf{0}$ in that domain and use it to affirm that $\mathbf{v}^{n+1}_s = \hat{\mathbf{v}}^{n+1}_s$. Note that this is not true in the interface nodes, which can lead to convergence problems. This observation will be discussed in detail in section 1.7.3 but, for now, it allows to rewrite equation (1.13) using the same fractional step scheme as the fluid:

$$\mathbf{r}^{n+1}_{s1} = \mathbf{f}^{n+1} - \rho^s \mathbf{M}^s \hat{\mathbf{a}}^{n+1} - \mathbf{C}^s \hat{\mathbf{v}}^{n+1} - \mathbf{K}^s \hat{\mathbf{u}}^{n+1} \tag{1.42}$$

$$\mathbf{r}^{n+1}_{s3} = -\rho^s \mathbf{M}^s \left(\frac{\mathbf{v}^{n+1} - \hat{\mathbf{v}}^{n+1}}{\gamma\,\Delta t} \right) \tag{1.43}$$

Again, the Newmark method has been assumed for time integration. Obviously, the solution of equation (1.43) is trivial, but it will be useful to write the complete coupled formulation. The terms $\hat{\mathbf{u}}^{n+1}$ and $\hat{\mathbf{a}}^{n+1}$ are introduced here for convenience. This notation indicates the "fractional step" values of displacement and velocity, obtained by using $\hat{\mathbf{v}}^{n+1}$ as the new value of velocity in the Newmark scheme

$$\hat{\mathbf{u}}^{n+1} = \left(\mathbf{u}^n + \Delta t \left(\frac{\beta}{\gamma} \hat{\mathbf{v}}^{n+1} + \left(1 - \frac{\beta}{\gamma} \right) \mathbf{v}^n \right) + \frac{\Delta t^2}{2} \left(1 - \frac{2\beta}{\gamma} \right) \mathbf{a}^n \right) \tag{1.44}$$

$$\hat{\mathbf{a}}^{n+1} = \left(\frac{1}{\gamma \, \Delta t} \left(\hat{\mathbf{v}}^{n+1} - \mathbf{v}^n \right) + \frac{\gamma - 1}{\gamma} \mathbf{a}^n \right) \tag{1.45}$$

From now on, the involved matrices will be arbitrarily divided in blocks to identify the rows and columns corresponding to the interface terms. In addition, the external force will be assumed constant[1].

The fluid-structure interaction problem can be solved by using equations (1.38–1.43). This process will be discussed here and an implementation of it will be tested in section 2, although it has limited practical interest. The solution it provides is the most exact of the various methods implemented in chapter 2, as the uncoupling of the structure and the fluid relies on additional assumptions. However, it is rarely used in practice because of its computational cost, as the matrices of the linear systems that have to be solved are larger that the ones that appear in uncoupled schemes and, more importantly, poorly scaled, as they combine terms obtained from different physical equations. Iterative methods, for their lower computational cost when compared to direct methods, are the only viable choice for large problems. Uncoupled methods, in which all equations are solved over a single domain, are much better scaled, which means that their solution is cheaper from a computational point of view.

The first step of this scheme involves solving the assembly of equation (1.38) for the fluid domain and equation (1.42) for the structural domain. Using Newton-Raphson:

$$\mathbf{A} := \left(\tfrac{\rho^s}{\gamma \, \Delta t} \mathbf{M}^s + \mathbf{C}^s + \tfrac{\beta \, \Delta t}{\gamma} \mathbf{K}^s \right)$$

$$\mathbf{B} := \left(\tfrac{\rho^f}{\gamma \, \Delta t} \mathbf{M}^f + \mathbf{K}^* \right)$$

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} + \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{0} & \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}_i \begin{pmatrix} \Delta \hat{\mathbf{v}}^s \\ \Delta \hat{\mathbf{v}}^{fs} \\ \Delta \hat{\mathbf{v}}^f \end{pmatrix}_{i+1} = \begin{pmatrix} \mathbf{r}^s_{s1} \\ \mathbf{r}^{fs}_{s1} + \mathbf{r}^{fs}_{f1} \\ \mathbf{r}^f_{f1} \end{pmatrix}_i \tag{1.46}$$

This system can be solved iteratively to find the fractional step value for the coupled problem. The next step is updating the pressure, which involves solving a lineal system. This can be done using the fluid domain only, as the pressure is not defined in the structural domain, but for the coupled problem

[1] If this wasn't the case, its derivate should be taken into account to build the Newton-Raphson Jacobian matrices.

we can symbolically write

$$\mathbf{F} := \frac{\gamma\,\Delta t}{\rho^f}\,\mathbf{D}(\mathbf{M}^f)^{-1}\mathbf{G} + \frac{\tau}{\rho^f}\,\mathbf{L}$$
$$\mathbf{b} := \hat{\mathbf{v}} + \frac{\tau}{\rho^f}\,(\mathbf{D}\boldsymbol{\Pi} - \mathbf{L}\mathbf{p}_{kn})$$
$$\begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{11} & \mathbf{F}_{12} \\ \mathbf{0} & \mathbf{F}_{21} & \mathbf{F}_{22} \end{pmatrix} \begin{pmatrix} \Delta\mathbf{p}^s \\ \Delta\mathbf{p}^{fs} \\ \Delta\mathbf{p}^f \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix} \tag{1.47}$$

Finally, the end of step velocity can be found by way of equations (1.31) and (1.43), which can be used to build an assembled non-linear system. This system can be solved using the scheme

$$\frac{1}{\gamma\,\Delta t} \begin{pmatrix} \rho^s\mathbf{M}^s_{11} & \rho^s\mathbf{M}^s_{12} & \mathbf{0} \\ \rho^s\mathbf{M}^s_{21} & \rho^s\mathbf{M}^s_{22} + \rho^f\mathbf{M}^f_{11} & \rho^f\mathbf{M}^f_{12} \\ \mathbf{0} & \rho^f\mathbf{M}^f_{21} & \rho^f\mathbf{M}^f_{22} \end{pmatrix}_i \begin{pmatrix} \Delta\mathbf{v}^s \\ \Delta\mathbf{v}^{fs} \\ \Delta\mathbf{v}^f \end{pmatrix}_{i+1} = \begin{pmatrix} \mathbf{r}^s_{s3} \\ \mathbf{r}^{fs}_{s3} + \mathbf{r}^{fs}_{f3} \\ \mathbf{r}^f_{f3} \end{pmatrix}_i \tag{1.48}$$

The algorithm to solve the fluid-structure interaction using the approach described in this section can be written as:

1. Predict the end of step velocity (in this case, using the Newmark method).
2. Find the fractional step velocity by means of equation (1.46).
3. Use equation (1.47) to update the pressure.
4. Determine the end of step velocity using equation (1.48).
5. Check convergence in velocity and pressure.
6. If convergence was not achieved, go to step 2.

1.7.2 Uncoupled schemes

Uncoupled approaches are based on dividing the problem in a structural domain and a fluid domain. There are several methods that can be used to solve the uncoupled fluid-structure interaction problem, but Dirichlet-Neumann scheme will be the main focus of this section.

When splitting the two domains, it is critical to handle the interface properly. One approach that has a widespread use in practical applications is the segregated Dirichlet-Neumann coupling (used, for example, in [9]), although this method presents severe convergence difficulties in some problems. In particular, problems are typically encountered in those cases where the fluid is heavier than the structure or the structure is very flexible.

The idea behind segregated Dirichlet-Neumann coupling is to model the effect of the structure over the fluid as a prescribed velocity (a Dirichlet condition), with the associated mesh deformation, and to model the effect of the fluid over the structure as an applied load (a Neumann condition). Although multiple variants exist, a common implementation of this method can be stated as

1. Solve the structure by means of equation (1.42), using a known value of the pressure
2. Map the displacements of the structure interface to the fluid domain and deform the mesh accordingly.
3. Solve equation (1.38) to find the fractional step velocity for the fluid, using the last known value of the pressure.
4. If convergence was not achieved, go back to step 1.
5. Use equation (1.40) to find a new value for the fluid pressure.
6. Find the end of step velocity thanks to equation (1.41).
7. If not converged, use the new values of pressure to update the external loads over the structure side of the interface and go to step 1.

This scheme has two nested loops. The inner loop imposes the dynamic equilibrium using a intermediate value of velocity (in this sense, it is equivalent to solving equation (1.46) in the monolithic linear momentum solution), while the other corrects the value of the velocity so it satisfies the mass conservation equation.

1.7.3 Some considerations about convergence

According to [19], the divergence problems of the segregated Dirichlet-Neumann scheme are a consequence of the outer loop, which can be modified to improve convergence. The reason why in some cases this method fails to find a solution are ultimately derived from the use of equation (1.47). The fractional step scheme developed in section 1.5 holds for the fluid domain, but it is not strictly correct when it is applied over the interface nodes.

When this scheme was adapted to the fluid-structure interaction problem as equation (1.40), the effect of the interface was not taken into account. Instead, the interface nodes were just treated as regular fluid nodes. To correct this oversight, the fractional step scheme will be reformulated for the interface. Going back to equation (1.23), it can be observed that, for the interface terms, the discrete conservation of linear momentum will include terms derived from the structural domain. Assuming, for consistency with the scheme used for the structure, $\theta = 1$, equations (1.23) and (1.24) can be rewritten for the interface as

$$\begin{aligned} &\left(\rho^s \mathbf{M}^s \mathbf{a}^{n+1} + \mathbf{C}^s \mathbf{v}^{n+1} + \mathbf{K}^s \mathbf{u}^{n+1}\right) + \\ &\left(\rho^f \mathbf{M}^f \mathbf{a}^{n+1} + \mathbf{K}^* \mathbf{v}^{n+1} + \mathbf{G}\mathbf{p}^{n+1}\right) = \mathbf{f}^{\,n+1} \end{aligned} \tag{1.49}$$

$$\mathbf{D}\mathbf{v}^{n+1} + \frac{\tau}{\rho^f}\mathbf{S}\mathbf{p}^{n+1} = \mathbf{0} \tag{1.50}$$

Where $\mathbf{a}^{n+1}$ and $\mathbf{u}^{n+1}$ were obtained from equations (1.19) and (1.18), respectively.

Introducing the fractional step velocity $\hat{\mathbf{v}}^{\mathbf{n+1}}$ and assuming that

$$\mathbf{K}^*\left(\hat{\mathbf{v}}^{n+1}\right)\hat{\mathbf{v}}^{n+1} \approx \mathbf{K}^*\left(\mathbf{v}^{n+1}\right)\mathbf{v}^{n+1} \quad ; \quad \mathbf{C}^s\hat{\mathbf{v}}^{n+1} \approx \quad \mathbf{C}^s\mathbf{v}^{n+1}$$

equations (1.49–1.50) can be rewritten as

$$\begin{aligned}&\left(\rho^s\mathbf{M}^s\hat{\mathbf{a}}^{n+1}+\mathbf{C}^s\hat{\mathbf{v}}^{n+1}+\mathbf{K}^s\hat{\mathbf{u}}^{n+1}\right)\\&+\left(\rho^f\mathbf{M}^f\hat{\mathbf{a}}^{n+1}+\mathbf{K}^*\hat{\mathbf{v}}^{n+1}+\mathbf{G}\mathbf{p}^n\right)=\mathbf{f}^{\,n+1}\end{aligned}\tag{1.51}$$

$$\begin{aligned}&\left(\rho^s\mathbf{M}^s+\rho^f\mathbf{M}^f\right)\left(\frac{\mathbf{v}^{n+1}-\hat{\mathbf{v}}^{n+1}}{\gamma\,\Delta t}\right)+\mathbf{K}^s\left(\frac{\beta\,\Delta t}{\gamma}\left(\mathbf{v}^{n+1}-\hat{\mathbf{v}}^{n+1}\right)\right)\\&+\mathbf{G}\left(\mathbf{p}^{n+1}-\mathbf{p}^n\right)=\mathbf{0}\end{aligned}\tag{1.52}$$

$$\mathbf{D}\mathbf{v}^{n+1}+\frac{\tau}{\rho^f}\mathbf{S}\mathbf{p}^{n+1}=\mathbf{0}\tag{1.53}$$

As in section 1.5, these three equations can be rewritten to allow a segregated solution:

$$\begin{aligned}&\left(\rho^s\mathbf{M}^s\hat{\mathbf{a}}^{n+1}+\mathbf{C}^s\hat{\mathbf{v}}^{n+1}+\mathbf{K}^s\hat{\mathbf{u}}^{n+1}\right)\\&+\left(\rho^f\mathbf{M}^f\hat{\mathbf{a}}^{n+1}+\mathbf{K}^*\hat{\mathbf{v}}^{n+1}+\mathbf{G}\mathbf{p}^n\right)=\mathbf{f}^{\,n+1}\end{aligned}\tag{1.54}$$

$$\mathbf{D}\left(\frac{\rho^s}{\gamma\,\Delta t}\mathbf{M}^s+\frac{\rho^f}{\gamma\,\Delta t}\mathbf{M}^f+\frac{\beta\,\Delta t}{\gamma}\mathbf{K}^s\right)^{-1}\mathbf{G}\,\Delta\mathbf{p}^{n+1}=\mathbf{D}\hat{\mathbf{v}}^{n+1}\tag{1.55}$$

$$\left(\rho^s\mathbf{M}^s+\rho^f\mathbf{M}^f\right)\left(\frac{\mathbf{v}^{n+1}-\hat{\mathbf{v}}^{n+1}}{\gamma\,\Delta t}\right)+\mathbf{K}^s\frac{\beta\,\Delta t}{\gamma}\left(\mathbf{v}^{n+1}-\hat{\mathbf{v}}^{n+1}\right)+\mathbf{G}\Delta\mathbf{p}^{n+1}=\mathbf{0}\tag{1.56}$$

The interesting point here is that, while equations (1.54) and (1.56) are identical to the interface terms obtained for the monolithic linear momentum solution, equation (1.55) contains two additional terms that haven't been taken into account until now. To include them in the monolithic scheme, equation (1.47) should be rewritten as

$$\begin{pmatrix}\mathbf{I}&\mathbf{0}&\mathbf{0}\\\mathbf{0}&\mathbf{F}_{11}+\mathbf{D}\left(\frac{\rho^s}{\gamma\,\Delta t}\mathbf{M}^s+\frac{\beta\,\Delta t}{\gamma}\mathbf{K}^s\right)^{-1}\mathbf{G}&\mathbf{F}_{12}\\\mathbf{0}&\mathbf{F}_{21}&\mathbf{F}_{22}\end{pmatrix}\begin{pmatrix}\Delta\mathbf{p}^s\\\Delta\mathbf{p}^{fs}\\\Delta\mathbf{p}^f\end{pmatrix}=\begin{pmatrix}\mathbf{0}\\\mathbf{b}_1\\\mathbf{b}_2\end{pmatrix}\tag{1.57}$$

Note that this expression can also be used for the uncoupled approach, if the the rows and columns corresponding to the structure-only nodes are omitted.

It can be observed that:

- the additional term appears exclusively on the FSI interface,
- the additional term is larger when the structure is lightweight, a case where Dirichlet-Neumann coupling tends to fail, and
- the additional term will vanish when the stiffness of the structure dominates, a case where Dirichlet-Neumann is satisfactory.

The term $\left(\frac{\rho^s}{\gamma\,\Delta t}\mathbf{M}^s+\frac{\beta\,\Delta t}{\gamma}\mathbf{K}^s\right)^{-1}$ is not usable in practical problems, as it would involve the inversion of the structural stiffness matrix $\mathbf{K}$. Fortunately, the effect of this term can be simulated relatively painlessly.

An interesting observation is that, for small time steps, the inertia term is greater:

$$\mathbf{D}\left(\frac{\rho^s}{\gamma\,\Delta t}\mathbf{M}^s+\frac{\beta\,\Delta t}{\gamma}\mathbf{K}^s\right)^{-1}\mathbf{G}\approx\mathbf{D}\left(\frac{\rho^s}{\gamma\,\Delta t}\mathbf{M}^s\right)^{-1}\mathbf{G}$$

The right hand side of this expression is similar to the $\mathbf{D}(\mathbf{M}^f)^{-1}\mathbf{G}$ that has appeared before, but it is defined for the structural domain. By using diagonal mass matrices, which guarantees that both matrices will have the same structure, it can be considered that the structure matrix is proportional to its fluid counterpart or, in mathematical terms:

$$\beta \left(\frac{\rho^f}{\gamma\, \Delta t} \mathbf{M}^f_{fs} \right)^{-1} = \left(\frac{\rho^s}{\gamma\, \Delta t} \mathbf{M}^s_{fs} \right)^{-1} \quad ;$$

$$\beta = \frac{\frac{\rho^f}{\gamma\, \Delta t} \mathbf{M}^f_{fs}}{\frac{\rho^s}{\gamma\, \Delta t} \mathbf{M}^s_{fs}}$$

where $\mathbf{M}^s_{fs}$ and $\mathbf{M}^f_{fs}$ are the rows and columns of the structure and fluid mass matrices restricted to the interface degrees of freedom. It is interesting that the values of the coefficients of the mass matrices are proportional to the size h of the elements used to mesh their respective domains. With this in mind, it can be said that

$$\beta \approx \frac{\rho^f (h^f)^3}{\rho^s (h^s)^3} \tag{1.58}$$

which provides an estimate for β.

Now the value of β can be used to model the effect of the structural terms in equation (1.57) using only fluid matrices as

$$\begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{F}}_{11} & \mathbf{F}_{12} \\ \mathbf{0} & \mathbf{F}_{21} & \mathbf{F}_{22} \end{pmatrix} \begin{pmatrix} \Delta \mathbf{p}^s \\ \Delta \mathbf{p}^{fs} \\ \Delta \mathbf{p}^f \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix} \tag{1.59}$$

$$\tilde{\mathbf{F}}_{\mathbf{11}} := \frac{\gamma\, \Delta t}{\rho^f} \left(1 + \frac{\rho^f (h^f)^3}{\rho^s (h^s)^3} \right) \mathbf{D}(\mathbf{M}^f)^{-1}\mathbf{G} + \frac{\tau}{\rho}\, \mathbf{L} \tag{1.60}$$

This approximation will be sufficient for the examples implemented in the following section, but a method to modify the value of the β parameter to take into account the effect of the structural stiffness matrix for "not so small" time steps is introduced in [19].

Chapter 2

An example implementation

Once the basic theory of the fluid-structure interaction problem has been introduced, this chapter aims to implement it in a simple practical example using Matlab 7.0. This exercise provides a benchmark to compare the different approaches to the problem discussed in section 1.7, which will prove useful to understand its practical aspects.

The implemented problem is an unidimensional model, composed of a structural domain of length $l_s = 10\,m$, next to a fluid domain of the same length ($l_f = 10\,m$). On a first example, shown in figure 2.1, an external load is applied over the right end of the fluid domain, while the position of the left end of the structural domain is fixed. In a second example, shown in figure 2.2, the positions of the structure and the fluid are reversed so the load is applied over the structural domain.

For both examples, the physical properties of the structural domain are:

- A length of $l_s = 10\,m$.
- A cross section area of $A_s = 1\,m^2$.
- Young Modulus: $E = 10^4\,MPa$.
- No dampening is considered, which means that $\mathbf{C}^s = \mathbf{0}$.

On the other hand, the fluid domain is characterized by:

- A length of $l_f = 10\,m$.
- A cross section area of $A_f = 1\,m^2$.
- The viscosity of the fluid is considered negligible.

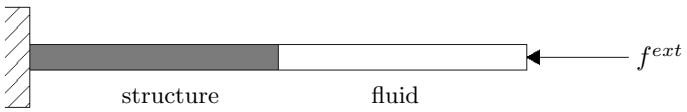

Figure 2.1: First example domain.

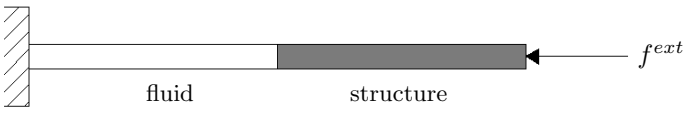

Figure 2.2: Second example domain.

For both examples, a force of $30\,KN$ is applied over the right end of the model, while the left end's position is fixed.

