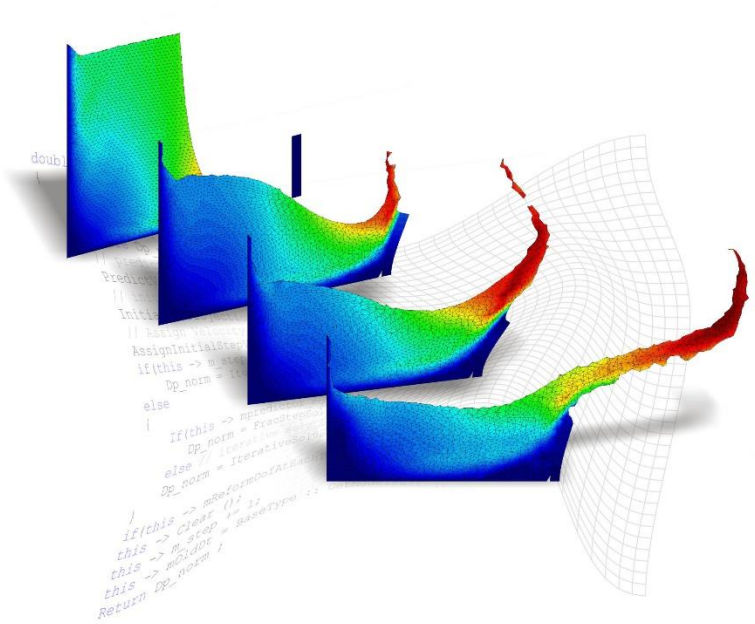
# A Framework for Developing Finite Element Codes for Multi-Disciplinary Applications

**Pooyan Dadvand**
**Eugenio Oñate**



Monograph CIMNE Nº-109, January 2008

# A Framework for Developing Finite Element Codes for Multi-Disciplinary Applications

**Pooyan Dadvand**
**Eugenio Oñate**

**Monograph CIMNE Nº-109, January 2008**

**Centro Internacional de Métodos Numéricos en Ingeniería**
Gran Capitán s/n, 08034 Barcelona, España

CENTRO INTERNACIONAL DE MÉTODOS NUMÉRICOS EN INGENIERÍA
Edificio C1, Campus Norte UPC
Gran Capitán s/n
08034 Barcelona, Spain

Primera edición: Januray 2008

**A FRAMEWORK FOR DEVELOPING FINITE ELEMENT CODES FOR MULTI-DISCIPLINARY APPLICATIONS**
Monografía CIMNE M109

To Hengameh

# Abstract

The world of computing simulation has experienced great progresses in recent years and requires more exigent multidisciplinary challenges to satisfy the new upcoming demands. Increasing the importance of solving multi-disciplinary problems makes developers put more attention to these problems and deal with difficulties involved in developing software in this area.

Conventional finite element codes have several difficulties in dealing with multi-disciplinary problems. Many of these codes are designed and implemented for solving a certain type of problems, generally involving a single field. Extending these codes to deal with another field of analysis usually consists of several problems and large amounts of modifications and implementations. Some typical difficulties are: predefined set of degrees of freedom per node, data structure with fixed set of defined variables, global list of variables for all entities, domain based interfaces, IO restriction in reading new data and writing new results and algorithm definition inside the code. A common approach is to connect different solvers via a master program which implements the interaction algorithms and also transfers data from one solver to another. This approach has been used successfully in practice but results duplicated implementation and redundant overhead of data storing and transferring which may be significant depending to the solvers data structure.

The objective of this work is to design and implement a framework for building multi-disciplinary finite element programs. Generality, reusability, extendibility, good performance and memory efficiency are considered to be the main points in design and implementation of this framework. Preparing the structure for team development is another objective because usually a team of experts in different fields are involved in the development of multi-disciplinary code.

Kratos, the framework created in this work, provides several tools for easy implementation of finite element applications and also provides a common platform for natural interaction of its applications in different ways. This is done not only by a number of innovations but also by collecting and reusing several existing works.

In this work an innovative variable base interface is designed and implemented which is used at different levels of abstraction and showed to be very clear and extendible. Another innovation is a very efficient and flexible data structure which can be used to store any type of data in a type-safe manner. An extendible IO is also created to overcome another bottleneck in dealing with multi-disciplinary problems. Collecting different concepts of existing works and adapting them to coupled problems is considered to be another innovation in this work. Examples are using an interpreter, different data organizations and variable number of dofs per node. The kernel and application approach is used to reduce the possible conflicts arising between developers of different fields and layers are designed to reflect the working space of different developers also considering their programming knowledge. Finally several technical details are applied in order to increase the performance and efficiency of Kratos which makes it practically usable.

This work is completed by demonstrating the framework's functionality in practice. First some classical single field applications like thermal, fluid and structural applications are implemented and used as benchmark to prove its performance. These applications are used to solve coupled problems in order to demonstrate the natural interaction facility provided by the framework. Finally some less classical coupled finite element algorithms are implemented to show its high flexibility and extendibility.

# Resumen

El mundo de la simulación computacional ha experimentado un gran avance en los últimos años y cada día requiere desafíos multidisciplinares más exigentes para satisfacer las nuevas demandas. El aumento de la importancia por resolver problemas multidisciplinares hizo poner más atención a la resolución de estos problemas y a los problemas que éstos implican en el área de desarrollo de software.

Los códigos convencionales de elementos finitos tienen varias dificultades para enfrentar se con problemas multidisciplinares. Muchos de estos códigos se diseñan y desarrollan para solucionar ciertos tipos de problemas, implicando generalmente un solo campo. Ampliar estos códigos para resolver problemas en otros campos del análisis, normalmente es difícil y se necesitan grandes modificaciones. Los ejemplos más comunes son: grados de libertad predefinidos para los nodos, estructura de datos capaz de guardar sólo una serie de variables definidas, lista global de las variables para todas las entidades, interfaces basadas en los dominios, capacidad del Input/Ouput para leer nuevos datos o escribir nuevos resultados y definición del algoritmo dentro del código. Un método común para resolver estos problemas es conectar varios modulos de calculo a través de un programa principal que implemente los algoritmos de la interacción y también transfiera datos de un modulo de calculo a otro. Este método se ha utilizado en la práctica con éxito, pero resulta en muchas duplicaciones del código y exceso de almacenamiento y tiempo de ejecución, dependiendo de la estructura de datos de los modulos de calculo.

El objetivo de esta tesis es diseñar e implementar un marco general para el desarrollo programas de elementos finitos multidisciplinares. La generalidad, la reutilización, la capacidad de ampliación, el buen rendimiento y la eficiencia en el uso de la memoria por parte del codigo son considerados los puntos principales para el diseño e implementación de este marco. La preparación de esta estructura para un fácil desarrollo en equipo es otro objetivo importante, porque el desarrollo de un código multidisciplinar generalmente requiere expertos en diferentes campos trabajando juntos.

Kratos, el marco creado en este trabajo, proporciona distintas herramientas para una fácil implementación de aplicaciones basadas en el método de los elementos finitos. También proporciona una plataforma común para una interacción natural y de diferentes maneras entre sus aplicaciones. Esto no sólo está hecho innovando, sino que además se han recogido y usado varios trabajos existentes.

En este trabajo se diseña y se implementa una interface innovadora basada en variables, que se puede utilizar a diferentes niveles de abstracción y que ha demostrado ser muy clara y extensible. Otra innovación es una estructura de datos muy eficiente y flexible, que se puede utilizar para almacenar cualquier tipo de datos de manera "type-safe". También se ha creado un Input/Ouput extensible para superar otras dificultades en la resolución de problemas multidisciplinares. Otra innovación de este trabajo ha sido recoger e integrar diversos conceptos de trabajos ya existentes, adaptándolos a problemas acoplado.Esto incluye el uso de un intérprete, diversas organizaciones de datos y distinto número de grados de libertad por nodo. El concepto de núcleo y aplicación se utiliza para separar secciones del codigo y reducir posibles conflictos entre desarrolladores de diversos campos. Varias capas en la estructura de Kratos han sido diseñadas considerando los distintos niveles de programación de diferentes tipos de desarrolladores. Por último, se aplican varios detalles técnicos para aumentar el rendimiento y la eficacia de Kratos, convirtiendo lo en una herramienta muy útil para la resolución de problemas prácticos.

Este trabajo se concluye demostrando el funcionamiento de Kratos en varios ejemplos prácticos. Primero se utilizan algunas aplicaciones clásicas de un solo campo como prueba patrón de rendimiento. Después, estas aplicaciones se acoplan para resolver problemas multidisciplinares, demostrando la facilidad natural de la interacción proporcionada por Kratos. Finalmente se han implementado algunos algoritmos menos clásicos para demostrar su alta flexibilidad y capacidad.

# Contents

Chapter 1

# Introduction

The automotive, aerospace and building sectors have traditionally used simulation programs to improve their products or services, focusing their computations in a few major physical process: fluid dynamics, structural, metal forming, etc. Nevertheless, the new needs for safer, cheaper and more efficient goods together with the impressive growth of the computing facilities, demand an equivalent solution for multi-disciplinary problems. Moreover, new applications in the food, chemical and electromagnetic industry, among others, are not useful at all without a multi-disciplinary approach.

Some illustrative cases can be found in the design of sails and sailboats where structural and fluid dynamic computations have to be taken in account, sterilization processes (thermal and fluid dynamic analysis), strong magnet design (structural, thermal and magnetic analysis) or photovoltaic cells (thermal and electrical computations), among many others.

The increasing importance of solving multi-disciplinary problems makes developers put more attention to these problems and deal with difficulties involved in developing software in this area.

## 1.1 Motivation

The world of computing simulation has experienced great progresses in recent years and requires more exigent multidisciplinary challenges to satisfy the new demands.

Due to the industrial maturity of one-purpose codes for the different fields, the production sector has increased its expectations, as they realize the need for solving in a more realistic way the problems they deal with, in order to stay competitive. Strong simplifications are the reason for which difficult problems could be solved in the past. These simplifications lead to a model as different from the real one, as the severity of the assumptions made. If we add every required accuracy in a concurrent world, then we need to relax the assumptions and become more general in the way of solving multidisciplinary problems.

The situation is not really new. What is novel is the need for solving multidisciplinary problems combining most of the information coming from different fields, obtaining a more precise information and therefore better optimization methods. This means solving more complex problems. There are many strategies to approach the solution of this kind of problems. A simple approach assumes that existing analysis methods are good and that people should create interfaces between already existing codes in order to solve multi-disciplinary problems. Unfortunately this strategy will simply not work for the monolithic solution of highly coupled problems and usually has

execution time overhead due to the data format conversion and memory overhead produced by duplicated data. These shortcomings provide a strong motivation for creating a multi-disciplinary programming framework which will allow dealing with these kind of problems in a more natural and flexible way.

It has been also planned to integrate another interesting topic in the multi-disciplinary solutions such as optimization. The vast amount of research works for the solution of industrial optimization problems constitutes a strong motivation to provide an affordable library as a numerical engine for optimization programs.

## 1.2 Problem

One of the relevant topics in the finite element method nowadays is the combination of different analysis (thermal, fluid dynamic, structural) with optimization methods, and adaptive meshing in one global software package with just one user interface and the possibility to extend the implemented solution to new types of problems, as an approach to a multi-disciplinary simulation environment.

Conventional finite element codes encounter several difficulties in dealing with multi-disciplinary problems involving the coupled solution of two or more different fields (i.e. fluid-structure, thermal-mechanical, thermal-fluid, etc.). Many of these codes are designed and implemented for solving a certain type of problems, generally involving a single field. For example to solve only structures, fluids, or electromagnetic problems. Extending these codes to deal with another field of analysis usually requires large amounts of modifications.

A common approach is to connect different solvers via a master program which implements the interaction algorithms and also transfers data from one solver to another. This approach has been used successfully in practice but has its own problems. First of all having completely separate programs results in many duplicated implementations, which causes an additional cost in code maintenance. In many cases the data transfer between different solvers is not trivial and, depending on the data structure of each program, may cause a redundant overhead of copying data from one data structure to another. Finally, this method can be used only for master and slave coupling and not for the monolithic coupling solution.

Some newer implementations have used the modern software engineering concepts in their design to make the program more extendible. They usually achieve the extendibility to new algorithms in their program but extending them to new fields is beyond their intent.

Using the object-oriented paradigm helps in improving the reusability of codes. This is considered to be a key point for streamlining the implementation of modules for solving new types of problems. Unfortunately many domain specific concepts in their design restricts their reusability for other modules.

The typical bottlenecks of existing codes for dealing with multi-disciplinary problems are:

- Predefined set of degrees of freedom per node.
- Data structure with fixed set of defined variables.
- Global list of variables for all entities.
- Domain based interfaces.
- IO restriction in reading new data and writing new results.
- Algorithm definition inside the code.

These shortcomings require extensive rewriting of the code in order to extend it to new fields.

Many programs have a predefined set of degrees of freedom per node. For example in a three dimensional structural program each node has six degrees of freedom $d_x, d_y, d_z, w_x, w_y, w_z$ where $d$ is the nodal displacement and $w$ the nodal rotation. Assuming all nodes to have just this set of degrees of freedom helps the developers to optimize their codes and also simplifies their implementation tasks. But this assumption prevents the extension of the code to another field with a different set of degrees of freedom per node.

Usually the data structure of programs is designed to hold certain variables and historical values. The main reasons for this design are: easier implementation, better performance of data structure and less effort in maintenance. In spite of these advantages using rigid data structures usually need important changes revision to hold new variables from another fields.

Another problem arises when the program's data structure is designed to hold the same set of data for all entities. In this case adding a nodal variable to the data structure implies adding this variable to all nodes in the model. In this implementation adding variables of one domain to data structure causes redundant spaces to be allocated for them in another domain. For example in a fluid-structure interaction problem, this design causes each structural node to have pressure values and all fluid nodes to have displacement values stored in memory. Even though this is not a restriction, it severely affects the memory efficiency of program.

Additionally, in single purpose programs, it is common to create domain specific interfaces in order to increase the code clarity. For example providing a `Conductivity` method for element's properties to get the element's conductivity as follow:

```
c = properties.Conductivity()
```

Though this enhances the code clarity, it is completely incompatible with extendibility to new fields.

IO is another bottleneck in extending the program to new fields. Each field has its sets of data and results, and a simple IO usually is unable to handle new set of data. This can cause significant implementation and maintenance costs that come from updating IO for each new problem to solve.

Finally introducing new algorithms to existing codes requires internal implementation. This causes closed programs to be nonextensible because there is no access to their source code. For open source programs, this requires the external developers learn about the internal structure of the code.

## 1.3 Objectives

The objective of this work is to design and implement a framework for building multi-disciplinary finite element programs. This framework is called Kratos and will help to build a wide variety of finite element programs from the simplest formulation, for example a heat conduction problem, to the most complex ones, like multi-disciplinary optimization techniques. From one side it will provide a complete set of flexible tools for fast implementation of experimental academic algorithms and from the other side it must be fast and efficient to be used for real industrial analysis.

Generality is the first objective in our design. A framework can be used if its generality is sufficient to fit a wide variety of finite element algorithms. The generality becomes more important when it comes to multi-disciplinary analysis in which, the code structure has to support the wide variety of algorithms involved in different areas.

Reusability is another objective in this design. Finite element methodology has several steps that are similar between different algorithms even for different types of problems. A good de-

sign and implementation can make all these steps reusable for all algorithms. This reduces the implementation effort and the maintenance cost of the code.

Another objective is providing a high level of extendibility for Kratos. The continuous innovation in finite element methodology may result in new algorithms with completely different requirements in the future. Thus supporting just a large number of current algorithms cannot guarantee the usefulness of the code in future. The solution is providing some ways to extend different parts of the code to new cases. For designing a multi-disciplinary code, extendibility plays an even more important role. Extendibility in a multi-disciplinary code is also necessary to support new problems in different fields. So extendibility is considered as an important objective for Kratos.

Good performance and memory efficiency are also objectives of Kratos. Solving multi-disciplinary industrial problems is the goal of Kratos and requires good performance and also good level of memory efficiency. Generality and flexibility are against the performance and efficiency of the code. So it is obvious that Kratos cannot achieve the same performance of a fully optimized single domain code. But the idea is to keep it as fast as possible and meanwhile increase the total performance in solving multi-disciplinary problems by reducing the data conversion and transfer between domains.

## 1.4 Solutions

Applying the object-oriented paradigm has shown to be very helpful in increasing the flexibility and reusability of codes. In this work, object-oriented design is successfully used to organize different parts of the code in a set of objects with clear interfaces. In this way, replacing one object with another is very easy, which increases the flexibility of the code, and reusing an object in some other places is also becomes more practical.

Design and implementation of a multi-disciplinary conceptual interface is another solution provided to previous problems. In the Kratos design, interfaces are defined in a very generic way and independent from any specific concept. The variable base interface resulting from this design is very general and solves the interface problems arising when extending the program to new fields.

A flexible and extendible data structure is another solution used to guarantee the extendibility of the code to new concepts. The proposed data structure is able to store any type of data associated to any concept. It also provides different ways for global organization of data required when dealing with multi-domains problems. The same strategy is applied to give flexibility in assigning any set of degrees of freedom to any node for solving new problems.

An interactive interface is provided in order to increase the flexibility of the code when implementing different algorithms. In this way a new algorithm can be introduced without the need to be implemented inside the program. This gives a high level of extendibility to the code and it is very useful for the implementation of optimization and multi-disciplinary interaction algorithms.

An automatic configurable IO module is added to these components providing the complete set of solutions necessary for dealing with multi-disciplinary problems. This IO module uses different component lists to adjust itself when reading and writing new concepts originating from different fields of analysis.

## 1.5 Organization

This work presents the design of a new object oriented finite element framework. The idea is to present a general view of its design and then divide the design procedure into its main parts while

describing each of them individually in separate chapters. Following this idea the layout of the work is organized as follows:

**Chapter 2: Background** In this chapter a background of finite element programming is given.

**Chapter 3: Concepts** It starts with a brief description of numerical analysis in general and continues by explaining the finite element method and its concepts with some detail. Finally it describes different design patterns used throughout this work and gives a brief description of some advanced C++ techniques used for writing high performance numerical applications.

**Chapter 4: General Structure** Explains the object oriented design of Kratos, its main objects, the code organization and also the separation between kernel and applications.

**Chapter 5:Basic Tools** Different tools provided to help developers in implementing their program are described in this chapter. Design of quadrature methods, linear solvers, and geometries are explained, and key points in their reusability and generality are mentioned.

**Chapter 6:Variable Base Interface (VBI)** Presents a new variable base interface to be used at different levels of the code. First, a motivation for designing the new interface is given. Then, this new interface is described and some applications are explained. After that there is a section about its implementation and the ways of designing interfaces using this method. Finally, some examples are given to show the capability of the VBI in practice.

**Chapter 7:Data Structure** This chapter first explains the basic concepts in programming containers and continues by describing different containers. Then presents new containers designed for finite element programming. Finally, a global scheme of data distribution in Kratos is provided.

**Chapter 8:Finite Element Implementation** Focuses on the components representing the finite element methodology implemented in Kratos. It describes the design of elements and conditions and also explains the structure of processes and strategies in Kratos. Finally it explains elemental expressions and formulations and their purpose of design.

**Chapter 9:Input Output** First it explains different approaches in designing IO modules and presents a flexible and generic IO structure. Then it continues with a part dedicated to interpreter writing, which consists of a small introduction to concepts and also a brief explanation on the use of related tools and libraries. Finally, it describes the use of Python as the interpreter with some description about the reasons for using Python, the interface library, and some examples to show its great abilities.

**Chapter 10:Validation Examples** It gives some examples of different finite element applications for multi-disciplinary problems implemented in Kratos. It also includes some benchmarks to compare the efficiency of Kratos with existing programs.

**Chapter 11:Conclusions and Future Work** Gives an overview of solutions and their effectiveness in solving multi-disciplinary problems and explains the future directions of this project.

Chapter 2

# Background

The history of object-oriented design for finite element programs turns back to the early 90's or even earlier. In 1990, Forde, *et al.* [44] published one of the first detailed descriptions of applying object-oriented design to finite element programs. They used the essential components of finite element method to design the main objects. Their structure consists of `Node`, `Element`, `DispBC`, `ForcedBC`, `Material`, and `Dof` as finite element components and some additional objects like `Gauss`, and `ShapeFunction` for assisting in numerical analysis. They also introduced the element group and the list of nodes and elements for managing the mesh and constructing system. Their approach has been reused by several authors in organizing their object-oriented finite element program structures. This approach was focused on structural domain and the objects' interfaces reflect this dependency.

In those years other authors started to write about the object-oriented paradigm and its applications in finite element programming. Filho and Devloo [43] made an introduction to object oriented design applying it to element design. Mackie [64] gave another introduction to the object-oriented design of finite element programs by designing a brief hierarchial element. Later he published a more detailed structure for finite element program providing a data structure for entities and also introduced the use of iterators [65]. Pidaparti and Hudli [80] published a more detailed object oriented structure with objects for different algorithms in dynamic analysis of structures. Raphael and Krishnamoorthy [82] also made an introduction to object-oriented programming and provided a sophisticated hierarchy of elements for structural applications. They also designed some numerical expressions for handling the common numerical operations in finite element method. The common point of all these authors was their awareness about the advantages of object oriented programming with respect to traditional Fortran approaches and their intention to use these advantages in their finite element codes.

Miller [70, 71, 72] published an object-oriented structure for a nonlinear dynamic finite element program. He introduced a coordinate free approach to his design by defining the geometry class which handles all transformations from local to global coordinates. The principal objects in his design are `Dof`, `Joint`, and `Element`. `TimeDependentLoad` and `Constrain` are added to them in order to handle boundary conditions. He also defines a `Material` class with ability to handle both linear and nonlinear materials. The `Assemblage` class holds all these components and encapsulates the time history modeling of structure.

Zimmermann, *et al.* [106, 35, 34] have designed a structure for linear dynamic finite element analysis. Their structure consists of three categories of objects. First the finite element

method objects which are: `Node`, `Element`, `Load`, `LoadTimeFunction`, `Material`, `Dof`, `Domain`, and `LinearSolver`. The second category are some tools like `GaussPoint`, and `Polynomial`. The third category are the collection classes like: `Array`, `Matrix`, `String`, etc. They implemented first a prototype of this structure in *Smalltalk* and after that an efficient one in C++, which latter version provides a comparable performance respect to a Fortran code.

In their structure `Element` calculates the stiffness matrix $K^e$, the mass matrix $M^e$, and the load vector $f^e$ in global coordinates. It also assembles its components in the global system of equations and update itself after solving. `Node` holds its position and manages dofs. It also computes and assembles its load vector and finally update the time dependent data after solving. `Dof` holds the unknown information and also its value. It stores also its position in the global system. and provides information about boundary conditions. `TimeStep` implements the time discretization. `Domain` is a general manager which manages the components like nodes and elements and also manages the solving process. It also provides the input-output features for reading the data and writing the results. Finally `LinearSystem` holds the system of equation components: left hand side, right hand side and solution. It also performs the equation numbering and implements the solver for solving the global system of equations.

They also developed a nonlinear extension to their structure which made them redefined some of their original classes like `Domain`, `Element`, `Material`, and some `LinearSystems` [69].

Lu, *et al.* [61, 62] presented a different structure in their finite element code FE++. Their structure consists of small number of finite element components like `Node`, `Element`, `Material`, and `Assemble` designed over a sophisticated numerical library. In their design the `Assemble` is the central object and not only implements the normal assembling procedure but also is in charge of coordinate transformation which in other approaches was one of the element's responsibilities. It also assigns the equation numbers. The `Element` is their extension point to new formulations. Their effort in implementing the numerical part lead to an object-oriented linear algebra library equivalent to LAPACK [15]. This library provides a high level interface using the object-oriented language features.

Archer, *et al.* [17, 18] presented another object-oriented structure for a finite element program dedicated to simulate linear and nonlinear, static and dynamic structures. They reviewed features provided by different other designs on that time and combined them in a new structure adding also some new concepts.

Their design consists of two level of abstractions. In the top level, the `Analysis` encapsulates the algorithms and the `Model` represents the finite element components. `Map` relates the dofs in the model, to unknowns in the analysis and removes the dependency between these objects. It also transforms the stiffness matrix from the element's local coordinate to the global one and calculates the responses. Additional to these three objects, different handlers are used to handle model dependent parts of algorithm. The `ReorderHandler` optimizes the order of the unknowns. The `MatrixHandler` provides different matrices and construct them over given model. Finally `ConstraintHandler` provides the initial mapping between the unknowns of analysis and the dofs in the model.

In another level there are different finite element components representing the model. `Node` encapsulates a usual finite element node which holds its position, and dofs. `ScalarDof` and `VectorDof` are derived from the `Dof` class and represent the different degree of freedom's types. `Element` uses the `ConstitutiveModel` and `ElementLoad` to calculate stiffness matrix $K^e$, mass matrix $M^e$, damp matrix $C^e$ in local coordinate system. `LoadCase` consists of loads, prescribed displacements, and initial element state objects and calculates the load vector in the local coordinate system.

Cardona, *et al.* [25, 53, 54] developed the Object Oriented Finite Elements method Led by Interactive Executor (OOFELIE) [77]. They designed a high level language and also implemented

an interpreter to execute inputs given in that language. This approach enabled them to develop a very flexible tool to deal with different finite element problems. In their structure a `Domain` class holds data sets like: `Nodeset`, `Elemset`, `Fixaset`, `Loadset`, `Materset`, and `Dofset`. `Element` provides methods to calculate the stiffness matrix $K^e$, the mass matrix $M^e$, etc. `Fixaset` and `Loadset` which hold `Fixation`s and `Load`s handl the boundary conditions and loads.

They used this flexible tool also for solving coupled problems where their high level language interpreting mechanism provides an extra flexibility in handling different algorithms in coupled problems. They also added `Partition` and `Connection` classes to their structure in order to increase the functionality of their code in handling and organizing data for coupled problems. `Partition` is defined to handle a part of domain and `Connection` provides the graph of degrees of freedom and also sorts them.

Touzani [96] developed the Object Finite Element Library (OFELI) [95]. This library has an intuitive structure based on finite element methodology and can be used for developing finite element programs in different fields like heat transfer, fluid flow, solid mechanics, and electromagnetic.

`Node`, `Element`, `Side`, `Material`, `Shape` and `Mesh` are the main components of its structure and different problem solver classes implement the algorithms. This library also provides different classes derived form `FEEqua` class which implement formulations in different fields of analysis. It uses a static `Material` class in which each parameter is stored as a member variable. The `Element` only provides the geometrical information and finite element implementation is encapsulated via `FEEqua` classes. The `Element` provides several features which make it useful for even complex formulations but keeping all these members data makes it too heavy for standard industrial implementations.

Bangerth, *et al.* [22, 19, 21] created a library for implementing adaptive meshing finite element solution of partial differential equations called Differential Equations Analysis Library (DEAL) II [20]. They were concerned with flexibility, efficiency and type-safety of the library and also wanted to implement a high level of abstraction. All these requirements made them to use advanced features of C++ language to achieve all their objectives together.

Their methodology is to solve a partial differential equation over a domain. `Triangulation`, `FiniteElement`, `DoFHandler`, `Quadrature`, `Mapping`, and `FEValues` are the main classes in this structure. `Triangulation` despite its name is a mesh generator which can create line segments, quadrilaterals and hexahedra depending on given dimensions as its template parameter. It also provides a regular refinement of cells and keeps the hierarchical history of mesh. The `FiniteElement` encapsulates the shape function space. It computes the shape function values for `FEValues`. `Quadrature` provides different orders of quadratures over cells. `Mapping` is in charge of the coordinate transformation. `DoFHandler` manages the layout of degrees of freedoms and also their numbering in a triangulation. `FEValues` encapsulates the formulation to be used over the domain. It uses `FiniteElement`, `Quadrature`, and `Mapping` to perform its calculation and provides the local components to be assembled into the global solver.

Extensive use of templates and other advanced features of C++ programming language increases the flexibility of this library without sacrificing its performance. They created abstract classes in order to handle uniformly geometries with different dimensions. In this way they let users create their formulation in a dimension independent way. Their approach also consists of implementing the formulation and algorithms and sometimes the model itself in C++. In this way the library configures itself better to the problem and gains better performance but reduces the flexibility of the program by requiring it to be used as a closed package. In their structure there is no trace of usual finite element components like node, element, condition, etc. This makes it less familiar for developers with usual finite element background.