A Lagrangian description is used in both domains. This is an atypical choice for the fluid domain but it is justified in this particular problem, as the displacement of the fluid won't be large. Note that this implies that the mesh velocity $\mathbf{v}_M$ will be equal to the nodal velocity $\mathbf{v}$. This, combined with the fact that the fluid has no viscosity, implies that $\mathbf{K}^* = \mathbf{0}$.

For the solution of equation (1.40), the choice of the stabilization parameter τ has been $\tau = \gamma\,\Delta t$, where γ is the Newmark parameter.

Convergence control is achieved by imposing a tolerance on the velocity and pressure results

$$\mathrm{Err}(\mathbf{v}^{i+1}) = \frac{\left\|\Delta\mathbf{v}^{i+1}\right\|}{\|\mathbf{v}^{i+1}\|} \leq \mathrm{tol}_v \quad ; \quad \mathrm{Err}(\mathbf{p}^{i+1}) = \frac{\left\|\Delta\mathbf{p}^{i+1}\right\|}{\|\mathbf{p}^{i+1}\|} \leq \mathrm{tol}_p$$

In both cases, the maximum acceptable error has been set as 10^{-4}.

Both problems will be solved for different values of structure and fluid density, in order to compare the results of the different methods for the cases of heavy structure and light fluid, light structure and heavy fluid and equal density.

The domain is discretized using a single row of lineal elements, which means that the fluid-structure interface will be composed of a single node.

2.1 Practical aspects

This section introduces the specifics of adapting the general fluid-structure interaction problem to the examples in this chapter. The most important aspect of this process is obtaining the various matrices that have been defined, as their expression is dependent on the finite element discretization used. Only the particularities will be discussed here, as the finite element assembly is not the main point of interest of this project. General information about this subject can be found in any finite element textbook as, for example, [20].

To discretize both domains, two-node linear elements such as the ones represented in figure 2.3 have been used.

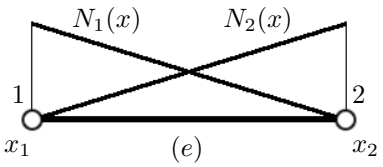

Figure 2.3: Shape functions.

The equations of the shape functions used to implement the problem are

$$N_1(x) = 1 - \frac{x - x_1}{l_e} \quad ; \quad N_1' = -\frac{1}{l_e}$$
$$N_2(x) = \frac{x - x_1}{l_e} \quad ; \quad N_2' = \frac{1}{l_e}$$

where e is the element number, x_1 and x_2 represent the x coordinate of the left and right nodes and $l_e := x_2 - x_1$ is the finite element length.

Once the shape functions have been given, the matrices involved in the problem can be computed. They will be defined here in their elemental form, as obtained by a Galerkin discretization, keeping in mind that the system matrices are built by the finite element assembly of the elemental matrices.

The mass matrix, that is required for both the fluid and the structure, takes the form

$$\mathbf{M}^e = \int_{\Omega_e} N_i \cdot N_j \, \mathrm{d}x \tag{2.1}$$
$$\mathbf{M}^e = \begin{pmatrix} \frac{l_e}{3} & \frac{l_e}{6} \\ \frac{l_e}{6} & \frac{l_e}{3} \end{pmatrix} \tag{2.2}$$

In some of the algorithms, a diagonal mass matrix is used, as its inverse can be easily computed. The diagonal mass matrix can be obtained by adding all the non-zero terms in any given row and placing the result in the diagonal. The diagonal mass matrix can be computed as the assembly of elemental matrices as the following:

$$\mathbf{M}^e_{diag} = \begin{pmatrix} \frac{l_e}{2} & 0 \\ 0 & \frac{l_e}{2} \end{pmatrix} \tag{2.3}$$

Note that the mass matrix always appears accompanied by the density. For an unidimensional problem as this one, the density should be understood as line density (per unit of length). Assuming that the structural and fluid domains are both homogeneous, the $\rho\mathbf{M}$ terms in the general fluid-structure interaction equations can be replaced for the unidimensional problem with $\rho A\mathbf{M}$, where A is the cross-section area.

For the structure part, the stiffness matrix will be required. It can be obtained by the assembly of

$$\mathbf{K}^e = \int_{\Omega_e} E\,A\,\nabla N_i \cdot \nabla N_j \, \mathrm{d}x \tag{2.4}$$
$$\mathbf{K}^e = \frac{E\,A}{l_e} \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \tag{2.5}$$

Note that the cross-section area A has been introduced in the equation. This is in line with the remarks made for the mass matrix.

For the fluid part, three additional equations are required. The first one of them is the matrix $\mathbf{G}$, obtained from the discretization of the gradient operator:

$$\mathbf{G}^e = -\int_{\Omega_e} \nabla N_i \cdot N_j \, \mathrm{d}x \tag{2.6}$$
$$\mathbf{G}^e = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & -\frac{1}{2} \end{pmatrix} \tag{2.7}$$

The discretization of the divergence operator yields matrix $\mathbf{D}$:

$$\mathbf{D}^e = \int_{\Omega_e} N_i \cdot \nabla N_j \, \mathrm{d}x \tag{2.8}$$

$$\mathbf{D}^e = \begin{pmatrix} -\frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix} \tag{2.9}$$

Finally, matrix $\mathbf{L}$, obtained from the discretization of the Laplacian operator:

$$\mathbf{L}^e = -\int_{\Omega_e} \nabla N_i \cdot \nabla N_j \, \mathrm{d}x \tag{2.10}$$

$$\mathbf{L}^e = \begin{pmatrix} -\frac{1}{l_e} & \frac{1}{l_e} \\ \frac{1}{l_e} & -\frac{1}{l_e} \end{pmatrix} \tag{2.11}$$

$\mathbf{L}$ is used in the pressure stabilization term and as an approximation to $\mathbf{D}\mathbf{M}^{-1}\mathbf{G}$.

2.2 Implemented algorithms

From the theory developed in the previous pages, three different algorithms have been implemented. One of them solves the dynamic equilibrium equation monolithically, as was already introduced in section 1.7.1. The other two are alternative implementations of the segregated Dirichlet-Neumann coupling scheme outlined in section 1.7.2, which will be discussed later in their own section.

Finally, an additional scheme has been implemented to check that the different methods are reaching the expected solution. This has been done by implementing a structure-only solver, which simulates the fluid domain as a structure with a much greater Young modulus. This stiffer "fluid" can be considered incompressible when compared to the solid domain, and provides a good estimate of the nodal displacements and velocities. Unfortunately, this model provides no information about the fluid pressure, which will have to be verified using other means.

2.2.1 Dirichlet-Neumann uncoupled solver

The first of the two uncoupled solvers implements the algorithm proposed in [19], which, given that this example uses a Lagrangian description of the fluid instead of a moving ALE mesh, can be described as

1. For the cases where the structure is heavier than the fluid, solve the structure by means of equation (1.13), using the result of the last time step as a value for the pressure.

2. Use the structure results to impose the velocity of the interface node to the fluid domain. If step 1 was skipped, use the velocity obtained in the last time step.

3. Impose the linear momentum conservation on the fluid domain thanks to equation (1.38), obtaining the fluid's fractional step velocity $\hat{\mathbf{v}}^{i+1}$.

4. Compute the new value of pressure using equation (1.40).

5. Use the new pressure to compute the forces that the fluid exerts on the structure as $\mathbf{f}^{fs} = 1/A\,\mathbf{G}\,\mathbf{p}$.

6. Solve the structure.

7. Find the end of step velocity for the fluid by means of equation (1.41).

8. Check for convergence and go back to step 4 if needed, using the current iteration's results as the new values of $\hat{\mathbf{v}}$ and $\mathbf{p}_{kn}$.

As, of all the methods used in this project, this one is the most used in practice, some further variations have been implemented. Several versions of this algorithm will be tested, with or without the additional interface term introduced in section 1.7.3 and with the possibility of using the Laplacian matrix $\mathbf{L}$ as an approximation to $\mathbf{D}\mathbf{M}^{-1}\mathbf{G}$ in the left hand of equation (1.36).

2.2.2 A simple uncoupled solver

The second uncoupled solver treats the two domains as independently as possible. Like the previous one, it uses Dirichlet-Neumann coupling, using the displacement of the structure as a boundary condition for the fluid and the fluid pressure to apply a load over the structure part. The main difference with the preceding example is that the results are only transferred between domains once each domain has been completely solved. The algorithm can be written as

1. Use the Newmark method to predict the velocity at the end of the time step.

2. Solve the fluid domain by means of equations (1.38–1.41), imposing the velocity of the interface node as a Dirichlet condition.

3. Use the end of step pressure to determine the load applied to the structural domain in the interface as $\mathbf{f}^{fs} = 1/A\,\mathbf{G}\,\mathbf{p}$.

4. Solve the structural domain using equation (1.13).

5. If convergence was not achieved, repeat step 2 using the results of step 4 to determine the velocity of the interface node.

2.3 Results

2.3.1 First example

The first example has been executed for five different combinations of structure and fluid density. Some of the obtained results are plotted in figures 2.4 to 2.6, where only the methods that achieved convergence are pictured. The numerical parameters adopted to solve each case can be found in table 2.1. Unless the opposite is explicitly stated, the Laplacian matrix $\mathbf{L}$ has been used as an approximation to $\mathbf{D}\mathbf{M}^{-1}\mathbf{G}$.

Before analyzing the results, some checks have been made to ensure that they are correct. The first thing that can be checked is that the obtained displacements are reasonably similar to the expected value, as calculated by assuming that the fluid is an "incompressible" structure. This can be verified

ρ^s/ρ^f	100	10	1	0.1	0.01
$\rho^s\ (kg/m^3)$	100	10	1	1	1
$\rho^f\ (kg/m^3)$	1	1	1	10	100
structure elements	10	10	25	25	45
fluid elements	10	10	25	25	45
time steps per period	90	90	200	200	220

Table 2.1: Problem parameters for the first series of tests.

by observing the mentioned figures, where the expected result has been plotted as a black line.

Some checks should also be made to verify the values of the pressure, but they are not as straightforward. A first thing that can be verified is that the pressure value in the node where the exterior force is applied coincides with the applied force (as it is applied of an area of $1\,m^2$). Unfortunately, this is be imposed as a boundary condition in all methods that use the Laplacian matrix to approximate $\mathbf{DM}^{-1}\mathbf{G}$, so it can be only used as a check for the last two Dirichlet-Neumann variants tested. In those cases, however, the results are satisfactory as they never differ significantly from the expected result (for an expected result of $30\,KPa$, the pressure in the end node oscillates between 29.998 and $30.002\,Pa$).

In the remaining cases, a possible check of the quality of the pressure result is to take into account the linear momentum equation for the fluid domain:

$$\rho\frac{\partial \mathbf{v}}{\partial t} - \nabla p = f_{ext} \tag{2.12}$$

For the interior nodes of the fluid (all of them except the rightmost one, where a force is applied, and the interface node), $f_{ext} = 0$. As the fluid is theoretically incompressible, all the fluid nodes move at the same velocity, which meas that, for any given time step, $\frac{\partial \mathbf{v}}{\partial t}$ will be a constant for all fluid nodes. This implies that equation (2.12) can be rewritten as

$$\nabla p = constant \tag{2.13}$$

which in turn implies that the pressure variation between the interior fluid nodes must be lineal for any given time step. Examining the results shows that this is true for all the methods that converged.

As several methods are being used to solve the same problem, a result with practical interest is the number of iterations that each method required to achieve convergence. The total number of iterations for each solver is shown in table 2.2. As was expected, the monolithic linear momentum equation approach requires the minimum amount of iterations, although each individual iteration is more expensive, followed by the different variations of the Dirichlet-Neumann uncoupled solver.

One of the first conclusions that can be extracted from the table is that, without taking into account the interface term introduced in section 1.7.3, the different methods are unable to tackle the cases where the fluid is heavier.

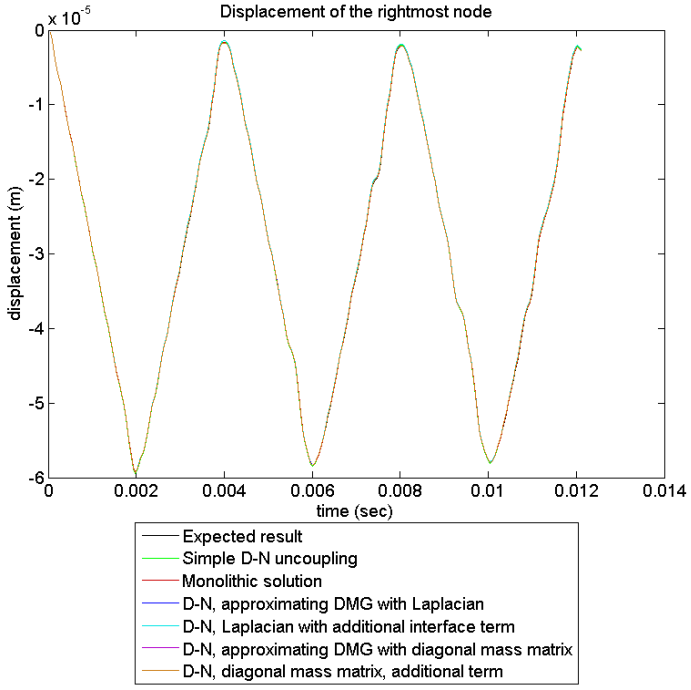

Figure 2.4: Displacement of the rightmost node of the first example for the case $\rho^s = 100\,\rho^f$.

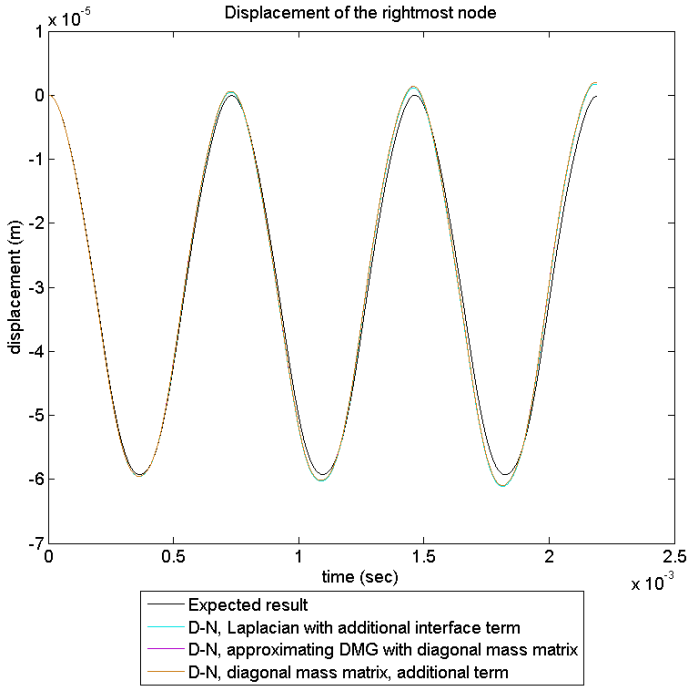

Figure 2.5: Displacement of the rightmost node of the first example for the case $\rho^s = \rho^f$.

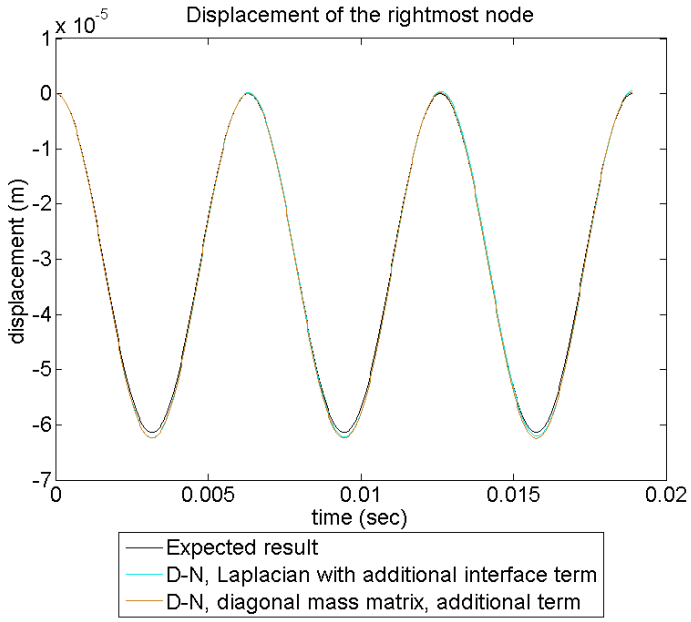

Figure 2.6: Displacement of the rightmost node of the first example for the case $\rho^f = 100\,\rho^s$.

This was expected as those are the cases where the Dirichlet-Neumann schemes typically experience problems. It should be noted that the lack of convergence appears relatively early, as some methods can not find a solution for the $\rho^s = \rho^f$ case. This is a consequence of the discretization. In most practical problems, the fluid mesh is finer, which means that interface nodes each have a smaller area of influence on the fluid side than on the structure side. This means that each interface node has less fluid mass associated than structure mass and this has a favorable effect in reducing the negative impact that the heavier fluid has on convergence. In this case, the fluid and structure elements have the same size, which means that the interface node has proportionally more fluid mass associated.

Another expected result is that the difference between using and not using the additional interface term is small for the cases where the structure is heavier than the fluid. This is a consequence of the choice of the β parameter that multiplies this added term, which was

$$\beta = \frac{\rho^f (h^f)^3}{\rho^s (h^s)^3}$$

However, it must be noted that the results in table 2.2 also show an unexpected result: the methods that compute $\mathbf{DM}^{-1}\mathbf{G}$ instead of approximating it with the Laplacian matrix are not consistently better: for extreme density ratios, they require more iterations than the rest. In theory, the use of $\mathbf{DM}^{-1}\mathbf{G}$ should result in a reduced number of iterations (although each individual iteration is more expensive) but, in this example, this is not always the case and, in some extreme cases, this method requires clearly more iterations.

ρ^s/ρ^f	100	10	1	0.1	0.01
Simple uncoupling	3564	7106	-	-	-
Monolithic lin. mom.	3131	4234	-	-	-
Dirichlet-Neumann	3457	5616	-	-	-
D-N with interface term	3512	2211	28195	9707	32700
D-N calculating $\mathbf{DM}^{-1}\mathbf{G}$	15439	5605	5433	-	-
D-N, interface term and $\mathbf{DM}^{-1}\mathbf{G}$	15471	5794	4760	9824	36025

Table 2.2: Required iterations to solve three full periods of the first example, for different density ratios.

$\rho^s\ (kg/m^3)$	10	7	5	2	1
$\rho^f\ (kg/m^3)$	1	1	1	1	1
Monolithic lin. mom.	5100	6682	7932	5440	3472
D-N, Laplacian	6716	10923	21083	32205	28195
D-N, $\mathbf{DM}^{-1}\mathbf{G}$	11447	9482	8146	7213	8855
$\rho^s\ (kg/m^3)$	1	1	1	1	-
$\rho^f\ (kg/m^3)$	2	5	7	10	-
Monolithic lin. mom.	4422	6533	7643	9085	-
D-N, Laplacian	12663	7958	8459	9707	-
D-N, $\mathbf{DM}^{-1}\mathbf{G}$	7944	8326	9266	10455	-

Table 2.3: Iteration totals for the second series of tests (3 full periods).

Once the importance of the interface term defined in section 1.7.3 was clear, a second series of tests were run on the same problem. This time, the additional interface term was added to the monolithic linear momentum solver, and the program was run again for a new series of density relations. The iteration totals are shown in table 2.3.

This time, the addition of the interface term allows the convergence of the monolithic linear momentum solver in the cases where the fluid is heavier. This solver is again the one that requires the least amount of iterations. The difference between the two Dirichlet-Neumann variants, again, is not as clear as expected. While not approximating $\mathbf{DM}^{-1}\mathbf{G}$ by a Laplacian is clearly more efficient in the cases where the fluid and the structure have roughly the same density, as soon as the difference of densities increases, the advantages of this approach are not as significant, to the point that, in some cases, the use of the Laplacian matrix is slightly more efficient. The number of iterations required for the Laplacian solver also varies somewhat erratically between the cases.