Patzák, *et al.* [79] published an structure used in the Object Oriented Finite Element Modeling

(OOFEM) [78] program. In this structure `Domain` contains the complete problem description which can be divided into several `Engineering Model`s which prepare the system of equations for solving. `Numerical Method` encapsulates the solution algorithm. `Node`, `Element`, `DOF`, `Boundary condition`, `Initial condition`, and `Material` are other main object of this structure. This program is oriented to structural analysis.

## 2.1 Discussion

A large effort has been done to organize the finite element codes trying to increase their flexibility and reducing the maintenance cost. Two classes of designing finite element program can be traced in literature. One consists of using the finite element methodology for the design which leads to objects like element, node, mesh, etc. Another approach is to deal with partial differential equations which results in object functions working with matrices and vectors over domains.

The work of Zimmermann, *et al.* [106, 35, 34] is one of the classical approaches in designing the code structure considering finite element methodology. However there is no geometry in their design and new components like processes, command interpreter etc. are not addressed.

The effort of Miller, *et al.* in order to encapsulate the coordinate transformation in geometry is useful for relaxing the dependency of elemental formulation to the specific geometry.

Cardona, *et al.* [25, 53, 54] added an interpreter to manage the global flow of the code in a very flexible way. The interpreter was used for introducing new solving algorithms to the program without changing it internally. This code is also extended to solve coupled problems using the interpreter for introducing interaction algorithms which gives a great flexibility to it. The drawback of their approach is the implementation of a new interpreter with a newly defined language beside binding to an existing one. This implies the maintenance cost of the interpreter itself and prevent them from using other libraries which may have interfaces to chosen script languages.

Touzani [96] designed an structure for multi-disciplinary finite element programs. His design is clear and easy to understand but uses field specific interfaces for its component which are not uniform for all fields. This reduces the reusability of the algorithm from one field to the other.

The approach of Lu, *et al.* [61, 62] is on the line of designing a partial differential solver with emphasizes on numerical components. Archer, *et al.* [17, 18] extended this approach to a more flexible and extendible point and more recently Bangerth, *et al.* used the same approach in designing their code. However the structure results from this design can be unfamiliar to usual finite element programmers. For instance in the latter design there are no objects to represents nodes and elements, which are the usual components for finite element programmers.

In this work the standard finite element objects like nodes, elements, conditions, etc. are reused from previous designs but modified and adapted to the multi-disciplinary perspective. There are also some new components like model part, process, kernel and application are added to cover new concepts mainly arising in multi-disciplinary problems. A new variable base interface is designed providing a uniform interface between fields and increasing the reusability of the code. The idea of using an interpreter is applied by using an existing interpreter. A large effort is also done to design and implement a fast and very flexible data structure enable to store any type of data coming from different fields and guarantee the uniform transformation of data. An extendible IO is also created to complete the tools for dealing with the usual bottlenecks when working with multi-disciplinary problems.

# Chapter 3

# Concepts

The objective of this work is to create a framework to implement multi-disciplinary finite element applications. Before starting, it is necessary to explain some basic concepts of the finite element method itself, multi-disciplinary problems and their solutions, and programming concepts related to its design and implementation.

In this chapter a brief introduction to finite element concepts is given first. Then some basic concepts of coupled systems are described. Finally some programming concepts and techniques are explained.

## 3.1 Numerical Analysis

In this section a brief introduction to numerical analysis in general will be given and a short description to different numerical methods, their similarities and their differences will be presented.

### 3.1.1 Numerical Analysis Scheme

There are several numerical analysis methods which are different in their approaches and type of applications. Beside their differences, they rely on a global scheme which make them similar in their overall methodologies and to some extent mixable or interchangeable. This overall scheme consists of three main steps as follows:

**Idealization** Defining a mathematical model which reflects a physical system. For this reason this step is also referred as *mathematical modeling*. In this step the governing equations of a physical system and its conditions are transformed into some general forms which can be solved numerically. Usually different assumptions are necessary to make this modeling possible. These assumptions make the model different from the physical problem and introduce the *modeling error* in our solutions.

**Discretization** Converting the mathematical model with infinite number of unknowns, called *degrees of freedom (dof)*, to a finite number of them. While the original model with infinite number of unknown cannot be solved numerically, the resulting *discrete model* with finite number of unknowns can be solve using a numerical approach. What is important to mention

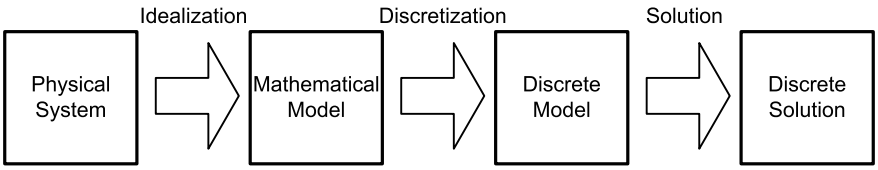



Figure 3.1: Three steps of Numerical analysis process.

here is the approximation involved in this step. This process introduces the *discretization error* to the solution which highly depends on the quality of discretization and the methodology used.

**Solution** Solving the discrete model and obtaining the dof and other related results. This step introduces the *solution error* coming from the inexactness of using algorithms, numerical precision of machine, or other sources.

Figure 3.1 Shows this global scheme. An important observation here is the existence of different errors and approximations introduced by different concepts. The accumulation of these errors can affect the validity of the results obtained by these methods. For this reason the validity of each step and reduction of the error in each process is one of the main challenges in using a numerical method.

### 3.1.2 Idealization

The idealization is the mathematical modeling of certain physical phenomena. The main task of this step is finding a proper mathematical model for a given physical problem.

Mathematical models are usually based on different assumptions. This dependency makes them useful for problems in which these assumptions are correct or near to reality. For this reason finding a good mathematical model requires a good knowledge not only about the problem but also about the assumptions of the model.

For example in solving a fluid problem the idealization consists of finding the best fluid model for the certain fluid in the problem. In this case the main questions are: Is this fluid Newtonian? Is it compressible or not? Is it a laminar flow? etc. Depending on these conditions one model can be considered more suitable than others for a certain problem. However the selected model may still introduce certain modeling errors to our solution.

There are different form of mathematical models. Some common ones are:

**Strong Form** Defines the mathematical model as a system of ordinary or partial differential equations and some corresponding boundary conditions.

**Weak Form** It expresses the mathematical equations in a particular modified form using a weighted residual approximation.

**Variational Form** In this form the mathematical model is presented as a functional whose stationary conditions generates the weak form.

#### Strong Form

As mentioned before the strong form defines the mathematical model as a system of ordinary or partial differential equations and some corresponding boundary conditions. Considering the

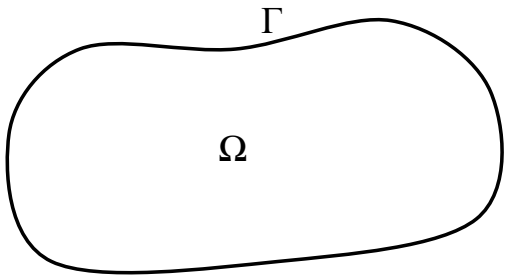



Figure 3.2: The problem of finding $u$ over a domain $\Omega$.

domain $\Omega$ with boundary $\Gamma$ shown in figure 3.2, this form defines the model by a set of equations over the domain and the boundary as follows:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}(u(x)) = q & x \in \Gamma \end{cases} \tag{3.1}$$

where $u(x)$ is the unknown and $\mathcal{L}$ is the operator applied over the domain $\Omega$ and $\mathcal{S}$ represents the operator applied over the boundary $\Gamma$. The first equation represents the governing equation over the domain and the second one represents the boundary conditions of this problem. For example, a thermal problem over the domain of figure 3.3 can be modeled with the following strong form:

$$\begin{cases} \nabla^T \mathbf{k} \nabla\theta(x) = Q & x \in \Omega \\ \theta(x) = \theta_\Gamma & x \in \Gamma_\theta \\ q(x) = q_\Gamma & x \in \Gamma_q \end{cases} \tag{3.2}$$

where $\theta(x)$ is the temperature in $\Omega$, $q$ is the boundary flux, $Q$ is the internal heat source, $\Gamma_\theta$ is the boundary with fixed temperature $\theta_\Gamma$, and $\Gamma_q$ is the boundary with fixed flux $q_\Gamma$.

### Weak Form

Let $V$ be a Banach space, Considering the problem of finding the solution $u \in V$ of equation:

$$\mathcal{L}(u) = p \qquad u \in V \tag{3.3}$$

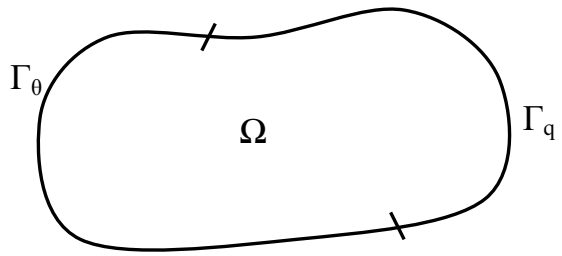



Figure 3.3: A Thermal domain $\Omega$ with fixed temperature boundary $\Gamma_\theta$ and fixed flux boundary $\Gamma_q$.

It can be verified that this problem is equivalent to finding the solution $u \in V$ such that for all $v \in V$ holds:

$$(\mathcal{L}(u), v) = (p, v) \qquad u \in V, \forall v \in V \tag{3.4}$$

Which is known as the *weak formulation* of the problem. The weak form defines the mathematical model using the weak formulation of the strong form. Now using the following scalar product:

$$(u, v) = \int_\Omega uv d\Omega \tag{3.5}$$

results in the *integral form*, which is the weighted integral representation of the model:

$$\int_\Omega \mathcal{L}(u)v d\Omega = \int_\Omega pv d\Omega \qquad u \in V, \forall v \in V \tag{3.6}$$

This formulation can be applied also to involve the boundary condition. For example, the same problem represented by equation 3.1 can be rewritten as follows:

$$\int_\Omega r(u)w d\Omega + \int_\Gamma \bar{r}(u)\bar{w} d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.7}$$

where $r$ and $\bar{r}$ are the residual functions defined over the domain and the boundary respectively:

$$r(u) = \mathcal{L}(u) - p \tag{3.8}$$
$$\bar{r}(u) = S(u) - q \tag{3.9}$$

and $w$ and $\bar{w}$ are arbitrary weighting functions over the domain and boundary. Usually it is convenient to use integration by parts in order to reduce the maximum order of derivatives in the equations and balance it by applying some derivatives to the weighting functions. Performing integration by parts on equation 3.7 yields:

$$\int_\Omega A(u)B(w) d\Omega + \int_\Gamma C(u)D(\bar{w}) d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.10}$$

Reducing the order of derivatives in $A$ and $C$ respect to $r$ and $\bar{r}$, allows for a lower order of continuity requirement in the choice of the $u$ function. However, now higher continuity for $w$ and $\bar{w}$ is necessary.

### Variational Form

The variational form usually comes from some fundamental quantities of the problem like mass, momentum, or energy whose stationary states are of interest. This form defines the mathematical model by a functional in the following form:

$$\Pi(u) = \int_\Omega F(u) d\Omega \tag{3.11}$$

The stationary state of this quantity is required, hence its variation is made equal to zero, which results in the following equation:

$$\delta\Pi(u) = \int_\Omega \delta(F(u)) d\Omega = 0 \tag{3.12}$$

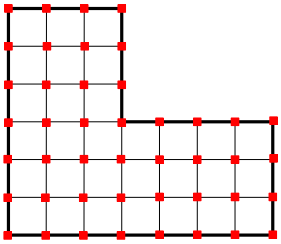


Figure 3.4: A regular domain discretized with a finite difference grid.

Deriving the variational equations from conservation laws is attractive for scientists as presents the same fundamental characteristics of the problem. Finally it is important to mention that the weak form can also be derived from the stationary state of the variational form.

### 3.1.3 Discretization

The first step of numerical analysis was the definition of a mathematical or *continuous* model corresponding to the real physical problem. In practice the continuous model cannot be solved analytically except for certain domains and conditions, which is the reason for numerical solution. The continuous model has an infinite number of unknowns corresponding to points in the domain and the boundary, and cannot be solved directly using numerical methods. So this second step is necessary for converting the continuous model to a discrete one with a finite number of unknowns which can be solved numerically.

There are several ways to perform this conversion which results in different numerical methods. The appropriate discretization method depends not only on the type of problem but also on the type of mathematical model describing it. A brief description on proper discretization for different models is given as follow.

#### Discretization of the Strong Form

Discretization of strong form is typically performed using the *finite difference method*. The idea here comes from the numerical calculation of derivatives by replacing them with differences. For example the first derivative of function $f(x)$ can be changed to its discrete form as follows:

$$\frac{df(x)}{dx} \approx \Delta_h^1(f,x) = \frac{1}{h}(f(x+h) - f(x)) \tag{3.13}$$

Where the discretization parameter $h$ is the distance of grid points. This method also can be applied to calculate higher derivatives of a function:

$$\frac{d^n f(x)}{dx^n} \approx \Delta_h^n(f,x) = \sum_{i=0}^{n} (-1)^{n-k} \left(\frac{1}{h}\right)^n \begin{pmatrix} n \\ i \end{pmatrix} f(x+ih) \tag{3.14}$$

The discretization is simply a cartesian grid over domain. Figure 3.4 shows a sample of grid in two dimensional space.

This method has been used practically in many fields and implemented in many applications. Its methodology is simple and also is easy to program. These made it one of the favorite methods in numerical analysis. However this method has its shortcomings. First, it works well for regular

domains, but for arbitrary geometries and boundary conditions encounters difficulties. For example the irregular domain of figure 3.5 (a) can be approximated by the discrete domain shown in figure 3.5 (b). It can be seen easily that this discretization changes the domain boundary for an arbitrary geometry.

(a) (b)

Figure 3.5: An arbitrary geometry and its finite difference discrete model.

Another disadvantage is its approximated solution, which can be obtained only in the grid points hence no information is provided on other points within the grid.

### Discretization of the Weak Form

Discretization of the continuous mathematical model consists of transforming the working space to some selected discrete one. Considering the following weak form in the continuous space $V$:

$$(\mathcal{L}(u), v) = (p, v) \qquad u \in V, \forall v \in V \tag{3.15}$$

This form can be transformed to the discrete space $V_h$ to approximate the solution by $u_h \in V$:

$$(\mathcal{L}(u_h), v_h) = (p, v_h) \qquad u_h \in V, \forall v_h \in V \tag{3.16}$$

As mentioned before, the weak form of the mathematical model can be represented by a weighted integral as:

$$\int_\Omega r(u) w d\Omega + \int_\Gamma \bar{r}(u) \bar{w} d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.17}$$

where $r$ and $\bar{r}$ are the residual functions defined over the domain and the boundary respectively. $w$ and $\bar{w}$ are arbitrary weighting functions over the domain and the boundary. Here selecting a discrete space $V_h$ as our working space results in the following discrete model:

$$\int_\Omega r(u_h) w_h d\Omega + \int_\Gamma \bar{r}(u_h) \bar{w}_h d\Gamma = 0 \qquad u_h \in V_h, \forall w_h, \bar{w}_h \in V_h \tag{3.18}$$

where $r$ and $\bar{r}$ are the residual functions. Equation 3.18 is a *weighted integral of residuals*. So this class of approximations is referred as the *weighted residual methods*. Several well known methods like the *Finite Element Method (FEM)*, *Finite Volume (FV)*, and *Least squares* fitting are subclasses of this method.

It is common to choose a discrete space $V_h$ made by a set of known *trial functions* $N_i$ and define the discrete solution as follows:

$$u_h \approx \sum_{j=1}^{n} a_j N_j \tag{3.19}$$

where $a_j$ are unknown coefficients, $N_j$ are known trial-functions, and $n$ is the number of unknowns. Substituting this in equation 3.18 results:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j) w_h d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j) \bar{w}_h d\Gamma = 0 \qquad \forall w_h, \bar{w}_h \in V_h \tag{3.20}$$

with:

$$w_h = \sum_{i=1}^{n} \alpha_i w_i \quad , \quad \bar{w}_h = \sum_{i=1}^{n} \alpha_i \bar{w}_i \tag{3.21}$$

where $\alpha_i$ are arbitrary coefficients, $w_i$ and $\bar{w}_i$ are arbitrary functions, and $n$ is the number of unknowns. Expanding equation 3.20 gives:

$$\sum_{i=1}^{n} \alpha_i \left[ \int_\Omega r(\sum_{j=1}^{n} a_j N_j) w_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j) \bar{w}_i d\Gamma \right] = 0 \qquad \forall \alpha_i, w_i, \bar{w}_i \in V_h \tag{3.22}$$

As $\alpha_i$ are arbitrary, all components of above sum must be zero in order to satisfy the equation. This results in the following set of equations:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j) w_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j) \bar{w}_i d\Gamma = 0 \qquad \forall w_i, \bar{w}_i \in V_h \quad , \quad i = 1, 2, 3, ..., n \tag{3.23}$$

There are several set of functions that can be used as weighting functions. Here is a list of some common choices:

**Collocation Method** Using *Dirac's delta* $\delta_i$ as the weighting function:

$$w_i = \delta_i \quad , \quad \bar{w}_i = \delta_i \tag{3.24}$$

where $\delta_i$ is a function such that:

$$\int_\Omega f \delta_i d\Omega = f_i \tag{3.25}$$

Substituting this weighting function in our reference equation 3.23 results in the following discrete model:

$$r_i(\sum_{j=1}^{n} a_j N_j) + \bar{r}_i(\sum_{j=1}^{n} a_j N_j) = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.26}$$

This method satisfies the equation just in the set of collocation points and gives a set of discrete equations similar to those obtained by the finite difference method.

**Subdomain Method** It is an extension of the previous method. It uses a weighting function $w_i$ which is identity in a subdomain $\Omega_i$ and zero elsewhere:

$$w_i = \begin{cases} I & x \in \Omega_i \\ 0 & x \notin \Omega_i \end{cases} \quad , \quad \bar{w}_i = \begin{cases} I & x \in \Gamma_i \\ 0 & x \notin \Gamma_i \end{cases} \tag{3.27}$$

The discrete model can be obtained by substituting this weighting function in the general weak form of equation 3.23:

$$\int_{\Omega_i} r(\sum_{j=1}^{n} a_j N_j) d\Omega_i + \int_{\Gamma_i} \bar{r}(\sum_{j=1}^{n} a_j N_j) d\Gamma_i = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.28}$$

This method provides a uniform approximation in each subdomain and establishes a way to divide the domain into subdomains for solving the problem.

**Least Square Method** This method uses the governing operator applied to the trial functions as its weighting functions:

$$w_i = \delta \mathcal{L}(N_i) \quad , \quad \bar{w}_i = \delta S(N_i) \tag{3.29}$$

Here is the resulted discrete model by substituting the above weighting function in equation 3.23:

$$\int_{\Omega} r(\sum_{j=1}^{n} a_j N_j) \mathcal{L}(N_i) d\Omega + \int_{\Gamma} \bar{r}(\sum_{j=1}^{n} a_j N_j) S(N_i) d\Gamma = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.30}$$

One can verify that the resulting equation is equivalent to minimizing the square of the global residual $\mathcal{R}$ over the domain:

$$\delta \mathcal{R} = 0 \tag{3.31}$$

where:

$$\mathcal{R} = \int_{\Omega} r^2(u_h) d\Omega + \int_{\Gamma} \bar{r}^2(u_h) d\Gamma = 0 \tag{3.32}$$

**Galerkin Method** It uses the trial functions as weighting functions:

$$w_i = N_i \quad , \quad \bar{w}_i = N_i \tag{3.33}$$

Substituting equation 3.33 in equation 3.23 results in the following Galerkin discrete model:

$$\int_{\Omega} r(\sum_{j=1}^{n} a_j N_j) N_i d\Omega + \int_{\Gamma} \bar{r}(\sum_{j=1}^{n} a_j N_j) N_i d\Gamma = 0 \quad , \quad i = 1, 2, 3, ..., n \tag{3.34}$$

This method usually improves the solution process because frequently, but not always, leads to symmetric matrices with some other useful features which makes it a favorite methodology and a usual base for the finite element solution.

### Discretization of the Variational Form

The *Rayleigh-Ritz* method is the classical discretization method for the variational form of the continuous model. It also was the first trial-function method. The idea is to approximate the solution $u$ by $\tilde{u}$ defined by a set of trial functions as follows:

$$\tilde{u} = \sum_{i=1}^{n} \alpha_i N_i \tag{3.35}$$

where $\alpha_i$ are unknown coefficients, $N_i$ are known trial-functions, and $n$ is the number of unknowns. Now considering the following continuous model in its variational form:

$$\delta\Pi(u) = \int_{\Omega} \delta(F(u)) d\Omega = 0 \tag{3.36}$$

Inserting the trial function expansion of equation 3.35 into equation 3.36 gives:

$$\delta\Pi(\tilde{u}) = \sum_{i=1}^{n} \frac{\partial \Pi}{\partial \alpha_i} \delta\alpha_i = 0 \tag{3.37}$$

As this equation must be true for any variation $\delta\alpha$, all its component must be equal to zero. This results into the following set of equations:

$$\begin{aligned} \frac{\partial \Pi(\tilde{u})}{\partial \alpha_1} &= \int_{\Omega} \frac{\partial (F(\tilde{u}))}{\partial \alpha_1} d\Omega = 0 \\ \frac{\partial \Pi(\tilde{u})}{\partial \alpha_2} &= \int_{\Omega} \frac{\partial (F(\tilde{u}))}{\partial \alpha_2} d\Omega = 0 \\ &\vdots \\ \frac{\partial \Pi(\tilde{u})}{\partial \alpha_n} &= \int_{\Omega} \frac{\partial (F(\tilde{u}))}{\partial \alpha_n} d\Omega = 0 \end{aligned} \tag{3.38}$$

### Mixed Discretization

Sometimes it is useful to mix two or more types of discretization in order to describe complex phenomena or just for simplifying some approximations while keeping a more detailed approach in other aspects of the problem. A typical case of mixing different forms is the modeling of time dependent problems, where one can use a finite difference discretization in time while using a weighted residual discretization in space.

### 3.1.4 Solution

The last step in the numerical methodology is the solution. This step consists of solving the discrete model using proper algorithms in order to find the main unknowns and also to calculate other additional unknowns of the problem. This process includes:

**Calculating Components** All components of a discrete model (like derivatives in the finite difference method, integrals in weighted residual or variational methods, etc.) are calculated.

**Creating the Global System** The discrete model's components are put together in order to create the global system of equations representing the discrete model.

**Solving the Global System** The global system must be solved to calculate the unknowns of the problem. For some models this leads to a system of linear equations which can be solved using linear solvers. Some algorithms create a diagonal global system and made the solving part completely trivial.

**Calculating Additional Results** In many problems not only the principal unknown, i.e. displacement in structural problems, are of interest, but also some additional results, like stresses and strains in structural problems, must be calculated.

**Iterating** In many algorithms some iterations are also needed to determine the unknown or calculate different sets of unknowns. Solving nonlinear problems, calculating time dependent unknowns, and optimization problems are examples of algorithms where iterations are needed.

## 3.2 Finite Element Method

In the previous section a brief introduction to numerical methods was given. As this work is based on finite element methodology, a brief description of this method and its basic steps are presented.

The finite element method (FEM) in general takes the integral form of the problem and uses piecewise polynomials as its trial functions. There are a wide range of formulations which lead to the FEM but the most used one is the Galerkin method, which is used here to describe the FEM and its basic steps.

### 3.2.1 Discretization

Considering a continuous problem:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}(u(x)) = q & x \in \Gamma \end{cases} \tag{3.39}$$

and its integral form as follows:

$$\int_{\Omega} r(u) w d\Omega + \int_{\Gamma} \bar{r}(u) \bar{w} d\Gamma = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.40}$$

where $r$ and $\bar{r}$ are the residual functions defined over the domain and the boundary respectively:

$$r(u) = \mathcal{L}(u) - p \tag{3.41}$$

$$\bar{r}(u) = \mathcal{S}(u) - q \tag{3.42}$$

Defining the discrete finite element space $V_h$ as composition of polynomial functions $N_i$:

$$x = \sum_{i=1}^{n} \alpha_i^x N_i \tag{3.43}$$

and transforming the equation 3.40 to this space results in:

$$\int_\Omega r(u_h) w_h d\Omega + \int_\Gamma \bar{r}(u_h) \bar{w}_h d\Gamma = 0 \qquad u_h \in V_h, \forall w_h, \bar{w}_h \in V_h \tag{3.44}$$

where:

$$u_h = \sum_{i=1}^{n} \alpha_i N_i \tag{3.45}$$

$$w = \sum_{i=1}^{n} \beta_i N_i \tag{3.46}$$

$$\bar{w} = \sum_{i=1}^{n} \beta_i N_i \tag{3.47}$$

Expanding of equation 3.44 with these definitions results in:

$$\int_\Omega r(\sum_{j=1}^{n} \alpha_j N_j) \sum_{i=1}^{n} \beta_i N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} \alpha_j N_j) \sum_{i=1}^{n} \beta_i N_i d\Gamma = 0 \qquad \alpha, N \in V_h \quad , \quad \forall \beta \in V_h \tag{3.48}$$

The following alternative form is obtained by taking out the $\beta_i$ from integrals:

$$\sum_{i=1}^{n} \beta_i \left[ \int_\Omega r(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Gamma \right] = 0 \qquad \alpha, N \in V_h \quad , \quad \forall \beta \in V_h \tag{3.49}$$

As $\beta_i$ are arbitrary, all components of above sum must be zero in order to satisfy the equation. This results in the following set of equations:

$$\int_\Omega r(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} \alpha_j N_j) N_i d\Gamma = 0 \qquad \alpha, N \in V_h \quad , \quad i = 1, 2, 3, ..., n \tag{3.50}$$

An observation here is the equivalence of the discrete form in equation 3.50 with the one in equation 3.34 obtained by the Galerkin method in section 3.1.3.

### Node and Degree of Freedom

In the previous section, the general process of converting the continuous integral form in equation 3.40 to its discrete form in equation 3.50 was explained. In the finite element method, each unknown value is referred as a *degree of freedom (dof)* and it is considered to be the finite element solution $u_h$ at a domain point called *node*.

$$\alpha_i = a_i \tag{3.51}$$

where $a_i$ is the approximate solution $u_h$ at node $i$. Using this assumption the set of equations 3.50 can be rewritten as follows:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j)N_i d\Omega + \int_\Gamma \bar{r}(\sum_{j=1}^{n} a_j N_j)N_i d\Gamma = 0 \qquad a, N \in V_h \quad , \quad i = 1, 2, 3, ..., n \tag{3.52}$$

This set of equations can be solved to obtain directly the unknowns at each node of the domain. Substituting equation 3.51 into 3.45 results:

$$u_h = \sum_{i=1}^{n} a_i N_i \tag{3.53}$$

which relates the approximated solution $u_h$ over the domain with the nodal values obtained from solving the previous set of equations 3.52. In this way, the approximate solution can be obtained not only at all nodes but also at any other points of the domain.

### Shape Functions and Elements

Returning to the equation 3.52 a set of trial functions $N_i$ are necessary to define the problem. In the finite element method these functions are called *shape functions*. The correct definition of the shape functions plays an important role in the correct approximation of the solution and its many important properties.

The discretization introduced in the previous section reduced the infinite number of unknowns in equation 3.40 to a finite number $n$ in the set of equations 3.52. This is a big step in making the model numerically solvable but still is not complete for solving it in practice. The problem comes from the fact that each equation in set of equations 3.52 involves a complete integration over the domain and a complete relation between the unknowns in the equations which makes the solution very costly. To avoid these problems, let us divide the domain $\Omega$ into several sub-domains $\Omega^e$ as follows:

$$\Omega^1 \cup \Omega^2 \cup ... \cup \Omega^e \cup ... \cup \Omega^m = \Omega \quad , \quad \Omega^1 \cap \Omega^2 \cap ... \cap \Omega^e \cap ... \cap \Omega^m = \emptyset \tag{3.54}$$

where each partition $\Omega^e$ is called an *element*. Dividing the integrals in equations 3.52 yields:

$$\sum_{e=1}^{m} \left[ \int_{\Omega^e} r(\sum_{j=1}^{n} a_j N_j)N_i d\Omega^e + \int_{\Gamma^e} \bar{r}(\sum_{j=1}^{n} a_j N_j)N_i d\Gamma^e \right] = 0 \quad i = 1, 2, 3, ..., n \tag{3.55}$$

where $\Gamma^e$ is the part of boundary $\Gamma$ related to element $\Omega_e$ as shown in figure 3.6. Now let us define the shape function $N_i$ as follows:

$$N_i = \begin{cases} N_i^e & x \in \Omega^e \cup \Gamma^e \\ 0 & x \notin \Omega^e \cup \Gamma^e \end{cases} \tag{3.56}$$

where $\Omega^e$ is a partition of domain called *element*. Substituting this shape function into equation 3.52 results in the following equation:

$$\sum_{e=1}^{m_i} \left[ \int_{\Omega^e} r(\sum_{j=1}^{m_i} a_j N_j^e)N_i^e d\Omega^e + \int_{\Gamma^e} \bar{r}(\sum_{j=1}^{m_i} a_j N_j^e)N_i^e d\Gamma^e \right] = 0 \quad i = 1, 2, 3, ..., n \tag{3.57}$$

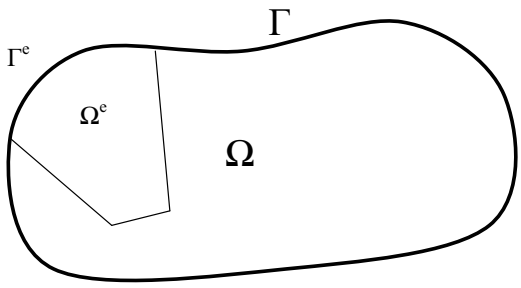



Figure 3.6: An element $\Omega^e$ and its boundary $\Gamma^e$.

Where $m_i$ is the number of elements containing node $i$. In this manner the relation between unknowns is reduced to those between the neighbors and in each equation the integration must be performed only over some elements.

### Boundary Conditions

Considering the following problem:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}(u(x)) = q & x \in \Gamma \end{cases} \tag{3.58}$$

and assuming that $\mathcal{L}$ contains at most $m$th-order derivatives. The boundary condition of such a problem can be divided in two categories: *essential* and *natural* boundary conditions.

The essential boundary conditions $\mathcal{S}_D$ are conditions that contain derivatives with order less equal to $m-1$. These conditions are also called *Dirichlet* conditions (named after Peter Dirichlet). For example the prescribed displacement in structural problems is a Dirichlet condition.

The rest of boundary conditions are considered to be natural boundary conditions $\mathcal{S}_N$. These conditions are also referred as *Neumann* boundary conditions (named after Carl Neumann). For example in a structural problem the boundary tractions are the Neumann condition of the problem.

Applying this division to equation 3.58 results in:

$$\begin{cases} \mathcal{L}(u(x)) = p & x \in \Omega \\ \mathcal{S}_D(u(x)) = q_D & x \in \Gamma_D \\ \mathcal{S}_N(u(x)) = q_N & x \in \Gamma_N \end{cases} \tag{3.59}$$

where $\mathcal{S}_D$ is the Dirichlet condition applied to boundary $\Gamma_D$ and $\mathcal{S}_N$ is the Neumann condition applied to boundary $\Gamma_N$ as can be seen in Figure 3.7. Transforming equation 3.59 to its integral form results:

$$\int_\Omega r(u)w d\Omega + \int_{\Gamma_D} \bar{r}_D(u)\bar{w} d\Gamma_D + \int_{\Gamma_N} \bar{r}_N(u)\bar{w} d\Gamma_N = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.60}$$

where:

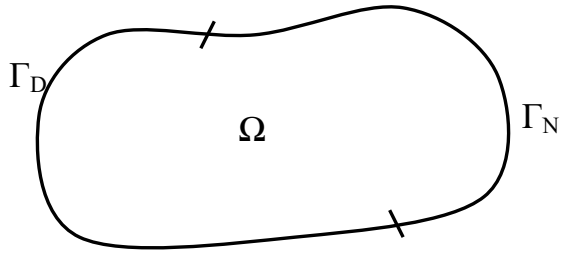



Figure 3.7: A domain and its Dirichlet and Neumann boundaries.

$$r(u) = \mathcal{L}(u) - p \tag{3.61}$$
$$\bar{r}_D(u) = \mathcal{S}_D(u) - q_D \tag{3.62}$$
$$\bar{r}_N(u) = \mathcal{S}_N(u) - q_N \tag{3.63}$$

If the choice of solution $u$ is restricted to functions that satisfy the Dirichlet condition on $\Gamma_D$, the integral over the Dirichlet boundary $\Gamma_D$ can be omitted by restricting the choice of $\bar{w}$ to functions which are zero on $\Gamma_D$. Using these restrictions results in:

$$\int_\Omega r(u) w d\Omega + \int_{\Gamma_N} \bar{r}_N(u)\bar{w} d\Gamma_N = 0 \qquad u \in V, \forall w, \bar{w} \in V \tag{3.64}$$
$$u = \bar{u} \qquad x \in \Gamma_D \tag{3.65}$$

where $\bar{u}$ is the solution over the Dirichlet boundary. Converting the above model to its discrete form using the same process described before results in the following discrete equations:

$$\int_\Omega r(\sum_{j=1}^{n} a_j N_j) N_i d\Omega + \int_{\Gamma_N} \bar{r}_N(\sum_{j=1}^{n} a_j N_j) N_i d\Gamma_N = 0 \quad a, N \in V_h \quad , \quad i = 1, 2, ..., n \tag{3.66}$$
$$u = \bar{u} \quad x \in \Gamma_D \tag{3.67}$$

### 3.2.2 Solution

In this section the global flow of a general finite element solution process will be described and some techniques used to improve the efficiency in practice will be explained.

#### Calculating Components

Applying the essential condition to equation 3.57 results in:

$$\sum_{e=1}^{m_i} \left[ \int_{\Omega^e} r(\sum_{j=1}^{m_i} a_j N_j^e) N_i^e d\Omega^e + \int_{\Gamma_N^e} \bar{r}(\sum_{j=1}^{m_i} a_j N_j^e) N_i^e d\Gamma_N^e \right] = 0 \quad i = 1, 2, ..., n \tag{3.68}$$
$$u = \bar{u} \quad x \in \Gamma_D \tag{3.69}$$

If the differential equations are linear we can write the equation above as follows:

$$\mathbf{Ka} + \mathbf{f} = 0 \tag{3.70}$$

where:

$$\mathbf{K}_{ij} = \sum_{e=1}^{m} \mathbf{K}_{ij}^{e} \tag{3.71}$$

$$\mathbf{f}_{i} = \sum_{e=1}^{m} \mathbf{f}_{i}^{e} \tag{3.72}$$

This step consist of calculating the shape functions and their derivatives in each element and then perform the integration for each element.

The usual technique here is to calculate these components in local coordinates of the element and transform the result to global coordinates. Usually the shape functions are defined in terms of local coordinates and their values and gradients with respect to local coordinates are known. However the elemental matrices contain gradients of shape functions with respect to global coordinates. These gradients can be calculated using the local ones and the inverse of some matrix **J** known as the *jacobian matrix*. Considering the global coordinate $x$,$y$,$z$ and elemental local coordinates $\xi$,$\eta$,$\zeta$, it can be seen that the gradients of the shape functions with respect to the local coordinates can be written in terms of the global ones as follows:

$$\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix}$$

where **J** is the jacobian matrix. Now the gradients of the shape functions with respect to the global coordinates can be calculated as follows:

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix}$$

After calculating the gradients of the shape functions respect to the global coordinates, we can integrate them over the elements. This can be done by transforming the integration domain from global to local coordinates as follows:

$$\int_{\Omega^e} f d\Omega^e = \int_{\Omega^e} f \; det\mathbf{J} \; d\xi \, d\eta \, d\zeta \tag{3.73}$$

Now all elemental matrices can be calculated using local coordinates and transformed to global coordinates. The usual way to calculate the integrals over the elements is using a *Gaussian Quadrature* method. This method converts the integration of a function over the domain to a weighted sum of function values at certain sample points as follows:

$$\int_{a}^{b} f(x) \, dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{3.74}$$

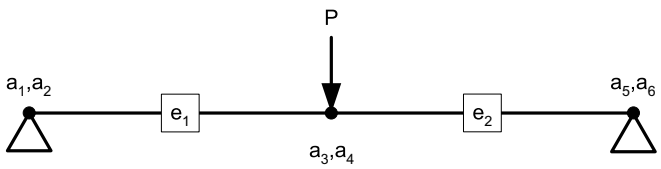



Figure 3.8: A simple beam example with two elements and three nodes.

This method uses nearly half the sample points to achieve the same level of accuracy of other classical quadratures. Thus it is an effective method for calculating elemental integrals in FEM. This method as well as some other integration methods will be explained later in section 5.1.

### Creating the Global System

As mentioned earlier for linear differential equations the set of equations 3.68 can be written as a global system of equations in the form of:

$$\mathbf{K}\mathbf{a} = \mathbf{f} \tag{3.75}$$

where:

$$\mathbf{K}_{ij} = \sum_{e=1}^{m} \mathbf{K}_{ij}^{e} \tag{3.76}$$

$$\mathbf{f}_{i} = \sum_{e=1}^{m} \mathbf{f}_{i}^{e} \tag{3.77}$$

However the sums in the equation above must be applied to the corresponding coordinates. Having the elemental matrices and vectors $\mathbf{K}^e$ and $\mathbf{f}^e$, the procedure of putting them together in order to create the global system of equation 3.75 is called *assembly* and consists of finding the position of each elemental component in the global equations system and adding it to the value in its position.

This procedure first assigns a sequential numbering to all dofs. Sometimes its useful to separate the *restricted* dofs, ones with Dirichlet conditions, from others. This can be done easily at the time of assigning indices to dofs. After that the procedure goes element by element and adds their local matrices and vectors to the global equations system using the following assembly operator $\bigsqcup$:

$$\mathbf{K}_{ij} \bigsqcup_{\mathbf{I}^e} \mathbf{K}_{ij}^{e} = \mathbf{K}_{\mathbf{I}_i^e \mathbf{I}_j^e} + \mathbf{K}_{ij}^{e} \tag{3.78}$$

$$\mathbf{f}_{i} \bigsqcup_{\mathbf{I}^e} \mathbf{f}_{i}^{e} = \mathbf{f}_{\mathbf{I}_i^e} + \mathbf{f}_{i}^{e} \tag{3.79}$$

where $I^e$ is the vector containing the global position, which is the index of the corresponding dof, of each row or column. For example considering the beam problem of figure 3.8 with two elements and the following elemental matrices and vectors:

$$\mathbf{K}^{(1)} = \begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} & K_{13}^{(1)} & K_{14}^{(1)} \\ K_{21}^{(1)} & K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} \\ K_{31}^{(1)} & K_{32}^{(1)} & K_{33}^{(1)} & K_{34}^{(1)} \\ K_{41}^{(1)} & K_{42}^{(1)} & K_{43}^{(1)} & K_{44}^{(1)} \end{bmatrix} \quad , \quad \mathbf{f}^{(1)} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} \\ f_3^{(1)} \\ f_4^{(1)} \end{bmatrix} \tag{3.80}$$

and:

$$\mathbf{K}^{(2)} = \begin{bmatrix} K_{11}^{(2)} & K_{12}^{(2)} & K_{13}^{(2)} & K_{14}^{(2)} \\ K_{21}^{(2)} & K_{22}^{(2)} & K_{23}^{(2)} & K_{24}^{(2)} \\ K_{31}^{(2)} & K_{32}^{(2)} & K_{33}^{(2)} & K_{34}^{(2)} \\ K_{41}^{(2)} & K_{42}^{(2)} & K_{43}^{(2)} & K_{44}^{(2)} \end{bmatrix} \quad , \quad \mathbf{f}^{(2)} = \begin{bmatrix} f_1^{(2)} \\ f_2^{(2)} \\ f_3^{(2)} \\ f_4^{(2)} \end{bmatrix} \tag{3.81}$$

Giving sequential indices to dofs $a_1$ to $a_6$, the index vectors $I^1$ and $I^2$ of elements $e_1$ and $e_2$ will be:

$$I^{(1)} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad , \quad I^{(2)} = \begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \tag{3.82}$$

Finally, assembling the elemental matrices and vectors using the index vectors above results in the following system:

$$\mathbf{Ka} = \mathbf{f} \tag{3.83}$$

with:

$$\mathbf{K} = \begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} & K_{13}^{(1)} & K_{14}^{(1)} & 0 & 0 \\ K_{21}^{(1)} & K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 & 0 \\ K_{31}^{(1)} & K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{13}^{(2)} & K_{14}^{(2)} \\ K_{41}^{(1)} & K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{23}^{(2)} & K_{24}^{(2)} \\ 0 & 0 & K_{31}^{(2)} & K_{32}^{(2)} & K_{33}^{(2)} & K_{34}^{(2)} \\ 0 & 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{43}^{(2)} & K_{44}^{(2)} \end{bmatrix} \tag{3.84}$$

and:

$$\mathbf{f} = \begin{bmatrix} f_1^{(1)} \\ f_2^{(1)} \\ f_3^{(1)} + f_1^{(2)} \\ f_4^{(1)} + f_2^{(2)} \\ f_3^{(2)} \\ f_4^{(2)} \end{bmatrix} \tag{3.85}$$

Another task to be done when building the global system of equations is the application of essential boundary conditions. This can be done easily by eliminating the rows and columns corresponding to restricted dofs from the global matrix and vector and apply their corresponding value to the right hand side. This procedure can be done without reordering of equations but it is more convenient to separate the restricted equations from others in order to simplify the process. Considering the following equation system where the components corresponding to Dirichlet degrees of freedom are separated from others:

$$\begin{bmatrix} \mathbf{K}_{NN} & \mathbf{K}_{DN} \\ \mathbf{K}_{DN} & \mathbf{K}_{DD} \end{bmatrix} \begin{bmatrix} \mathbf{a}_N \\ \mathbf{a}_D \end{bmatrix} = \begin{bmatrix} \mathbf{f}_N \\ \mathbf{f}_D \end{bmatrix} \tag{3.86}$$

where $\mathbf{a}_N$ are unknowns and $\mathbf{a}_D$ are known dofs with Dirichlet boundary condition and $\mathbf{f}_D$ their corresponding boundary unknown. Let us divide the above system in two restricted and not restricted part as follows:

$$[\mathbf{K}_{NN}]\,[\mathbf{a}_N] + [\mathbf{K}_{ND}]\,[\mathbf{a}_D] = [\mathbf{f}_N] \tag{3.87}$$
$$[\mathbf{K}_{DN}]\,[\mathbf{a}_N] + [\mathbf{K}_{DD}]\,[\mathbf{a}_D] = [\mathbf{f}_D] \tag{3.88}$$

Knowing the Dirichlet boundary condition values $\mathbf{a}_D$ let us move them to the right hand side:

$$[\mathbf{K}_{NN}][\mathbf{a}_N] = [\mathbf{f}_N] - [\mathbf{K}_{ND}][\mathbf{a}_D] \tag{3.89}$$

This system of equations can be solved to obtain the unknowns $\mathbf{a}_N$. Considering the previous beam example of figure 3.8. The dofs $a_1$ and $a_5$ are restricted. In order to partition the global system let us reorder the dofs as follows:

$$\tilde{\mathbf{a}} = \{a_2, a_3, a_4, a_6, a_1, a_5\} \tag{3.90}$$

which results in the following index vectors:

$$I^{(1)} = \begin{bmatrix} 5 \\ 1 \\ 2 \\ 3 \end{bmatrix} \quad , \quad I^{(2)} = \begin{bmatrix} 2 \\ 3 \\ 6 \\ 4 \end{bmatrix} \tag{3.91}$$

Now let us assemble the global matrix $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 & K_{21}^{(1)} & 0 \\ K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{14}^{(2)} & K_{31}^{(1)} & K_{13}^{(2)} \\ K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{24}^{(2)} & K_{41}^{(1)} & K_{23}^{(2)} \\ 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{44}^{(2)} & 0 & K_{43}^{(2)} \\ K_{12}^{(1)} & K_{13}^{(1)} & K_{14}^{(1)} & 0 & K_{11}^{(1)} & 0 \\ 0 & K_{31}^{(2)} & K_{32}^{(2)} & K_{34}^{(2)} & 0 & K_{33}^{(2)} \end{bmatrix} \tag{3.92}$$

and vector $\mathbf{f}$:

$$\mathbf{f} = \begin{bmatrix} f_2^{(1)} \\ f_3^{(1)} + f_1^{(2)} \\ f_4^{(1)} + f_2^{(2)} \\ f_4^{(2)} \\ f_1^{(1)} \\ f_3^{(2)} \end{bmatrix} \tag{3.93}$$

Finally applying the Dirichlet boundary condition using equation 3.89:

$$\mathbf{K_{NN}} = \begin{bmatrix} K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 \\ K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{14}^{(2)} \\ K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{24}^{(2)} \\ 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{44}^{(2)} \end{bmatrix} \tag{3.94}$$

and:

$$[\mathbf{R}_N] = [\mathbf{f}_N] - [\mathbf{K}_{ND}][\mathbf{a}_D] = \begin{bmatrix} f_2^{(1)} \\ f_3^{(1)} + f_1^{(2)} \\ f_4^{(1)} + f_2^{(2)} \\ f_4^{(2)} \end{bmatrix} - \begin{bmatrix} K_{21}^{(1)} & 0 \\ K_{31}^{(1)} & K_{13}^{(2)} \\ K_{41}^{(1)} & K_{23}^{(2)} \\ 0 & K_{43}^{(2)} \end{bmatrix} \begin{bmatrix} a_1 \\ a_5 \end{bmatrix} \tag{3.95}$$

$$\begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} = \begin{bmatrix} f_2^{(1)} - K_{21}^{(1)} a_1 \\ f_3^{(1)} + f_1^{(2)} - K_{31}^{(1)} a_1 - K_{13}^{(2)} a_5 \\ f_4^{(1)} + f_2^{(2)} - K_{41}^{(1)} a_1 - K_{23}^{(2)} a_5 \\ f_4^{(2)} - K_{43}^{(2)} a_5 \end{bmatrix} \tag{3.96}$$

results in the following equation to be solved:

$$\begin{bmatrix} K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & 0 \\ K_{32}^{(1)} & K_{33}^{(1)} + K_{11}^{(2)} & K_{34}^{(1)} + K_{12}^{(2)} & K_{14}^{(2)} \\ K_{42}^{(1)} & K_{43}^{(1)} + K_{21}^{(2)} & K_{44}^{(1)} + K_{22}^{(2)} & K_{24}^{(2)} \\ 0 & K_{41}^{(2)} & K_{42}^{(2)} & K_{44}^{(2)} \end{bmatrix} \begin{bmatrix} a_2 \\ a_3 \\ a_4 \\ a_6 \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \tag{3.97}$$

Another way to apply the Dirichlet conditions is to use a *penalty method*. This applies a big coefficient to the diagonal elements corresponding to the restricted degrees of freedom. This method is easier to program but is less robust than the previous one. The problem is to find the correct coefficient because a high number may cause the system to be ill conditioned and a low one is less realistic.

The global system matrix obtained through the FEM typically has a lot of zeros in it. Holding all its values in a *dense* matrix structure, which stores all elements of matrix, implies a large overhead in memory due to the storage of zero elements. There are several alternative structures which hold a useful portion of matrix for solving. For example a *banded matrix* structure holds a band around the diagonal of a matrix and assumes all element outside the band are zero. There are also several *sparse* matrix structures like: *compressed sparse row (CSR)* which stores the nonzero elements of each row with their corresponding numbers of columns, or *compressed sparse column (CSC)* which is the transposed of compressed sparse row as a column major structure. Another common structure is the *symmetric matrix* structure that uses the symmetry property of the matrix to hold approximately half of the elements and can be combined with a sparse structure to store half of the nonzeros in matrix.

### Solving the Global System

In the previous section the global system of equations was prepared and applying the essential conditions made it ready to be solved. This system of equations can be solved using conventional solvers. There are two categories of solvers, *direct solvers* and *iterative solvers*.

Direct solvers try to solve the equation by making the coefficient matrix upper triangular, lower triangular, diagonal, or sometimes decomposing it to upper lower form and calculating the unknowns using this form of matrices. Solvers like *Gaussian elimination* [81], *frontal solution* [23], *LU decomposition* [81] for general matrices, and *Cholesky* [81] for symmetric matrices are examples of this category.

Iterative solvers start with some initial values for the unknown and try to find the correct solution by calculating the residual and minimize it over iterations. *Conjugate gradient (CG)* [90], *Biconjugate gradient (BCG)* [90], *Generalized minimal residual method (GMRES)* [90] are examples of this category of solvers.

Direct solvers are very fast for small systems of equations and are less dependent on the conditioning of the matrix. The only exception is the existence of *pivot*, a zero in diagonal, which needs a special treatment. These solvers are very slow for large systems while the number of operations grows with order $O(N^3)$ where $N$ is the size of system. Algorithms like *multi frontal solution* [36] are used to solve big systems in parallel machines taking advantage of using several processors in parallel. A way to reduce the number of operations needed for solving the problem is to reduce the bandwidth of the system matrix.

Iterative solvers on the contrary are highly dependent on the condition of the system which affects considerably their convergence. Usually for small systems the direct solvers are faster while

for medium to large systems the iterative ones are more suitable, still depending on the condition of system. These methods are also easier to implement and optimize than direct methods.

As mentioned before the solving cost of direct solvers is highly depended on the bandwidth of the system matrix. For this reason a procedure called *reordering* is recommended to reduce the bandwidth of system matrix. This procedure consists of changing the order of rows and columns of the matrix in order to reduce the bandwidth before the solution and then permuting the result back after solving. In practice these algorithms can be applied to renumber the degrees of freedom in optimum way once they are created and then solve the system as usual. The *Cuthill McKee* [90] algorithm is a classical example of these algorithms. Sometimes the reordering process is applied before using iterative solvers to reduce the cache miss produced by the large sparsity of matrix.

Sometimes it is recommended to prepare the system matrix before solving it using an iterative solver. This procedure is called *preconditioning* and consists of transforming the system of equations to and equivalent but better conditioned one for the solution with iterative solvers. *Diagonal* preconditioner [90] for diagonal dominant systems, *Incomplete LU* with tolerance and filling [90] for general nonsymmetric systems, and *Incomplete Cholesky* [90] for symmetric systems are examples of popular preconditioners. Unfortunately, finding the best combination of solver and preconditioner for a certain problem is a question of experience and there is not a single best combination for all problems.

### Calculating Additional Results

In a linear problem after solving the global system of equations, the principal results are obtained and in some sense the problem is solved. But in many cases there are some additional results which are of interest and must be calculated. For example in structural analysis the nodal displacements can be obtained by solving the global system of equations. However the stress in elements is also important and has to be calculated. These values usually are calculated using the primary result, i.e. displacements for each element. For example the elemental stress in a structural problem can be calculated using the displacement values $\mathbf{a}^e$, obtained by solving the global system using the following equation [75]:

$$\sigma = \mathbf{DBa}^e - \mathbf{D}\varepsilon_0 + \sigma_0 \tag{3.98}$$

where $\sigma$ is elemental stress, $\mathbf{D}$ is the elasticity matrix, $\mathbf{B}$ is the strain matrix, $\varepsilon_0$ is the initial strain, and $\sigma_0$ is the initial stress of element. However the results obtained by this equation are usually discontinuous over the domain. This means that the stress result for a node from different elements connected to it is different. For this reason different averaging methods are implemented to smooth the discontinuous results. An alternative is to use recovery methods which try to reproduce continuous gradient results with a better approximation [104].

### Iterating

One may note that the previous sections where mainly based on linear differential equations. So what has to be done if the problem is not linear? There are several methods to deal with nonlinear problems. Unfortunately these methods cannot obtain the results as simply as before and usually need to perform iterations to find the results. Considering the following nonlinear system of equations:

$$\mathbf{K}(u)\mathbf{u} = \mathbf{f} \tag{3.99}$$

One can plan an iterative solution procedure where each set of unknowns $\mathbf{u}_n$ is used to calculate the system of equations and calculate the next set of unknowns $\mathbf{u}_{n+1}$:

$$\mathbf{K}(u_n)\mathbf{u}_{n+1} = \mathbf{f} \tag{3.100}$$

There are several methods for accelerating the convergence of this procedure. Some examples are *Newton method*, *Modified Newton method*, *Line search* [104], etc. As mentioned before all these methods need iterations over the solution.

## 3.3 Multi-Disciplinary Problems

The objective of this work is creating a framework to deal with multi-disciplinary problems. So before getting any further, it is important to give a general description of these problems and describe some important features of them briefly.

### 3.3.1 Definitions

There are different definitions for multi-disciplinary problems. A multi-disciplinary solutions is usually defined as solving a *coupled system* of different physical models together. A coupled system is assumed to be a collection of dependent problems put together defining the model.

In this work a multi-disciplinary problem, also called *coupled problem*, is defined as *solving a model which consists of components with different formulations and algorithms interacting together*. It is important to mention that this difference may come not only from the different physical nature of the problems but also from their different type of mathematical modeling or discretization.

A *field* is a subsystem of multi-disciplinary model representing a certain mathematical model. Typical examples are a fluid field and a structure field in a fluid-structure interaction problem. In a coupled system a *domain* is the part of a modeled space governed by a field equation, i.e. a structure domain and a fluid domain.

### 3.3.2 Categories

The definition given for multi-disciplinary problem includes a wide range of problems with very different characteristics. These problems can be grouped into different categories reflecting some of their aspects affecting the solution procedure. One classification can be made by how different subsystems interact with each other. Another classification can be done reflecting the type of domain interfaces.

#### Weak and Strong Coupling

One may classify multi-disciplinary problems by the type of coupling between the different subsystems. Consider a problem with two interacting subsystem as shown in figure 3.9.

The problem is calculating the solutions $u_1$ and $u_2$ of subsystems $S_1$ and $S_2$ under applied forces $F(t)$. There are two types of dependency between the subsystems:

**Weak Coupling** Also called *one-way* coupling where one domain depends on the other but this can be solved independently. A thermal-structure problem is a good example of this type of coupling. In this problem the material's property of the structure depends on the temperature while the thermal field can be solved independently, assuming the temperature change due to the structural deformation is very small. Figure 3.10 shows this type of coupling.

**Strong Coupling** Also referred as *two-way* coupling when each system depends on the other and hence none of them can be solved separately. The fluid-structure interaction problems for structures with large deformations fall into this category. In these problems, the structure deforms by the pressure coming from the fluid, and the fluid velocity and pressure depend on the shape of the deformed structure. Figure 3.11 shows this type of coupling.

#### Interaction Over Boundary and Domain

As mentioned before one classification of multi-disciplinary problems relies on how subsystems interact together. Another classification can be done by looking not on how they interact but where they interact with each other. There are two categories of multi-disciplinary problems using this criteria [104]:

**Class I** In this category the interaction occurs at the boundary of the domains. For example in a fluid-structure interaction problem the interaction occurs at the boundary of the structure in contact with fluid and vice versa. Figure 3.12 shows an example of this type of problems.

**Class II** This category include problems where domains can overlap totally or partially. A thermal-fluid problem is a good example of this class of problems where the domain of the fluid and the thermal problem overlap. Figure 3.13 shows an example of this type of problems.

### 3.3.3 Solution Methods

There are several different approaches for solving multi-disciplinary problems. Finding a suitable approach for each case, highly depends on the category of the problem and the different details of each field specially for time dependent problems. In this section an overview of different methodologies for solving these problems will be discussed.

#### Sequential Solution of Problems with Weak Coupling

The solving procedure of one-way coupled problems is trivial. Considering the problem of figure 3.10 with two subsystem $S_1$ and $S_2$ where $S_2$ depends on $u_1$ (the solution of $S_1$). This problem can be solved easily by solving $S_1$ first and using its solution $u_1$ for solving $S_2$ as shown in figure 3.14. For transient problems this can be done at each time step.