2.3.2 Second example

For the second example, the only segregated methods applicable are those that compute $\mathbf{DM}^{-1}\mathbf{G}$. This is a consequence of what was introduced in section 1.5: using the Laplacian matrix produces a linear system that can only be solved if some values of pressure are known and applied as boundary conditions. Unfortunately, in this case there is no fluid point whose pressure can be known a priori. For this reason, only the last two Dirichlet-Neumann variants used in the last example are applicable here. The values used and the iteration totals are shown in table 2.4

$\rho^s\ (kg/m^3)$	10	1	1
$\rho^f\ (kg/m^3)$	1	1	5
structure elements	25	25	100
fluid elements	25	25	100
time steps per period	200	200	300
Dirichlet-Neumann	62916	73827	-
D-N with interface term	63763	73931	226965

Table 2.4: Parameters and iteration totals for the second series of tests (2 full periods).

As it can be seen in the table, the tests run on the second example require a much greater number of iterations compared to the first one. This is a result of the way in which the problem was defined, this example is more demanding than the first, as the incompressible fluid is placed next to a solid boundary, which means that the fluid can't experience any deformation. Just as in the previous example, it can be noted that the added interface term is required to achieve convergence in the case where the fluid is heavier.

2.4 Conclusions

To finish this chapter, the most significant results obtained will be commented briefly.

- One of the clearest results obtained from the examples exposed in this chapter is that the added interface term in the mass conservation equation is critical for the convergence of the cases with a heavier fluid. With it, both the monolithic linear momentum solver and the variants of the segregated Dirichlet-Neumann coupling scheme are able to reach a satisfactory solution.

- Clearly, the cases where the structure is heavier than the fluid can be solved more easily than the opposite, as predicted by the theory.

- Solving the monolithic linear momentum equation requires less iterations that the segregated approaches. Unfortunately, this advantage can not be

exploited in practical applications, as the system matrices in the coupled problem are poorly scaled, because they combine terms used to describe different physical problems (the structure and the fluid), which can have different orders of magnitude.

- One of the results does not coincide with the theory: in the examples analysed, the use of $\mathbf{DM}^{-1}\mathbf{G}$ instead of the Laplacian matrix does not always result in a reduced iteration total. In fact, in some cases, the use of the Laplacian matrix results in considerably faster convergence that not approximating $\mathbf{DM}^{-1}\mathbf{G}$.

Some reasons that could account for the last point are that the examples studied are extremely simplistic, while the solution methods implemented are simplified versions of the ones used in practice. For example, in real problems the Lagrangian description is never used for the fluid, and more sophisticated versions of the Newmark method are used for time stepping. Perhaps the results obtained are not as representative as expected of the practical aspects of the problem. While the solutions obtained are correct, the behaviour of the method with regards to convergence is somewhat erratic.

Chapter 3

A new input tool

3.1 Introduction

This chapter will discuss the development of a support program for the implementation of finite element applications within the Kratos framework. In particular, this program generates the required configuration files for GiD, a pre and post-processing tool, to interact with Kratos.

The following pages will explain why it is interesting to be able to automate the process of configuring GiD to produce files for the different Kratos applications and how the code required to achieve this goal has been designed and implemented.

3.1.1 Kratos

Kratos [1] is a framework for the development of finite element solvers, which provides a common platform to solve any type of finite element problem. In addition, it is designed to provide an easy implementation of multi-disciplinary problems as, for example, the fluid-structure interaction problem that is the main subject of this project.

From a practical point of view, the core of Kratos has been written using C++, and relies in the Python programming language [21] for interfacing with the end user.

3.1.2 GiD

GiD [22] is a graphical user interface for geometrical modelling, data input and visualisation of results for all types of numerical simulation programs. Like Kratos, GiD is not designed for a specific pysical problem, and it can be used for any problem as long as it is used in conjunction with an appropriate solver.

An important point to be made here, as it will be relevant later, is that the geometric data introduced in GiD is organized in four levels of geometrical entities: points, lines, surfaces and volumes. Entities of higher level are constructed over entities of lower level: volumes are defined as the space comprised between surfaces. For example, a rectangular prism is defined by six quadrilateral surfaces (its faces). Likewise, surfaces are defined by their edges and lines

are created by joining the points at their ends. This geometrical *hierachy* will have a storng impact in the design some of the characteristics of our program.

3.1.3 Using GiD as a pre-process tool for Kratos solvers

As a pre-process tool, GiD's task is generating all the required input for a solver program. To do this, it relies on the information provided by the user, who will draw a geometry and assign a series of properties to it. These properties can be any type of data, so GiD will need to know which information the user has to provide. For example, in a structural problem, the user will be required to specify the model's displacement constraints and the physical properties (such as its density or its Young Modulus) of the materials that constitute its parts.

In addition to this information, which is generally related to the physical aspects of the finite element domain and its boundary conditions, GiD also requires information of the implementation of the solver. Specifically, GiD needs to know how to provide the physical information to the solver application, how it expects to find its input. Typically, this data is read from one or more input files, where the information is organized in some way so the solver can know if the numbers written on the file are, for example, the coordinates of a node or the connectivity of an element. To provide the required data to Kratos, GiD will need to know how to write those input files.

All this information about the finite element problem and its solver is provided to GiD in what it calls a problem type. A problem type is a folder containing a series of configuraion files that GiD reads to know how to handle the input for a specific solver. The main objective of this chapter will be designing and implementing a way to automate the creation of these files.

3.1.4 A brief overview of GiD problem types

In order to provide a better understanding of the program developed in this project, this section intruduces the technical aspects of configuring GiD. Obviously, this is only a brief explaination of the basics and the aspects that have had a greater influence in the implementation of the new tool, so it shouldn't be considered a comprehensive guide. For a more detailed approach to GiD configuration, refer to [23] or [24].

As it has been mentioned in section 3.1.3, pre-processing a problem with GiD involves drawing the geometry of our problem's domain and assigning data such as boundary conditions or the physical properties of the materials that compose the model. GiD can then be used to generate a finite element mesh from the geometry and assign to its nodes and elements the information that the user has provided. The resulting model is used to write an input file, which the finite element solver will read. GiD knows which information the user will be required to assign and how to write the input file thanks to a problem type. A problem type is a folder containing several configuration files, each one of which stores an aspect of the problem:

- The general configuration file, identified by its `.prb` extension, contains general information about the problem and data that is not related to any specific node or element.

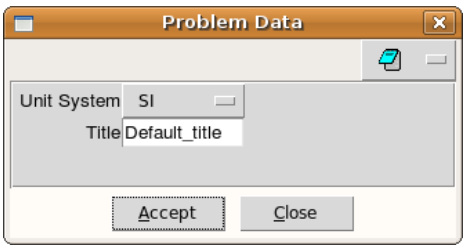

Figure 3.1: An example `.prb` file and the resuling GiD window.

- The materials file (`.mat`) configures the materials available for the model's elements and the physical properties that need to be defined for each one of them.
- The conditions file (`.cnd`) stores the available boundary conditions and, in general, any information related to specific nodes or elements.
- Template files (`.bas`) store the format in which GiD will write the input files for the solver.
- Batch files (`.bat`) contain instructions that will be executed when the *calculate* button is clicked inside GiD. They are used to call the solver application.
- The Tcl extension file (`.tcl`) is an optional file that contains extensions to GiD written in the Tcl/Tk programming language. This file will be used to provide some additional features to Kratos problem types, and will be discussed in detail in section 3.4.3.

All these files are text based. The first three use a common syntax (with particularities for each one), in which pairs of *question* and *value* fields are defined. When the user attempts to assign a condition or material or modify the values of the general parameters, a window will appear containing a series of fields where a value can be introduced, with what is found in the *value* field as default, accompanied by a name identifying what should be written in it, read from the *question* field. For example, the following configuration file results in the window that can be seen in figure 3.1:

```
PROBLEM DATA
QUESTION: Unit_System#CB#(SI,CGS,User)
VALUE: SI
QUESTION: Title
VALUE: Default_title
END GENERAL DATA
```

Template files have a different syntax. Without going into detail, they are a combination of text that will be present in the finished input file and special commands (words preceded by an asterisk), which will be replaced with information about the model when the input file is written. These commands can be used to ask information about the mesh, its nodes and elements, or the various conditions and values that the user can assign to the model.

Finally, batch files contain instructions to be executed once the model has been finished and the input files written. There are two of them: one for Windows environments (written in DOS batch language) and one for Unix environments (written in Bash shell language).

3.1.5 The need for a problem type generator

As each Kratos-based application solves a different finite element problem, it will have its own variables, boundary conditions and materials. This means that it will require its own GiD problem type. On the other hand, Kratos has a standard input format, which means that all Kratos applications will recieve their input written in the same way. In this context, a program able to write problem types from a template, requiring minimal input from the coder of the solver, can be a helpful tool. This is specially true when a solver is being developed, which means that, to test any new features as they are being implemented, the problem type files have to be modified regularly. This is a cumbersome process, as it typically involves modifying at least two files: a configuration file and a template file. In addition, there is no tool to check if the changes made are being written using proper syntax, which means that it's easy to introdce errors.

This fact becomes more relevant when taking into account that many of the users of Kratos are degree or post-graduate students who, in many cases, don't have a previous experience of using neither Kratos nor GiD. As configuring GiD is a relatively advanced task, specially because it requires a certain degree of knowledge about how the program works, an automated tool to generate problem types for the applications they develop or modify would reduce the time they have to invest into learning to use the available tools.

Continuing with this attempt to make using GiD and Kratos together easier, this new program will also take advantage of GiD's extension capabilities to simplify the process of creating models for Kratos applications. This will be explained with greater detail in the following pages and specially in section 3.4.3.

It is worth mentioning that, in the case of fluid-interaction problems, the interaction between GiD and Kratos is specially not user-friendly. For example, the fact that the fluid-structure interaction application expects to find two different models for each problem, one containing the structural domain and another one containing the fluid part, is inconvenient from the point of view of GiD. GiD stores the input files it creates in a different folder for each model, but Kratos expects to find the input for both parts in the same folder. In consequence, the user has to copy the files to a new folder before running Kratos, and this has to be done again every time a change is made in the model, such as modifying the boundary conditions or using a different mesh. This procedure could be easily automated but, when this project started, the existing problem type was not able to do it alone.

3.1.6 The initial situation

When this project began, the problem types used for the different Kratos applications were generated using a pre-existing Python program. The initial goal was to expand the capabilities of this program to make the problem types it wrote more user-friendly, according to what has been mentioned in the previous

paragraphs. To do so, the program would be adapted to generate an additional configuration file, which would use GiD's extension capabilities to automate certain parts of the process of creating a new model.

In addition to this, the Kratos developement team decided to change the format for the input data, replacing the existing way to introduce the information for a simpler, more compact one. This means that the new problem types would need to be able to generate input in this new format.

Soon after starting the task, it became evident that it was excedingly difficult to modify the existing program to provide those additional capabilities. In consecuence, it was decided that a new programn would be written from scratch, designed to support the new features from the start.

3.2 Designing a problem type generator

3.2.1 Design principles

Once it became clear that the existing program to generate problem types could not be easily adapted to provide additional features, the process of designing a new one began. To ensure that the new program would be useful, the following goals were set:

- The program has to be able to write all the required files to configure GiD to work with Kratos solvers. This point should be obvious by now, as it is the main objective.
- The new program has to be reasonably easy to learn. As this program is designed for people with little or no experience in modifying GiD, its use has to be relatively simple.
- This simplicity should not mean introducing limitations. The problem type generator will only be able to implement a subset of all the possibilities available when directly modifying the problem type files, but it should be designed keeping the possibility of future expansion in mind. The program should be a tool to help the user, not a limiting factor in what can be done.
- The problem type files should be written with minimal input from the user. This point is important because it reduces an important source of errors. Once the finished program has been debugged and tested, the user can be certain that the GiD configuration files written for the problem type have proper syntax, eliminating a common cause of problems. It also has the side advantage of simplifying the process of defining new problem types.
- The program has to be able to adapt to changes in the input format. This last point is important, as the Kratos input format is being changed and the program should still be useful once this change is finished.

As it will become clear in the followng pages, the goal that has influenced the most the final program has been the need to be able to adapt to changes in the input format. To provide this feature, as well as the possibility of expansion, the problem type generator has been designed in two different parts. One is the core

of the program, and contains the basic instructions to create the problem type files and write the Tcl code required to expand GiD capabilities. The second half of the program contains the information that is specific to the solver application and its input format.

This design was chosen because it is modular, as there is a clear division between the internal code of the program and the desired format of the resulting problem type. By creating different sets of files defining different input formats, the program can be used to generate problem types for different solvers.

In addition, the program has been designed to be as independent as possible from an input format. Unfortunately, this means that the problem type generator needs to be configured before it can produce input in a certain format. It a certain sense, the task of writting the required files for the problem type still has to be done, but now its done to configure the problem type genrator instead of configuring GiD directly. However, this approach provides an advantage: the problem type configuration only has to be done once, as the same files can be used to define all the problem types that share the same input format.

As the problem type generator has been concieved reuse the configuration files, once it has all the information, individual problem types for the different Kratos applications can be generated quickly and relatively painlessly from a single input file, in which the user defines the conditions, elements and materials that will be available in GiD. In contrast, working directly with GID would mean writing multiple interrelated files for each application.

The required configuration files for both the old and the new Kratos input format are already provided with the problem type generator. In addition, it could be concievably expanded to work with other input formats by designing new configuration files. The process of writting the required files will only be discussed briefly here, but all the required information can be found in greater detail in the manual for the problem type generator, which is included as the appendix A to this document.

3.2.2 The *data groups* approach

One of the key points in dividing the program in a common part and a problem type dependent part is providing a basic language to communicate between the two. This has been achieved by defining several basic types of data. The main program modules define five data groups and, for each one of them, the required functions to write the code that defines them in the problem type. Each of those groups represents one type of data that the user will be required to provide for the models, such as the boundary conditions or the phisical properties of the materials, defining the basics of its implementation in GiD. The divisions between data groups are somewhat arbitrary, but each one of them has to be implemented in a different way. For example, they are defined in different GiD files or require similat Tcl extension code.

The process of configuring the problem type generator for a specific input format involves defining derived classes and defining the input format for them. For example, the basic condition class can be used to define a scalar condition and a vector condition. This is done by providing the input format for those conditions, written in a similar format to that of GiD template files but using provisional names instead of the final name of the condition. When the problem type generator is run, reads the conditions it will define for GiD from an input

file. If, for example, a vector condition is found, it reads the template code defined for its class, replaces the provisional name for the final one and writes it in the finished file.

One of the most useful data groups is the **condition** data type. It is used to implement information related to specific nodes or elements in the mesh and its intended use is to create boundary conditions or initial values, although, for example, in the Kratos problem type this class has been also used to define flags that the solver application uses to identify certain parts of the model. From the point of view of its implementation, conditions write code for GiD's condition file (`.cnd`) and template files (`.bas`). Another important remark is that, while GiD conditions are defined for a single entity type (points, lines, surfaces or volumes), the condition template can be used to write the required code for multiple entities at once.

The **element** type is used to define the type of finite elements used to the solver. In the case of Kratos, it is necessary because, for example, a problem can use several types of tetrahedra elements and providing just their number of nodes is not enough to defie them. This class makes use of the Tcl extension capabilities to automate the task of defining the mesh. For example, if an element is defined as quadrilateral, GiD will automatically create a mesh that uses quadrilaterals where that element is used (instead of the default triangles). The element data type usually modifies the GiD condition file, the Tcl file (`.tcl`) and the template files, altough this can be changed when defining its derived classes.

Due to limitations imposed by the implementation of the functions that generate the Tcl code, the element type can only be used to define elements for a single entity type (for example, only for volumes or surfaces). It is thought that this won't become a restriction, as finite elements are also dependent on entities: a quadrilateral element only makes sense when used to mesh surfaces, while a tetraheral element can only be used in volumes.

The **materials** type is used to define GiD materials, which are used to identify the phisical properties of the different parts of the model. Typically, defining new materials involves modifying the materials (`.mat`) file and at least one template file.

The **general data** type is used to define problem parameters and general information not related to specific nodes or elements. This kind data is defined in GiD general properties file (`.prb`) and will be provided to the solver by including it in some of the template files.

Finally, there is one additional data type, the **part**. Parts are used to provide Tcl features to the problem type, and recieve their name because they are used to identify regions of the model. Parts are implemented as GiD conditions and can be assigned by the user or automatically: there are several subtypes of parts, and some of them can be assigned automatically to the boundary of the model, to all entities, or to all entities except those that have been assigned a different part by the user. Parts are used to assign conditions and elements automatically, and provide several Tcl enhancements that will be explained in section 3.4.1.

3.3 Implementation

As mentioned in the previous sections, the problem type generator implementation is organized in a common part that can be combined with multiple format-specific modules. The main files provide a basic set of tools to write a problem type, while the format-specific ones define the various types of data that the solver can accept as input. This section will outline this organization and explain the interaction between the two.

To write the problem type files, the problem type generator uses a basic template of the file and a series of definition files. The templates provide the basic layout of the finished file, while the definition files provide the code required to add a single condition or, in general, a unit of data belonging to one of the classes derived from the basic data groups, like an element or a material.

The template files are structured, meaning that the code created from the definitions is added to them at specific points. This approach produces cleaner and more organised input files, as the order in which each insividual condition is defined has little influence in the position in which its related code is inserted in each configuration file. This reduces problems when the order in the input file is relevant. The organisation is achieved by including commented out lines in the template files, which won't appear in the finished file. This comment is used as a *bookmark* to identify a given point in the file. The definition files provide, for each piece of code, the name of the file and the bookmark comment line that will be used to know where to add it.

For a given problem type format, several files are required:

- A Python file defining the new data classes, each one of them based on one of the five data types. For each class, the name of its definition file is provided, along with other information.

- A folder named `defnitions` containing definition files. Each one of those definition files contains the GiD code required to define a condition belonging to a given data class, including code for configuration files (`.cnd`, `.mat` or `.prb`), template code (`.bas`) or Tcl extension code.

- A folder named `files` containing basic versions of all problem type files, with special lines (identified with a bookmark comment) where definition code will be inserted.

A flowchart describing the use of the problem type generator is included as figure 3.2. The input for the problem type generator is a file containig the names and default values of the conditions, materials, elements and properties that will be available for the problem type generator, as well as some additional commands to include Tcl enhancements. This file also provides the name of the folder that contains the format-specific information. With this information, the program writes the problem type files. For each item defined in the input file, the problem type generator reads the code included in its class' definition file, replaces any generic bits with the values provided by the input and adds this code to the corresponding problem type file.

The finished problem type files allow the user to define finite element problems using GiD, producing the required input files. Those files can then be read by the appropriate Kratos application, which will solve the problem.

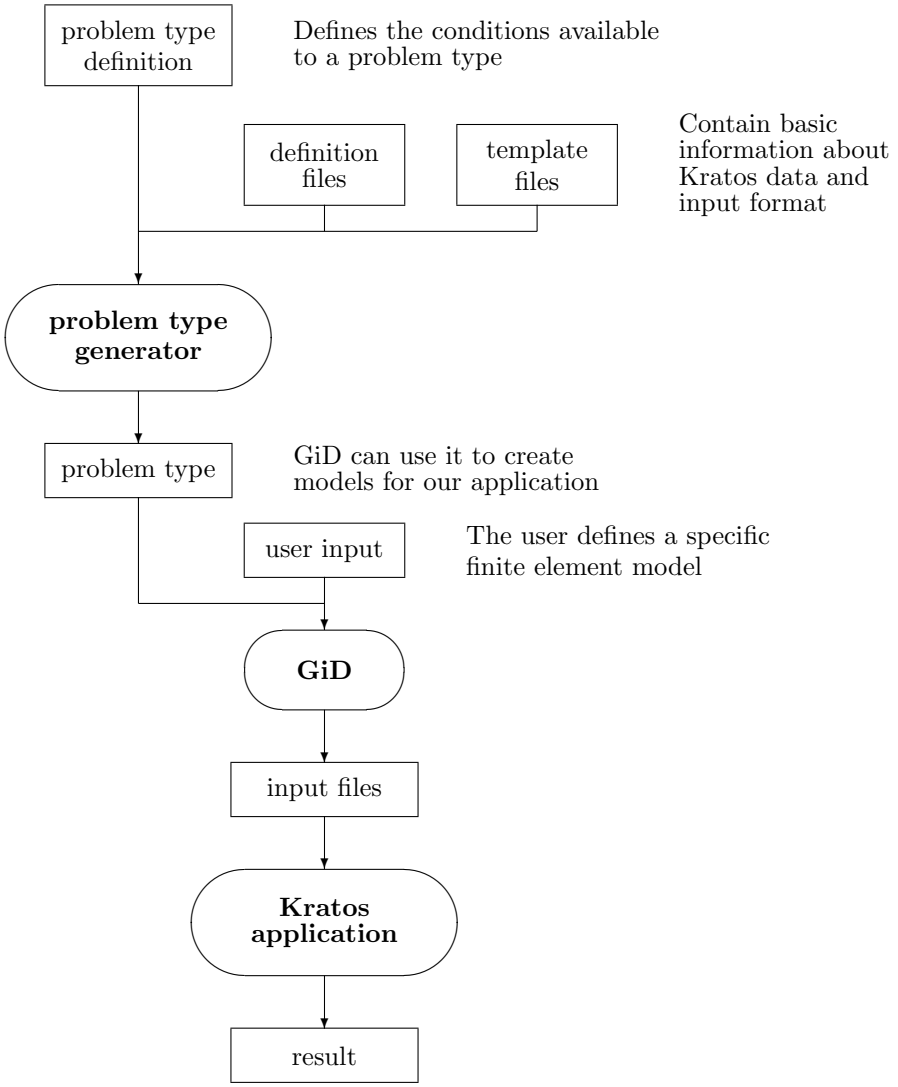

Figure 3.2: Flowchart for problem types created using the problem type generator.

3.4 Tcl features

An important feature of the new problem type generator is that it has the capability of including Tcl extension code in a problem type, which can be used to modify to a certain degree the way GiD works. This is useful because GiD is a tool for pre-process in general, and it wasn't conceived with Kratos in mind. Creating some of the input that Kratos requires is needlessly complex if the original GiD functions are used, and the Tcl extension code can provide a more user-friendly environment for Kratos models.

The Tcl extension features have proved specially useful in joining tasks related to assigning GiD conditions with the process of editing the mesh. For example, as Kratos applications can use different types of finite elements, the elements of the mesh have to be identified using a GiD condition. If a given element is quadrilateral, the user has to assign it and then tell GiD to mesh the regions where this element is used with quadrilaterals instead of the default triangles. If the element is defined as a quadrilateral element when the problem type is created, the program can add the necessary Tcl code to instruct GiD to mesh with quadrilaterals all entities that have the element assigned without the intervention of the user.

Similarly, the problem type generator can add the necessary code to mesh entities that GiD doesn't mesh by default. This is needed in Kratos applications, as their conditions use a syntax similar to that of finite elements, but applied over the faces of the model instead of over its body. To include them in the input file properly, GiD has to create a mesh over those faces, something it won't do by default.

3.4.1 Parts

Most of the Tcl extension features have been implemented with the help of parts. The idea behind parts is to change the way the user defines the boundary conditions in a GiD model. Instead of assigning boundary conditions to the entities, the user identifies in the model a series of areas with physical significance. For example, in the fluid domain of a fluid-structure interaction problem, the following elements will usually exist:

- An inlet, where the fluid enters the domain with a given velocity or pressure.
- An outlet, where the fluid exits the domain.
- The interface: the area in contact with the structural domain.
- A solid contour that won't be deformed and that the fluid can't cross.

Each one of these zones will have different boundary conditions in the model. In the problem types created using the problem type generator, those areas can be identified using parts. Thanks to the Tcl extension code, those parts will be translated into the corresponding boundary conditions internally.

The advantage of this approach is that it can be used to define independent areas with clear borders. Typically, the boundary between two of those parts can require different conditions than those assigned to either of them. The problem type can be programmed to assign specific boundary contitions to the entities

in the bondary between several parts. This feature can be useful to simplify the process of assigning boundary conditions to a model, eliminating a common source of errors. Another minor advantage is that some of the parts can be assigned by default. Following the fluid domain example, the solid contour can be defined as the part of the boundary that is neither inlet, outlet nor interface, and it will be assigned automatically by the program.

It sould be noted that there are problems where the division in parts with physical meaning is not as clear as in this example. For this reason, this feature is entirely optional, and will only be used if parts and their interaction are defined when the problem type is generated. In addition, before assigning any condition, the Tcl code will check that no conditions of the same type have been assigned to the same entity, ensuring that automatically assigned data won't override data assigned by the user.

3.4.2 Tcl extension in GiD

The Tcl extension features for a given GiD problem type must be included in a file with the `.tcl` extension found inside that problem type's folder. As detailed in [23], this file must contain several procedures which GiD will call at different points of the process of creating a problem type. The Tcl files used by the problem types created using our program implement three of those procedures: `InitGIDProject`, which is run when the problem type is selected, `BeforeMeshGeneration`, which is called just before creating the finite element mesh, and `AfterMeshGeneration`, executed after the mesh generation ends.

These three procedures were used because they are called at critical points of the process of generating a model. The first one is called as soon as the problem type information is accessed, which means that no Kratos-specific information has been used yet. This is the perfect time to initialize global variables or create the custom menus that the problem type adds. The other two procedures are called before and after meshing. In GiD, meshing is typically one of the last steps of pre-process, just before writting the input files. This means that, at this point, the user will have provided all the information about the boundary conditions, elements and materials that the model will use, so it will be available to the procedures. Obviously, this is the perfect moment to do all automatic operations over the entities and the problem data: assigning conditions, changing the direction of normals or modifying GiD's meshing behaviour. This is achieved thanks to special Tcl functions included in GiD that can be used to obtain information about the model and modify it.

3.4.3 Writting the Tcl file

To add new Tcl capabilities for the problem type generator, such as aligning normals or transferring a condition or part from a surface entity to its boundary lines, the chosen implementation has been, when possible, to write a procedure in the template Tcl file and add a call to that procedure for each condition that uses them when the problem type file is generated. It is felt that this implementation is the best for code generated by another application, as it reduces the amount of lines that have to be added during file generation and the total length of the Tcl file, making it more readable and easier to modify.

The main difficulties found while designing the Tcl extension code derive from the fact that this code has to be written automatically every time the problem type generator is used. An example of the kind of problems that this situation introduces is the implementation of conditions that identify finite element types. One of the features that conditions implemented using the element data group provide is that they can change the element type that GiD will use when meshing. If, for example, a quadrilateral element is assigned automatically because there is a part that is always meshed using it, the instruction to assign the element to entities that belong to the part has to be placed *before* the instruction telling GiD that all entities that will use the element must be meshed with quadrilaterals. For this reason, the `BeforeMeshGeneration` procedure that the problem type generator writes is quite structured, as the order in which the instructions are executed is important.

In addition, a first block of instructions resets everything that can be added automatically by the procedure. This is done because, if the user has made changes in the model, some of the conditions assigned by the procedure in a previous meshing might no longer be correct.

After the model has been reset and everything that wasn't explicitally assigned by the user is erased, the instructions are organized in entities: first all operations related to volume entities are executed, followed by surface entities, lines and, finally, points. This is necessary to ensure that parts and their boundaries are properly identified. To achieve this, the condition that identifies a given part is transferred from entities to their boundaries, step by step: from a volume to its faces, from a surface to its contour lines and from a line to its ends. Each entity block follows the same steps: first, part conditions are checked and transferred from the higher entity, then automatic part conditions are assigned (for example, for parts assigned to the boundary) and finally part interaction is checked. This is done looping over the entities and checking which part conditions have been assigned to each one. If an entity belongs to a single part, the conditions or elements related to that part are assigned. If it belongs to multiple parts, the conditions specified in the input file for that part combination are assigned or, if that particular combination was not defined, a warning is displayed, informing the user that that particular entity won't recieve any conditions.

The last block of the `BeforeMeshGeneration` procedure changes GiD's mesh criteria for entities that need it. In the Kratos example in particular, this is required for all entities with a face condition assigned, so this section is filled with calls to the `meshelement` procedure. This procedure lists all entities with a given condition assigned and instructs GiD to mesh them.

Details of the specific procedures used in the Tcl file can be found in section A.7.3 of the problem type generator manual, included as an appendix to this document.

3.5 The finished program

This section will review the basic characteristics of the current implementation of the problem type generator. The main module of the program is called `problemtype.py`. This file reads the input—a file defining the conditions, materials and other data available to the problem type—line by line and executes

the required instructions to translate what is being read into GiD configuration files. To do this, it relies in the classes introduced by the format-specific module `new_classes.py`. Each one of those classes is based on one of the bassic data classes in `core_definitions.py`, that implement the basic data groups. This provides an easier way to introduce new conditions and allows the main module to recognize them.

3.5.1 Format-specific files

The format-dependent files are organized in three categories. The first of them is the Python file, named `new_classes.py`, which defines the data types specific to the format. As explained in the following section, data types are Python classes used to define and implement concepts like *vector condition* or *problem parameter*. The second type of files are template files, stored in the `files` folder. Those files are the basic versions of the finished problem type files, outlining the basic organization of the problem data in the `.cnd`, `.prb` and `.mat` files, the basic extension procedures in the Tcl file and the input format in the template files. Finally, the definition files outline the input format for each of the new data types. When the problem type definition is read, new conditions are written according to `new_classes.py`. Those conditions, when the user assigns them, will apppear in the input file according to the model that was read from the definition file associated to that condition type.

3.5.2 The `core_definitions` module

The `core_definitions.py` file defines the basic classes used. The most important of those classes are the those that contain the required methods to implement GiD conditions belonging to one of the five basic data types, but it also defines other relevant classes.

The first one of them is the `code` class. All code that has to be written in a problem type file is stored in memory as a code instance. This is needed because sometimes the code will be reused (for example, parts reuse pieces of code from the conditions and elements they assign), but it is also useful to avoid creating incomplete files. All code is stored as soon as it is created, but the files are only written once the entire problem type definition has been read. In this way, files are only written if no errors were found while reading the input, which avoids creating the problem type files and leaving them incomplete when an error is found.

The `code` class is fundamentally a string containing the code, with some additional attributes describing the name of the file where it will be written and the line where it should be added (using a bookmark comment). In addition, the class two relevant methods: one writes the code to the destination file while the other can be used to replace some parts of the code with new strings, which is useful to replace the generic unknown parameters that included in the definition files with the actual values for a given condition.

As several `code` objects are generated for each condition or element defined, there are two classes intended as containers and organizers of code instances. The first of them is the `code_entry` class, which contains all code related to a single condition. It also stores a list of the missing unknown parameters for each code piece and identifies it with a string. This string is used to know which

code instances contain code ready to be written or code that needs to be edited (by replacing the unknown parts) before it can be used. The `code_entry` also defines some additional methods to manage the code instances, such as searching all code instances with a specific identifying string or writing all code that is ready to be written to the corresponding files.

The `code_container` class is used to define a collection of code entries, each one containing the code for a single condition or element. The problem type generator uses an instance of this class (a variable called `code_db`) to manage all the code instances involved in a project. It is basically an organizer for `code_entry` objects: it has methods to activate the functions of multiple `code_entry` instances matching certain criteria, which allow the program to do things like finding all code instances from all defined conditions containing code ready to be written and writing them to the relevant files.

The last part of the `core_definitions.py` file defines the basic data types. Each one of them (`condition`, `element`, `material`, `gendata` for properties and a class for each part type) has two methods, `__init__` and `add`, that define the generation of that type of code. The first one defines some generic variables and creates some common code that will be used to define them in GiD, while the second generates specific code using the information obtained from the input file. Note that all data types are derived from the `base` class, which defines some auxiliary methods used by all of them.

3.5.3 Reading and writing files

The main file of the problem type generator is `problemtype.py`. It reads the input file and uses the classes defined in `core_definitions` and in the format-specific files to create code. It is fundamentally a loop over the lines of the problem type definition file, identifying which kind of information each one provides and sending that line to the relevant class method for processing. To do so, it uses the auxiliary functions defined in `read_tools.py`. The module also generates the Tcl code required to define the interaction between parts using the functions from `tcl_functions.py`, which is added to the `code_db` variable to be written in the Tcl extension file. After the entire input file has been read, if no errors have been detected, the problem type files will be created and the code generated for each condition will be added to them. The final section of the file adds some additional Tcl code, including a custom menu, and erases any unused data books from the problem type files. A book is the way GiD uses to organize the conditions defined in a given problem type. The basic versions of the problem type files used by the problem type generator define books that will be used to separate different kinds of conditions, but if the books remain empty (because no condition of that type is used in that particular Kratos application) they will cause problems when GiD tries to read them.

The `read_tools` module contains several helper functions to read the definition files. One of them, `read_definitions` is specially important because it reads the classes defined the format-specific files (using the functions of the Python module `inspect`) and their definition files, and sends them to the main module, organized by the data type they are derived from. This is fundamental because it allows the problem type generator to use custom classes, giving the user the ability to modify the input format and the problem type's structure with relative ease.

There is an additional module, `file_functions`, used while generating the files. It defines some basic file manipulation functions to read, write and modify the problem type files.

3.6 Creating problem types for Kratos applications

This section will discuss briefly some of the particularities of the problem types created to be used with Kratos applications, both using the old and the new input format. All the problem types generated for the new and the old Kratos formats include some properties by default, which are used to provide information that they will always need.

To solve finite element problems using Kratos a Python file is used. This file loads the required solver applications, feeds them the input data created by GiD and controls the solution of the problem. Problem types created for Kratos can include an example Python file for that Kratos application, and GiD will copy it and provide the values for some of the variables it uses based on the user input. These variables are implemented as GiD general data, and can be accessed and modified while creating a new model by opening the `Problem Parameters` window.

Most of these variables are defined by adding them in the problem type definition file, but there is one that will be included by default in all problem types because all Kratos applications require it. It is the `domain_size`, which takes a value of 2 in 2D problems and 3 in 3D problems. Thanks to the Tcl extesion features, GiD will try to set it automatically based on each model, but this won't be enough in some cases. For example, there are shell problems where the original geometry is 2D, but we are interested in the tri-dimensional behaviour of the shell. In those cases, the domain size should be introduced by the user. To this end, a flag called `Let GiD determine domain size` is added to the `Problem Parameters` window. If it is unchecked, the user will be asked to provide a value for the `domain_size` variable.

Ocasionally, the user will need to modify the Python file for a given problem. The normal behaviour of the problem type is creating a new Python file every time the template files are modified (by clicking the `calculate` button in GiD), which would overwrite any changes made by the user. To avoid this and allow the user to continue using the modified Python file, there is a flag in the `Problem Parameters` window that can change the default behaviour. The `Write a new Python script file for Kratos` flag, if checked, will tell GiD to copy any Python files found in the problem type folder to your madel folder. By unchecking it, the modified version of the Python file won't be overwritten every time the model is modified.

A third flag included by default in the `Problem Parameters` window is the `Transfer materials to lower entities`. If checked, this flag will make GiD transfer materials assigned to volume entities to surfaces, lines and points (in 3D problems) or from surfaces to lines and points (if no volumes are found). This guarantees that Kratos conditions and elements assigned to those surfaces or lines will recieve the same material as the volume. By itself, GiD assigns to an element the material of the entity it belong to. This is a problem if the element

is a Kratos condition assigned, for example, to the faces of a volume. As the user typically assigns the material only to the volume, its surfaces won't have a material by themselves. The problem type generator makes the surfaces inherit the volume's material to simplify the process of assigning materials, but there are cases in which the volume and the surface need to have different materials. In those cases, this flag should be unchecked and the materials assigned materials manually to all entities.

3.7 Further modification

Modifying the problem type generator to work with new input formats can be done in different levels, depending on how different is the new format from the existing ones. The simplest level of modification is adding new types of conditions, elements or materials to an existing problem type format. To implement them, the user would have to create a new custom data type, defined as a class in the `new_classes` module, and create a definition file that contains the information that GiD needs to make it available to the user and include it with the correct format in the input files for the solver.

A second level of customization involves modifying the input format itself: the number of required files or the way in which elements or nodes are introduced to the solver. To do this, the user would need to modify the template files that the problem type generator uses, which are stored in the `files` folder for each input format. Those files are basic versions of GiD template files, which are filled with informtion relative to the conditions and other problem-specific data when the problem type generator is executed.

Finally, if the existing customisiation possibilities are not enough, or if the user needs to implement a new type of data that the problem type generator can't handle, there is the option of modifying the problem type generator source code. This could be needed, for example, if a solver required input that could not be generated by a class derived from one of the basic data types.

Chapter 4

Practical application

To conclude the discussion of the design and implementation of the problem type generator, this chapter will show a problem implemented using a problem type created with this program. As the aim of this section is just to provide a proof that the program is useful, the analysis of the results won't be very deep.

The problem analyzed is a test case described in [25], where a fluid flows inside a curved cylinder. The behaviour of the structure is modeled with non-linear elastodynamic equations (St. Venant-Kirchoff material), while the fluid is described by the incompressible Navier-Stokes equations with an ALE formulation. The geometry of the problem is defined by

- The fluid vessel has a radius of $0.5\,cm$ and its axis describes a curve of 180^o with a radius of $2\,cm$.
- The fluid is surrounded by a structure with a thickness of $0.1\,cm$.

The physical parameters are the following:

- For the fluid, viscosity $\mu = 0.035\,poise$ and density $\rho^f = 1\,g/cm^3$.
- The structure is defined by density $\rho^f = 1.2\,g/cm^3$, Young modulus $E = 3 \times 10^6\,dynes/cm^2$ and Poisson ratio $\nu = 0.3$.

Both systems are initially at rest. The structure is clamped at the inlet and the outlet. An over-pressure of $1.3332 \times 10^4\,dynes/cm^2$ is imposed, over the inlet boundary, for 5×10^{-3} seconds. The time step is fixed at $\Delta t = 10^{-4}$ seconds.

Once both domains were defined in GiD, the files generated were used by the Kratos fluid-structure interaction application to solve the problem. The mesh used for the fluid domain is represented in figure 4.1.

Once the calculation ends, the solver provides the results of the fluid domain, which are shown for different time steps in figures 4.2 to 4.7. The results show the expected behaviour, where the temporary pressure applied over the inlet produces a pressure wave that propagates along the fluid domain.

Figure 4.3 shows a time step just after the inlet over-pressure has stopped. Subsequent figures show the displacement of the pressure wave unit it reaches the outlet and exits the domain. The observed behaviour of the fluid matches the solution obtained in the reference.

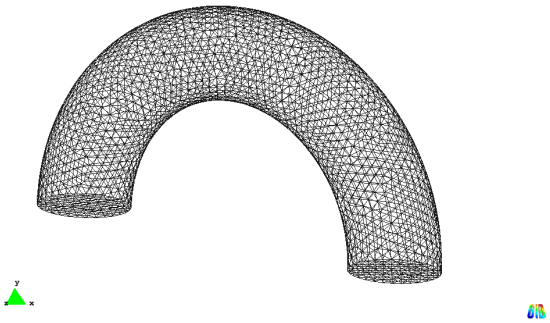

Figure 4.1: Mesh for the fluid vessel.

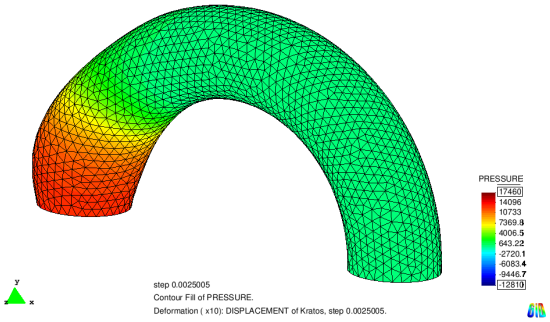

Figure 4.2: Pressure and solid deformation (exaggerated) of the fluid vessel after 0.0025 *sec.*

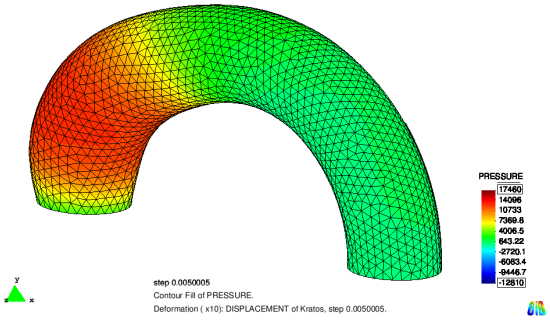

Figure 4.3: Pressure and solid deformation (exaggerated) of the fluid vessel after 0.0050 *sec.*

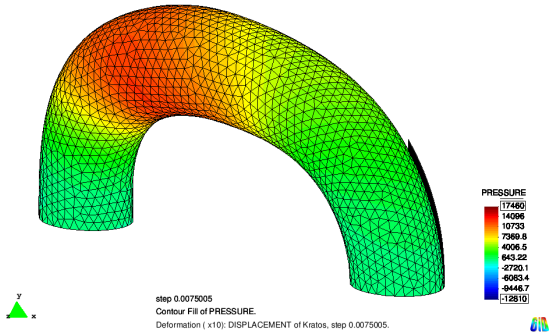

Figure 4.4: Pressure and solid deformation (exaggerated) of the fluid vessel after 0.0075 *sec*.