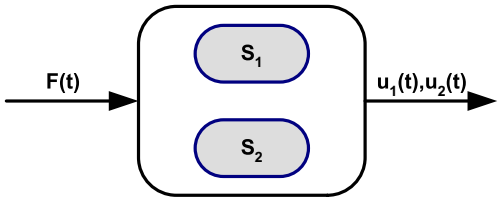



Figure 3.9: A general multi-disciplinary problem with two subsystems.

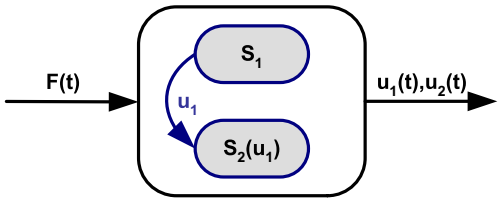



Figure 3.10: A weak coupled system where the subsystem $S_2$ depends on the solution of subsystem $S_1$.

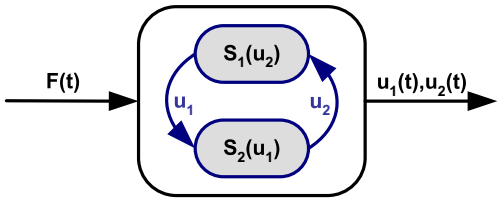



Figure 3.11: A strong coupled system where not only the subsystem $S_2$ depends on the solution of subsystem $S_1$ but also subsystem $S_1$ depends on $S_2$.

### Monolithic Approach

In this approach the interacting fields are modeled together which results in a coupled continuous model and finally a multi-disciplinary element to be used directly. Consider the problem with strong coupling in figure 3.11 where not only subsystem $S_2$ depends on the solution of subsystem $S_1$, but also subsystem $S_1$ depends on $S_2$:

$$\mathcal{L}_1(u_1, u_2, t) = f_1(t) \tag{3.101}$$
$$\mathcal{L}_2(u_1, u_2, t) = f_2(t) \tag{3.102}$$

Applying the discretization over time and space one can rewrite the above equation in the following form for each time step:

$$\begin{bmatrix} \mathbf{K}_1 & \mathbf{H}_1 \\ \mathbf{H}_2 & \mathbf{K}_2 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1(t) \\ \mathbf{f}_2(t) \end{bmatrix} \tag{3.103}$$

where $\mathbf{K}_1$ and $\mathbf{K}_2$ are the field system matrices corresponding to the field variables and $\mathbf{H}_1$ and $\mathbf{H}_2$ are the field system matrices corresponding to interaction variables. These equations can be solved at each time step in order to calculate the solutions of both fields. Figure 3.15 shows this scheme.

Though this approach seems to be very easy and natural, in practice it encounters difficulties. One problem is the difficulty of the formulation. The multi-disciplinary continuous models, are usually complex by nature and this complexity makes the discretization process a tedious task. However by using computer algebra systems like *Mathematica* [103], and *Maple* [66], the symbolic derivation in this approach becomes more feasible.

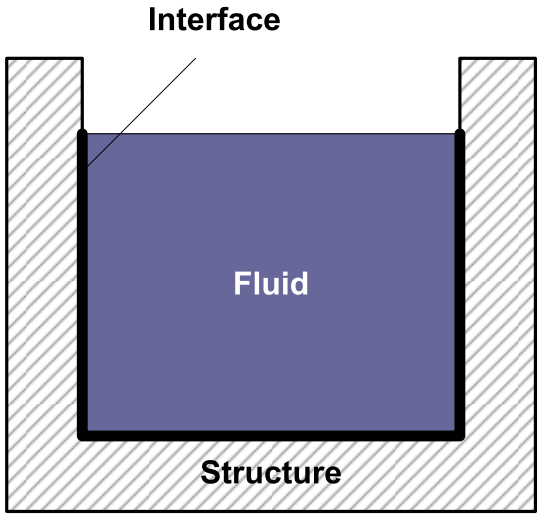



Figure 3.12: A class I fluid-structure interaction problem. The interface, shown by the thick black line, is just at the boundary of the fluid and structure domains.

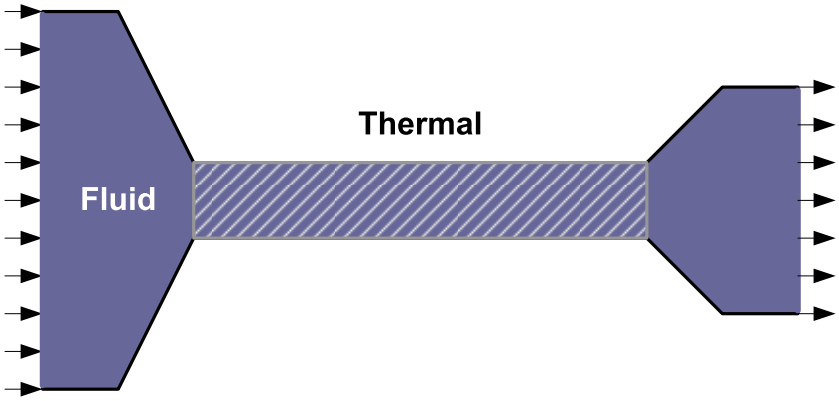



Figure 3.13: A thermal-fluid interaction problem. Here the thermal domain and the fluid domain overlap in the heating pipe part.

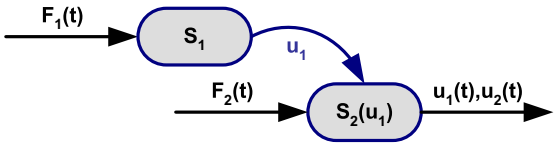



Figure 3.14: Sequential solving of one-way coupled problems.

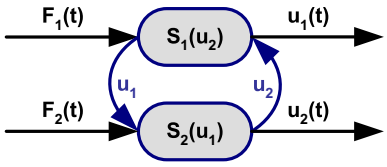



Figure 3.15: Monolithic scheme for solving multi-disciplinary problems.

Another problem is the size and bandwidth of the global system. In this method all fields have to be solved together which make it an expensive approach.

An additional disadvantage is the implementation cost. Implementing the interaction of a certain field with any new field requires the interface matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ to be customized to reflect the new variables. These are not reusable for another interaction of that field. Also any existing codes, for solving each field individually, cannot be reused as they are and in many cases severe modifications are necessary for adapting them to this approach.

In spite of these problems, this approach perfectly models the interaction and results in a more robust and more stable formulation for solving coupled problems.

### Staggered Methods

The intention of *staggered methods* is to solve each field separately and simulate the interaction by applying different techniques for transforming variables from one field to another. Some common techniques for staggered methods are described below:

**Prediction** This technique consists of predicting the value of the dependent variables in the next step. For example in the two-way coupled problem of figure 3.11 a prediction of the variable $u_2^{(n+1)}$ can be used to solve the $S_1$ subsystem separately. This technique is widely used to decouple different fields in problems with strong coupling. Figure 3.16 shows the prediction's scheme.

There are different methods for predicting the solution for the next step. One common method is the *last-solution predictor* which uses the actual value of the variable as the predicted value for the next step:

$$u_p^{(n+1)} = u^{(n)} \tag{3.104}$$

Another common choice is the prediction by solution gradient which applies a predicted variation to the actual value of variable to obtain the predicted value in the next step:

$$u_p^{(n+1)} = u^{(n)} + \Delta t \dot{u}^{(n)} \tag{3.105}$$

with:

$$\Delta t = t^{(n+1)} - t^{(n)} \tag{3.106}$$

$$\dot{u}^{(n)} = \left(\frac{\partial u}{\partial t}\right)^{(n)} \tag{3.107}$$

**Advancing** Calculating the next time step of a subsystem using the calculated or predicted solution of other subsystem. Figure 3.17 shows this technique.

**Substitution** Substitution is a trivial technique which uses the calculated value of one field in another field for solving it separately. Figure 3.18 shows its scheme.

**Correction** Considering the $S_1$ subsystem which is solved using the predicted solution $u_{2p}^{(n+1)}$ to obtain $u_1^{(n+1)}$. Advancing the subsystem $S_2$ using $u_1^{(n+1)}$ results in $u_2^{(n+1)}$. The correction step consist of substituting $u_2^{(n+1)}$ in place of the predicted value $u_{2p}^{(n+1)}$ and solve again $S_1$ to obtain a better result. Obviously this procedure can be repeated several times. Figure 3.19 shows this procedure.

An staggered method can be planed using the techniques above. For example, returning to the problem of figure 3.11, one can plan the following staggered method:

1. Prediction: $u_p^{(n+1)} = u_2^{(n)} + \Delta t \dot{u}_2^{(n)}$
2. Advancing: $S_1^{(n+1)}(u_p^{(n+1)}) \rightarrow u_1^{(n+1)}$
3. Substitution: $u_1^{(n+1)} = u_1^{(n+1)}$ for $S_2$
4. Advancing: $S_2^{(n+1)}(u_1^{(n+1)}) \rightarrow u_2^{(n+1)}$

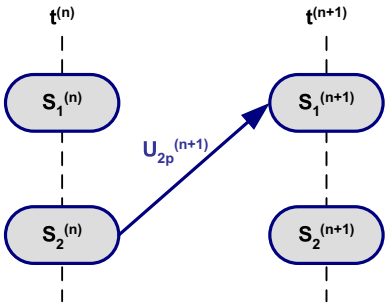



Figure 3.16: The prediction technique consists of predicting the value of the interaction variable for the next step and use it to solve the other field.

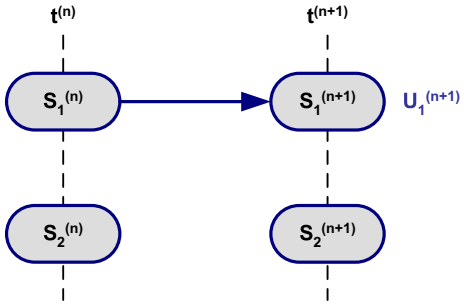



Figure 3.17: The advancing is calculating the solution of the next time step ($S_1$) using the calculated or predicted solution of other subsystem ($S_2$).

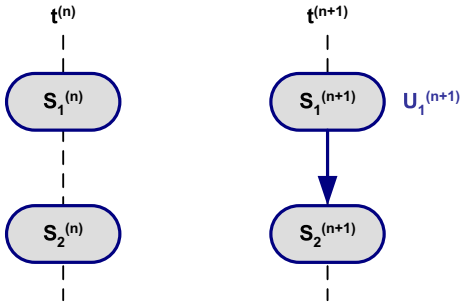



Figure 3.18: The substitution technique consists of substituting the calculated interaction variables in the field $S_1$ into the field $S_2$ to calculate it separately.

Figure 3.20 shows this procedure.

More information about staggered methods and their techniques can be found in [41, 42].

Staggered methods use less resources than the monolithic approaches because in each step solve only one part of problem. This can be a great advantage in solving large problems. Also this gives an idea to use the same process for solving large single field problems over several machines in parallel.

The other advantage is the possibility of reusing existing single field codes for solving multi-disciplinary problems almost without modification. This can be done by writing an small program to control the interaction between independent programs for each field. This strategy is referred as *master and slave method* and widely used for solving multi-disciplinary problems.

This approach also enables the use of different discretizations for each field. It also lets each field have its own mesh characteristics. This can be a great added value for solving large and complex coupled problems.

As usual, beside all advantages there are some disadvantages. Staggered methods require a careful formulation to avoid instability and obtain an accurate solution. In general the staggered method is less robust than the monolithic approach and needs more attention in time of modeling and solving.

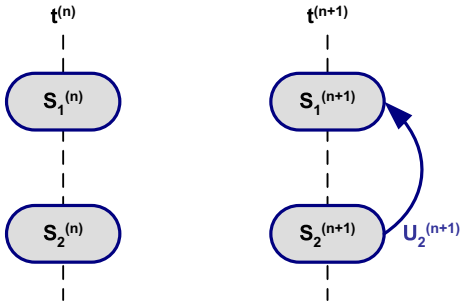



Figure 3.19: The correction consist of replacing the predicted solution with recently calculated one and resolve the subsystem to obtain a better solution.

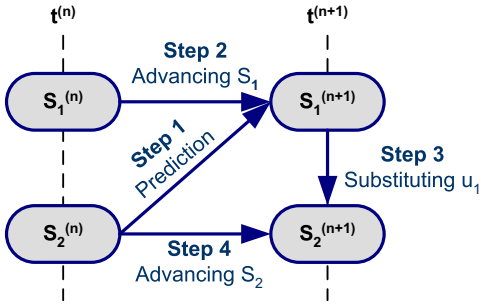



Figure 3.20: An staggered method for solving a coupled system.

## 3.4 Programming Concepts

Designing and implementing a new software is a hard task. Using proper software engineering solutions and also advanced programming techniques can significantly increase the quality of the program. This chapter describes different software engineering solutions and programming techniques which are useful for designing a finite element program.

### 3.4.1 Design Patterns

Designing usually consists of several decisions that affect the feature reusability, flexibility, and extendability of the code. However there are several classical problems that appearing during the design and can be solved easily by applying existing *Design Patterns*. Design patterns are some reusable patterns for designing a part of program representing a known problem. In this section a set of patterns that can be used in designing a finite element program are briefly explained.

#### Strategy Pattern

*Strategy pattern* defines a family of algorithms by encapsulating each algorithm in one separate class and making them interchangeable via a uniform interface established by their base class.

Figure 3.21 shows this pattern.

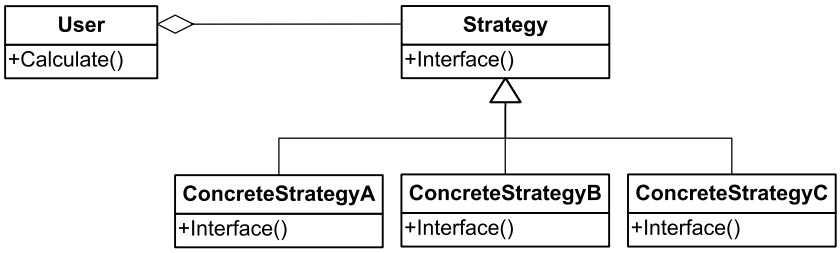



Figure 3.21: Strategy pattern's structure

In this structure `Strategy` declares the interface for all strategies. `User` has a reference to a `Strategy` object and uses the `Strategy` interface to call the algorithm. `User` also may let `Strategy` access its data via an interface. Finally each `ConcreteStrategy` implements an algorithm using the `Strategy` interface.

There are many points in finite element program design where this pattern can be used. Linear solvers, geometries, elements, condition, processes, strategies, etc. Figure 3.22 shows an example of using this pattern for designing a linear solver's structure:

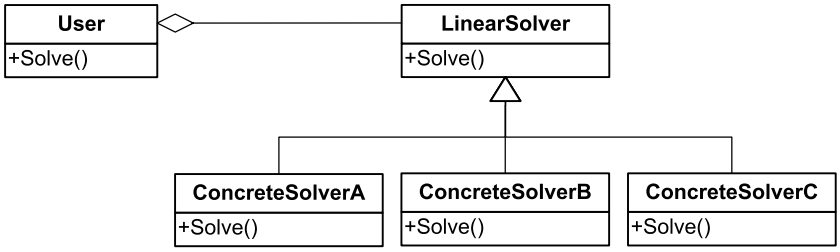



Figure 3.22: Structure designed for linear solvers using strategy pattern

Using this pattern each class derived form `LinearSolver` encapsulates one solving algorithm separately. This encapsulation makes a library easier to extend. The interface is defined by the `LinearSolver` base class and is uniform for all derived solvers. User keeps a pointer to `LinearSolver` base class which may point to any member of the solver family and use the interface of the base class to call different procedures.

### Bridge pattern

The *bridge* pattern decouples the abstraction and its implementation in such a way that they can change independently. Figure 3.23 shows the structure of this pattern.

In this pattern `Abstraction` defines the interface for the user and also holds the implementor reference. `AbstractionForm` create a new concept and may also extend the Abstraction interface.

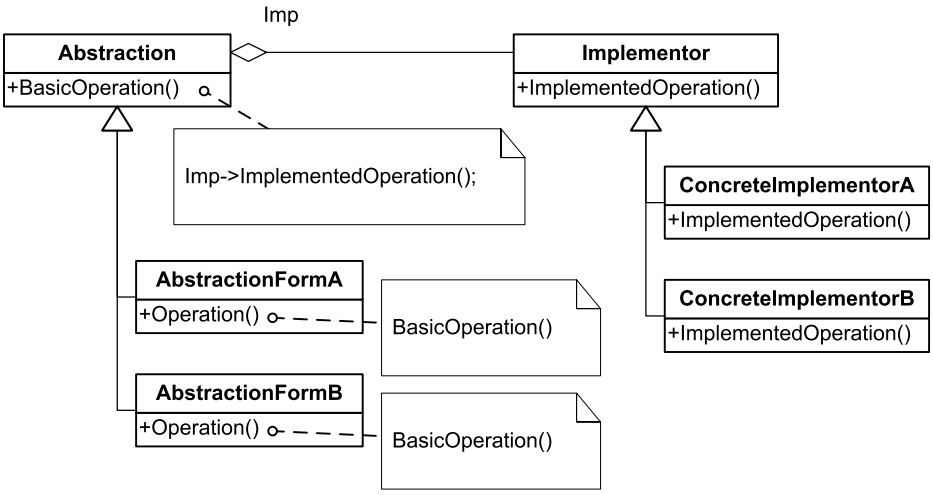



Figure 3.23: Bridge pattern's structure.

`Implementor` defines the interface for implementation part which is used by abstractions. Each `ConcreteImplementor` implements the implementor interface for a concrete case.

Bridge pattern is useful for connecting to concepts with hierarchical structure. In a finite element program this pattern can be used to connect elements to geometries, linear solvers to their reorderer, or to connect iterative solvers to preconditioners.

For example, applying a bridge pattern to an `Element`'s structure results in the structure shown in Figure 3.24.

This pattern lets each `Element` combine its formulation to any `Geometry`.

### Composite Pattern

The *Composite* pattern lets users group a set of object in one composite object and treat individual objects and compositions of objects uniformly. Figure 3.25 shows the structure of this pattern.

In this pattern `Component` defines the interface for object operations and also declares the interface for accessing and managing the child components. It may also define an interface for accessing the component's parent in reversible structures. `Leaf` has no children and implements just the operation. It can be used as a basic unit in composition. `Composite` stores its children and implements the child management interface. It also implements its operation by using operations of its children. Finally user can use the component interface to work with all objects in composition uniformly.

This pattern can be used to design processes which can be constructed by a set of processes, geometries with ability of grouping them in a composite one, or even elements grouping different elements in one and mixing formulations.

For example applying this pattern to process results in the structure shown in figure 3.26.

This structure lets users to combine different process in one and use it like any other process.

### Template Method Pattern

The *Template Method* pattern defines the skeleton of an algorithm separately and defers some steps to subclasses. In this way template method pattern lets subclasses redefine certain steps

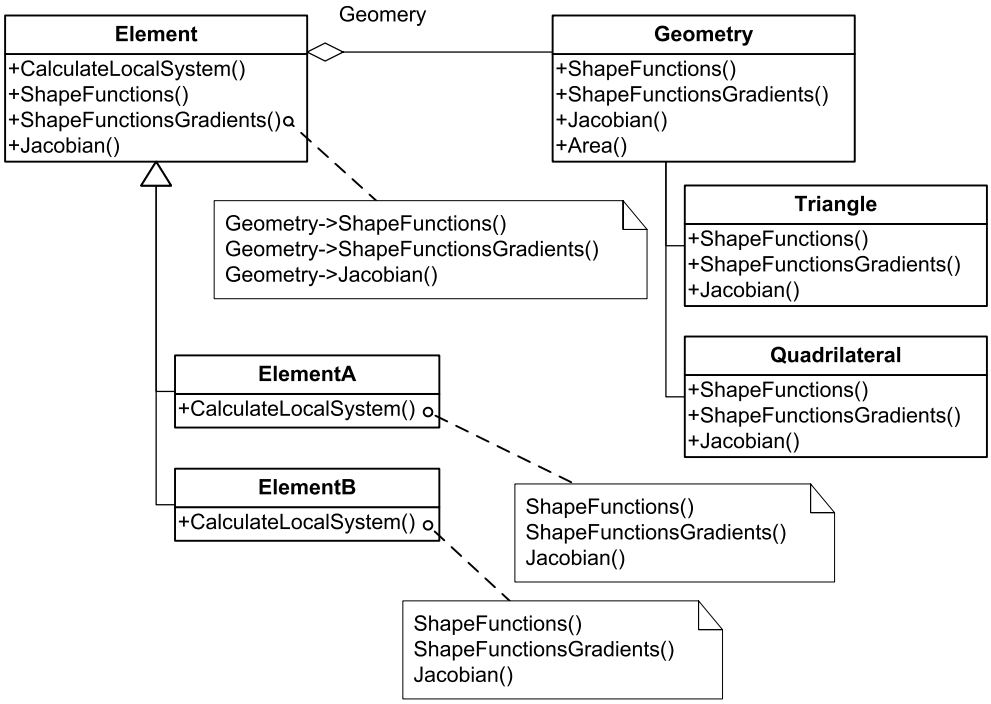



Figure 3.24: Element's structure using bridge pattern.

of an algorithm without changing the algorithm's structure. Figure 3.27 shows structure of this pattern.

In this structure `AbstractClass` defines abstract primitive operations and also implements the skeleton of an algorithm in a template method. `ConcreteClass` implements the primitive operations which will be used by the template method as changed steps of algorithm.

This method is useful in situations when various algorithms differ in some of their steps but not in global. Strategies can be designed using this pattern to provide a category of algorithms changeable by schemes or applying this pattern to linear solver design can make them independent from the matrix and vectors and their operations.

Figure 3.28 shows an example of using this pattern in designing strategies.

### Prototype Pattern

The *Prototype* pattern provides a set of prototypes of objects to be created. User clones the prototype to create a new object of that type. System is extendible to any new type whose prototype is available. Figure 3.29 shows the structure of this pattern.

Prototype provides the cloning interface and is the common base class useful to create prototypes list. Each concrete prototype implements the cloning operation for itself. User creates a new object by asking its related prototype to clone itself.

Prototype pattern is very useful for designing an extendible IO. IO can use a prototype of new object and create it without problem. Figure 3.30 shows a use of this pattern in IO for creating elements.

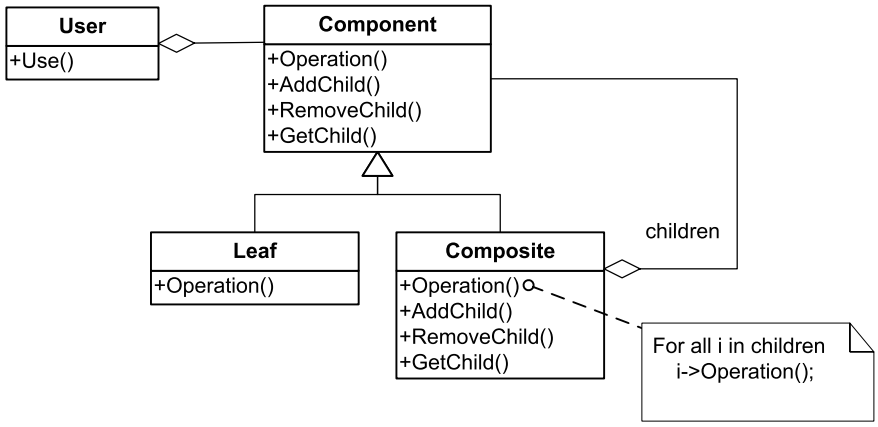



Figure 3.25: Composite pattern.

### Interpreter Pattern

The *Interpreter* pattern creates a representation for given grammar and then use it to interpret the langauge. This pattern has simple structure as shown in figure 3.31.

`AbstractExpression` defines the `Interpret` interface for all nodes in syntax tree. `TerminalExpression` represents a terminal symbol in grammar and implements the Interpret method for it. For each terminal symbol in a sentence an instance of it has to be created. `NonterminalExpression` represents a nonterminal symbol in context free grammar. It holds instances of all expressions in its sentence. It also implements the `Interpret` method which usually consist of calling it members `Interpret` method.

### Curiously Recursive Template Pattern

The *Curiously Recursive Template (CRT)* pattern, also called as *Barton and Nackman Trick*, consists of giving the derived class to its base class as its template argument. The idea is configuring

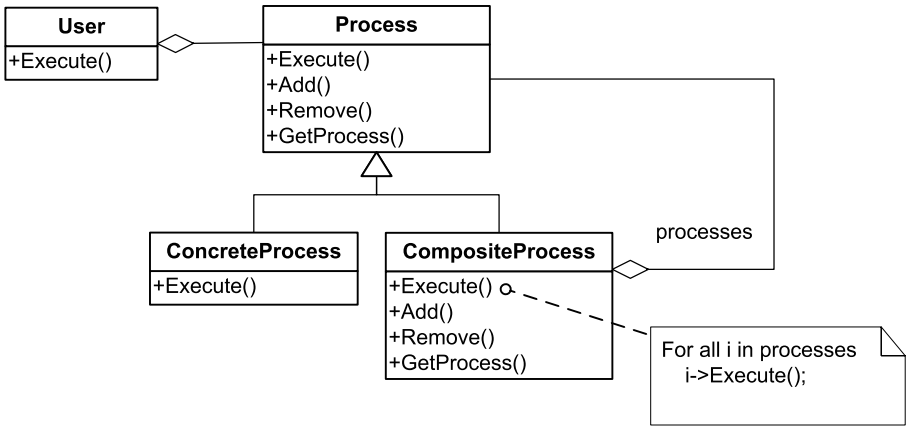



Figure 3.26: Applying composite pattern to processes' structure.

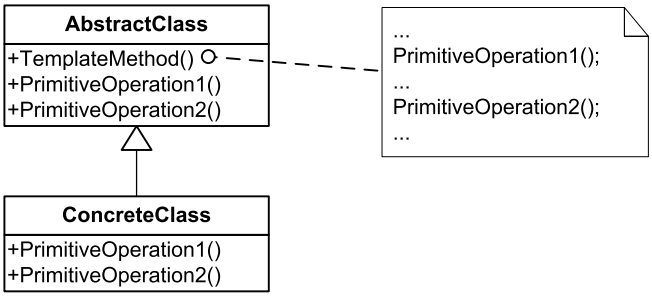



Figure 3.27: Template Method pattern structure.

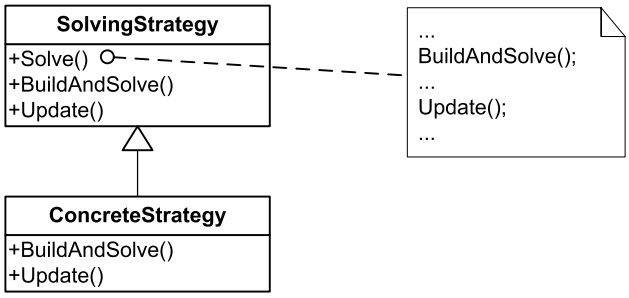



Figure 3.28: Template Method pattern applied to solving strategy.

a base class depending on its derived class and providing a type of static polymorphism which can be much more efficient than usual polymorphism by deriving and overriding the virtual functions. Figure 3.32 shows this pattern.

Using this pattern lets developer customize the base class without losing efficiency in operation. This pattern is more effective when the operation is very simple and the overhead of virtual function calling is considerable. For example a matrix library can use this pattern to let a symmetric matrix derived class change the operators of a base matrix class. In this way methods like access methods, assignments, etc. can be overridden without producing performance overhead. Figure 3.33 shows this pattern applied to the matrix example above.

The patterns described in this section are the ones used in the design of the Kratos. Description for other patterns and also more detailed explanation of patterns mentioned before can be found in [45].