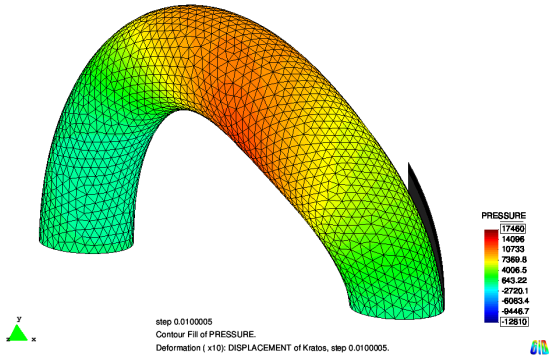

Figure 4.5: Pressure and solid deformation (exaggerated) of the fluid vessel after 0.0100 *sec*.

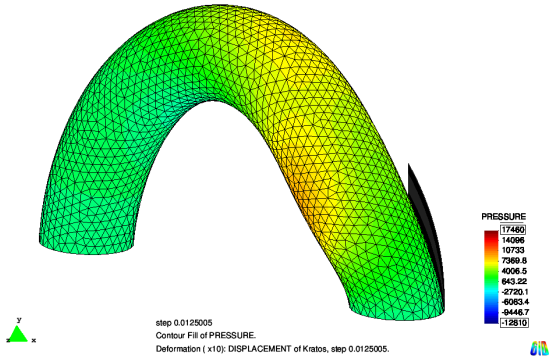

Figure 4.6: Pressure and solid deformation (exaggerated) of the fluid vessel after 0.0125 *sec*.

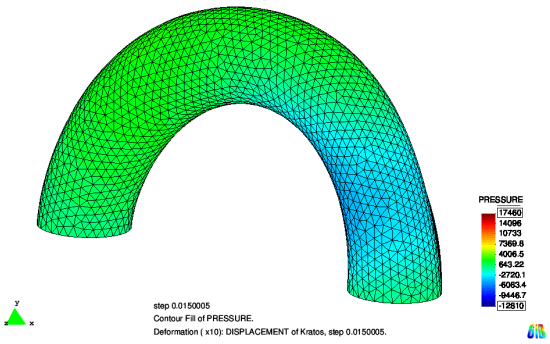

Figure 4.7: Pressure and solid deformation (exaggerated) of the fluid vessel after 0.0150 *sec*.

Chapter 5

Conclusions

In the first part of this project, different methods that can be used to solve the fluid-structure interaction problem have been analyzed. The most accurate solution method is the monolithic approach, that combines the velocity of both the fluid and the structure and the pressure of the fluid domain in a single system. The solution provided by this method is good, as no additional hypotheses or simplifications have been assumed to uncouple the problem, and requires a relatively small amount of iterations. Unfortunately, this method has little practical application, as each individual iteration is exceedingly expensive. This is a consequence of the fact that the system that has to be solved is large, as all variables are solved at the same time, and badly scaled, as it contains terms associated to different physical magnitudes and different constitutive equations. This last point is critical, as a good scale factor is a requisite for iterative solvers to perform adequately. For large problems, iterative methods are the only viable choice as the alternative, direct solvers, requires a number of operations of the order of the cube of the system's dimension.

On the other hand, segregated schemes, where each variable is solved separately, result in smaller and better conditioned linear systems, where iterative solvers can perform much better and, as a result, are more desirable for practical problems. In the case of fluid-structure interaction, there are two different divisions to be made. On one hand, the fluid equations have to be uncoupled to allow for a separate solution of the pressure and the velocity. This is achieved using fractional step methods, introduced in section 1.5. On the other hand, the velocity has to be split between the two domains. In section 1.7.2, the segregated Dirichlet-Neumann scheme was introduced to allow for a separate treatment of the velocities.

In segregated problems, a key point is to simulate properly the behaviour of the interface between the two domains. In the case of Dirichlet-Neumann uncoupling, the failure to do so results in a scheme that is unable to solve the cases where the fluid is heavier than the structure. For this reason a modification to the standard formulation was introduced in section 1.7.3.

Another important factor in the practical implementation of fluid-structure interaction problems was analyzed in chapter 2. With the examples provided, the convenience of approximating the term $\mathbf{D}\left(\mathbf{M}^f\right)^{-1}\mathbf{G}$ with a Laplacian matrix in the mass conservation equation was analyzed. It was concluded that not doing so resulted in faster convergence (although with more expensive individual

iterations) and that the use of the Laplacian matrix can result in convergence problems in some cases, such as the one studied in the second example.

The second part of the project involves a part of the solution of finite element problems that is often overlooked by civil engineers. It is understandable, as it is secondary to the solution of the problem. The main concern of a finite element developer is to devise a scheme that can solve the problem at hand, and the pre-process is secondary to that. However, in large problems, it soon becomes apparent that defining the geometric and mathematical model in a practical way is a considerable challenge. The tools that tackle it have to be carefully designed to ensure that they are versatile enough so that they can accommodate to problems with some characteristics that are different than those its designer had in mind when writing it.

Until now, the performance of the problem type generator has been satisfactory. Some of the problem types it has generated have already been used to solve test problems as, for example, the one introduced in chapter 4. In those problems, the additional Tcl features implemented have proved useful to simplify the pre-process of the input for Kratos applications.

It could be said, however, that it is too early to tell if the problem type generator is a success, as it has been designed with the possibility of adapting it to new problems and formats in mind. In this sense, it the problem type generator will only prove useful if, some time from now, it can be adapted to deal with changes in the input format.

Bibliography

[1] Kratos. `http://kratos.cimne.upc.es/`.

[2] C. Felippa, K.C. Park, and C Farhat. Partitioned analysys of coupled mechanical systems. *Comput. Methods Appl. Mech. Eng.*, (190):3247–3270, 2001.

[3] Ch. Forster. *Robust methods for fluid-structure interaction with stabilised finite elements*. PhD thesis, Institut fur Baustatik und Baudynamik, Universitat Stutgart, 2007.

[4] Jean Donea and Antonio Huerta. *Finite Element Methods for Flow Problems*. J. Wiley and Sons, 2002.

[5] D.P. Mok and Wall W.A. Partitioned analisys schemes for the transient interaction of incompressible flows and nonlinear flexible structures. In *Trends in computational structural mechanics*. Barcelona, 2001.

[6] U. Kuttler and W.A. Wall. Fixed-point fluid-structure interaction with stabilised finite elements. *Journal of Computational Mechanics*.

[7] J. Vierendeels, L. Lanoye, J. Degroote, and P. Verdonck. Implicit coupling of partitioned fluid-structure interaction problems with reduced order models. *Computers and Structures*, (85):970–976, 2007.

[8] O.C. Zienkiewicz, R.L. Taylor, and P. Nithiarasu. *The Finite Element Method for Fluid Dynamics*. Butterworth-Heinemann, 2005.

[9] Riccardo Rossi. *Light-weight structures — Numerical Analysis and Coupling Issues*. PhD thesis, University of Padova, Italy, 2005.

[10] M.A. Fernández, J.-F. Gerbeau, and C. Grandmont. A projection semi-implicit scheme for the coupling of an elastic structure with an incompressible fluid. *International Journal for Numerical Methods in Engineering*, (69):794–821, 2007.

[11] S. Badia, A. Quaini, and A. Quarteroni. Splitting methods based on algebraic factorization for fluid-structure interaction. *SIAM Journal on Scientific Computing*, (30):1778–1805, 2008.

[12] P. Causin, J.-F. Gerbeau, and F. Nobile. Added-mass effect in the design of partitioned algorithms for fluid-structure problems. *Comput. Methods Appl. Mech. Eng.*, (194):4506–4527, 2005.

[13] S. Badia, F. Nobile, and C. Vergara. Fluid-structure partitioned procedures based on robin transmission conditions. *Journal of Computational Physics*, (227):7027–7051, 2008.

[14] W. Dettmer and A. Peric. A computational framework for fluid-structure interaction: Finite element formulation and applications. *Comput. Methods Appl. Mech. Eng.*, (195):5754–5779, 2006.

[15] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method*. Butterworth-Heinemann, fifth edition, 2000.

[16] Ramon Codina. Pressure stability in fractional step finite element methods for incompressible flows. *Journal of Computational Physics*, pages 112–140, 2001.

[17] T.J.R. Hughes. *The Finite Element Method, Linears Static and Dynamic Finite Element Analysis*. Dover, 2000.

[18] R. Codina and J. Blasco. A finite element formulation for the stokes problem allowing equal velocity-pressure interpolation. *Comput. Methods Appl. Mech. Eng.*, (143):373, 1997.

[19] R. Rossi, S. Idelsohn, E. Oñate, and F. del Pin. Improving a strongly coupled method for FSI by a simple approximation of the prssure tangent matrix. Submitted to CMAME.

[20] Eugenio Oñate. *Cálculo de Estructuras por el Método de Elementos Finitos*. CIMNE, 1995.

[21] The Python Programming Language. `http://www.python.org/`.

[22] GiD Introduction. `http://gid.cimne.upc.es/intro/`.

[23] GiD 9.0 reference manual. `http://gid.cimne.upc.es/support_team/gid_toc/gid_toc.html`.

[24] GiD ProblemType tutorial. Available from `http://gid.cimne.upc.es/support_team/su05.html`.

[25] E. Burman and M.A. Fernández. Stabilization of explicit coupling in fluid-structure interaction involving fluid incompressibility. *Comput. Methods Appl. Mech. Eng.*, 2008.

Appendix A

The problem type generator manual

A.1 Introduction

The problem type generator is a tool to automate the process of writing problem types for GiD. In the GiD context, a problem type is a collection of files used to configure GiD for a particular type of analysis. It contains the information required to create a model and input it to a solver, from the boundary conditions and finite element types that will be available and how they will be presented to the user to the format in which the data has to be written so the solver can read it.

The problem type generator was developed as a support tool for kratos, a framework for building multi-disciplinary finite element programs. As such, kratos can be used to solve different physical problems, which involve different variables, different boundary conditions and different materials. Each of these problems requires its own GiD problem type, but the input files that GiD generates for them always have the same basic format. In this situation, a program able to write the problem types from a template, requiring minimal input from the coder of the solver, can be a helpful tool.

A secondary objective of the problem type generator is to use GiD's extension capabilities to automate several tasks, such as aligning the normals of the model's surface entities or explicitly assigning a value for a condition to points where lines with different values for that condition meet. This has the double advantage of reducing the time required to create a model and eliminating common causes of beginner's mistakes.

This manual explains how to use the problem type generator, from its basic functionality of creating a problem type for kratos, to more complex tasks like defining new types of conditions and elements or modifying the input format.

A.2 Generating a problem type: the basics

To create a new problem type for kratos, a text file, called the *input file*, is required. In its simplest form, the input file contains basic information about

the problem such as which conditions will be available or which are the relevant physical properties for each material, but the writer can provide additional information to automate certain aspects of the input of new models, such as conditions to be assigned by default to certain parts of the model.

The first part of the input file specifies the name of the problem type we want to generate and the folder which contains the template files. For example:

```
PROBLEMTYPE fsi_structure
DEFINITION FOLDER fsi_structure_problemtype
```

will create a problem type called *incompressible_fluid* based on the files found inside the `fsi_structure_problemtype` folder. Note that the program expects to find both the input file and the folder inside its main directory, `problemtype_generator`.

The following lines define the data that will be available when creating a model. For the problem type generator, there are five basic types of data: conditions, elements, materials, properties and parts, which will be described in detail in the following sections. Each one of these data types has been implemented as a class that can be used to generate the information needed to define and use it in GiD. It must be noted that the base classes don't contain all the required input. This is the main reason why a definition folder is required: it contains the actual format of the files we want to generate. For example, if we want to define a velocity boundary condition, the problem type generator will look for the format in which a vectorial condition must be given to the solver in the definition folder and use that information to generate the velocity boundary condition. The main advantage of this implementation is that each one of the base classes can be used to write more specific information. The condition base class defines the methods used to read a condition definition from the input file and include all the required information in the problem type files. Then, this base class can be used to define several derived classes (for example flags, scalar conditions or vector conditions) which provide the desired format.

The problem type generator reads the input file one line at a time. Each line of the input file represents a command. Commands are sequences of words, a word being something delimited by white spaces. Most commands in the input file have the same structure: the first words identify the kind of data we want to define, followed by the name we want to give it. If it is a condition, element or part, then we must specify the GiD entities (points, lines, surfaces or volumes) over which it can be assigned. Finally, we write any additional information required to define it, which is different for each data type but generally includes the default values.

Most data types can be defined using a single command, but those based on the material or part classes usually need some additional lines. Those extra commands must be written before a new material or part is defined (otherwise the program will believe that they contain information related to the newer one), so it is recommended that they are written immediately after the main definition line. In addition to this restriction, it must be noted that the definition of parts involves other conditions or elements and that these conditions or elements must have been defined before they can be used by the part. The easiest way to avoid problems with missing or mismatched definitions is to write all commands defining the same material or part together and to write part definitions last.

Another possible source of errors is using the same name to define two different materials (even if they belong to different material families) or a condition and a part that assigns the condition to, for example, the problem's boundary. In general, repeating names in definitions should be avoided, as it causes errors that can be difficult to identify.

A final remark: while reading the input file, the problem type generator can find and understand two special "words": a single `#` character at the beginning of a line comments out the whole line, while a single `\` character at the end of a line means that the command continues in the following line.

A.3 Generating problem types for kratos

This section explains how to define kratos conditions, elements and materials, as well as some properties that can be used to define Python variables. The commands described here are defined in a Python file read from the folder given as `DEFINITION FOLDER`, which means that some of them can be missing or have a different format depending on the definition folder. Likewise, some additional conditions could be available for other problems. When writing the existing definition files, I have tried to use the same commands when possible but, in case of doubt, all commands available will be defined as the `call` attribute of their class, which can be found in the `new_classes.py` file from each folder.

Note that what is defined in this section can be used to generate problem types for the new kratos input format, which uses a single input file with extension `.mdpa`. There is a definition folder that can be used to generate problem types for the old kratos format (with five different files), but it doesn't have as many features available. To use the problem type generator to write problem types in the old format, refer to section A.8.

The different commands available will be grouped here by the kind of input they generate: properties, elements, conditions, nodal values, elemental values and conditional values.

A.3.1 Materials

The materials data type is used to create GiD materials, which store the physical properties needed to define the materials used in the model and a library of materials with the corresponding values for these properties. The problem type generator implements materials by defining them in "families". First a material type is defined, providing the name and type of the physical properties involved, and then one or more example materials can be added to the problem type, giving values for those properties. Each material "family" is stored in a different *book* in the `.mat` file, which means that it will have its own menu option. Some example material definitions follow:

```
DEFINE MATERIAL Structure DENSITY SCALAR YOUNG_MODULUS SCALAR \
POISSON_RATIO SCALAR BODY_FORCE 3DVECTOR THICKNESS SCALAR
ADD MATERIAL Aluminium 2700 70000000000  0.3 (0.0,-26487.0,0.0) 1.0
ADD MATERIAL Steel     7600 210000000000 0.3 (0.0,-74556.0,0.0) 1.0
ADD MATERIAL Concrete  2500 30000000000  0.3  (0.0,-24525.0,0.0) 1.0

DEFINE MATERIAL Fluid DENSITY SCALAR VISCOSITY SCALAR
ADD MATERIAL Air 1.2 0.000017
ADD MATERIAL Example 1 1
```

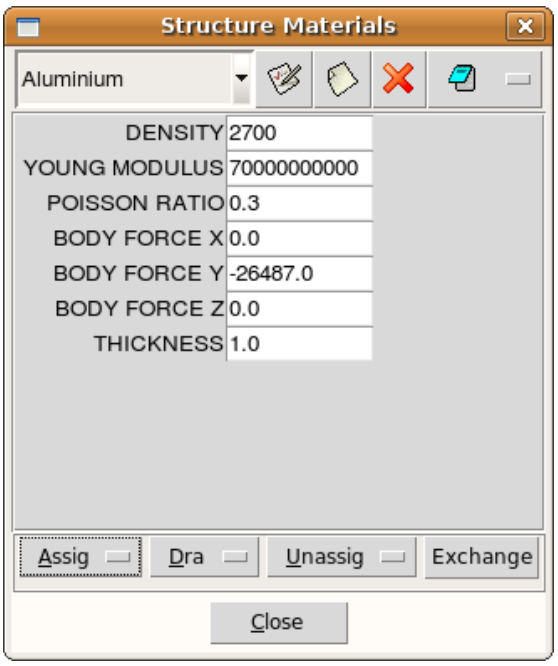

Figure A.1: A material window

Property type	Example input
SCALAR	0.0
2DVECTOR	(0.0,-9.8)
2X2MATRIX	((1.0,0.0),(0.0,1.0))
3DVECTOR	(0.0,-9.8,0.0)
3X3MATRIX	((1,0,0),(0,1,0),(0,0,1))
TEXT	alphanumeric_characters

Table A.1: Property names and their input format for materials

In each one of the examples, the first instruction, beginning with `DEFINE MATERIAL`, creates a material family (called `Structure` in the first example and `Fluid` in the second) and defines some physical properties for them. Subsequent commands create materials that belong to that family, specifying the values of the physical properties (in the same order they where defined). Figure A.1 shows the result of the first example. Note that the physical properties can be scalars, vectors or matrices. You must specify which kind of property you want to create by adding one of the commands from table A.1. Then, when you create materials you must introduce the default values for each property using the same format as the examples in that table. It is specially important to avoid introducing white spaces when giving the default values, as the program won't be able to read them properly.

A.3.2 Elements and conditions

Elements and conditions are both defined using the problem type generator's element class, and the commands to add them are similar, so they will be described together in this section. The basic element class from the problem type generator can be used for any kind of data that is applied to the elements of the mesh. It has an important limitation: the GiD conditions it generates can only be applied to a single type of entities (such as lines or surfaces). This shouldn't be a problem because its intended use is to define finite elements, which typically can only be used in a single entity type (for example, a triangle element can be used to mesh surface entities, but not lines or volumes)

Some example commands to generate elements can be found after this paragraph. As you can see, to define an element for kratos you start the line writing `ELEMENT`, followed by the name you want to give it and the entity over which it can be applied (only `line`, `surface` or `volume`, elements applied over point entities have their own syntax). Finally, there is an additional parameter, which can be used to tell GiD to mesh entities that have this element assigned with a specific element type.

```
ELEMENT TotalLagrangian2D3N surface
ELEMENT TotalLagrangian3D4N volume Tetrahedra
ELEMENT TotalLagrangian3D8N volume Hexahedra
ELEMENT IsoShellElement surface
```

The element type must be one already known by GiD: "Triangle", "Quadrilateral", "Linear" and "Circle" can be used for surface elements, while "Tetrahedra", "Linear", "Hexahedra", "Prism", "Only Points" or "Sphere" can be used for volume elements (the names are case-sensitive). Note that if you use this additional parameter, the element type choice will always override changes to the element type made by the user when creating a new model, and that no check is made to verify that the elements used in different entities are compatible, which could lead to problems if, for example, you attempt to mesh a volume with tetrahedra and one of its faces with quadrilateral elements.

The syntax to generate conditions is the same as for elements, only that the line must start with `FACE CONDITION` instead of `ELEMENT`. Some examples follow:

```
FACE CONDITION Face2D line
FACE CONDITION Face3D3N surface
FACE CONDITION Face3D4N surface Quadrilateral
```

The difference between kratos elements and conditions is that usually the entities over which conditions are applied are not meshed by GiD by default. For example, on a 2D problem, elements will be applied over surfaces, while conditions can also be assigned to lines and points. Conditions defined with the problem type generator add some Tcl code to the problem type that will tell GiD to mesh them automatically without the user's intervention.

An important remark is that GiD (at least until version 9.1.1b) doesn't mesh points by itself. To generate point elements from GiD conditions assigned over point entities some additional code is required, which means that the problem type generator needs to create point elements and conditions from their own, different template. To generate a point element or condition, write `POINT ELEMENT` or `POINT CONDITION` followed by its name in the template file. Note that this is only important for elements and conditions assigned manually to points, to mesh

a volume with point elements you should use the standard `ELEMENT` command, applied over `volume` entities and with the `Only Points` parameter.

```
POINT CONDITION PointForce3D
POINT CONDITION PointForce2D
```

A.3.3 Nodal Values

Nodal data is the input that kratos requires to identify conditions assigned to nodes of the mesh. From a GiD point of view, nodal data can be assigned to all types of entities (points, lines, surfaces or volumes). For each nodal data type defined in the input file, GiD will generate a list of the mesh nodes with a given value for it, specifying if this value is fixed or can change during the calculations.

There are three types of nodal data currently implemented for kratos: flag conditions, scalar conditions and vector conditions. To add them to a problem type, you must write their names and default values in the input file. Two different default values must be given for each condition: the actual value of the physical magnitude and if it will be considered constant during the analysis (for scalars and vectors, this means that the same condition can be used to represent both initial conditions and boundary conditions). Some examples of condition definitions can be seen in the following lines:

```
VECTOR CONDITION DISPLACEMENT point line surface volume fixed 0.0 0.0 0.0
SCALAR CONDITION PRESSURE point line surface volume free 0.0
FLAG CONDITION IS_BOUNDARY point line surface 1.0 2.0 3.0 4.0 5.0
```

The first line defines a vector condition called `DISPLACEMENT`, which can be applied over point, line, surface and volume entities. Its default values (which the user will be able to edit from GiD's Conditions window) are "fixed", which identifies it as a boundary condition (as opposed to an initial value) and a velocity of zero in the direction of each of the three axes of the model. The problem type generator can use this information to generate a GiD window like the one shown in figure A.2.

The second line creates a scalar condition called `PRESSURE`, which will be available to point, line, surface and volume entities, with a default value of 0.0 and marked as an initial value by default. Finally, the third line creates a flag condition called `IS_BOUNDARY`, which is used to identify certain parts of the model's boundary. This command is somewhat different from the previous ones, as all available values for the flag must be given (instead of only the default one). This flag will be able to take five values, from 1.0 to 5.0, and 1.0 (the first value in the list) will be selected by default. Note that this line does not specify if the condition will be "free" or "fixed" by default. In this situation, the problem type generator will assume that it will be "fixed"[1].

There are two special words that can be used to reduce typing when introducing entities. The first one of them is `all`, which is equivalent to `point line surface volume`, while the second one is `"`, which means "use the same entities as the last definition". This means that the following code is equivalent to the last example:

[1] For those interested in implementing new condition types, the way in which the additional input (everything after the entities) is read is defined by the parseinput method of the new data class.

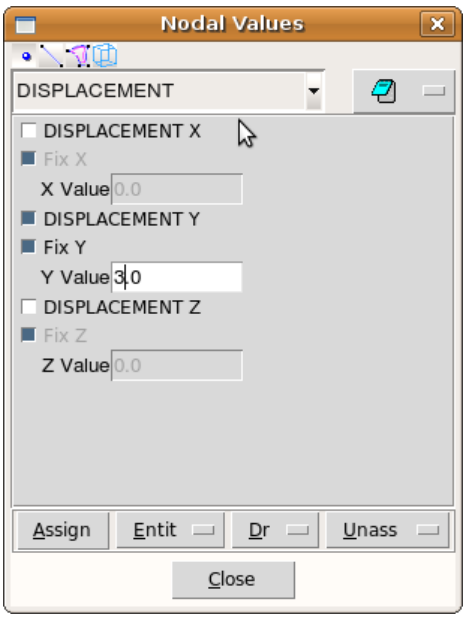

Figure A.2: A vector condition produced by the problem type generator.

```
VECTOR CONDITION DISPLACEMENT all fixed 0.0 0.0 0.0
SCALAR CONDITION PRESSURE " free 0.0
FLAG CONDITION IS_BOUNDARY point line surface 1.0 2.0 3.0 4.0 5.0
```

A.3.4 Elemental and Conditional Data

Kratos elemental and conditional data is similar to nodal data, but it is applied to elements or conditions instead of to the nodes. There is another difference between them: elemental and conditional values don't change when the problem is solved, so they don't require the "fixed" or "free" parameter.ental and conditional data is similar to nodal data, but it is applied to elements or conditions instead of to the nodes. There is another difference between them: elemental and conditional values don't change when the problem is solved, so they don't require the "fixed" or "free" parameter.

They can be defined using the similar commands as nodal values, replacing `CONDITION` for `ELEMENTAL VALUE` to define elemental data or `FACE VALUE` to define conditional data.

```
VECTOR ELEMENTAL VALUE <name> <entities> 0.0 0.0 0.0
SCALAR ELEMENTAL VALUE <name> <entities> 0.0
FLAG ELEMENTAL VALUE <name> <entities> 1.0 2.0 3.0 4.0 5.0

VECTOR FACE VALUE DISPLACEMENT <name> <entities> 0.0 0.0 0.0
SCALAR FACE VALUE PRESSURE <name> <entities> 0.0
FLAG FACE VALUE IS_BOUNDARY <name> <entities> 1.0 2.0 3.0 4.0 5.0
```

The "shortcuts" used to define entities in nodal values can also be used for conditional and elemental data.

A.3.5 Properties

The property class is used to create general data, which is used to define data related to the entire model, such as the direction of gravity, or parameters for the solver, such as the time step. The current implementation of general data in the problem type generator is to assign the values given in GiD to variables and writing them to an auxiliaryy Python file, so they can be imported and read in the Python script file. There is one limitation: in the current implementation, only strings or scalar variables can be used. The input required is `PROPERTY <NAME> <DEFAULT VALUE> <VARIABLE>`. `<NAME>` is the name of the property in GiD, which will appear in the General Data Window, `<VALUE>` is its default value and `<VARIABLE>` is the name of the Python variable we will assign the value to. Some examples follow:

```
PROPERTY Time_step 0.1 Dt
PROPERTY Max_time 0.1 max_time
PROPERTY Number_of_steps 10 nsteps
```

Those variables will be written to a file called `name_var.py`, where `name` is the name of your problem type. The intended use of this Python file (which will be explained in detail in section A.5) is to be imported in the Python application file when solving the problem.

There is an additional property type which can be used to select a file. It was designed for the fluid-structure interaction problem, in which the structure and the fluid part are introduced as two different models. In that problem, this property is used to give the path to the structure `.mdpa` file as a variable in the fluid's Python file. Kratos is then run from the fluid's folder, knowing that it will find the fluid file in the current folder and the structure file in the path stored in a Python variable.

To create this property, write

```
FILE SELECTION structure_file structure.mdpa str_file
```

This will create a GiD property called `structure_file` with a "select file" button to navigate through your folder structure. `structure.mdpa` is the default value of the property and `str_file` is the name of the Python variable.

A.4 Additional commands

Besides the commands already defined, there are some additional commands that can be used to provide a certain degree of automation to the task of introducing the model to GiD. They are not specific to kratos, so they can be used regardless of the definition folder used.

A.4.1 Parts

For the problem type generator, parts are a special type of conditions that control the assignation of other conditions and elements. Typically, parts are used to identify a group of entities that represent the same physical reality in a model. For example, in fluid problems, a part can be defined to represent an inlet (where the fluid enters the domain at a given velocity) and another part can be used to define the solid contour of the domain, where the fluid's velocity will be zero (as the fluid can't cross it). Assume that we are defining

a 2D model: a group of lines will be the inlet, with a fixed velocity condition, and some other lines will define a solid wall, with a zero velocity condition. There will be some points where inlet lines and wall lines meet. As the lines have different velocity conditions, GiD won't assign a velocity condition to that point. Using the problem type generator, we can define which condition must be assigned to points where inlet and wall parts meet and have GiD assign them during mesh generation. The parts described in our example could be defined in the input file as:

```
DEFINE MODEL PART Inlet point line surface LOWER
ADD CONDITION VELOCITY fixed 0.0 0.0 0.0
DEFINE MODEL PART Wall point line surface LOWER
ADD CONDITION VELOCITY fixed 0.0 0.0 0.0
```

This code creates an inlet part and a wall part for point, line and surface entities and makes both of them assign a velocity condition (note that $(0, 0, 0)$ is the default velocity value, but that the user will be able to change it when assigning them). The `LOWER` parameter means that the part will be transferred from surfaces to their boundary lines and from lines to their end points when generating the mesh. Note that, in our previous example, what allows us to detect that the point belongs to an Inlet line and to a Wall line is that both lines transfer their respective conditions to it. If you don't want the part to transfer to lower entities, replace `LOWER` with `NO LOWER`. Note that this last parameter is optional. If nothing is found, `LOWER` will be assumed.

The `ADD CONDITION` line has some variants: `ADD ELEMENT` to assign data types derived from the element template, `ADD 2D CONDITION` or `ADD 2D ELEMENT` to assign the condition only in 2D problems and `ADD 3D CONDITION` or `ADD 3D ELEMENT` to assign them only for 3D problems.

By default, the problem type generator will try to assign elements and conditions to all entities over which both the part and the condition are defined. In some cases it can be more desirable to assign the condition only to a certain types of entities instead of assigning it to all available. In such cases, add the name of the desired entities after the name of the condition.

An important remark about 2D and 3D: for the purpose of assigning conditions, the code generated by the problem type generator assumes that a problem is 2D if it can't find any volume entity in the model and 3D if there are volume entities. Some problems (shell problems for example) will be assigned the "2D" conditions even when they are tri-dimensional, because they are made of surface entities. The main advantage of this behaviour is knowing that the body elements in a "2D" problem will always be surface elements and its boundary will always be made of lines, while the body elements in a "3D" problem will be volume elements and its boundary will be made of surfaces.

Note that the part and the conditions can be defined for different entities. For example, a part defined for lines and surfaces can assign a line element, which is only defined for lines, and a surface element, only defined for surfaces. The part only will try to assign its conditions (or elements) to entities where both the part and the condition are available.

Another feature of parts is that they will never override information introduced by the user. When a part assigns a condition to an entity, it checks that the user has not assigned that condition to the entity first, and avoids overwriting it. This is more useful when there are parts assigned by default, which we will discuss in the following lines.

There are other types of parts which can be assigned by default to specific areas of the model, without the user's intervention. One of them is `BOUNDARY PART`, which is automatically assigned to the entire boundary of the model. This part has an optional feature, which can make the all normals of the boundary entities to point in the same direction. Let's see an example definition:

```
DEFINE BOUNDARY PART Boundary
ADD CONDITION DISPLACEMENT fixed 0.0 0.0 0.0
ADD 2D ELEMENT CONDITION2D2N
ADD 3D ELEMENT CONDITION3D
```

This will assign a zero displacement condition and a face condition (a line condition, `CONDITION2D2N`, for "2D" problems or `CONDITION3D`, a surface condition, for "3D" problems) to the model's boundary. Additionally, if we want to align the normals, we must replace the first line for one of the following:

```
DEFINE BOUNDARY PART Boundary OUTWARDS NORMALS
DEFINE BOUNDARY PART Boundary INWARDS NORMALS
```

A second type of default model part is `DEFAULT BOUNDARY PART`. It will be assigned to all boundary entities except those with some other parts assigned. In the previous incompressible fluid example, the boundary entities will define either an inlet, an outlet or the solid wall. We could let the user select the inlet and the outlet and assume that everything else is a wall. To do so, we would first define inlet and outlet parts with `DEFINE MODEL PART` commands an then define the wall as:

```
DEFINE DEFAULT BOUNDARY PART Wall NOT Inlet Outlet
ADD CONDITION VELOCITY fixed 0.0 0.0 0.0
```

Both `BOUNDARY` and `DEFAULT BOUNDARY` are always transferred to lower entities (there isn't a `NO LOWER` option as in model parts)

Another available model part is `ALL ENTITIES PART`, which is automatically assigned to all entities of the selected types. The structure of the command used to define this part follows the example:

```
DEFINE ALL ENTITIES PART All_surfaces surface
ADD 2D ELEMENT NDFluid2D
```

In this case, a part called `All_surfaces` is created to assign the element `NDFluid2D` to all surfaces in a "2D" problem.

Note that the values given for conditions assigned using default parts such as `BOUNDARY` and `DEFAULT BOUNDARY` are not visible to the user: these values will be always assigned and the user won't be able to modify them.

A.4.2 Part interaction

The final section of the input file defines the interaction between parts: what will happen when an entity belongs to more that one part (because it has more than one part assigned). This usually happens in the boundary between two parts: If two surfaces share a line any belong to different parts, that line will belong to both parts[2] and we can define a special behaviour for it: it can inherit the conditions from either or both surfaces or a completely different set of conditions.

[2] Assuming that both parts are automatically transferred to lower entities

Part interaction is defined in two steps. First, parts are grouped. Parts that don't belong to the same group just "ignore" each other, so an entity with two parts from different groups will just inherit the conditions from both parts. Parts don't belong to any group are considered a group by themselves, so all entities with the part assigned will automatically receive all conditions and elements related to it. To define a group, write the following line:

```
GROUP Inlet Outlet Wall IN point line surface
```

This instructs the problem type generator to treat the Inlet, Outlet and Wall parts as a group in point, line and surface entities. Note that a part can belong to different groups for different entities (a part belonging to two groups for the same entity would cause problems, though). After defining a group, the interaction between its parts must be specified. This is done with a command starting with the structure "`<Part1 Part2 ... PartN> ASSIGN`" (where `Parti` is a part from the group), followed with one or more assignations of conditions or elements. There are two ways to assign conditions and elements: "`<Condition> FROM <Part>`" assigns `<Condition>` with the value it has in `<Part>` (one of the parts that appears in the same line before `ASSIGN`) and "`<Condition> <Value>`" assigns `<Condition>` with a value of `<Value>`. Some examples:

```
GROUP Inlet Outlet Wall IN point line surface
Inlet Wall ASSIGN VELOCITY FROM Inlet
Outlet Wall ASSIGN PRESSURE FROM Outlet VELOCITY fixed 0.0 0.0 0.0

GROUP Velocity_Inlet No_Slip_Condition Slip_Condition  \
Pressure_Inlet/Outlet IN point line surface
Pressure_Inlet/Outlet No_Slip_Condition ASSIGN IS_BOUNDARY 4.0
Pressure_Inlet/Outlet Slip_Condition ASSIGN IS_BOUNDARY 4.0
Velocity_Inlet No_Slip_Condition ASSIGN IS_BOUNDARY 1.0
Velocity_Inlet Slip_Condition ASSIGN IS_BOUNDARY FROM Velocity_Inlet
```

Note that in those examples not all possible combinations of parts are defined (including combinations of three or more parts). This is because the remaining combinations are unlikely in practical problems. However, if one of such combinations is found in a given entity from a model, GiD will display a message warning that an unexpected combination of parts has been found and that no conditions have been assigned to that entity. In this case, if a condition has to be assigned, the user will have to assign it to that entity manually and mesh again.

A.4.3 The `OPTION` command

There is a special command that can be used in combination with model parts to assign an element or condition to a certain group of entities automatically. This is useful, for example, to assign an element to all entities of a certain type, or a condition to the entire boundary of the model. The actual element or condition that will be assigned can be chosen from a list found in the c menu option.

In the following example, we want an element assigned automatically to the body of model (which will be composed either of surface entities or volume entities). With this code, assuming that all involved elements have been defined before in the input file, we can just draw the model and chose the element in

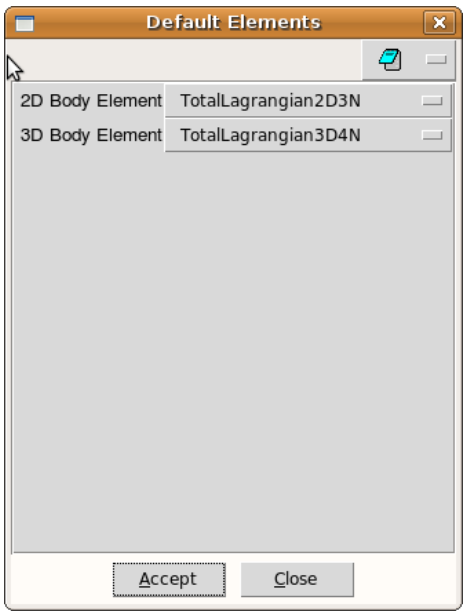

Figure A.3: The Default elements window for an example problem type.

the `Default elements` menu, and it will be automatically assigned when the mesh is generated.

```
OPTION 2D_Body_Element surface TotalLagrangian2D3N IsoShellElement
OPTION 3D_Body_Element volume TotalLagrangian3D4N TotalLagrangian3D8N

DEFINE ALL ENTITIES PART Body surface volume
ADD 2D ELEMENT 2D_Body_Element surface
ADD 3D ELEMENT 3D_Body_Element volume
```

Then, to assign an element to all entities in the problem, the user would just have to open the `Default elements` window, as seen in figure A.3, and choose an element from the list.

A.5 Using generated problem types

Once the input file is written, save it in the `problemtype_generator` folder and run `problemtype.py`. For example if we saved our file as `my_problemtype.txt`, open a console window in the `problemtype_generator` folder and type:

```
python problemtype.py my_problemtype.txt
```

If no errors are found in the input file, a folder containing all required files will be created. Copy it into GiD's `problemtypes` folder and start GiD. You should be able to select your problemtype in the `Data/Problem Type` menu.

All kratos problem types based on the kratos template should have a custom menu. This menu provides access to all data books found in the problem type

files[3]. This menu also has one extra option, called `Model Status`, that gives some basic information about the model such conditions assigned or if some entity is missing a material.

Everything else about the problem type should work exactly as in regular GiD, except that forcing GiD to mesh additional entities is no longer necessary (all entities with a face condition will be meshed automatically) and assigning parts should be faster than assigning the conditions manually.

There are a couple of important remarks to be made about problem types generated for kratos. All the folders for the new kratos format provided with the original version of the problem type generator have some special features to ease the process of writing and executing the problem's Python file.

First of all, it should be noted that the python file requires the folder in which kratos is installed as input. This information can be automatically added to the Python variables file that the problemtype generates if the path to kratos is stored as a system variable. In Ubuntu, this means adding the following lines to the end of the `.bashrc` file (which is a hidden file found in your home folder), replacing the example path given here with the path to your kratos folder:

```
KRATOS_PATH='/home/user/kratosR1'
export KRATOS_PATH
```

If you use Windows, to add a system variable open the `system` window from the control panel. Go to the `Advanced options` tab and click the `environment variables` button. Then click the `New` button under system variables (not the one under user variables) to add the new variable, giving `KRATOS_PATH` as name and the path as value.

Kratos problem types have some properties added by default to the `Problem Parameters` window. The `Write a new Python script file for Kratos` flag, if checked, will tell GiD to copy any Python files found in the problem type folder to your model folder. Usually, the only Python file included in the problem type folder will be an example Python script. If you intend to modify this file for your problem, uncheck this flag so your modified version won't be overwritten when you make changes to the model.

The `Transfer materials to lower entities` flag will make GiD transfer materials assigned to volume entities to surfaces, lines and points (in 3D problems) or from surfaces to lines and points (if no volumes are found). This guarantees that conditions and elements assigned to those surfaces or lines will receive the same material as the volume. By itself, GiD assigns to an element the material of the entity it belong to. This is a problem if the element is a kratos condition assigned, for example, to the faces of a volume. As the user typically assigns the material only to the volume, its surfaces won't have a material by themselves. The problem type generator makes the surfaces inherit the volume's material to simplify the process of assigning materials, but there are cases in which the volume and the surface need to have different materials. In those cases, this flag should be unchecked and the materials assigned materials manually to all entities.

Finally, there is a last flag that sets the kratos `domain_size` variable automatically. This variable takes a value of 2 in 2D problems and 3 in 3D problems.

[3]Except the Default one, which contains automatic part types that should be assigned without the user's intervention. The Default book is still available from the Data/Conditions menu

The problem type will try to set it automatically based on your model, but this won't be enough in some cases. For example, there are shell problems where the original geometry is 2D, but we are interested in the tri-dimensional behaviour of the shell. In those cases, the domain size should be given manually by unchecking the `Let GiD determine domain size` flag and introducing its value.