### 3.4.2 C++ advanced techniques

Performance and memory efficiency are two crucial requirement for finite element programs. It has been shown that an optimized implementation of numerical methods in C++ can provide the same performance of Fortran implementations [100] and usually the inefficiency of the C++ codes comes from the developer's misunderstanding of the language [54]. For this reason it can be helpful to take a look at different techniques used for implementing high performance and efficient numerical algorithms in C++. These techniques are used in different parts of Kratos to improve its efficiency

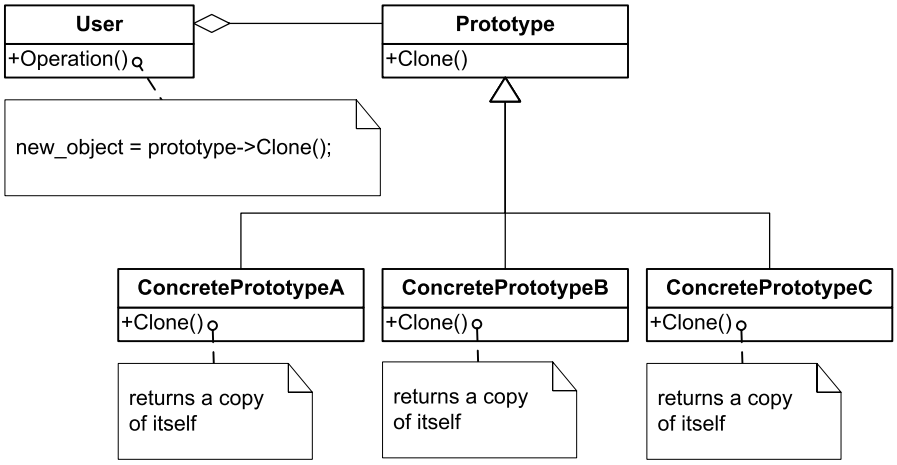



Figure 3.29: Prototype pattern.

while providing a clear and easy to use interface.

### Expression Templates

*Expression Templates* is a technique which is used to pass expressions to a function argument in a very efficient way [97]. For example passing a function to an integration procedure to be integrated. This technique is also used in high performance linear algebra libraries to evaluate vectorial expressions [98]. In this way expressions consisting of operation over matrices and vectors can be evaluated without creating any temporary object and in a single loop.

The idea is to create a template object for each operator and constructing the whole expression by combining these templates and their relative variables. For example, considering the function $f$ as follows:

$$f(x,y) = \frac{1}{x+y}$$

Converting this function to its expression templates form can be done in three steps. First we need expressions to represent the constant and variables as follows:

```
class ConstantExpression {
  double mValue;
public:
  ConstantExpression(double Constant) : mValue(Constant){}
  double Value(){return mValue;}
};

class ReferenceExpression {
  double& mReference;
public:
  ReferenceExpression(double& Variable) : mReference(Variable){}
  double Value(){return mReference;}
};
```

Then some templates are necessary to handle the operators:

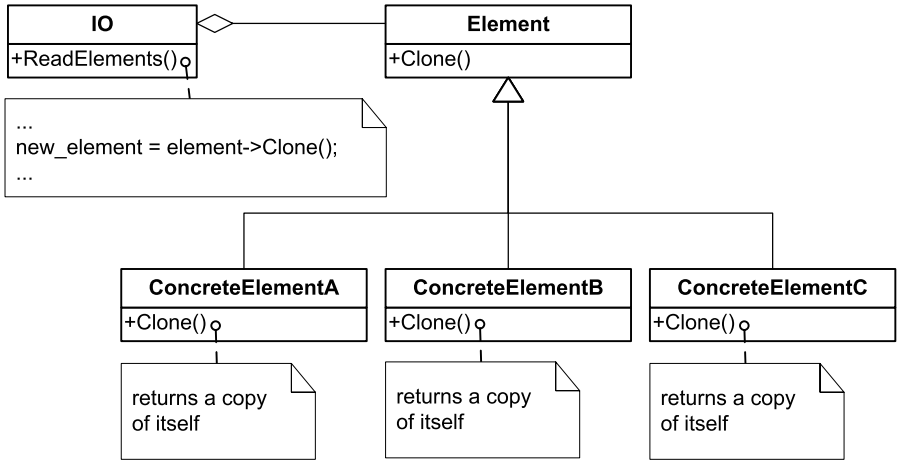



Figure 3.30: Using prototype pattern in IO for creating elements.

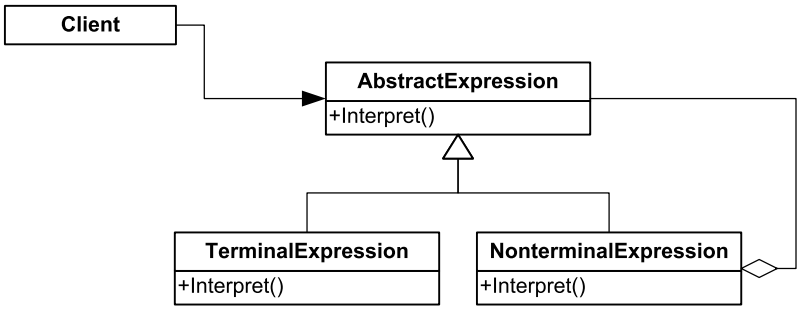



Figure 3.31: Interpreter pattern structure

```
template<class TExpression1, class TExpression2> class
SumExpression {
  TExpression1 mExpression1;
  TExpression2 mExpression2;

public:
  SumExpression(TExpression1 Expression1,
                TExpression2 Expression2) :
                mExpression1(Expression1),
                mExpression2(Expression2){}

  double Value(){return mExpression1.Value() +
                        mExpression2.Value();}

};

template<class TExpression1, class TExpression2> class
```

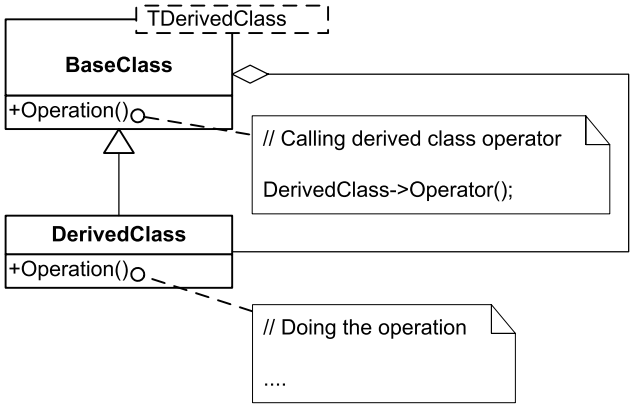



Figure 3.32: Curiously recursive template pattern.

```
DivideExpression {
  TExpression1 mExpression1;
  TExpression2 mExpression2;

public:
  DivideExpression(TExpression1 Expression1,
                   TExpression2 Expression2) :
                   mExpression1(Expression1),
                   mExpression2(Expression2){}

  double Value(){return mExpression1.Value() /
                        mExpression2.Value();}

};
```

And finally the expression template version can be written using previous components. Con-

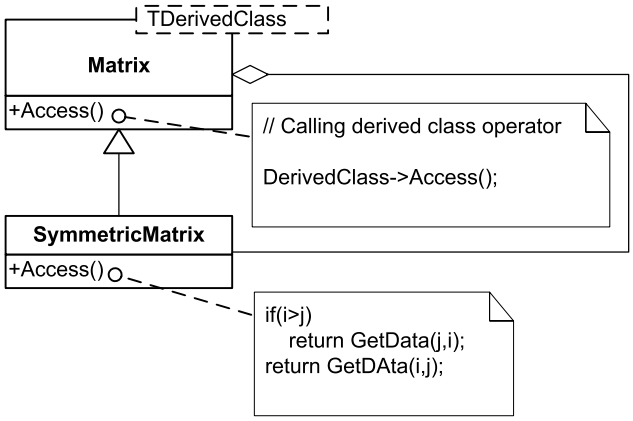



Figure 3.33: Using CRT pattern in matrix structure desing.

verting the $x + y$ expression results in:

```
SumExpression<ReferenceExpression,
              ReferenceExpression>(ReferenceExpression(x),
                                   ReferenceExpression(y));
```

Using above expression the whole function can be written as follows:

```
typedef SumExpression<ReferenceExpression,
                      ReferenceExpression> sum_expression;

typedef DivideExpression<ConstantExpression,
                         sum_expression> expression;

expression f = expression(ConstantExpression(1),
                          sum_expression(ReferenceExpression(x),
                                         ReferenceExpression(y)));
```

Writing these expressions manually is really impractical but fortunately carefully overloaded operators can do this conversion automatically. As mentioned earlier this technique can be used to evaluate matrix and vector expressions without creating temporaries and in one pass. A simple sum operation using overloaded operators over vectors or matrices can result in many redundant loops and overhead. For example considering the following innocent code:

```
// a,b,c and d are vectors
d = a + b + c;
```

This simple code to sum three vectors and assign it to another one using simple overloaded operators can generate a code equivalent to:

```
Vector t1 = b + c; Vector t2 = a + t1; d = t2;
```

In the first step the overloaded operator is used to calculate the sum of two vectors and put them in the temporary vector `t1`. Again the overloaded operator is used to calculate the sum of vector `a` and temporary vector `t1` and the result is stored in another temporary vector `t2`. Finally the second temporary vector is assigned to left hand side vector `d`. It can be seen that this operation is done by performing three loops over all vector elements and creating two temporary vectors which make it very inefficient. Using expression templates can eliminate all this overhead and make it as efficient as a hand coded procedure.

The idea is to overload operators to create the expression without evaluating it. The evaluation of the expression is postponed to the assigning time. The right hand side of this expression can be converted to the following form:

```
class ReferenceExpression {
  Vector& mReference;
public:
  ReferenceExpression(Vector& Variable) : mReference(Variable){}
  double Value(int i){return mReference[i];}
};

template<class TExpression1, class TExpression2> class
SumExpression {
  TExpression1 mExpression1;
  TExpression2 mExpression2;

public:
```

```
  SumExpression(TExpression1 Expression1,
                TExpression2 Expression2) :
                mExpression1(Expression1),
                mExpression2(Expression2){}

  double Value(int i){return mExpression1.Value(i) +
                             mExpression2.Value(i);}

};

typedef SumExpression<ReferenceExpression,
                      SumExpression<ReferenceExpression,
                                    ReferenceExpression> rhs_expression;

d = rhs_expression(ReferenceExpression(a),
                   SumExpression<ReferenceExpression,
                                 ReferenceExpression>(b,c));
```

Now an overloaded assignment operator can complete the procedure:

```
template<class TExpression>
  operator = (Vector& a, TExpression Expression)
  {
    for(int i = 0 ; i < size ; i++)
        a[i] = Expression.Value(i);
  }
```

Passing our `rhs_expression` to this assignment operator results a code equivalent to:

```
for(int i = 0 ; i < size ; i++)
  d[i] = rhs_expression.Value(i);
```

Inlining the first `Value` method and references inside it results the following code:

```
for(int i = 0 ; i < size ; i++)
  d[i] = a[i] + SumExpression<ReferenceExpression,
                              ReferenceExpression>(b,c).Value(i);
```

And inlining the second `Value` method and its all the references results the optimized code:

```
for(int i = 0 ; i < size ; i++)
  d[i] = a[i] + b[i] + c[i];
```

### Template Metaprogramming

Templates were added to C++ for a better alternative to existing macros in old C. Eventually they didn't eliminate the usage of macros but gave us much more than anyone could expect. For example, template meta programming. Everything began when Erwin Unruh tricked the compiler to print a list of prime numbers at compile time. This extends the algorithm writing from standard form in C++ to a new form which makes the compiler run the algorithm and results in a new specific algorithm to run.

The Template Metaprogramming technique can be used to create an specialized algorithm at the time of compiling. This technique makes compiler interpret a subset of C++ code in order to generate this specialized algorithm. Different methods are used to simulate different programming

statements, like loops or conditional statements, at compile time. These statements are used to tell the compiler how to generate the code for our specific case.

This technique uses recursive templates to encourage the compiler into making a loop. When a template calls itself recursively the compiler has to make a loop for instantiating templates until a specialized version used as stop rule is reached.Here is an example of how template meta programming can be used to make a compile time power function. First a general template function class is needed to hold the power function class:

```
template<std::size_t TOrder> struct Pow {
  static inline
  double Value(double X)
  {
    // Calculatin result ...
    return result;
  }
};
```

This class takes order as a template argument of the class. Note that `TOrder` must be a positive integer and also known at compile time. So it cannot be used as a normal runtime power function. Now we take a recursive algorithm to compute the $n$-th power of a value $x$:

$$x_i = x_{i-1} * x, i = 2, ..., N \tag{3.108a}$$

$$x_1 = x \tag{3.108b}$$

Implementing this using template meta programming is relatively straightforward. Recursive templates can be used here to make the compiler perform a for loop and in each pass add an $x$ multiplication to our code. Stoping the loop after repeating $k$ times will generate a code equivalent to manually written one. First we introduce the recursive part in `Value` method:

```
template<std::size_t TOrder> struct Pow {
  static inline
  double Value(double X)
  {
    return X * Pow<TOrder - 1>::Value(X);
  }
};
```

And the stop rule in equation 3.108b as a specialized template

```
template<> struct Pow<1> {
  static inline
  double Value(double X)
  {
    return X;
  }
};
```

Now the power function is ready to use. Here is an example of calling it to calculate $x^4$:

```
double x = 2.00; double y = Pow<4>::Value(x); assert(y == 16.00)
```

In the time of Compiling, the compiler will try to inlining the `Value` function which results

```
double y = x * Pow<3>::Value(x)
```

And continuing the recursive calling to order 1

```
double y = x * x * x * Pow<1>::Value(x)
```

At this point the template specialization stop it from going into a infinite loop because `Value` method of `Pow<1>` is not recursive. Compiler tries to inline this `Value` method and generates the following code:

```
double y = x * x * x * x;
```

There is an advantage using this class. If `x` is known at compile time `y` also be calculated in compile time. This can be used to optimize the code for instance, loops can be unrolled when the repeat number is a known value and so on.

Template specialization is can also be used to make compiler simulate conditional statements or switch and cases. These statements can be used to generate different codes according to some conditions. For example an assignment operator for matrices may change its algorithm depending on row or column majority of a given matrix:

```
template<bool c> class Assign{};

class Assign<true> { public:
  template<class Matrix1, class Matrix2>
  Assign(Matrix1& A, Matrix2& B)
  {
    // Assigning row by row;
  }
}


class Assign<false> { public:
  template<class Matrix1, class Matrix2>
  Assign(Matrix1& A, Matrix2& B)
  {
    // Assigning column by column;
  }
}

template<class Matrix2>
  operator = (Matrix2& B)
  {
    Assign<Matrix2::IsRowMajor>(*this, B);
  }
```

In this form the compiler generates an specialized algorithm for each assigning statement.

Chapter 4

# General Structure

In this chapter first the objectives and also the users considered in the design of Kratos are described then the methodology to design the structure is given.

## 4.1 Kratos' Requirements

Kratos is designed as a framework for building multi-disciplinary finite element programs [31]. Generality in design and implementation is the first requirement. Kratos has to provide the general tools necessary for finite element solution. It also has to remove many restrictions that exist in other codes in order to achieve enough flexibility to handle a wider variety of algorithms than before. For example, restrictions like using certain degrees of freedom (dof).

Kratos must provide a flexible structure in order to handle a wide variety of methods and algorithms. This flexibility has to be provided not only in its global layout and basic assumptions but also in its implementation details. Minimization of restrictive assumptions is necessary in order to let developers configure this library as they want at different levels.

Kratos as a library must provide a good level of reusability in its provided tools. The key point here is to help users develop easier and faster their own finite element code using generic components provided by Kratos, or even other applications.

Kratos has to be extendible, at different levels of implementation. It must be extendible to new formulations, algorithms, and concepts. Supporting a wide variety of problems that can be coupled in a multi-disciplinary problem requires very different formulations and algorithms to be implemented. These formulations and algorithms may also use new concepts and variables. So Kratos must provide an extendible design for all of its components in order to support new methods.

Another important requirements are good performance and memory efficiency. This features are necessary for enabling applications implemented using Kratos, to deal with industrial multi-disciplinary problems. These requirements are very important and are the reason for most of the restrictions in Kratos.

Finally it has to provide different levels of developers' contributions to the Kratos system, and match their requirements and difficulties in the way they extend it. Developers may want to just make a plug-in extension, create an application over it, or using IO scripts to make Kratos perform a certain algorithm. Kratos has to provide not only all these capabilities but also hide the unnecessary difficulties from each group of developers.

## 4.2 Users

One of the important factors in design is to determine who will work with the program, what are their needs and how the program can help them. In essence Kratos is defined to be used by three groups of users at different levels:

**Finite Element Developers** Kratos is defined to be used by finite element developers to implement a multi-disciplinary formulation easily. These developers, or users from Kratos point of view, are considered to be more expert in FEM, from the physical and mathematical points of view, than C++ programming. For this reason, Kratos has to provide their requirements without involving them in advanced programming concepts.

**Application Developers** Kratos can be used as a finite element engine for other applications. This ability favors another teams of developers to work with Kratos. These users are less interested in finite element programming and their programming knowledge may vary from very expert to higher than basic. They may use not only Kratos itself but also any other applications provided by finite element developers, or other application developers. Developers of optimization programs or design tools are the typical users of this kind.

**Package Users** Engineers and designers are other users of Kratos. They use the complete package of Kratos and its applications to model and solve their problem without getting involved in internal programming of this package. For these users Kratos has to provide a flexible external interface to enable them use different features of Kratos without changing its implementation.

Kratos has to provide a framework such that a team of developers with completely different fields of expertise as mentioned before, work on it in order to create multi-disciplinary finite element applications.

## 4.3 Object Oriented Design

History of object-oriented design for finite element programs turns back to early 90's, and even more. Before that, many large finite element programs were developed in modular ways. Industry demands for solving more complex problems from one side, and the problem of maintaining and extending the previous programs from the other side, has lead developers to target their design strategy towards an object-oriented one [44, 43, 64, 80, 82].

The main goal of an object-oriented structure is to split the whole problem into several objects and to define their interfaces. There are many possible ways to do this for each kind of problem we want to program and the functionality of the resultant structure depends largely on it. In the case of finite element problems there are also many approaches such as constructing objects based on partial differential equations solving methods [22] or in the finite element method itself [44, 106, 35, 34].

In Kratos we have chosen the second approach and have constructed our objects based on a finite element general methodology. This approach was selected because our goal was to create a finite element environment for multidisciplinary problems. Also our colleagues were, in general, more familiar with this methodology than with physical properties. In addition, this approach has given us the necessary generality mentioned above in the objectives of Kratos. Within this scope main objects are taken from various parts of the FEM structure. Then, some abstract objects are defined for implementation purposes. Finally their relation are defined and their responsibilities are balanced. Figure 4.1 shows the main classes in Kratos.

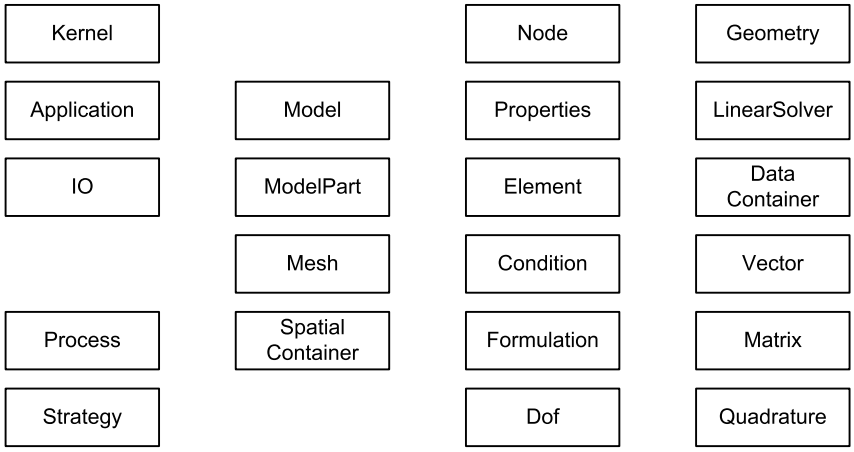



Figure 4.1: Main classes defined in Kratos.

`Vector`, `Matrix`, and `Quadrature` are designed by basic numerical concepts. `Node`, `Element`, `Condition`, and `Dof` are defined directly from finite element concepts. `Model`, `Mesh`, and `Properties` are coming from practical methodology used in finite element modeling completed by `ModelPart`, and `SpatialContainer`, for organizing better all data necessary for analysis. `IO`, `LinearSolver`, `Process`, and `Strategy` are representing the different steps of finite element program flow. and finally `Kernel` and `Application` are defined for library management and defining its interface.

These main objects are described below:

`Vector` Represents the algebraic vector and defines usual operators over vectors.

`Matrix` Encapsulate matrix and its operators. There are different matrix classes are necessary. The most typical ones are dense matrix and compressed row matrix.

`Quadrature` Implements the quadrature methods used in finite element method. For example the gaussian integration with different number of integration points.

`Geometry` Defines a geometry over a list of points or `Node`s and provides from its usual parameter like area or center point to shape functions and coordinate transformation routines.

`Node` `Node` is a point with additional facilities. Stores the nodal data, historical nodal data, and list of degrees of freedom. It provides also an interface to access all its data.

`Element` Encapsulates the elemental formulation in one objects and provides an interface for calculating the local matrices and vectors necessary for assembling the global system of equations. It holds its geometry that meanwhile is its array of `Node`s. Also stores the elemental data and interface to access it.

`Condition` Encapsulates data and operations necessary for calculating the local contributions of `Condition` in global system of equations. Neumann conditions are example of `Condition`s which can be encapsulated by derivatives of this class.

`Dof` Represents a degree of freedom (dof). It is a lightweight object which holds the its variable, like `TEMPERATURE`, its state of freedom, and a reference to its value in data structure. This

class enables the system to work with different set of dofs and also represents the Dirichlet condition assigned to each dof.

`Properties` Encapsulates data shared by different `Element`s or `Condition`s. It can stores any type of data and provide a variable base access to them.

`Model` Stores the whole model to be analyzed. All `Node`s, `Properties`, `Element`s, `Condition`s and solution data. It also provides and access interface to these data.

`ModelPart` Holds all data related to an arbitrary part of model. It stores all existing components and data like `Node`s, `Properties`, `Element`s, `Condition`s and solution data related to a part of model and provides interface to access them in different ways.

`Mesh` Holds `Node`s, `Properties`, `Element`s, `Condition`s and represents a part of model but without additional solution parameters. It provides access interface to its data.

`SpatialContainer` Containers associated with spacial search algorithms. This algorithms are useful for finding the nearest `Node` or `Element` to some point or other spacial searches. Quadtree and Octree are example of these containers.

`IO` Provides different implementation of input output procedures which can be used to read and write with different formats and characteristics.

`LinearSolver` Encapsulates the algorithms used for solving a linear system of equations. Different direct solvers and iterative solvers can be implemented in Kratos as a derivatives of this class.

`Strategy` Encapsulates the solving algorithm and general flow of a solving process. Strategy manages the building of equation system and then solve it using a linear solver and finally is in charge of updating the results in the data structure.

`Process` Is the extension point for adding new algorithms to Kratos. Mapping algorithms, Optimization procedures and many other type of algorithms can be implemented as a new process in Kratos.

`Kernel` Manages the whole Kratos by initializing different part of it and provides necessary interface to communicate with applications.

`Application` Provide all information necessary for adding an application to Kratos. A derived class from it is necessary to give kernel its required information like new `Variable`s, `Element`s, `Condition`s, etc.

The main intention here was to hide all difficult but common finite element implementations like data structure and IO programming from developers.

## 4.4 Multi-Layers Design

Kratos uses a *multi-layer* approach in its design. In this approach each object only interfaces with other objects in its layer or in layers below its layer. There are some other layering approaches that limited the interface between objects of two layers but in Kratos this limitation is not applied.

Layering reduces the dependency inside the program. It helps in the maintenance of the code and also helps developers in understanding the code and clarifies their tasks.

In designing the layers of the structure different users mentioned before are considered. The layering are done in a way that each user has to work in the less number of layers as possible. In this way the amount of the code to be known by each user is minimized and the chance of conflict between users in different categories is reduced. This layering also lets Kratos to tune the implementation difficulties needed for each layer to the knowledge of users working in it. For example the finite element layer uses only basic to average features of C++ programming but the main developer layer use advanced language features in order to provide the desirable performance.

Following the current design mentioned before, Kratos is organized in the following layers:

**Basic Tools Layer** Holds all basic tools used in Kratos. In this layer using advance techniques in C++ is essential in order to maximize the performance of these tools. This layer is designed to be implemented by an expert programmer and with less knowledge of FEM. This layer may also provides interfaces with other libraries to take benefit of existing work in area.

**Base Finite Element Layer** This layer holds the objects that are necessary to implement a finite element formulation. It also defines the structure to be extended for new formulations. This layer hides the difficult implementations of nodal and data structure and other common features from the finite element developers.

**Finite Element Layer** The extension layer for finite element developers. The finite element layer is restricted to use the basic and average features of language and uses the component base finite element layer and basic tools to optimize the performance without entering into optimization details.

**Data Structure Layer** Contains all objects organizing the data structure. This layer has no restriction in implementation. Advanced language features are used to maximize the flexibility of the data structure.

**Base Algorithms Layer** Provides the components building the extendible structure for algorithms. Generic algorithms can also be implemented here to help developer in their implementation by reusing them.

**User's Algorithms Layer** Another layer to be used by finite element programmers but at a higher level. This layer contains all classes implementing the different algorithms in Kratos. Implementation in this layer requires medium level of programming experience but a higher knowledge of program structure than the finite element layer.

**Applications' Interface Layer** This layer holds all objects that manage Kratos and its relation with other applications. Components in this layer are implemented using high level programming techniques in order to provide the required flexibility.

**Applications Layer** A simple layer which contains the interface of certain applications with Kratos.

**Scripts Layer** Holds a set of IO scripts which can be used to implement different algorithms from outside Kratos. Package users can use modules in this layer or create their own extension without having knowledge of C++ programming or the internal structure of Kratos. Via this layer they can activate and deactivate certain functionalities or implement a new global algorithm without entering into Kratos implementation details.

Figure 4.2 shows the multi-layer nature of Kratos.

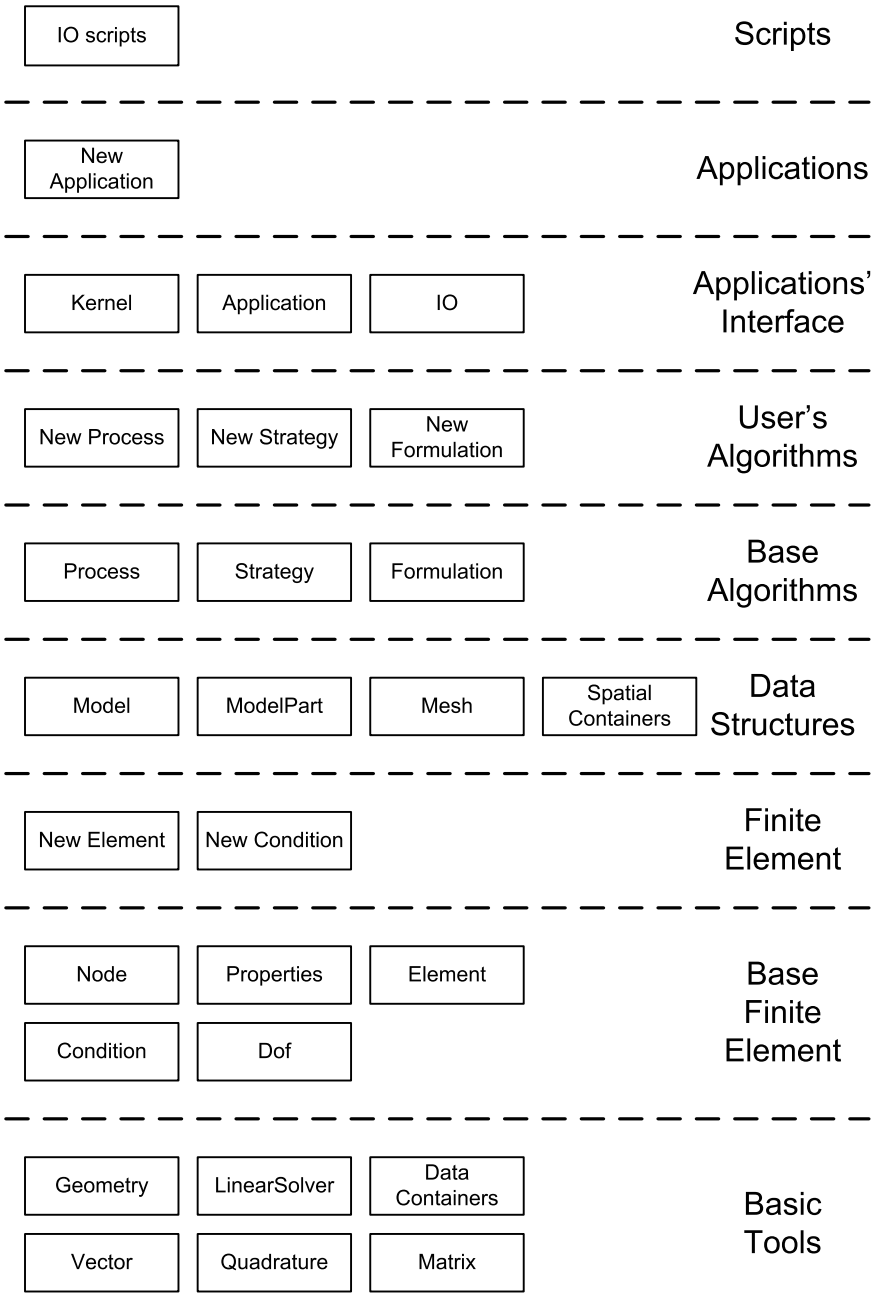



Figure 4.2: Dividing the structure into layers reduces the dependency.