A.6 Customizing the problem type generator

A.6.1 Basic concepts and required files

To define a new problem type, at least the following files must be written:

- The general configuration file (.prb), which defines general physical and numerical information, such as the value and direction of gravity or the value of the time step used.
- The condition definition file (.cnd), that defines the different kinds of boundary and initial conditions that can be applied to the model.
- The material definition file (.mat). It defines the materials available and their relevant physical properties.
- At least one data format file (.bas). This file is a template that GiD uses as a reference when writing the input for the solver.
- An execution file for Unix environments (.unix.bat) and/or one for Windows (.win.bat), which manages the generated files and calls the solver.

There are several additional files that GiD can use: .tcl, which contains extensions to GiD written in Tcl/Tk programming language, .xml for XML-based configuration, .uni for unit systems, .sim for condition symbols and .geo for geometrical definition symbols.

The problem type generator was designed to write general configuration, condition and material files, any number of data format files and Tcl extension files. Other file types aren't directly written by the generator, but will be added to the finished problem type if they are found in the template folder.

To define a problem type, this program requires four types of files:

- An input file for each problem type, naming specific conditions, elements, materials and properties to include in the problem type.
- Template files contain skeletal versions of al
- A Python file defining the custom data types as classes.
- Definition files, which describe the input format for each new data type and, if needed, the Tcl extension code they require.

Depending on the level of customization required, different files will have to be modified. The simplest customization is defining new problem types using the input file, which has been explained in section A.2. Modifying the Python file and creating new definition files, we can add new types of conditions, elements,

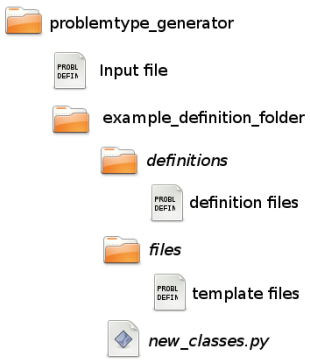

Figure A.4: Folder structure for the required files. Names in italics cannot be changed.

materials, properties or classes. This will be explained in section A.6.3. Finally, section A.6.4 explains how the template files are used, which will allow us to modify the input format.

A.6.2 File organization

The problem type generator expects to find each of the reference files we just described in a specific location and, generally, with a specific name. It has already been said that the input file is expected to be in the folder where `problemtype.py` (the main file for the problem type generator) is found, which is `problemtype_generator` unless the user decides to change it. The program expects to find any remaining files inside a folder specified in the input file (with the `DEFINITION FOLDER` line, as shown in page 56, the `kratos` folder contains the basic kratos data types). A Python file called `new_classes.py`, containing the custom classes required for the problem type, must be placed inside this folder. Definition files must be placed inside a sub-folder named `definitions` (`kratos/definitions` in our example). The name of each definition file can be chosen freely, but it must be specified in the Python file. Finally, template files must be placed in the `files` sub-folder. Their names must be the names of the finished files they define, replacing the name of the problem type with `problemtype`. For example, the template for the conditions file will be called `problemtype.cnd`, the template for the Unix execution file will be `problemtype.unix.bat` and the template for the kratos' nodal coordinates file will be `004_problemtype.node.bat`. An example folder tree for the definition files is shown in figure A.4.

A.6.3 Adding new data types

A simple example

We will start by providing a basic explanation on how to define a scalar condition. A finished problem type will contain at least two types of information about a scalar condition: a definition of the condition itself in the cnd file and a definition of the input that must be produced from it in the template files. A

very simple example can be seen in the following two files. First, let's take a look at a simple conditions file, `example.cnd`.

```
# example.cnd
CONDITION: point_PRESSURE
CONDTYPE: over points
CONDMESHTYPE: over nodes
QUESTION: PRESSURE
VALUE: 0.0

CONDITION: line_PRESSURE
CONDTYPE: over lines
CONDMESHTYPE: over nodes
QUESTION: PRESSURE
VALUE: 0.0

CONDITION: surface_PRESSURE
CONDTYPE: over surfaces
CONDMESHTYPE: over nodes
QUESTION: PRESSURE
VALUE: 0.0
```

This file defines three GiD conditions, that can be applied over point, line and surface entities respectively. When the user assigns them to an entity from the conditions menu, he or she will have to give a value for the pressure. Later, when a mesh is generated, this value of pressure will be transferred to all nodes that belong to that entity. The template file determines how this information will be given to the solver. Assume that we have written the following block in a bas file:

```
*# Example.bas
*Set cond point_PRESSURE *nodes
*Add cond line_PRESSURE *nodes
*Add cond surface_PRESSURE *nodes
*loop nodes *OnlyInCond
NODES[*NodesNum](PRESSURE,0) = *cond(PRESSURE);
*end nodes
```

If node 17 belongs to a line with a value of pressure of 5.0 assigned and node 23 belongs to a surface with a pressure of 7.5, this block of code will produce the following two lines in the input files[4]:

```
NODES[17](PRESSURE,0) = 5.0;
NODES[23](PRESSURE,0) = 7.5;
```

How can we tell the problem type generator to produce all the required code for our pressure conditions? A first important remark is that definitions for the problem type generator are independent of the entities over which it can be assigned. Therefore, lines like `CONDTYPE: over points` or `*Add cond line_PRESSURE *nodes` are not required and should not be written in definition files, they will be generated automatically based on the entities given in the input file. The names and default values of the condition ("PRESSURE" and "0.0" in this example) are not part of the generic definition, and should be replaced by temporary values. To generate a scalar condition using the basic condition class, only the following code is required:

[4]Check GiD's reference manual or help files for an explanation of all available commands in a .bas file.

```
QUESTION: <QUESTION_NAME>
VALUE: <DEFALUT_VALUE>
```

for the condition file and:

```
*loop nodes *OnlyInCond
NODES[*NodesNum](<QUESTION_NAME>,0) = *cond(<QUESTION_NAME>);
*end nodes
```

for the template file. Later, we would use the input file to define a condition called "PRESSURE", which has a default value of 0.0 and can be defined over point, line and surface entities, based on the scalar condition model. The condition code must be introduced in the Python file, while template code is stored in the definition files, but this will be explained in detain in the next section.

Implementing new data types

The process of adding a new data type to an existing problem type involves defining a new class in the Python file, creating a definition file to store the template code (or Tcl extension code) and possibly editing the existing template files to create a book for new condition, element or property classes. To explain how all this is done, we will take a look at how the existing data types for kratos have been implemented.

The data class contains the basic information about how to add new conditions (or elements or materials or ...) based on the information read from the input file. Each new class must be derived from one of the five basic data classes that the problem type generator knows[5]: `condition`, `element`, `material`, `gendata` (for properties) and `part`. (note that those basic conditions are defined in the `core_definitions` module, so you must always import it in your Python file). This allows the program to identify what kind of data it is reading from the input file and reduces the amount of code required to define new data types, as a part of the required GiD code is already known by the basic class.

We will examine the existing condition types first. The `new_classes.py` file defines scalar conditions, vector conditions and flag conditions as classes derived from `core_definitions.condition`. For each one of these classes, several attributes must be defined. Using the the flag condition as an example:

```
class condition_flag(core_definitions.condition):
    # A flag condition
    call='FLAG CONDITION'
    definition_file='scalar_condition'
    insert_in='# Nodal Values'

    questions='QUESTION: Fixed#CB#(1,0)\n'+\
              'VALUE: <TYPE>\n'+\
              'QUESTION: <NAME>#CB#(<VALUES>)\n'+\
              'VALUE: <DEFVALUE>\n'

    additional_input=('<TYPE>','<DEFVALUE>','<VALUES>')
```

[5]They could also be based on a class derived from them, as the problem type generator will check their class tree until it finds a basic class.

The `call` attribute is the name by which the condition can be called in the input file. When a line in the input file is found beginning with `FLAG CONDITION`, the problem type generator will assume that it defines a condition based on this class. The `definition_file` attribute is the name of the file containing the template code (which must be found in the `definitions` folder), while the `questions` attribute is the code that will be added to the condition file, as it was described in the previous section[6]. Note that the GiD condition code is input as a string, so newlines must be written as `\n`. The `insert_in` attribute is used to determine where the condition code will be placed in the condition file. There is a line in the condition file template (`problemtype.cnd` in the `files` folder) which reads `# Add Conditions here`. When adding flag conditions, the problem type generator will write their condition code just before that line, which will place them inside the `Conditions` book. The technique of using a commented out line as a reference to know where to write code is widely used by the problem type generator and, in this manual, the line in the input file is called a "bookmark". Finally, the `additional_input` attribute is a tuple containing the names of the "unknown" parts of the code: the extra information that must be provided by the input file. Note that `<NAME>` does not appear in that list. `<NAME>`, and `<ENTITY>` for elements, are already assumed to be parameters that need to be specified in the input file by the basic data classes, so they don't have to be declared as input in their derived classes.

The fact that extra input is required creates the problem of how the problem type generator will find it in the input file. This is achieved defining two extra methods for our class, that will be used to handle this additional data. One is the `parseinput` method, which is used when the condition is defined, and the other is the `valuestring` method, which allows the Tcl extension file to assign this condition automatically based on the use of model parts.

We will take a look at `parseinput` first. When the problem type generator reads the input file, it takes each line and turns it into a list of words, which are removed from the list as soon as they are identified. As described in section A.2, to create a condition based on the flag condition class, a instruction like the following will be needed:

```
FLAG CONDITION IS_BOUNDARY point line surface \
fixed 1.0 2.0 3.0 4.0 5.0
```

The line begins with our `call` attribute, it then specifies the name of the condition, `IS_BOUNDARY`, and the GiD entities over which it can be applied and, finally, it gives the default values for our two extra questions. As the problem type generator reads this line, it will recognize all words until it reaches the "`fixed 1.0...`" part and then it will call the class' `parseinput` method to know what to do with it, with a list (`line=['fixed','1.0','2.0', ... '5.0']`) as an argument. The `parseinput` method is the expected to return a tuple containing the desired values for the extra input, in the same order as they appear in the `extra_input` attribute. (In this case, the desired result would be `('fixed','1.0','1.0,2.0,3.0,4.0,5.0')`).

The `parseinput` method currently used for flag condition can be found in `new_classes.py`:

```
def parseinput(self,extra_input):
```

[6] In this example, the `#CB#` option is used to define a GiD combo box.

```
        if extra_input[0]=='free':
            cond_type='0'
            extra_input=extra_input[1:]
        elif extra_input[0]=='fixed':
            cond_type='1'
            extra_input=extra_input[1:]
        else:
            cond_type='1'
        if len(extra_input)==0:
            print 'ERROR: No default values found for this condition'
            return
        defvalue=str(extra_input[0])
        first=True
        for val in extra_input:
            if first==False:
                values=values+','+str(val)
            else:
                values=str(val)
                first=False
        return cond_type,defvalue,values
```

There is a default `parseinput` method defined for the base classes, which simply returns the list it receives as an argument. It is enough for simple classes, but it has the problem that it can't check if the information it receives is correct: it can't detect errors like introducing more (or less) parameters that required or make sure that numeric parameters are not given alphanumeric values. Thus, a custom `parseinput` method is always desirable, if only to check that there are enough parameters.

The `valuestring` method must use the same input to generate a string containing a series of values that are a valid answer to the `QUESTION` statements we defined in the `questions` attribute, separated by blank spaces. It is called when using the condition in a class. For example, if `IS_INTERFACE` has been defined as a flag condition, we can define a part like:

```
DEFINE MODEL PART Structure point line surface LOWER
ADD CONDITION IS_INTERFACE fixed 1.0
```

To understand this line, the problem type generator will call the flag condition's `valuestring` method with `['fixed','1.0']` as an argument. As the flag condition has two `QUESTION:` lines in its `questions` attribute (`Type` and `<NAME>`, which, in this case, will be replaced with `IS_INTERFACE`) so, in order to assign the condition, two values are needed. As the flag condition's valuestring method is[7]:

```
def valuestring(self,extra_input):
        if len(extra_input)==0:
            print 'ERROR: No values found for this condition'
            return
        elif extra_input[0]=='fixed':
            fixed='1'
            value=extra_input[1]
```

[7] Authors note: If you take a look at the real Python file, you will notice that I'm cheating you here: this method is defined for the scalar condition. The flag condition class is derived from the scalar condition class and it inherits the `valuestring` method from it.

```
            if len(extra_input)>2:
                print 'More values than expected found, only the'+\
' first two will be used'
                print '\t'+str(extra_input)
        elif extra_input[0]=='free':
            fixed='0'
            value=extra_input[1]
            if len(extra_input)>2:
                print 'More values than expected found, only the'+\
' first two will be used'
                print '\t'+str(extra_input)
        else:
            fixed='1'
            value=extra_input[0]
            if len(extra_input)>1:
                print 'More values than expected found, only the' +\
' first one will be used'
                print '\t'+str(extra_input)
            extra_input=extra_input[0:2]
        return fixed+' '+value
```

The string `'fixed 1.0'` will be returned. Again, a simple `valuestring` that transforms the list into a string is provided by default, but a more complex one is recommended, and in fact will be required, in many cases. For example, the vector condition has more questions than additional input arguments. Let's take a look at its `questions` attribute (edited from the original in the Python file for readability):

```
QUESTION: <NAME>_X#CB#(1,0)
VALUE: 1\
DEPENDENCIES:(0,SET,X_Value,0.0,SET,Fix_X,#CURRENT#) ...
QUESTION: Fix_X#CB#(1,0)
VALUE: <FIXX>
QUESTION: X_Value
VALUE: <VALX>\n
QUESTION: <NAME>_Y#CB#(1,0)
VALUE: 1
DEPENDENCIES:(0,SET,Y_Value,0.0,SET,Fix_Y,#CURRENT#) ...
QUESTION: Fix_Y#CB#(1,0)
VALUE: <FIXY>
QUESTION: Y_Value
VALUE: <VALY>
QUESTION: <NAME>_Z#CB#(1,0)
VALUE: 1\n
DEPENDENCIES:(0,SET,Z_Value,0.0,SET,Fix_Z,#CURRENT#) ...
QUESTION: Fix_Z#CB#(1,0)
VALUE: <FIXZ>
QUESTION: Z_Value
VALUE: <VALZ>

additional_input=('<FIXX>','<VALX>','<FIXY>','<VALY>','<FIXZ>','<VALZ>')
```

As you can see, we have three questions for each direction, one that controls if a value is being imposed in that direction (it takes a value of 1 if we are imposing it, 0 otherwise), one that controls if the value given will be fixed or

free to change while solving the problem and a last one containing the value itself. Also note that we now have a total of nine `QUESTION` and `VALUE` pairs, while only three or values are given to define a vector condition (typically four values are given: "fixed" or "free" and the three components, although the "fixed/free" one can be omitted) This means that the default `valuestring` method will not be sufficient, as it won't be able to produce the nine values by itself. This class needs a new `valuestring` method that, given a value for each direction, imposes a magnitude in all three directions by assigning ones to the `<NAME>_X` questions:

```
def valuestring(self,extra_input):
        if len(extra_input)<=4:
            if extra_input[0]=='fixed':
                fixed='1'
                extra_input=extra_input[1:]
            elif extra_input[0]=='free':
                fixed='0'
                extra_input=extra_input[1:]
            else: # no fixed/free argument
                fixed='1'

            if len(extra_input)<3:
                print 'ERROR: Not enough default values '+\
'found for this condition'
                print '\t'+str(extra_input)
                return # This crashes the problem type generator
            elif len(extra_input)>3:
                'More default values than expected found, '+\
'some will be ignored'
                print '\t'+str(extra_input)
                extra_input=extra_input[0:3]

            ValX=extra_input[0]
            ValY=extra_input[1]
            ValZ=extra_input[2]

            return '1 '+fixed+' '+ValX+' 1 '+fixed+' '+ValY+\
' 1 '+fixed+' '+ValZ
```

In addition to the condition class, we must create a definition file with the name we specified as the `definition_file` attribute (see page 72). This file will any contain pieces of code that must be repeated for each condition added (generally template code written in `.bas` files, but other uses, such as Tcl code, are possible). As with the condition code we added to the Python file, the template code must follow the same syntax as regular GiD bas file code, with the particularity that `*Set` and `*Add` clauses, which depend on the specific entities on which the part has been defined, must be omitted and will be added later by the program.

Continuing with the flag condition example, the input that kratos requires for it is a Nodal Data block in the `.mdpa` file, specifying the name of the condition and, for each node that has it assigned, the number of the node, `1` if the value assigned is fixed or `0` otherwise and the value of the condition:

```
Begin NodalData Condition_Name
```

```
Node_Id Is_Fixed Value
...
End NodalData
```

To generate this input for a condition that, for example, can be assigned over point, line and surface entities, the following code has to be written in the template file:

```
*Set cond volume_Condition_Name *nodes
*Add cond surface_Condition_Name *nodes
*Add cond line_Condition_Name *nodes
*Add cond point_Condition_Name *nodes
*if(CondNumEntities > 0)
Begin NodalData Condition_Name
*loop nodes *OnlyInCond
*format "%i%i%f"
*NodesNum *cond(Fixed) *cond(Condition_Name)
*end nodes
End NodalData
*endif
```

This will tell GiD to check if the condition `Condition_Name` has been used. If it has, it will write the `Begin NodalData ...` and `End NodalData` lines and loop over all nodes with this condition assigned producing a line for each of them. If we remove the `*Set` and `*Add` clauses and replace the name of the condition for the `<NAME>` parameter, we obtain a generic code template that can be used by all flag conditions. The only additional information needed to write a definition file is where this code has to be written. This is provided by the `file` and the `where` lines. The first one contains the name of an existing template file, while the second references a bookmark comment in that file.

```
file problemtype.bas
where *# Nodal Variable blocks

*if(CondNumEntities > 0)
Begin NodalData <NAME>
*loop nodes *OnlyInCond
*format "%i%i%f"
*NodesNum *cond(Fixed) *cond(<NAME>)
*end nodes
End NodalData

*endif
```

Note that any of the strings included in `additional_input` attribute can be used in the definition file and will be replaced by their actual values once a condition is defined. In addition, as seen in this example, the string `<NAME>` will be replaced by the condition's name for conditions, elements and properties and `<ENTITY>` will be replaced by the element type (`point`, `line`, `surface` or `volume`) for GiD conditions derived from the element base class.

Defining new material types is slightly different to defining other data types because materials are grouped in families, collections of materials with the same physical properties defined. To create new material types, the definition file must contain blocks to be written for each family, and not for each individual material. We will clarify this with an example. The `kratos_material` class defined in `new_classes.py` is the reference used to define kratos materials:

```
class material(core_definitions.material):
    call='MATERIAL'
    definition_file='material'
```

Note that this class is much simpler that the condition ones. It has no `insert_in` attribute because materials are just added to the end of the `.mat` file. It doesn't have a `questions` attribute either, because the code for the materials file is automatically generated from the property names given in the input file. As there are no additional questions, the `additional_input` attribute and the `valuestring` and `parseinput` methods are not required. There are several strings that can have to be used as unknown parameters in the definition file: `<TYPE>`, `<PROPNAME>`, and the special `<FOR EACH PROPERTY>` block. As usual, we will explain them with an example, in this case from the kratos material definition file.

```
file problemtype.bas
where *# Property blocks

*loop materials
*if(strcmp(MatProp(Type),"<TYPE>")==0)
Begin Properties  *MatNum
<FOR EACH PROPERTY>
<BEGIN SCALAR>
*format "%f"
<PROPNAME> *MatProp(<PROPNAME>,real)
<END SCALAR>
<BEGIN 2DVECTOR>
*format "%f%f"
<PROPNAME> [2] (*MatProp(<PROPNAME>_X,real),*MatProp(<PROPNAME>_Y,real))
<END 2DVECTOR>
<BEGIN 3DVECTOR>
*format "%f%f%f"
<PROPNAME> [3] (*MatProp(<PROPNAME>_X,real),*\
*MatProp(<PROPNAME>_Y,real),*MatProp(<PROPNAME>_Z,real))
<END 3DVECTOR>
<BEGIN 2X2MATRIX>
*format "%f%f%f%f"
<PROPNAME> [2,2] *\
((*MatProp(<PROPNAME>_XX,real),*MatProp(<PROPNAME>_XY,real)),*\
(*MatProp(<PROPNAME>_YX,real),*MatProp(<PROPNAME>_YY,real)))
<END 2X2MATRIX>
<BEGIN 3X3MATRIX>
*format "%f%f%f%f%f%f%f%f%f"
<PROPNAME> [3,3] (*\
(*MatProp(<PROPNAME>_XX,real),*MatProp(<PROPNAME>_XY,real),*\
*MatProp(<PROPNAME>_XZ,real)),(*MatProp(<PROPNAME>_YX,real),*\
*MatProp(<PROPNAME>_YY,real),*MatProp(<PROPNAME>_YY,real)),*\
(*MatProp(<PROPNAME>_ZX,real),*MatProp(<PROPNAME>_ZY,real),*\
*MatProp(<PROPNAME>_ZY,real)))
<END 3X3MATRIX>
<BEGIN TEXT>
<PROPNAME> *MatProp(<PROPNAME>)
<END>
End Properties
```

```
*endif
*end materials
```

This block of code will be repeated in the kratos properties file for each material family defined in the input file using a `DEFINE MATERIAL` line[8]. The `<TYPE>` string will be replaced by the family name (from the `DEFINE` line). In this case, it is used to check a hidden property that all materials based in the `core_definitions.material` class have, which identifies which family they belong to. This is important because if our problem type has two different material types and we define different physical properties for each type, we have to make sure that we don't attempt to read a property defined for one type from a material that belongs to the other (and has no property with that name).