## 4.5 Kernel and Applications

In the first implementation of Kratos all applications were implemented in Kratos and also were compiled together. This approach at that time produced several conflicts between applications and was requiring many unnecessary recompiling of the code for changes in other applications. All these problems lead to a change in the strategy and to separating each application not only from others but also from Kratos itself.

In the current structure of Kratos each application is created and compiled separately and just uses a standard interface to communicate with the kernel of Kratos. In this way the conflicts are reduced and the compilation time is also minimized. The `Application` class provides the interface for introducing an application to the kernel of Kratos. `Kernel` uses the information given by `Application` through this interface to mange its components, configure different part of Kratos, and synchronize the application with other ones. The `Application` class is very simple and consists of registering the new components like: `Variable`s, `Element`s, `Condition`s, etc. defined in application. The following code shows a typical application class definition:

```
// Variables definition
KRATOS_DEFINE_VARIABLE(int, NEW_INTEGER_VARIABLE )
KRATOS_DEFINE_3D_VARIABLE_WITH_COMPONENTS(NEW_3D_VARIABLE);
KRATOS_DEFINE_VARIABLE(Matrix, NEW_MATRIX_VARIABLE)

class KratosNewApplication : public KratosApplication
{
public:
  virtual void Register();

private:

  static const NewElementType    msNewElement;
  static const NewConditionType  msNewCondition;
};
```

Here `Application` defines its new components and now its time to implement the `Register` method:

```
// Creating variables
KRATOS_CREATE_VARIABLE(NEW_INTEGER_VARIABLE )
KRATOS_CREATE_3D_VARIABLE_WITH_COMPONENT(NEW_3D_VARIABLE);
KRATOS_CREATE_VARIABLE(NEW_MATRIX_VARIABLE)

void KratosR1StructuralApplication::Register()
{
  // calling base class register to register Kratos components
  KratosApplication::Register();

  // registering variables in Kratos.
  KRATOS_REGISTER_VARIABLE(NEW_INTEGER_VARIABLE)
  KRATOS_REGISTER_3D_VARIABLE_WITH_COMPONENTS(NEW_3D_VARIABLE);
  KRATOS_REGISTER_VARIABLE(NEW_MATRIX_VARIABLE)

  KRATOS_REGISTER_ELEMENT("MyElement", msNewElement);
  KRATOS_REGISTER_CONDITION("MyCondition", msNewCondition);
```

```
}
```

This interface enables Kratos to add all these `Variable`s, `Element`s, and `Condition`s in the list of components. Kratos also synchronizes the variables numbering between different applications. Adding new components to Kratos, enables IO to read and write them and also configures the data structure to hold these new variables.

In the next chapter the basic tools layer will be declared.

Chapter 5

# Basic tools

A finite element program has several common procedures that can be implemented as basic tools to be used by other part of program. Integrating, calculating shape functions and other geometrical parameters, and linear solvers are some examples of these procedures. Basic tools are defined to implement these common tasks. Implementing basic tools reduces the implementation task by removing the duplicated procedure in program and also increases the reusability and compatibility between different parts of code via their uniform interface.

While many of this tools are used in the most inner part of the code their performance has a great importance. For example integration tools are called inside the `Element`s in time of build and any overhead in their performance will cause a great overhead in program executing time. From other point of view, a large proportion of executing time, is consumed by these tools. A good example is the linear solver which take a large amount of execution time just by itself. All these comments shows the importance of performance tuning in these tools.

Another important requirement for these tools is efficiency in memory. Again in a usual finite element program, a large proportion of used memory is consumed by these tools. This come from two facts, first, a large amount of these tools may be necessary to handle a finite element problem and second, they use a large amount of memory for their operations. For example many geometries must be created in order to model a real problem. So their efficiency in using memory is crucial in order to control the total memory used by program.

The last but not least feature is reusability of these tools. While these tools must be used by different part of the code, their generality and flexibility plays an important role in their successful usage.

In this section the design and implementation of different tools in Kratos will be explained.

## 5.1 Integration tools

Integrating some function over the domain or boundary is one of the fundamental operations in the FEM. `Element`s, `Condition`s and sometimes other parts of the code have to perform integration in an efficient way. All this makes necessary the designing and implementing an efficient integration tool which can handle different methods of integration with less overhead as possible. Before starting with designing integration tools let take a brief look to numerical integration methods used in finite element programs.

### 5.1.1 Numerical Integration Methods

It's clear that integrating a function analytically in many cases contains difficulties and in general is not always possible. This made numerical integration, also called quadrature, to be started in 18th and 19th centuries. However use of numerical integration for real problems was postponed to the time of development of computers.

A typical approach to approximate an integral of a function is to evaluate it in different points, apply specific weight to them and sum the weighted values to obtain the result.

$$\int_a^b f(x)\,d(x) \approx \sum_{i=1}^{n} w_i f(x_i) \tag{5.1}$$

Many classical method assume that sample points are in the same distance $h$ all the time.

$$x_i = x_0 + ih \tag{5.2}$$

They use different number of sample points and apply different weighting coefficient to obtain the result. Here there are some examples of methods in this category:

#### Classical Formulas

A very simple case is trapezoidal rule. It take to sample point and evaluate their corresponding points on the function then connect them with a line and simply calculate the area of trapezoidal obtained below this line. In the other word, it use the average of these two values to calculate the integral:

$$\int_a^b f(x)\,d(x) = \frac{h}{2}f(a) + \frac{h}{2}f(b) + O\left(h^3 f"\right) \tag{5.3}$$

It can be easily seen that this integration is exact for linear function and for other functions while second derivative is unknown to us just approximate the result

$$\int_a^b f(x)\,d(x) \approx \frac{h}{2}f(a) + \frac{h}{2}f(b) \tag{5.4}$$

According to equation 5.1 we can define sample points and weighted for this method:

$$x_1 = a, x_2 = b \tag{5.5a}$$

$$w_1 = w_2 = \frac{h}{2} \tag{5.5b}$$

Famous Simpson's rules are also in this category. They use more sample point to produce higher order approximation. Here is the first one:

$$\begin{aligned}\int_a^b f(x)\,d(x) = \frac{h}{3}f(a) + \frac{4h}{3}f(\frac{a+b}{2}) \\ + \frac{h}{3}f(b) + O\left(h^5 f^{(4)}\right)\end{aligned} \tag{5.6}$$

which is exact for a polynomial up to degree 2. While it approximate other functions better than trapezoidal rule:

$$\int_a^b f(x)\,d(x) \approx \frac{h}{3}f(a) + \frac{4h}{3}f(\frac{a+b}{2}) + \frac{h}{3}f(b) \tag{5.7}$$

Also we can introduce this in the global form using:

$$x_1 = a, x_2 = \frac{a+b}{2}, x_3 = b \tag{5.8a}$$

$$w_1 = w_3 = \frac{h}{3}, w_2 = \frac{4h}{3} \tag{5.8b}$$

There are also extension to this methods using the same contest which approximate with higher order using more sample points. In any case to give exact integral of a polynomial of order $n$ this methods use $2n+1$ sample points. This make them very expensive while there are other ways to achieve the same result with just $n$ sample points, which will be described as follow.

### Gaussian Quadrature

As mentioned before in previous methods, sample points assumed to be located in the same distance. If we take this restriction and chose their position in some other manner we can duplicate the order of approximation. This is the base idea of Gaussian quadrature formulas.

But how this magical method works? Here is a quick review over it. First, lets define a weighted scalar product of two functions $f$ over $g$ as

$$\langle f \mid g \rangle = \int_a^b W(x) f(x)\, g(x) dx \tag{5.9}$$

A set of polynomials can be funded which include exactly one polynomial $p_j(x)$ of order $j$, for each $j = 1, 2, 3, ....$ which are also mutually orthogonal over a given weight function $W(x)$.

Let extend the equation 5.1 to a more general case which is:

$$\int_a^b W(x) f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i) \tag{5.10}$$

Here $W(x)$ is added as a known weighting function applied to our integrand. This extension is to use a powerful feature of Gaussian quadrature where we can make an exact integral over a polynomial times some known weighting function $W(x)$. This weighting function help us to discomposing some integrable function to a polynomial and a complex but integrable part to make an exact integral. Also it can be chosen to remove singularities which are integrable. By the way we can assume equation 5.1 a special case of 5.10 with $W(x) \equiv 1$. Choosing this weighting function results so called *Gauss-legendre integration*.

Now it's time to apply the fundamental theorem of Gaussian quadratures, the abscissas of the N-point Gaussian quadrature formulas with weighting function $W(x)$ in the interval $(a, b)$ are precisely the roots of the orthogonal polynomial $p_n(x)$ for the same interval and weighting function.

For giving a practical way to find Gaussian weight and abscissas, we use a set of orthogonal polynomials defined as

$$p_0(x) \equiv 1 \tag{5.11a}$$

$$p_1(x) = [x - \frac{\langle xp_0|p_0\rangle}{\langle p_0|p_0\rangle}]p_0(x) \tag{5.11b}$$

$$p_{i+1}(x) = [x - \frac{\langle xp_i|p_i\rangle}{\langle p_i|p_i\rangle}]p_i(x) - \frac{\langle p_i|p_i\rangle}{\langle p_{i-1}|p_{i-1}\rangle}p_{i-1}(x) \tag{5.11c}$$

It can be shown that each polynomials $p_j(x)$ has exactly $j$ distinct roots and also there is exactly one root of $p_{j-1}(x)$ between each adjacent pair of them. This property comes very useful in the time of finding roots. A root finding scheme can be applied starting from $p_1(x)$ and continuing to higher orders using interval of previous roots to find all of them. Finally equation 5.10 used to make a system of equations for finding weights $w_i$. Considering that equation 5.10 must gives the correct answer for the integral of the first $N-1$ polynomials

$$a_{ij}w_j = b_i \tag{5.12a}$$

$$a_{ij} = p_{i-1}(x_j) \tag{5.12b}$$

$$b_i = \int_a^b W(x)p_{i-1}(x)\,dx \tag{5.12c}$$

$$i = 1, ..., N, j = 1, ..., N \tag{5.12d}$$

It can be shown that using the same weights $w_i$ the quadrature is exact for all polynomials of order up to $2N-1$. Also in equation 5.12a it's easy to verify that $b_i$ for $i = 2, ..., N$ is equal to zero considering the orthogonal nature of $p_i$'s.

There is also another, and more practical, way to find the weights $w_i$, using another sequence of polynomials $\varphi_i(x)$

$$\varphi_0(x) \equiv 0 \tag{5.13a}$$

$$\varphi_1(x) = p_1' \int_a^b W(x)dx \tag{5.13b}$$

$$\varphi_{i+1}(x) = [x - \frac{\langle xp_i|p_i\rangle}{\langle p_i|p_i\rangle}]\varphi_i(x) - \frac{\langle p_i|p_i\rangle}{\langle p_{i-1}|p_{i-1}\rangle}\varphi_{i-1}(x) \tag{5.13c}$$

Where $p_1'$ is the derivative of $p_1(x)$ with respect to $x$. Using these polynomials calculating the weights is straightforward

$$w_i = \frac{\varphi_N(x_i)}{p_N'(x_i)}, i = 1, ..., N \tag{5.14}$$

Also there is a formula just for Gauss-Legendre case

$$w_i = \frac{2}{(1-x_i^2)[p'_N(x_i)]^2} \tag{5.15}$$

**Multidimensional Integrals**

Passing from one dimensional integrals to multidimensional one consist of multiplying number of sample points and also complexity of applying boundaries. In this part just an easy case with simple boundary condition will be discussed. In FEM usually we map our geometry functions to some local coordinates and then reducing the integral to lower dimensionality using simplicity of boundaries in this local coordinate and also considering smoothness of our function. Considering a three dimensional case

$$F \equiv \int_{x_1}^{x_2} \int_{y_1(x)}^{y_2(x)} \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dxdydz \tag{5.16}$$

It can be reduced in 2 dimensional form

$$F = \int_{x_1}^{x_2} \int_{y_1(x)}^{y_2(x)} G(x,y)dxdy \tag{5.17a}$$

$$G \equiv \int_{z_1(x,y)}^{z_2(x,y)} f(x,y,z)dz \tag{5.17b}$$

And finally to one dimensional form

$$F = \int_{x_1}^{x_2} H(x)dx \tag{5.18a}$$

$$H(x) \equiv \int_{y_1(x)}^{y_2(x)} G(x,y)dy \tag{5.18b}$$

It can be seen that for implementing this we need to calculate subdimensional integrals in each sample point.

Finally, using Gaussian quadrature in this approach, results

$$F = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} w_i w_j w_k f(x_i, y_j, z_k) \tag{5.19}$$

### 5.1.2 Existing Approaches

There are several ways to implement integration methods in a program. They can be implemented using a very rigid structure or very flexible and dynamic one. In this part different approaches for implementing integration methods are discussed.

The first and most static and also rigid one is just to introduce manually the integration coordinates and weights in the integrand functions. This approach is used in finite element programs

to create an array of shape functions values in all integration points or directly into the weak form to integrate it over element. The advantage is integration will be extremely fast especially for low order simple elements, while all the values are known at compile time and all the loops can be eliminated. The drawback is that maintaining these structure for higher order elements is difficult and the code is not reusable for other formulations or elements. Here is an example:

```
double f(double x, double y) {
  // function body ...
}

double integral_of_f() {
  const double x1 = -sqrt(1.00 / 3.00)
  const double x2 =  sqrt(1.00 / 3.00)
  // w = 1.00

  return f(x1,x1) + f(x1,x2) +
         f(x2,x1) + f(x2,x2);
}
```

The previous approach can be enhanced by encapsulating the 1 dimensional sample points sets and their weights in a class and creating any $n$ dimensional instance of it just in the dimension loops over integrand. In this manner the integration points can be reused in other parts of program but program still is depended to a set of outer product of 1 dimensional sample points. In other words having a specific $n$ dimensional points is not supported in this manner.

```
array<double, 2>
  Gauss2 = {-sqrt(1.00 / 3.00),
             sqrt(1.00 / 3.00)};


double f(Point& x) {
  // function body ...
}

double integral_of_f() {
  double result = 0;
  const int size = Gauss2.size();
  // w = 1.00

  for(int i = 0 ; i < size ; i++)
    for(int j = 0 ; j < size ; j++)
    {
      Point g(Gauss2[i], Gauss2[j]);
      result += f(g);
    }

  return result;
}
```

Another alternative is to encapsulate an array of quadrature points and their weights in a quadrature class and implement an instance of this class for each set of integration points. This design is also common and relatively more reusable than previous one. The performance is highly depended on how these quadratures object are implemented and used but can be optimized as much

as previous approach. Its weakness is hand coding and also debugging it especially when extending it from 1 dimension to 2 and 3. In this cases for any set of gaussian integration sample points different quadrature sets for 1 and 2 and 3 dimensional integration space must be implemented and tested separately.

```
class Gauss2D2 : public array<Point<2>, 4> { public:
  Gauss2D2()
  {
    // Initializing array with
    // integration points
  }
};

double f(Point<2>& x) {
  // function body ...
}

double integral_of_f() {
  double result = 0;
  const int size = Gauss2D2::size();
  Gauss2D2 gauss_points;
  // w = 1.00

  for(int i = 0 ; i < size ; i++)
    result += f(gauss_points[i])
  return result;
}
```

A good extension to previous design is to automating the construction of 2 and 3 dimensional integration points sets from their 1 dimensional set. This extension make us a new alternative with more compact implementation which is easier to extend and also test. In Deal II [21] this approach is used to implement quadrature classes. The quadrature base class is in charge of expanding 1 dimensional set to its dimension (given by template parameter) and store them while providing the interface for them. Extensibility is the good point of this structure. Any new set of integration points can be added just by deriving it from general base class and any $n$ dimensional set created just by providing abscissas and weights and without changing the library.

### 5.1.3 Kratos Quadrature Library Design

Kratos quadrature's structure is very simple and straight forward. It consist of two classes `IntegrationPoint` and `Quadrature` with a set of classes representing array of sample points which referred as `IntegrationPoints`.

The library is divided in two parts, one is the library by itself which is not for modifying in term of extension and other is the extension which is the port for adding new methods in order to extend the library. This separation allows to encapsulate all difficulties in the library part and leave the extending part as simple as possible. This was one of the reasons not to use a CRT pattern in its design.

Each abscise and its corresponding weighting encapsulates in `IntegrationPoint` class. In more detail, `IntegrationPoint` derived from `Point` class which makes it possible to be passed as a point to any geometrical function. Also `Point` by itself is an interface and derived from `array_1d`

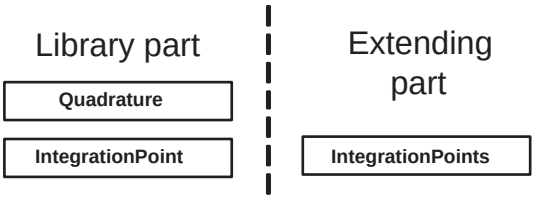



Figure 5.1: Quadrature overall scheme

which physically contains the coordinates value. All these make an `IntegrationPoint` usable in places where an array of coordinates or points needed. In Kratos also having a corresponding weight completes the encapsulation level in terms of storing, passing and using it to integrate a function. `IntegrationPoint` is a template with 3 parameters:

- `TDimension` is an unsigned integer which represents the dimension of this integration point as a point without weight as an extra dimension.
- `TDataType` is the coordinate type of integration point which is a `double` by default.
- `TWeightType` is the type of integration weight stored at an integration point and also is a double by default.

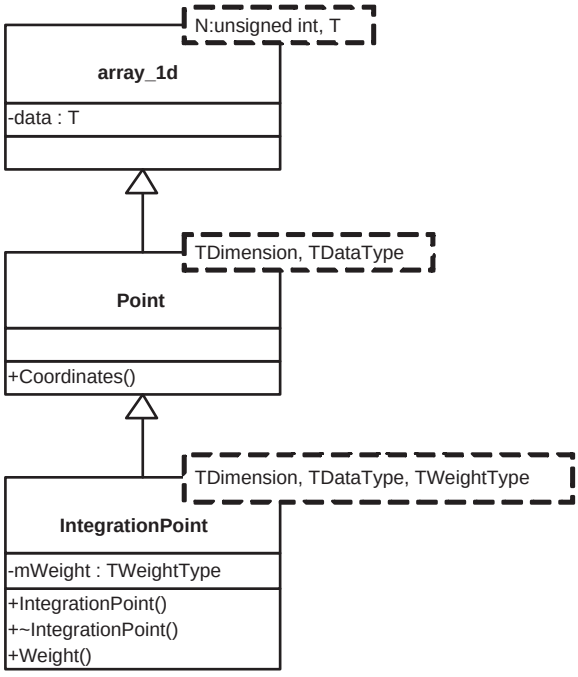



Figure 5.2: `IntegrationPoint` class

`IntegrationPoints` is just an specific set of integration points corresponding to some certain

integration algorithms. It is also the extending point of the library. Any class containing a set of user specified integration points can be used as integration points if confirming these conditions:

- Having a `Dimension` attribute known at compilation time.
- An static `size()` method must be provided and size must be known at compilation time.
- 0 base indexing is also required.

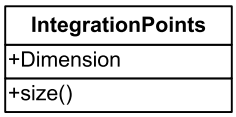



Figure 5.3: IntegrationPoints class

`Quadrature` is the engine of the library. Implements the interface for the user and also creates the integration points for higher dimensions using a Lagrangian multiplier. It can be customized by its three template parameters which are:

- `TQuadraturePointsType` is the given integration points to create a quadrature
- `TDimension` is an unsigned integer which represents the dimension of the quadrature which by default is the dimension of the given integration points.
- `TIntegrationPointType` is the integration point type used to create a quadrature. Its default value is `IntegrationPoint` with a given dimension.

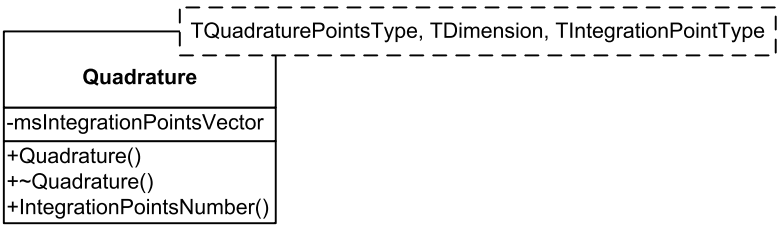



Figure 5.4: The `Quadrature` class

## 5.2 Linear Solvers

Implicit methods for solving finite element problems construct a system of equation representing the model and solve it with a linear solver. In this section the design of linear solver is discussed.

### 5.2.1 Existing Libraries

The independence nature of the solvers and also the essential and massive use of them in many analyzing methods motivates several groups to create solvers library. In Fortran there are successful libraries which are really matured and are widely used.

The *Linear Algebra PACKage (LAPACK)* is a classic library for solving linear system of equations. This library provides routines for solving a simultaneous systems of equations, eigenvalue and singular value problems, and least-squares solutions of linear system of equations. This library is written in Fortran 77 and is used widely in fortran finite element codes. *LAPACK95* is an extension of this library which provides better interface using Fortran 95 features. LAPACK is designed for running efficiently in shared-memory vector and parallel processors. To be more portable it uses the *Basic Linear Algebra Subprograms (BLAS)* for its internal operations. So by configuring the BLAS for any architecture a corresponding optimized LAPACK is obtained. The lack of iterative solvers is the only restriction of using this library for solving some big problems.

The *Linear Algebra PACKage in C++, (LAPACK++)* is an C++ interface for LAPACK. The original LAPACK++ was abandoned to start its successor *Template Numerical Toolkit (TNT)* which didn't arrive to its functionality. But the original LAPACK++ was improved by another group of developers and now has several additional features. LAPACK++ wasn't used in Kratos because in time of selecting a solver LAPACK++ was in its abandoned period.

There are also several solvers implemented in C or C++ which can be used directly in a C++ program.

The *Iterative Template Library (ITL)* provides a collection of iterative solvers and a set of preconditioners to be used with them. It also provides an abstract interface to use different basic linear algebra codes like *Matrix Template Library (MTL)* or *Blitz++*. This library can also use different type of matrices and vectors thanks to this interface. The first version of Kratos was implemented using ITL. Its clear structure and interface and its flexibility to handle different type of matrices was the reason to be selected. Unfortunately its performance was depending very much on the compiler's optimizer and basically its poor performance on Visual C++ compiler made us to put it aside.

Another important library is the *Portable, Extensible Toolkit for Scientific Computation (PETSc)*. It is constructed over BLAS, LAPACK, and *Message Passing Interface (MPI)* libraries and provides a large set of parallel linear and nonlinear equation solvers. The advantages of this library are its good performance and its support for parallel processing over distributed machines. This library is written in C and there are lack of C++ features in its structure for clean encapsulating of its tasks. This library wasn't used in implementing Kratos because parallelization at that time was not the objective of Kratos and supporting it for single process computing was too complex, (compiling and debugging this library specially under Microsoft Windows was a complex task). However there is an intention to support this library for the parallelization of Kratos in the future.

### 5.2.2 Kratos' Linear Solvers Library

In Kratos, a linear solvers library is implemented in order to have an small but efficient set of solvers to be used without the need to link some external solvers. This library is designed to be independent from the rest of the program, efficient in its calculation and flexible in the type of matrices and vectors it uses.

Three set of classes make the linear solvers library. Solvers encapsulate the solving algorithm. Preconditioners are used by solvers and modifying the equation system for better convergence and finally perform the inverse process to get the results. Reorderers change the equation order for less bandwidth or better cache use and also do inverse permutation for results.

### Spaces For Encapsulating Operations

One of the main problems to use a linear solver or incorporate a new one is the matrix and vector types they use which can be incompatible with those used by the application. In order to solve this problem another layer of abstraction is necessary in order to encapsulate matrices and vectors operations and to be used as an interface to new matrix and vector types.

The idea used here is the same as used in LAPACK and ITL but with some modifications. As mentioned before LAPACK implements the solver algorithm and uses BLAS for its algebraic operations. This structure makes it configurable for different machines. ITL also uses a somehow similar design. It has a set of functions defined in its namespace that can be overloaded by users to implement the operation over their matrices and vectors. While ITL uses templates it can also accept different type of matrices and operate them via customized operations provided by user.

The problem with the LAPACK approach is the type of matrices and vectors that cannot be changed. For ITL this problem is solved using templates but the interface doesn't provide the complete separation of different types of matrices and vectors. For example two developers can implement two interfaces by overloading the interface methods for two different types of matrices but using the same vector. In this case operations over vectors are the same in both interfaces and causes a conflict.

In Kratos this idea is improved by defining *Space* as a new concept. `Space` defines a matrix and a vector and also their operators. Encapsulating definitions and operators in one object lets different developers to implement their interfaces without any conflict. `Space` defines a simple interface with all operations which are required by linear solvers. Table 5.1 shows this interface. In the table $A$ and $B$ are matrices, $X$,$Y$, and $Z$ are vectors, and $\alpha$ and $\beta$ are scalar constants.

| **Method Name** | **Operation** |
|---|---|
| `Size` | Returns Size of the vector. |
| `Size1` | Return number of rows of matrix. |
| `Size2` | Return number of columns of matrix. |
| `Resize` | Resizes the vector |
| `Resize` | Resizes the matrix |
| `Copy` | $X \rightarrow Y$ |
| `Copy` | $A \rightarrow B$ |
| `Dot` | $X \cdot Y$ |
| `TwoNorm` | $\|X\|_2$ |
| `Mult` | $A \cdot X \rightarrow Y$ |
| `TransposeMult` | $A^T \cdot X \rightarrow Y$ |
| `RowDot` | $A_i \cdot X$ |
| `ScaleAndAdd` | $\alpha X + \beta Y \rightarrow Z$ |
| `ScaleAndAdd` | $\alpha X + \beta Y \rightarrow Y$ |
| `GraphDegree` | Number of nonzeros in given row. |
| `GraphNeighbors` | Columns of nonzeros in given row. |

Table 5.1: Interface of `Space`

It is important to mention that not all methods defined in `Space` has to be implemented for all user defined spaces. One can implement the sufficient set of them which are used by the solvers according to their needs.

### Linear Solvers

The first requirement for linear solvers is to be interchangeable with each other. While there are no best solver for all cases and in general choosing the proper solver depends also to the problem it has to solve, it is highly desirable for users to change from one solver to other in order to find the best one for their case. This feature can be provided using the Strategy pattern. Applying this pattern to linear solvers results in the structure shown in Figure 5.5.