The `<FOR EACH PROPERTY>` and `<END>` lines limit a special block of code that. For every property defined in the input file, the problem type generator will copy the lines corresponding to its type (determined according to the instructions in table A.1), replacing any generic parameters with their values. Inside this block, Inside this block, `<PROPNAME>` will be replaced for the name of the property.

For example, if we define a material family called `Example` with one scalar property called `DENSITY` and a matricial property called `CONDUCTIVITY` using the `3X3MATRIX` template, the program will produce the following template code:

```
*loop materials
*if(strcmp(MatProp(Type),"Example")==0)
Begin Properties  *MatNum
*format "%f"
DENSITY *MatProp(DENSITY,real)
*format "%f%f%f%f%f%f%f%f%f"
CONDUCTIVITY [3,3] (*\
(*MatProp(CONDUCTIVITY_XX,real),*MatProp(CONDUCTIVITY_XY,real),*\
*MatProp(CONDUCTIVITY_XZ,real)),(*MatProp(CONDUCTIVITY_YX,real),*\
*MatProp(CONDUCTIVITY_YY,real),*MatProp(CONDUCTIVITY_YY,real)),*\
(*MatProp(CONDUCTIVITY_ZX,real),*MatProp(CONDUCTIVITY_ZY,real),*\
*MatProp(CONDUCTIVITY_ZY,real)))
End Properties

*endif
*end materials
```

Finally, the definition of properties is similar to that of conditions, with the particularity that the `questions` attribute is optional. If none is given, a simple `QUESTION:` and `VALUE` pair will be generated, using the name of the property as `QUESTION`. There is one unknown parameter that can be used in the definition file: `<NAME>`, which is replaced by the name of the question in the `.prb` file.

A.6.4 Defining a new input format

The files in the `files` folder define the basic structure of all problem type files, including template files. By changing them we can modify the format and contents of the template files or add new function to the Tcl extension file. The basic GiD files (for example the `.cnd`, `.mat`, `.tcl` or `.unix.bat` files) must be named "problemtype" with the proper extension, so we will create `problemtype.cnd`, `problemtype.prb`, ...

[8] Note that `MATERIAL` is used here because its the class' `call` attribute.

A file named `problemtype.bas` is required by GiD, but if you use more that one `.bas` file, the names of the rest are free. Bear in mind, though, that the files that GiD generates for your from the `.bas` file don't retain their original names, and its up to the batch files (`.unix.bat` or `.win.bat` depending on your operative system) to give them their final names. An easy way to make sure that you give the right name to the right file (which the problem type generator will recognize and use properly when replacing the `problemtype` in the template file name with the problem type's name) is to add a prefix to their name to make sure that they are created in the right order, as GiD reads them ordered by name and then follows the instructions in the execution file to move them. For example, the template files for the kratos format use this trick, so they are called:

- `problemtype.bas`, which generates the `.mdpa` file
- `001_problemtype.aux.unix.bas` contains some additional shell instructions that can change depending on some options that the user can chose in the `Problem Parameters` window.
- `002_problemtype.aux.win.bas` is the DOS batch (for Windows OS) version of the last file.
- `003_problemtype_var.py.bas` contains some Python variable definitions

And the execution file relies in that order (first the `problemtype.bas` file, and then any other `.bas` files, ordered by name) to give them their definitive names.

We will take a look to the template files to explain how they should be written in order to be used by the problem type generator. In general, they will contain generic code (the parts of the code that don't depend on the conditions and materials found in the input file) or bookmarks to insert data-specific code. Bookmarks have already been introduced in section A.6.3 and are commented out lines that the problem type generator uses to know where to write the code created by custom classes and definition files. In the `.bas` template files, they usually begin with `*#`, which is used for comments in the GiD template format. The code included in template files is either not related to any specific condition, such as the code to generate node lists in `problemtype.bas`, or related to data included by default to all problem types using the kratos template (such as the kratos properties mentioned in A.5). For example, the `domain_size` variable, is already implemented as a GiD property in the template `problemtype.prb` file and will be assigned automatically when meshing thanks to a procedure included in the `problemtype.tcl` template.

It should be noted that there are three template files in the `default` folder which will be used in your problem type if a custom version is not found in the `files` folder. Those are a conditions file (`problemtype.cnd`), a general data file (`problemtype.prb`) and a Tcl extension file (`problemtype.tcl`). The first two are included because they contain the bookmarks used by default by the basic data types (which means that if you don't define any `insert_in` attribute in `new_classes.py` you won't need to add a `problemtype.cnd` or a `problemtype.prb` to your `files` folder), while the last one contains Tcl procedures that are required for parts to work properly.

The following is an example of a template conditions file used by the kratos problem types:

```
BOOK: Nodal Values
# Nodal Values

BOOK: Elements
# Elements

BOOK: Elemental Data
# Elemental Data

BOOK: Conditions
# Conditions

BOOK: Conditional Data
# Conditional Data

BOOK: Model Parts
# Add Parts Here

BOOK: Default
# Add Default Conditions Here
```

As you can see, in condition files the symbol used to comment out lines is `#`. the same is true for materials, properties and Tcl extension files.

Note that, while there isn't a default materials file template, the materials file has a very simple structure and will be created from scratch during file generation. If you wanted to, you could write a template `problemtype.mat` file and it would be used in your problem type.

There is one last important remark about books in conditions, materials and properties files: there can be problems if GiD finds an empty book. For this reason, once the problem type has been written, the problem type generator checks those files for empty books and deletes them.

Finally, we will briefly describe the execution files. `problemtype.unix.bat`, for example, contains the following instructions:

```
#!/bin/bash -f
#    OutputFile: $2/$1.info
#    ErrorFile: $2/$1.err
#delete previous result file
rm -f  $2/$1.flavia.res
rm $2/$1.info
rm $2/$1.flavia.dat
rm $2/$1.mdpa

mv $2/$1.dat $2/$1.mdpa
mv $2/$1-1.dat $2/${1}_aux.unix.bat
rm $2/$1-2.dat

chmod 700 $2/${1}_aux.unix.bat

./${1}_aux.unix.bat $1 $2 $3
```

GiD gives three parameters as input to this file: `$1` is your model's name, `$2` is the absolute path to your model's folder and `$3` is the path to the problem type folder (relative from GiD's `problemtypes` folder).

As you can see, this file removes any existing files and replaces them with the new templates. Note that the order in which the input files are created matches the order in which we named them. When writing files from the templates, GiD names them with the name of your model followed by a number. We can be sure of which file is each one because we ordered the templates by adding them a prefix. After this, it gives executing permission to the auxiliaryy batch file and runs it, giving it some extra input parameters.

This auxiliary batch file contains some additional commands, which are executed depending on some parameters that the user can modify freely. This is the reason why it is created as a GiD template file and not as a shell batch file. Its contents are as follows:

```
#!/bin/bash -i

write_python_file=*GenData(Write_a_new_Python_script_file_for_Kratos,Int)
problemtype_name=*Tcl(GiD_Info Project ProblemType)

mv $2/$1-3.dat $2/${problemtype_name}_var.py

echo "problem_name=\"${1}\"" >> ${problemtype_name}_var.py
echo "problem_path=\"${2}\"" >> ${problemtype_name}_var.py
echo "kratos_path=\"${KRATOS_PATH}\"" >> ${problemtype_name}_var.py

if [ $write_python_file = 1 ]
then
 cp $3/**.py $2/
fi
```

As you can see, this file adds some variables to the end of a Python file (this is the same file where properties described in section A.3.5) and copies any Python files from the problem type folder to the model's folder if the user chooses so. As mentioned before, by doing this it attempts to copy an example Python script file containing the basic instructions to solve the problem.

There is another execution file, `problemtype.win.bat`, that contains equivalent instructions for GiD installations running in Windows platforms.

A.7 An overview of the problem type generator source

This manual should be useful as an introduction to the problem type generator and its capabilities, but it is not an exhaustive guide. The best way to understand how the program works and how it can be modified is to examine its source files. For this reason, the last part of this manual will briefly describe the problem type generator's files and their function.

A.7.1 The definitions

The most important file for the program is `core_definitions.py`, as it describes the basic classes it uses. Some of them have already been defined in this manual: the five basic data classes, but there are some other important definitions.

The first one of them is the `code` class. All code that has to be written in a problem type file is stored as a code instance. It is fundamentally a string containing the code, with some additional attributes describing the name of the file where it will be written and the line where it should be added (using a "bookmark" comment). It also has a method to write the code to the destination file and another one to replace some parts of the code with new strings, which is used to change the generic unknown parameters that we added in the definition files (`<NAME>`, `<VALUE>` and the likes, see section A.6.3).

As several `code` instances are generated for each condition or element defined, there are two classes intended as containers and organizers of code instances. The first one of them is the `code_entry` class, that contains all code related to a single condition. It also stores a list of the missing unknown parameters for each code piece and identifies them with a string (stored in the `uses` attribute). This string is useful to identify which code instances contain code ready to be written or code that needs to be edited (by replacing the unknown parts) when a condition is defined. The `code_entry` also has some additional methods to manage the code instances, such as searching all code instances with a specific use or writing all code ready to be written to its file.

The `code_container` class is used to define a collection of code entries, each one containing the code for a single condition or element. The problem type generator uses an instance of this class (a variable called `code_db`) to manage all the code instances involved in a project. It is basically an organizer for `code_entry` objects: it has methods to activate the functions of multiple `code_entry` instances matching certain criteria, which allow the program to do things like finding all code instances from all defined conditions containing code ready to be written and writing them to the relevant files.

The last part of the `core_definitions.py` file defines the basic data types. Each one of them (`condition`, `element`, `material`, `gendata` for properties and a class for each part type) has two methods, `__init__` and `add`, that define the generation of that type of code. The first one defines some generic variables and common code based on the attributes we described in section A.6.3 (or their default values if they are not defined in the specific Python files), while the second generates specific code using the information obtained from the input file. Note that all data types are derived from the `base` class, which defines some auxiliary methods used by all of them (the default `valuestring` and `parseinput` for example).

A.7.2 Reading and writing files

The main file of the problem type generator is `problemtype.py`. It reads the input file and uses the classes defined in `core_definitions` and in format-specific files to create code. It is fundamentally a loop over the lines of the input file, identifying which kind of information each one gives and sending that line to the relevant class method for processing. To do so, it uses the auxiliary functions defined in `read_tools.py`. The module also generates the Tcl code required to define the interaction between parts using the functions in `tcl_functions.py`, which is added to the `code_db` variable to be written in the Tcl extension file. After the entire input file has been read, if no errors have been detected, the problem type files will be created and the code generated for each condition will be added to them. This is done after all input is read to

avoid creating the problem type files and leaving them incomplete when an error is found. The final section of the file adds some additional Tcl code, including a custom menu, and erases any unused data books from the problem type files.

The `read_tools` module contains several helper functions to read the definition files. One of them, `read_definitions` is specially important because it reads the classes defined in the format-specific files (using the functions of the Python module `inspect`) and their definition files and sends them to the main module, organized by their base class. This is fundamental because it allows the problem type generator to use custom classes, giving the user the ability to modify the input format and the problem type's structure with relative ease.

There is an additional module, `file_functions`, used while generating the files. It defines some basic file manipulation functions to read, write and modify the problem type files.

A.7.3 The Tcl extension file

In this section of the manual we will examine the Tcl extension file that the problem type generator creates, as it contains the procedures that streamline the process of generating kratos models using GiD. The Tcl extension file is divided in Tcl procedures. As explained in GiD's documentation, if a Tcl file is found when loading a problem type, GiD will look for several procedures and call them at specific times. The default tcl file implements three of those procedures: `InitGIDProject`, which is run when the problem type is selected, `BeforeMeshGeneration`, which is run just before meshing, and `AfterMeshGeneration`, which is run after the mesh generation ends. The former is used to generate a custom menu for the problem type, while the later are used for all automatic operations over the entities and the problem data: assigning conditions, changing the direction of normals or issuing meshing instructions. This is done thanks to special Tcl functions included in GiD that can be used to obtain information about the model and modify it[9].

Main GiD procedures

When adding new Tcl capabilities for the problem type generator, such as aligning normals or transferring a condition or part from a surface entity to its boundary lines, the chosen implementation has been, when possible, writing new procedures in the Tcl file template and adding a call to them in the `BeforeMeshGeneration` procedure for each condition that uses them when the problem type file is generated. It is felt that this implementation is the best for code generated by another application, as it reduces the amount of lines that have to be added during file generation and the total length of the Tcl file, making it more readable, but there is the possibility that some of the procedures will remain unused in a given generated file.

An important remark about the `BeforeMeshGeneration` procedure that the problem type generator writes it that it is quite structured, because the order in which the instructions are written is important. A first block of instructions resets everything that can be added automatically by the procedure. This is

[9] As usual, checking GiD documentation is recommended. The chapter *Tcl/TK extension* from the Online GiD User Guide is specially relevant for this section.

done because, if the user has made changes in the model, some of the conditions assigned by the procedure in a previous meshing might no longer be correct.

The instructions are then organized in entities: first all operations related to volume entities are done, followed by surface entities, lines and, finally, points. This is necessary to ensure that parts are correctly transferred from an entity to its boundary: from a volume to its faces, from a surface to its contour lines and from a line to its ends. Each entity block follows the same steps: first, parts are checked and transferred from the higher entity, then automatic parts are assigned (for example, parts assigned to the boundary) and finally part interaction is checked, first for parts defined as groups in the input file and later for parts that don't belong to a group. This is done looping over the entities and checking which parts each one has been assigned. If an entity belongs to a single part in a group, that part's conditions are assigned. If it belongs to multiple parts from the same group, the conditions specified in the input file are assigned or, if that particular part combination was not defined in the input file, a warning is displayed.

The last block of the `BeforeMeshGeneration` procedure changes GiD's mesh criteria for entities that need it. In the kratos example in particular, this is required for all entities with a face condition assigned, so this section is filled with calls to the `meshelement` procedure. This procedure lists all entities with a given condition assigned and instructs GiD to mesh them.

Additional procedures used by the problem type generator

We will finally list some of the procedures that have been implemented to write the problem type's Tcl extensions, as it is felt that some of them could be useful while defining new problem types.

- `alignlinenormals Direction` and `alignlinenormals Direction` make the normals of all the surfaces (for 3D problems) or lines (for 2D problems) of the boundary in the same direction. Use `Inwards` or `Outwards` as the `Direction` parameter.

- `cond_linetopoint Condition`, `cond_surfacetoline Condition`, and `cond_volumetosurface Condition` transfer the `Condition` condition from an entity that has it assigned to its boundary.

- `assign_materials` make GiD transfer materials assigned to volume entities to surfaces, lines and points (in 3D problems) or from surfaces to lines and points (if no volumes are found).

- `condfrompart Part Condition mode entity args` assigns the condition `Condition` to all entities of type `entity` (possible values are `point`, `line`, `surface` or `volume`) in the `args` list (a list of entity numbers), using the values for that condition given by the model part `Part`. The `mode` argument determines if the condition should be assigned in all models (`always`), only in 2D problems (`only2D`) or only in 3D problems (`only3D`).

- `meshelement Element Entity` tells GiD to mesh all entities of type `Entity` (use `line` or `surface`) that have the GiD condition `Element` assigned.

- `meshtype Element Entity Elemtype` tells GiD to use elements of the type `Elemtype` to mesh all entities of type `Entity` (use `line`, `surface` or `volume`) with condition `Element` assigned.
- `check_elemtype Element Entity Elemtype` verifies if non-default element types are being used. The input syntax of this command is rather confusing because, in its current implementation, all calls to it are written by the problem type generator while creating conditions based on the `element` template, with limited information. Basically, For each `Element` defined in the input file for `Entity` entities using the custom element type `Elemtype`, it will check if it has been used in the project. If the element has been used, it will reset the element types used by all entities of that type the next time a mesh is generated. This is done to ensure that all entities have the correct element type even if the user decides to change their assigned element.
- `findboundary entity` makes a list containing all boundary entities. use `line` as `entity` for models without volumes or `surface` for models made of volumes.
- `createlist entity Part` is one of the most widely used Tcl procedures in this file. Given an `entity` (`point`, `line`, `surface` or `volume`) and a condition name as `Part`, returns a list containing the numbers of all entities with that condition assigned.
- `cleanautomatic Condition args` unassigns automatically assigned GiD conditions `Condition` (a model part, condition or element) from entity types given as `args` (one or more of the following words: `point`, `line`, `surface` or `volume`)
- `assigndefault Condition Property args` assign `Condition`, single value condition to all entities in `args` (use it as in `cleanautomatic`) that don't have it assigned. The value given is taken from the field `Property` from General Data.
- `create_point_elems Condition` can be used in `AfterMeshGeneration` to create point elements for all points with condition `Condition` assigned.
- `assign_element_choice Option entity args` is used to assign an element chosen using the `OPTION` command (explained in section A.4.3). `Option` is the name of the element chooser, `entity` is the entity type over which the element will be assigned (`point`, `line`, `surface` or `volume`) and `args` are the names of all elements that can be assigned. The `args` parameter is necessary to check that any given entity doesn't receive more that one element (for example, in case that the user manually assigns an element to some entities).
- `cond_report` creates the Tk window that is generated when the menu option `Model Status` is selected.
- `TkwidgetFilePath` is used by the `FILE SELECTION` property to create the `Select File` window.
- `getmatnum matname` returns the internal material number of the material `matname`.

A.8 The old kratos format

This section explains how the problem type generator can be used to create problem types using the old kratos input format, which requires five different files:

```
000_problemtype.prop.bas
001_problemtype.elem.bas
002_problemtype.cond.bas
003_problemtype.init.bas
004_problemtype.node.bas
```

To generate problem types in the old format, use the `kratos` folder as `DEFINITION FOLDER` when writing your input file. There are some differences in the template files from the old and new formats, which means that, instead of what was explained in section A.3 only the commands described in this section can be used.

There are three condition types available to assign values for nodes, which can be used to generate flags, scalar conditions and vectorial conditions. Those conditions can then be used as boundary conditions or initial values. The syntax is the same as in the new format:

```
VECTOR CONDITION DISPLACEMENT point line surface volume fixed 0.0 0.0 0.0
SCALAR CONDITION PRESSURE point line surface volume free 0.0
FLAG CONDITION IS_BOUNDARY point line surface 1.0 2.0 3.0 4.0 5.0
```

Elements and conditions can also be generated as in the new format, but they can't be used for points:

```
FACE CONDITION CONDITION2D line
FACE CONDITION CONDITION3D surface
ELEMENT Fluid2D surface
ELEMENT Fluid3D volume
```

If you need to use point elements or conditions, you could use a modified version of the `point_element` class from the new format, using a custom definition file. You would also need to copy the `create_point_elems Condition` procedure to your Tcl file.

Conditional values are implemented, but there are no elemental values:

```
VECTOR FACE VALUE DISPLACEMENT <name> <entities> 0.0 0.0 0.0
SCALAR FACE VALUE PRESSURE <name> <entities> 0.0
FLAG FACE VALUE IS_BOUNDARY <name> <entities> 1.0 2.0 3.0 4.0 5.0
```

Finally materials and properties can also be generated as in the new format, but the `FILE SELECTION` property is not implemented.