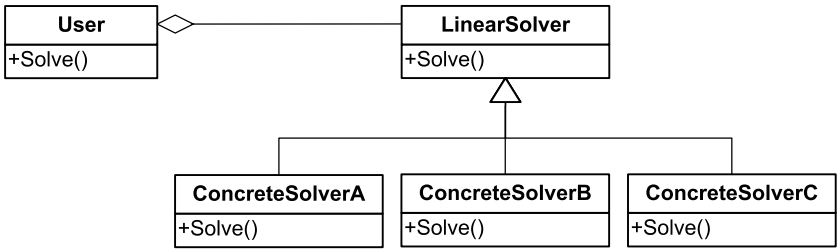



Figure 5.5: Linear solver's structure using Strategy pattern.

Using this pattern each `LinearSolver` encapsulates one solving algorithm separately and also make them interchangeable. In this way, adding a new solving algorithm is equivalent to adding a new `LinearSolver`. This encapsulation makes the library easier to extend. User keeps a pointer to the `LinearSolver` base class which may point to any member of the solver's family and use the interface of the base class to call different procedures.

This structure can be improved by dividing it into two branches, direct solvers and iterative solvers. Figure 5.6 shows this structure. Separating these two categories from each other allows special interface to be added to them.

The `IterativeSolver` class stores the tolerance, number of iterations and also the maximum number of iteration. It provides also methods for controlling the convergence and needed iterations. The `DirectSolver` is just a layer of abstraction without adding any new feature to its base class.

Linear solvers may use reorderers to reduce the bandwidth of the matrix which is important in the solution cost them using direct solvers. Reordering also can be used by iterative solvers for improving the use of cache memory. Our design must provide an easy way to add a new reordering algorithm and also let them to be combined with any linear solver.

Strategy pattern here is used to encapsulate each reordering algorithm in one class with an standard interface. In this way the extendibility is guaranteed and interchangeability is provided. Figure 5.7 shows this structure.

Bridge pattern is used to connect linear solvers and reorderers. Using this pattern users can combine each reorderers with any linear solver without problem. Figure 5.8 shows the resulted combined structure.

The next step is designing the preconditioners. They are used by iterative solvers for enhancong the convergence of the solution. Their structure must allow developers to add new preconditioning algorithms easily. It is also necessary to let users changing one preconditioner to other without problem. An strategy pattern is used in designing this structure. In this way each preconditioning algorithm is encapsulated in one object and can be added later independently to the rest of the program. Also their uniform interface established by `Preconditioner` base class allows the user to change one algorithm by another without any problem. Figure 5.9 shows this structure.

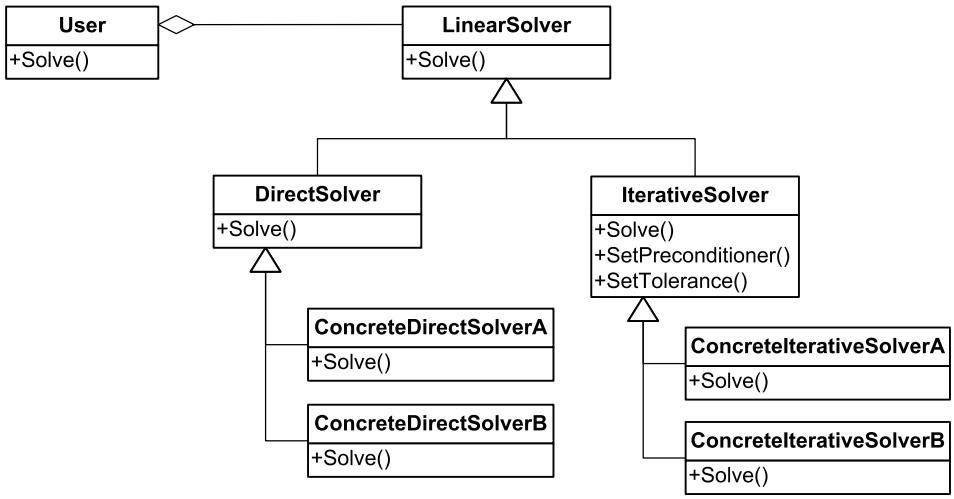



Figure 5.6: Separating direct solvers from iterative solvers.

As mentioned before preconditioners are used by iterative solvers. A bridge pattern is used to connect these two structure in a way that combining any iterative solver with any preconditioner will be possible. Figure 5.10 shows the combined structure using the bridge pattern.

All these classes take two spaces as their template parameters. One sparse space for defining operations over sparse matrices and vectors and a dense space which defines the operator over dense matrices and vectors which used as auxiliary matrices and vectors or for providing multiple right hand sides. For solving an small equation with dense matrices one can create these classes with two dense spaces without problem.

### 5.2.3 Examples

Here a simple conjugate gradient solver is implemented as an example to show how an algorithm can be implemented over a given space:

```
template<class TSparseSpaceType,
         class TDenseSpaceType,
         class TPreconditionerType
         class TReordererType >
class CGSolver : public IterativeSolver<TSparseSpaceType,
                                        TDenseSpaceType,
                                        TPreconditionerType,
                                        TReordererType>
  {
    public:
      CGSolver(double NewTolerance,
               unsigned int NewMaxIterationsNumber)
        : BaseType(NewTolerance, NewMaxIterationsNumber){}

      CGSolver(double NewMaxTolerance,
```

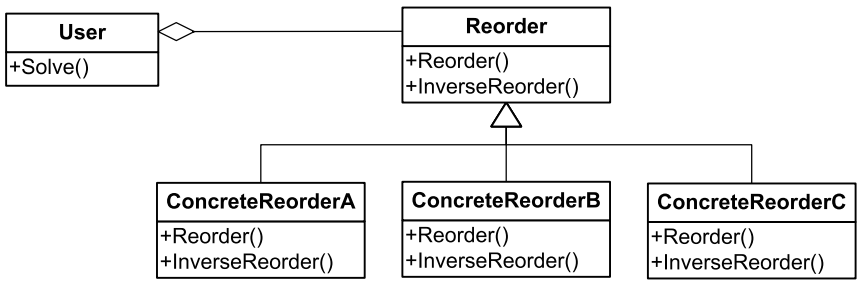



Figure 5.7: Using the Strategy pattern for designing the structure of reorderers.

```
            unsigned int NewMaxIterationsNumber,
            PreconditionerPointerType pNewPreconditioner)
    : BaseType(NewMaxTolerance,
               NewMaxIterationsNumber,
               pNewPreconditioner){}

  virtual ~CGSolver(){}

  bool Solve(SparseMatrixType& rA,
             VectorType& rX,
             VectorType& rB)
  {
    if(IsNotConsistent(rA, rX, rB))
      return false;

    GetPreconditioner()->Initialize(rA,rX,rB);
    GetPreconditioner()->ApplyInverseRight(rX);
    GetPreconditioner()->ApplyLeft(rB);

    bool is_solved = IterativeSolve(rA,rX,rB);

    GetPreconditioner()->Finalize(rX);

    return is_solved;
  }

private:

  bool IterativeSolve(SparseMatrixType& rA,
                      VectorType& rX,
                      VectorType& rB)
  {
    const int size = TSparseSpaceType::Size(rX);

    mIterationsNumber = 0;

    VectorType r(size);
```

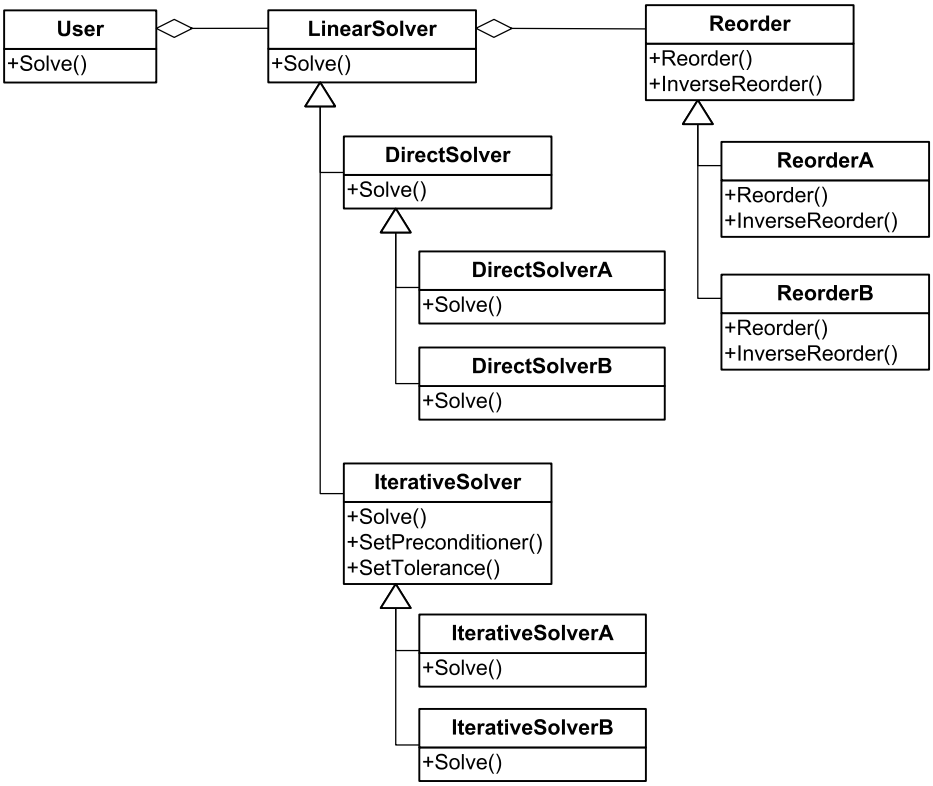



Figure 5.8: Applying the bridge pattern for connecting linear solvers and reorderers.

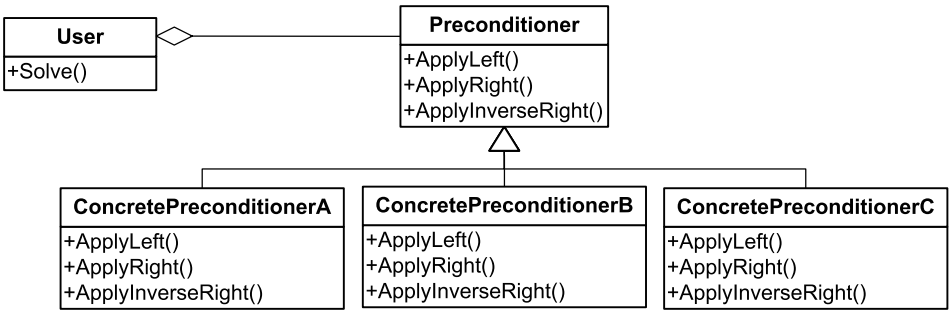



Figure 5.9: Strategy pattern is used for designing the structure of preconditioners.

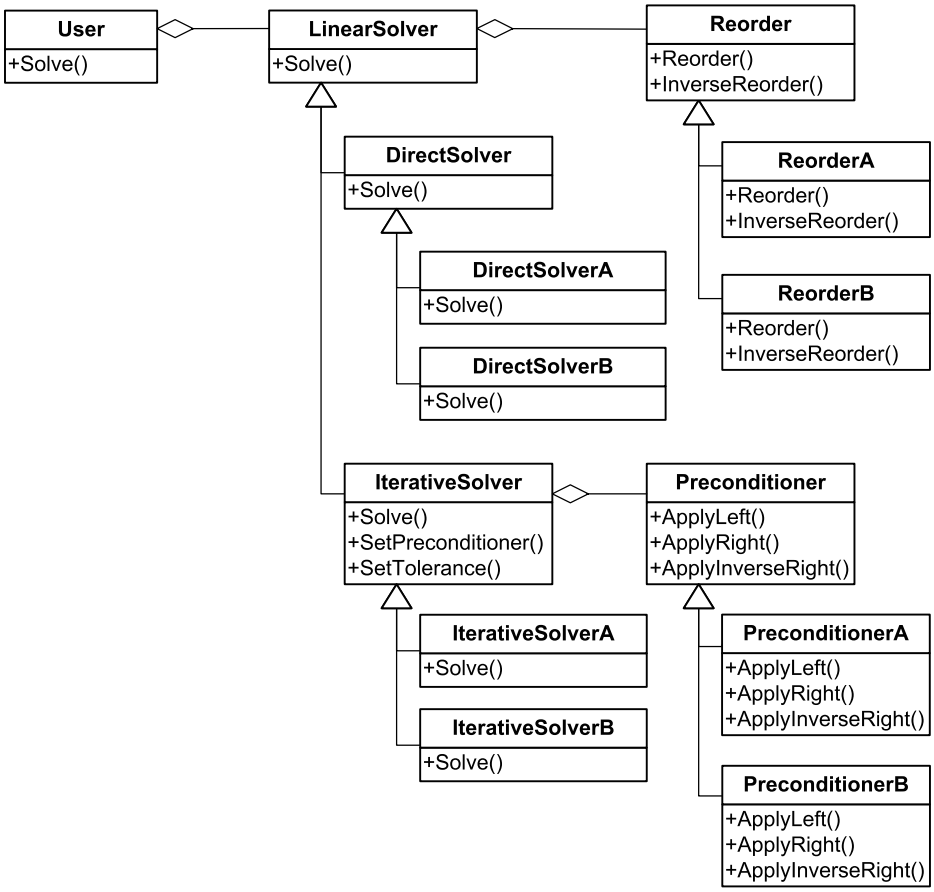



Figure 5.10: Applying the bridge pattern in connecting iterative solvers and preconditioners.

```
PreconditionedMult(rA,rX,r);
TSparseSpaceType::ScaleAndAdd(1.00, rB, -1.00, r);

mBNorm = TSparseSpaceType::TwoNorm(rB);

VectorType p(r);
VectorType q(size);

double roh0 = TSparseSpaceType::Dot(r, r);
double roh1 = roh0;
double beta = 0;

if(roh0 == 0.00)
return false;

do
{
  PreconditionedMult(rA,p,q);
```

```
            double pq = TSparseSpaceType::Dot(p,q);

            if(pq == 0.00)
              break;

            double alpha = roh0 / pq;

            TSparseSpaceType::ScaleAndAdd(alpha, p, 1.00, rX);
            TSparseSpaceType::ScaleAndAdd(-alpha, q, 1.00, r);

            roh1 = TSparseSpaceType::Dot(r,r);

            beta = (roh1 / roh0);
            TSparseSpaceType::ScaleAndAdd(1.00, r, beta, p);

            roh0 = roh1;

            mResidualNorm = sqrt(roh1);
            mIterationsNumber++;
          } while(IterationNeeded() && (roh0 != 0.00));

          return IsConverged();
        }

};// Class CGSolver
```

## 5.3 Geometry

Solving a problem using finite element method requires the discretization of model which consist of dividing the model into several small partitions with a defined shape. Geometries are designed to encapsulate these shapes, manage their data, and calculate their parameters. In this section the design and implementation of geometries in Kratos is described.

### 5.3.1 Defining the Geometry

As described in section 3.2 in the FEM a partial differential equation is converted to its equivalent integral form in discrete space which has a general form:

$$\int_{\Omega} L_1(Na)d\Omega + \int_{\Gamma} L_2(Na)d\Gamma = 0$$

where $N$ is the shape function and $a$ is the interpolated parameter. This integral form is divided into sub integrals over domain and boundary and which results in the following elemental form:

$$\sum_{e}^{n_e} \int_{\Omega_e} L_1(Na)d\Omega_e + \sum_{c}^{n_c} \int_{\Gamma_c} L_2(Na)d\Gamma_c = 0$$

where $n_e$ is the number of elements and $n_c$ number of conditions. This form consists of integrals over elements or boundary domain. The value of the shape functions $N$ and their derivatives are necessary for evaluating the integrals as operators $L_1$ and $L_2$ usually have derivatives operators.

The derivatives of shape functions are computed via the inverse of the jacobian matrix **J** as it can be seen in following equations for the three dimensional space:

$$\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix}$$

To perform the integration one way is to change from global coordinates to local ones. This transformation requires the calculation of the jacobian matrix $J$ and its determinant. All this process is described to define the geometry and help us to identify what can be encapsulated by the geometry.

`Geometry` encapsulates all the geometrical information like its dimension, points, edges and also some auxiliary operations like calculating the center point, area, volume, etc. Beside this geometrical information, it also provides the integration points and also the value of the shape functions and their local and global derivatives. It also calculates the jacobian matrix **J** , the inverse of the jacobian $\mathbf{J}^{-1}$ and its determinant $|\mathbf{J}|$.

All these operations are encapsulated to enable the element developers to write their `Element`s in a generic form and independent from the type of geometry.

### 5.3.2 Geometry Requirements

First of all `Geometry` has to be very lightweight and memory efficient. This requirement comes form the fact that modeling a real problem usually needs to store a large number of geometries in memory and process them. Having any unnecessary or even less necessary overhead for each geometry, results a significant overhead in memory used by program.

`Geometry` also has to be fast in its operations. `Element`s use geometries to perform calculation in most inner loops of the code, so their performance severely affects the global performance of the code specially in time of building equations system. In this way all parameters that can be calculated once must be kept to be used without recalculations. Also interfaces and algorithms must be designed and implemented in a way that minimum creation of temporaries be necessary.

Another requirement is enabling users to combine a generic `Element` with any `Geometry` without changing it. To achieve this objective the geometry structure has to provide an standard interface for all geometries. Doing this changing from one geometry to other will be easy while the interface to be used will be the same.

Finally adding a new geometry must be done without any need to change other part of the code. A good encapsulation and clear connection to other part of the code can help to achieve this objective.

### 5.3.3 Designing Geometry

Like in the linear solver's structure, an strategy pattern is used in the geometries structure. Using this pattern makes them interchangeable via a uniform interface. It also encapsulates all `Geometry` data and operations in one object which makes them more independent from each other and easier to be extended. Figure 5.11 shows this structure.

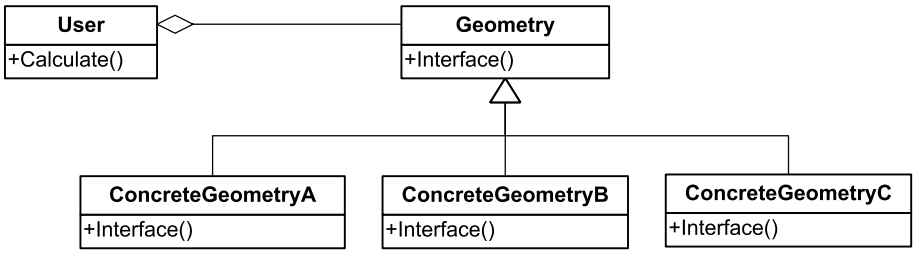



Figure 5.11: Designing the geometries structure, using the strategy pattern.

A composite pattern also can be used to let users combine different geometries to form a complex geometry. This structure may be useful in some situations where a complex geometry has to be defined. Figure 5.12 shows proposed structure.

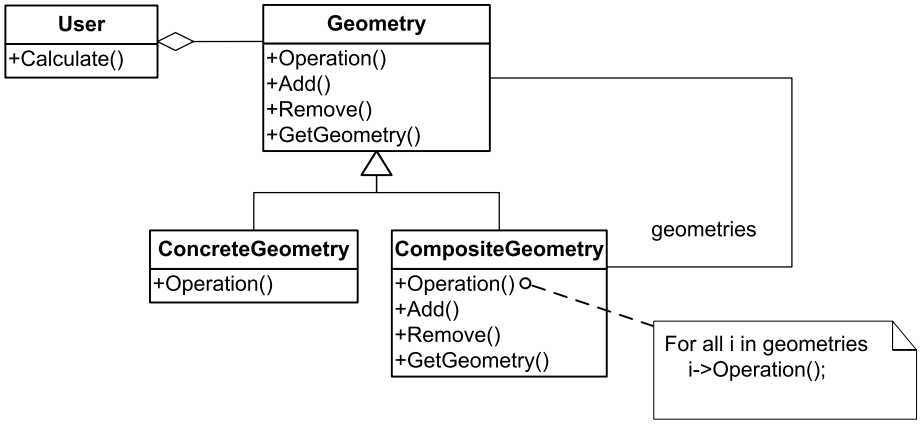



Figure 5.12: Composite pattern lets users combine different geometries in one complex geometry.

There are different ways of implementing geometries. Some common ways are to defined them as a set of points and constructive rules. The other way is an hierarchy form in which a three dimensional shape is defined by its two dimensional edges. These edges like any other two dimensional geometry are defined by their 1 dimensional edges. Finally these edges are defined by points. Geometries implemented in this form provide a complete set of information to users and are useful in situations that detailed information about the geometry and its edges is required.

In Kratos this implementation is considered to be too sophisticated and the first approach is used. There are two main reason for this decision. Algorithms implemented in Kratos work with geometry points (`Node`s for `Element` geometry). For this reason a fast access to `Node`s is more important than to its edges. The second reason is the memory overhead for holding all hierarchy is considerable and redundant for usual elemental algorithms. All these made us to use the first approach but keeping an empty place in geometry for inserting edges data base. `Geometry` is derived from points array. This approach lets users to operate with geometry like an array of points. For example move it just by moving all points in a loop or applying C++ standard library

| Method | Information |
|---|---|
| `Dimension` | Dimension of the geometry |
| `WorkingSpaceDimension` | Dimension of space which geometry is definied |
| `LocalSpaceDimension` | Geometries local dimension |
| `PointsNumber` | Number of points creating geometry |
| `Length` | Length of 1 dimensional geometries |
| `Area` | Area of 2 dimensional geometries |
| `Volume` | Volume of 3 dimensional geometries |
| `DomainSize` | Length, area, or volume depending to dimension |
| `Center` | Center point of geometry |
| `Points` | Geometries' points |
| `pGetPoint` | Pointer to $i$-th point of geometry |
| `GetPoint` | $i$-th point of geometry |
| `EdgesNumber` | number of edges of this geometry |
| `Edges` | Edges of geometry |
| `pGetEdge` | Pointer to $i$-th edge of geometry |
| `GetEdge` | $i$-th edge of geometry |
| `IsSymmetric` | True if geometry is symmetric |

Table 5.2: `Geometry` methods implementing geometrical operations.

to it using its point iterator.

The interface of `Geometry` reflects all operations provided by it. Table 5.2 shows methods implementing geometrical operations and table 5.3 shows methods providing finite element operations.

In order to optimize the performance of `Geometry` all interface which produces results in form of vector, matrix, or tensor accept their results as an additional argument. Here is an example:

```
Matrix& Jacobian(Matrix& rResult,
                 IndexType IntegrationPointIndex) const
{
  // Calculating the jacobian...

  return rResult;
}
```

The alternative design is getting parameters as arguments and return the calculated results. Here is the previous example using this alternative design:

```
Matrix Jacobian(IndexType IntegrationPointIndex) const
{
  Matrix result;

  // Calculating the jacobian...

  return result;
}
```

This alternative seems to be more elegant but produces a significant overhead in `Geometry` performance due to creating several unnecessary temporaries.

The information provided by `Geometry` can be divided in two different categories:

| Method | Information |
|---|---|
| `HasIntegrationMethod` | True if implements the integration method |
| `IntegrationPointsNumber` | Number of integration points |
| `IntegrationPoints` | Array of integration points |
| `GlobalCoordinates` | Global coordinates related to given local one |
| `Jacobian` | Jacobian matrix $\mathbf{J}$ |
| `DeterminantOfJacobian` | Determinant of jacobian $\|\mathbf{J}\|$ |
| `InverseOfJacobian` | Inverse of jacobian $\mathbf{J}^{-1}$ |
| `ShapeFunctionsValues` | shape functions' values in integration points |
| `ShapeFunctionValue` | Value of shape function $i$ in integration point $j$ |
| `ShapeFunctionsLocalGradients` | |
| `ShapeFunctionLocalGradient` | |
| `ShapeFunctionsFirstDerivatives` | |
| `ShapeFunctionsSecondDerivatives` | |
| `ShapeFunctionsThirdDerivatives` | |

Table 5.3: `Geometry` methods providing finite element operations.

**Constants** Information which depends only to the type of geometry and are equal for all geometries with the same type. Dimension, integration points, shape functions values, and shape functions local gradients are examples of operations in this category.

**Nonconstants** The second category contains the rest of information which can change from one geometry to another. For example points, jacobian, and shape functions gradients are in this category.

In order to optimize even more the performance of geometry, the different nature of these two categories must be considered. First, constant information can be calculated once and stored to be used if there are necessary. Also there is no need to access this information using virtual methods. Virtual functions can reduce the performance due to their function call delay specially for methods with small operations like returning a value. So making these methods no virtual can increase their performance.

As mentioned earlier constant information can be calculated once and stored to be used later. The draw back of this idea is the memory required by pointers pointing to these data for each geometry. While the memory used by these pointers is relatively small, but creating a large number of geometries makes it considerable. A good solution to this problem is creating an auxiliary object and put all constant information in it. In this way only one pointer in each geometry is necessary to point to this object and access all constant information. `GeometryData` is defined and implemented to hold all constant information in geometry. Figure 5.13 shows this class and its attributes.

`Geometry` keeps a reference to the geometry data and use it to provide constant information by its no virtual methods. Figure 5.14 shows the complete structure.

For each type of geometry an static variable of `GeometryData` is created and its reference is given to geometry in construction time.

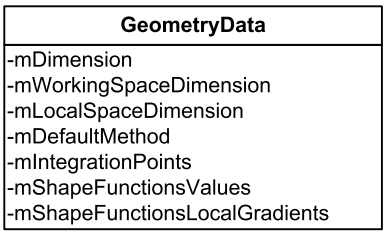



Figure 5.13: `GeometryData` class and its attributes.

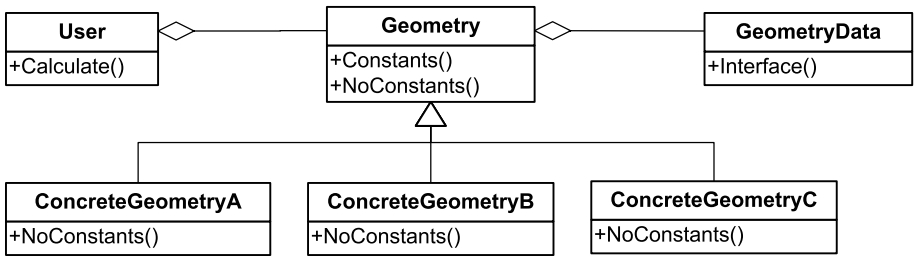



Figure 5.14: `Geometry` uses `GeometryData` for providing constant information and its derived classes only implement the rest of operations.

Chapter 6

# Variable Base Interface

## 6.1 Introduction

Finite element codes has been changed a lot during the evolution of the Finite Element Method, In its early times, finite element applications were developed to solve a certain type of problems in an specific domain. Many applications for solving shells, plates, stokes fluid etc. were created in this period. By the time, engineers began to apply FEM in engineering problems which mostly consist of more than one type of problem. First they tried to solve it by separating the problem and solve each part by one module meanwhile developers began to gather previous small modules related to the same domain in a domain specific application which could handle more complex problems. Following this line various applications for solving structural, fluid and other problems created.

Applying FEM to solve a coupled problem was a natural step forward from single domain problems. Reusing the existing codes and connect them via an interface is the most common and approved way to deal with these problems. In this way complexity reduced and existing modules for solving each domain can be reused to solve coupled problem. But what happen if the interfaces are inconsistent? Writing a file and reading it in another module is a common way while there is more elegant way to do it using libraries which can handle objects in a generic way. There are many successful examples using file interface or libraries like CORBA [101] or omniORB, though using them can cause big overheads in the performance of the code. The latency time of invoking a request on a CORBA object is approximately 500-5000 times higher than doing it by a function call in C++ [58].

In these days still developing and extending finite element applications to solve new and more complex problems is a challenge. Also using FEM in any new area creates a new demand for developing an application to make the method applicable in practice. So flexibility, extensibility and reusability are the key features in the design and development of modern finite element codes.

Though the previous strategy to chain different modules in an application works fine, the level of reusability is respectively low. The reason is that in this manner we can reuse the whole module but not a part of it. For example each module has its own data structure and IO routines which cannot be used by another method or new modules in the same way. This is the motivation to establish a variable base interface which can be used at high and low levels in the same manner.

## 6.2 The Variable Base Interface Definition

In many connection points between different parts of a finite element program we are asking a value of some variable or mapping some variables and so on. The methodology to design this interface is sending each request with variable or variables it depends on. It sounds simple and natural, meanwhile it is a powerful trick to make generic algorithms and modules using variables.

First let us see what we need in general to create some finite element generic algorithms:

- Type of a variable is an important information while many operation and also storing mechanism are depended to it
- Sometimes we need also the name of variable, (i.e. to print the results).
- While each variable can be identified by its name but using name in checkpoints give a big unnecessary overhead (comparing two strings is respectively slow) so an identifier number is essential, for example in case of fast searching and database working.
- Another useful information is a zero value. Though this information seems unnecessary but it helps a lot in case of initializing zero vectors and matrix with the correct size.
- Also in some modules we may need to work just with some component of a variable. In such a cases we need a mechanism to identify the components owner.
- Another useful feature is to work with raw pointers. This can be useful specially in the case of connecting external modules and extracting value form pointer passed by modules.

Keeping in mind all above requirements now a global design is possible. In this variable base interface:

- A variable encapsulate all information needed by different objects to work in a generic way over different variables. Doing this helps to simplifying modules interfaces. Considering a `PrintNodalResult` module which normally need result name, an index to retrieve nodal results and a zero value if some `Node`s don't have results. All this information can be passed by one variable in this design.
- Each module must configure itself in term of variable or variables given to it.
- Type-safety reached by statically typing variables.
- Each instance of variable class has the same name as its represented variable name. This is a great added value to code readability.

## 6.3 Where Can be used?

As mentioned earlier, in a multi-disciplinary application always there is a need to exchange data between different domains. So one important point is to exchange data in a robust and safe way. Developing different modules by the VBI (Variable Base Interface) make data exchanging between them trivial, though each domain has different data stored and sometimes the internal data structure may be different. For example a fluid application may have velocities and pressures stored while a thermal domain just temperature. When using variables the interface provides the means to recover their values as well as to store new values for given variable.

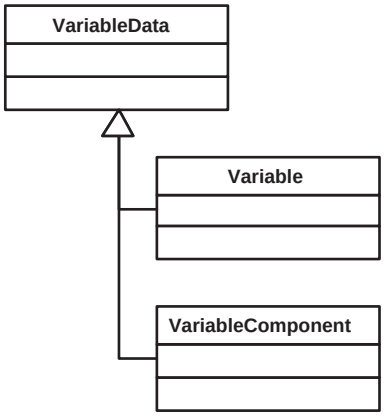



Figure 6.1: Variable classes structure

Now let us go one level down, by developing a set of generic algorithms and data structures in a VBI conforming manner. This can boost up reusability of the code. Also code management becomes easier while any new extension can be done without changing the interface. There are several algorithms which can be equipped with VBI. For example:

**Data Structure** First and more important place is to store and then retrieve a value in data structure. Here type, id and zero value are needed to create a typesafe data container. All this information is represented by a variable. The interface is `GetVAlue` or `SetValue` and so on for given variable.

**Reading input file** Each variable has its type and name, so they can be used to validate tokens. In this way the input interpreter can be extended to read new data just by defining a variable presenting this new data.

**Printing output** Printing output needs name of variable and its value and zero in case of no data. Using a data structure with VBI, and sending a request to print a variable gives all information to print even for new variables.

**Mapping or interpolating** Again using a VBI conforming data structure we can generalize, or map and interpolate algorithms.

## 6.4 Kratos variable base interface implementation

In Kratos `VariableData` and its derivatives `Variable` and `VariableComponent` represent interface information.

### 6.4.1 VariableData

`VariableData` is the base class which contains two basic information about the variable it represents; Its name `mName`, and its key number `mKey`.

`VariableData` has trivial access methods for these two attributes while also has virtual and empty methods for raw pointer manipulation. The reason of not implementing these methods here is the fact that `VariableData` has not any information about the variable type it represents. Lack

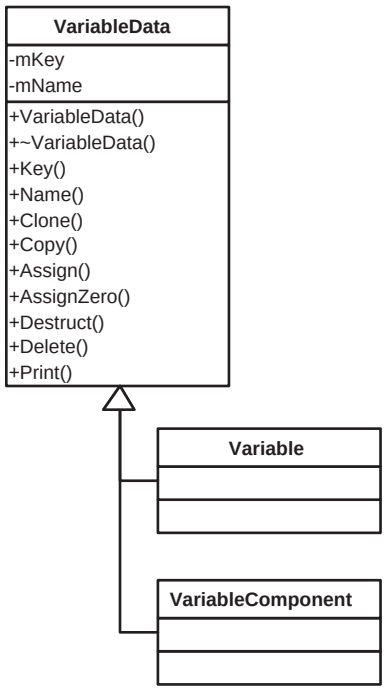



Figure 6.2: VariableData class

of type information in `VariableData` make it unsuitable and less usable to pass it in many parts of the interface but the idea of this class is to provide a lower level of information which is common between all types of variables and their components and use it as a place holder when there is no distinction between variables and their components. Also in this implementation we use a virtual method base to dispatch various operations on raw pointers. This may result a poor performance in some cases while the function call overhead seems to be considerable.

### 6.4.2 Variable

`Variable` is the most important class in this structure. It has information represented by `VariableData` which is derived from it. Also it has another important information which is the type of variable representing.

Using C++ as implementing language `Variable` has its data type as a template parameter. In this manner interface can be specialize for each type of data and also type-safety can be achieved. Another important advantage of having `variable` in template form with its data type as template parameter is to have a restriction form in interface. If we want to restrict a method to accept just matrices for instance, then by passing a `variable<Matrix>` as its argument we can prevent users to pass another type by mistake. This feature shows to be important especially in some cases which there are different types representing the same variable. (Constitutive matrix and its corresponding vector is a good example of this case). Variable data type `TDataType` must satisfy following requirements:

- Copy constructible.

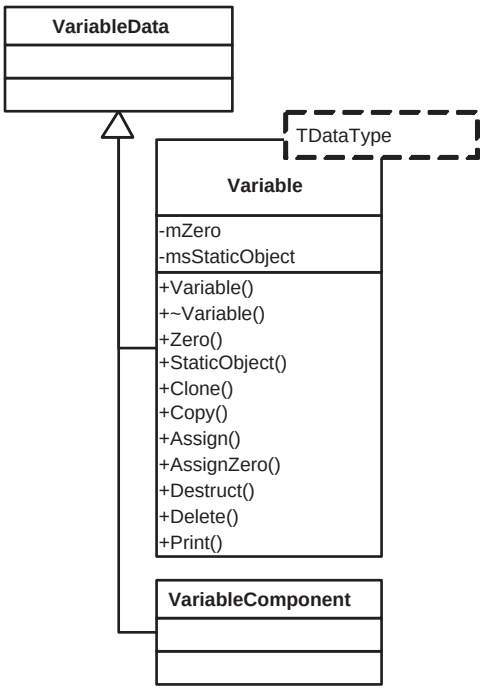



Figure 6.3: Variable class

- Assignable.
- streaming operator defined

In `Variable` by knowing the data type raw pointer manipulating methods are implemented. These methods use raw pointers to perform basic operations over its type:

- `Clone` create a copy of this object using copy constructor of class. It takes a `void*` as its argument and returns also `void*` which is pointing to the memory allocated for cloned object. `Clone` is useful to avoid shallow copying of complex objects and also without actually having information about variable type.
- `Copy` is very similar to `Clone` except that destination pointer also passed as a `void*` argument to it. it is a helpful method specially to create a copy of heterogeneous data arrays .
- `Assign` is very similar to `Copy`. It just differs in using assignment operator beside copy constructor. `Copy` is to create a new object while `Assign` do the assignment for both existing objects.
- `AssignZero` is a special case of `Assign` which variable zero value used as source. This method is useful for initializing arrays or resetting values in memory.
- `Delete` removes an object of variable type from memory. It takes a `void*` as its argument which is the location of object to be removed. `Delete` calls destructor of object to prevent memory leak and frees the memory allocated for this object assuming that object allocated in heap.

- `Destruct` is to destruct an object maintaining the memory it is using. It takes a `void*` as its argument which is the location of object to be destructed. It calls destructor of object but unlike `Delete` it does nothing with memory allocated for it. So it is very useful in case of reallocating a part of memory.

- `Print` is an auxiliary method to produce output of given variable knowing its address. For example writing an heterogenous container in output stream can be down using this method. This method assumes that streaming operator is defined for the variable type.

All this methods are available for low level usage. They are useful because they can be called by a `VariableData` pointer and equally for all type of data arranged in memory but maintaining typesafety using these methods is not straightforward and needs special attention.

Zero value is another attribute of `Variable`, stored in `mZero`. This value is important specially when a generic algorithms needs to initialize a variable without losing generality. For example an algorithm to calculate the norm of a variable for some `Element`s must return a zero if there is no `Element` at all. In case of double values there is no problem to call default constructor of variable type but applying same algorithm to vector or matrix values can cause a problem because default constructor of this types will not have the correct size. But returning a zero value instead of default constructed value keeps generality of algorithms even for vectors and matrices, assuming that variables are defined properly.

There is another method which is `StaticObject`. This method just returns `None` which is an static variable of this type with None as its name and can be used in case of undefined variable (like null for pointers). It is just an auxiliary variable to help managing undefined, no initialized or exceptional cases.

### 6.4.3 VariableComponent

As mentioned before, there are situations that we want to deal with just component of a variable but not all of it. `VariableComponent` implemented to help in these situations.

`VariableComponent` like `Variable` derived from `VariableData`.

`VariableData` is a template taking an adaptor as its argument. Adaptor is the extending point of component mechanism. For any new type's component a new adaptor needed to be implemented. This adaptor type requirements are:

- `GetSourceVariable` method to retrieve parent variable.

- `GetValue` method to convert extract component value from source variable value.

- `StaticObjec` used to create none component.

Unlike `Variable`, `VariableComponent` has not been implemented to have zero value or raw pointer manipulators. A zero value can be extracted from the source value so there is no need to have it here. Operations over raw pointers are not allowed here by purpose. This interface manages variables entirely and not just some part of them. In fact a part of an object cannot be copied, cloned, deleted or destroyed. So these methods are not implemented to protect objects from unsafe memory operations.

Having adaptor as template parameter helps compiler to optimize the code and eliminating overheads. In this manner adaptor's `GetValue` method will be inlined in `VariableComponent`'s one so there won't be any overhead due to decomposition while extensibility reached.

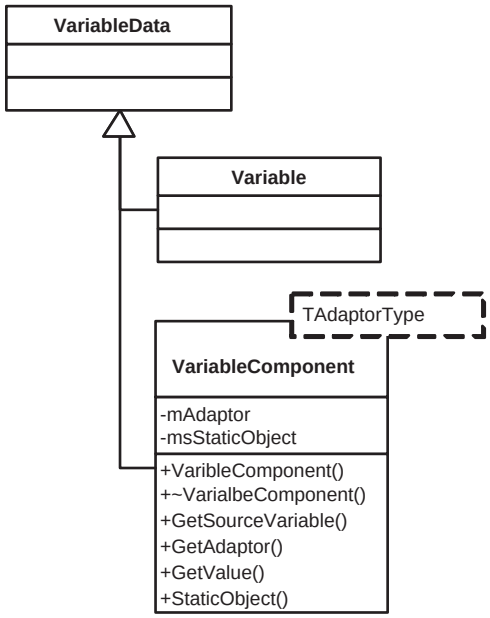



Figure 6.4: VariableComponent class

## 6.5 How to Implement Interface

Sending any request with variables is the way this interface works. But to make the things working in real world a uniform way to getting and setting variables value in objects must be defined.

### 6.5.1 Getting Values

In many interfaces there are methods like `GetDisplacement` or `Flux` to get values of for example displacement or thermal flux. This cannot work for a generic interface while the variables to be accessed can be completely different from one domain to another. The idea is to specify the variable not by the method name but by the variable passed to a generic method. In this manner any previously defined algorithm can be reused for new domain and new variables using this generic access method.

Having a `GetValue` method is essential for each class which contains a data to be processed. Note that the name of this method doesn't change with the variable we want to get . For classes with just one type or few types of data it can be written as a normal method using specific variable

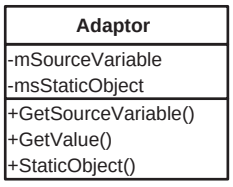



Figure 6.5: Adaptor class

type as argument:

```
DataType GetValue(VariableType const&) const

DataType const& GetValue(VariableType const&) const

DataType& GetValue(VariableType const&)
```

These three versions are different in returning the values which make them usable in different places.

The first version is more suitable for returning some calculated and not stored value and also for small objects like doubles.

The second one is useful for an internal value access method returning just a reference to the internal variable without any cost.

The third version is for left value representation of an internal variable via this interface. The returned reference is not constant so the value can be changed and can be used as a left value.

In the case of classes containing arbitrary type variables like heterogenous containers `GetValue` can be implemented in template form maintaining generality of the code

```
template<TDataType> TDataType GetValue(Variable<TDataType> const&)
const

template<TDataType> TDataType const& GetValue(Variable<TDataType>
const&) const

template<TDataType> TDataType& GetValue(Variable<TDataType>
const&)
```

In this way the interface is open to not only any new variable but also to any new type of variables. Here template member specialization can be used as a very useful tool to define exceptional cases and also to optimize the code using different specialized implementations.

Sometimes it is necessary to take a variable component and not hole variable. Note that `VariableComponent` is not convertible to `Variable` therefore cannot be pass as argument in above patterns. So to make this possibility another set of methods with `VariableComponent` as their argument needed to be implemented. As before, an specific implementation can be done using known adaptor for specific component:

```
DataType GetValue(
  VariableComponentType const&) const

DataType const& GetValue(
  VariableComponentType const&) const

DataType& GetValue(VariableComponentType const&)
```

Also template implementation for more generic interface:

```
template<TAdaptorType> typename TAdaptorType::Type GetValue(
  VariableComponent<TAdaptorType> const&
  ) const

template<TAdaptorType> typename TAdaptorType::Type const&
GetValue(
```

```
  VariableComponent<TAdaptorType> const&
  ) const

template<TAdaptorType> typename TAdaptorType::Type& GetValue(
  VariableComponent<TAdaptorType> const&)
```

In above patterns having the type of component defined in adaptor used to specify the type of return value. In this manner adaptor is responsible to specify the return value type and extensibility guaranteed.

Finally methods for getting variables and those for getting components can be combined in one set of methods creating a uniform interface:

```
template<TVariableType> typename TVariableType::Type GetValue(
  TVariableType const&) const

template<TVariableType> typename TVariableType::Type const&
GetValue(
  TVariableType const&) const

template<TVariableType> typename TVariableType::Type& GetValue(
  TVariableType const&)
```

### 6.5.2 Setting Values

This part of the interface has the same characteristic of getting values. Though the third version of getting values can be used to setting it as a left value but in some cases a separate set of methods needed to ensure generality of the interface.

To set a value of a variable it is necessary to pass both the variable and its value as `SetValue` argument. Again implementation depends on the variety of variables that have to be accessed via this interface. For few variable types accessing `SetValue` method can be implemented by knowing data type and its variable type:

```
void SetValue(VariableType const&,
         DataType     const&)
```

Not that here we have just one version, due to the fact that `SetValue` method cannot be constant. Again to implement a more general version templates are useful:

```
template<TDataType> void SetValue(VariableType const&,
         TDataType    const&)
```

In the case of setting a component the two above versions must be modified in order to accept a `VariableComponent` instead of `Variable`.

```
void SetValue(VariableComponentType const&,
         DataType              const&)
```

or template version using adaptors:

```
template<TAdaptorType> void SetValue(VariableComponentType
const&,
     typename TAdaptorType::Type const&)
```

Finally the uniform version combining variables and their components in the same method:

```
template<TVariableType> void SetValue(TVariableType
const&,
     typename TVariableType::Type const&)
```

### 6.5.3 Extended Setting and Getting Values

In some cases there are additional information needed to access a variable inside a class. For simple cases an additional index is enough to indicate solution step or `Node`'s number which are passed as extra argument.

In more complex interfaces an information object with a VBI interface also is passed as argument to extend its generality. In this way any new information just passed through existing interface and can be retrieved by the interface.

```
template<TVariableType> typename TVariableType::Type
GetValue(TVariableType const&,
         InfoType const&) const

template<TVariableType> typename TVariableType::Type const&
GetValue(TVariableType const&,
         InfoType const&) const

template<TVariableType> void SetValue(TVariableType
const&,
     typename TVariableType::Type const&,
     InfoType                    const&)
```

Using the variable base interface to extending itself is a very powerful trick to create extremely flexible interfaces without sacrificing the clarity of the code. In this way not only the type and the variable to be passed is extendible but also the number and type of arguments to be passed can be arbitrary extended.

### 6.5.4 Inquiries and Information

Not all the time we need to transfer data by the interface and there are some situations where an inquiry is needed or some information about an object respect to a certain variable is required. Assuming that in this operations the type and components of a variable are not important, the base class `VariableData` can be used in interface to deal with all kinds of variables and their components in the same way.

### 6.5.5 Generic Algorithms

Now it is time to implement generic algorithms using this interface. In general algorithm interface can use any forms described above for getting and setting specially the extended ones. In fact each algorithm is very different in the arguments it accepts but what is important here is the variable or variables needed which are passed through the interface. The point is to use the the interface of objects inside algorithms using variables passed to it. In this way the algorithm is transparent to the working variable and can be applied generally to any new variable in new domains.

Many algorithms can be generalized in this way, reading inputs, print results, mapping and interpolating values, calculating norms and errors, etc. All these algorithms have a common property: they operate over the value of some variable. A carefully implementation of these algorithms can free them from knowing really which variable they are working on. They just work

on the variable which the user passes to them and use that to extract values it is needed. In other word, algorithms are implemented between two layer of interfaces.

The idea looks simple, but in practice results in reusable algorithms which can be used in any new area of work with a new set of variables.

## 6.6 Examples

To see how above idea works in practice and also to make some details more clear, some examples from different levels of interface are presented next.

### 6.6.1 Nodal interface

A very first example is to access nodal values in a finite element program. We will consider that any new extension to the program may introduce new set of variables to access. Using variable base interface can be a solution for extensibility needed in these cases. A basic get and set value interface implemented to give basic accessibility is as follow:

```
// Getting a reference
template<class TVariableType> typename TVariableType::Type&
Node::GetValue(TVariableType const&) {
// Accessing to database and
// returning value ...
}

// Getting a constant reference
template<class TVariableType> typename TVariableType::Type const&
Node::GetValue(TVariableType const&) const {
// Accessing to database and
// returning value  ...
}

template<class TVariableType> void Node::SetValue(TVariableType
const&, typename TVariableType::Type const&) {
// Accessing to database and
// setting value  ...
}
```

It can be seen that the uniform version of the interface adapts here perfectly and prevent us from having different versions of `GetValue` and `SetValue` for variables and variables components. And finally overwriting the `[]` operators make the syntax easier to use:

```
template<class TVariableType> typename TVariableType::Type&
Node::operator[](const TVariableType&) {
  return GetValue(rThisVariable);
}
```

Also some inquiries can be implemented to see if the variable exists:

```
template<class TDataType> bool Node::Has(Variable<TDataType>
const&) const {
// Check if the variable
// stored before ...
}
```

```
template<class TAdaptorType> bool
Node::Has(VariableComponent<TAdaptorType> const&) const {
// Check if the variable component
// stored before ...
}
```

Here, the separate variable and components methods used to implement this part of interface. As mentioned before, This can be done just by using the `VariableData` in the interface:

```
bool Node::Has(VariableData const&) const {
// Check if the variable
// stored before ...
}
```

Note that, to use the `VariableData` version a uniform search by key id is required from the internal containers.

Now it is easy to use this `Node` in the code and access any variable through interface:

```
// Getting pressure of the center node

double pressure = center_node[PRESSURE];

// Setting velocity of node 1

Nodes[1][VELOCITY] = calculated_velocity;

// Printing temperature of the
// nodes to output

for(IteratorType i_node = mNodes.begin() ;
    i_node != mNodes.end() ; i_node++)
{
    std::cout << "Temperature of node #"
        << i_node->Id()
        << " = "
        << i_node->GetValue(TEMPERATURE)
        << std::endl;
}

// Setting displacement of nodes to zero

Vector zero = ZeroVector(3); for(IteratorType i_node =
mNodes.begin() ;
    i_node != mNodes.end() ; i_node++)
{
  i_node->SetValue(DISPLACEMENT, zero);
}
```

Accessing to history of nodal variable is an example of a simple extended interface. Keeping previous arguments for `GetValue` and `SetValue` an additional parameter can be passed to indicate for example time step needed.

```
template<class TVariableType> typename TVariableType::Type&
Node::GetValue(const TVariableType&,
```

```
  IndexType SolutionStepIndex)
{
// Accessing to database and
// returning value ...
}

template<class TVariableType> typename TVariableType::Type const&
Node::GetValue(const TVariableType&,
  IndexType SolutionStepIndex) const
{
// Accessing to database and
// returning value ...
}
```

Also `()` operator can be overridden to provide an easy interface. (Note: `[]` operator cannot accept more than one argument and cannot be used here)

```
template<class TVariableType> typename TVariableType::Type&
Node::operator()(const TVariableType&,
  IndexType SolutionStepIndex)
{
  return GetValue(rThisVariable,
                  SolutionStepIndex);
}
```

In practice this interface can be more complicated due to the fact that the history must not be stored for all variables and it is useful to have separate containers to store historical and not historical variables. For example in Kratos there are two sets of access methods to give the possibility of storing only needed history and not all of them.

### 6.6.2 Elemental Interface

In `Elements`, access methods can be implemented like `Nodes`. it is better to keep the interface as unchanged as possible to avoid extra effort due to the inconsistence interfaces. So let's leave the access methods as before and take some other methods to make examples.

Methods to calculate local matrices and vectors also can be implemented using VBI. But what is the advantage of using VBI here? There are several additional parameters needed for calculating local contributions of each `Element`, like time step, current time, delta time and so on. These arguments can be completely different from one formulation to the other and passing all of them as individual arguments is somehow impossible. An approach is to create a helper class to encapsulate these arguments and pass all of them through this helper class. Again the VBI can be used here to make a uniform interface also at this level of working. In Kratos there is a helper class named `ProcessInfo` which is passed to the methods which calculating the local systems as follows:

```
virtual void SomeElement::CalculateLocalSystem(
    MatrixType& rLeftHandSideMatrix,
    VectorType& rRightHandSideVector,
    ProcessInfo& rCurrentProcessInfo)
{
  // Getting process information
  double time = rCurrentProcessInfo[TIME];
```

```
  // Calculating local matrix and
  // vector ...
}
```

### 6.6.3 Input-Output

Writing a generic input-output with enough flexibility is a complex task. Making extensions causes problems in many codes while the IO is not flexible enough to process new variables.

In modern applications a parser used to read the input file and understand the input grammar [54, 77]. Normally this input file parser is a complex part of the code and modifying it for any new variable added by extensions is expensive and not free of bugs. A good solution is to make the parser work with a list of variables and change the list each time reading a new variable is needed. Doing this adding new extensions to the input parser can be much easier while the parser itself won't be changed. The input file parser is too big to add here as an example and interested readers can find a working version of it in Kratos `DatafileIO` class.

Writing output files like reading inputs can be generalized using variables. In this way any new extension to the library can write its results using existing output procedures. Here is an example of using variables to write a generic output procedure for GiD [84, 83]:

```
void GidIO::WriteNodalResults(Variable<double> const& rVariable,
                          NodesContainerType& rNodes,
                          double SolutionTag,
                          std::size_t SolutionStepNumber)
{
  GiD_BeginResult( (char*)(rVariable.Name().c_str()),
    "Kratos",
    SolutionTag,
    GiD_Scalar,
    GiD_OnNodes, NULL, NULL, 0, NULL );

  for(NodesContainerType::iterator i_node = rNodes.begin();
      i_node != rNodes.end() ; ++i_node)
      GiD_WriteScalar( i_node->Id(),
          i_node->GetSolutionStepValue(rVariable,
              SolutionStepNumber));

  GiD_EndResult();
}

void GidIO::WriteNodalResults(Variable<Vector> const& rVariable,
                          NodesContainerType& rNodes,
                          double SolutionTag,
                          std::size_t SolutionStepNumber)
{
  GiD_BeginResult( (char*)(rVariable.Name().c_str()),
    "Kratos",
    SolutionTag,
    GiD_Vector,
    GiD_OnNodes, NULL, NULL, 0, NULL );

  for(NodesContainerType::iterator i_node = rNodes.begin();
      i_node != rNodes.end() ; ++i_node)
```

```
  {
    array_1d<double, 3>& temp =
      i_node->GetSolutionStepValue(rVariable, SolutionStepNumber);
    GiD_WriteVector( i_node->Id(), temp[0], temp[1], temp[2] );
  }

  GiD_EndResult();
}
```

In above examples the GiD interface has different rules for scalar and vectorial variables. Knowing the type of variable helps to implement customized versions of `WriteNodalResults` for each type of variable. This is an important feature of this interface which can handle exceptional cases for certain types and handle them with a uniform syntax for users:

```
// Writing temperature of all the nodes
gid_io.WriteNodalResults(TEMPERATURE, mesh.Nodes(), time, 0);
// Writing velocity of all the nodes
gid_io.WriteNodalResults(VELOCITY, mesh.Nodes(), time, 0);
```

Each variable as mentioned before has its name which can be accessed via `Name` method. This gives us necessary information to print the variable in output. In this way there is no need to pass the name of the variable as an argument by itself. Though This wouldn't be difficult, it keeps the simplicity of the interface.

### 6.6.4 Error Estimator

Writing an error estimator is another example of making a generic and reusable code using VBI. Here is an example of a simple recovery error estimator [104] implemented in a generic way:

```
virtual void EstimateError(const VariableType& ThisVariable,
                           ModelPart& rModelPart)
{
  mpRecovery->Recover(ThisVariable, rModelPart);

  double e_sum = double();

  typedef ModelPart::ElementIterator iterator_type;

  for(iterator_type element_iterator = rModelPart.ElementsBegin();
      element_iterator != rModelPart.ElementsEnd();
      ++element_iterator)
  {
    double error = CalculateError(ThisVariable,*element_iterator);
    element_iterator->GetValue(ERROR) = error;
    e_sum += error;
  }

  SetGlobalError(e_sum);

}

double CalculateError(const VariableType& ThisVariable,
                      Element& rThisElement)
{
```

```
  Element::NodesArrayType& element_nodes = rThisElement.Nodes();

  if(element_nodes.empty())
    return double();

  double result = 0.00;

  typedef Element::NodesArrayType::iterator iterator_type;

  for(iterator_type node_iterator = element_nodes.begin() ;
      node_iterator != element_nodes.end() ; ++node_iterator)
  {
    TDataType error = CalculateError(ThisVariable,
                                     rThisElement,
                                     *node_iterator);

    result += sqrt(error * error);
  }

  result *= rThisElement.GetGeometry()->Area();
  return result / element_nodes.size();
}

TDataType CalculateError(const VariableType& ThisVariable,
                         Element& rThisElement,
                         Node& rThisNode)
{
  TDataType result = rThisNode[ThisVariable];

  result -= rThisElement.Calculate(ThisVariable, rThisNode);
  return result;
}
```

In this manner the error estimator is not depended to the domain and can work in the same way with thermal flow or pressure gradient.

## 6.7 Problems and Difficulties

As usual nothing is perfect and the VBI is not excluded from that. There are some difficulties and problems still arising using this interface. Though they are not so important in some cases but in some other ones they need more attentions.

A problem is to store a variable which can be replaced with a component. Where can this happen? Any process which works over variables or their components in the same way and wants to store the variable or component causes some difficulties. A typical example is a process over some doubles, like calculating the norm, and applied to a double variable, like temperature, or a component of a vector, velocity_x, which also wants to keep a given variable to work on it after. In many cases this can be easily solved just by replacing the data type template parameter with a variable type parameter. The pattern can be same to the uniform template version of `GetValue` and `SetValue` methods.

But still there are some cases which above method cannot solve so easily. Storing the variable of a dof is a complex task. Assuming that each dof wants to store its variable and then search

database to find its value for example to update the results:

```
// Updating
typedef EquationSystemType::DofsArrayType::iterator iterator_type;
for(iterator_type i_dof = equation_system.DofsBegin() ;
    i_dof != equation_system.DofsEnd() ; ++i_dof)
{
  if(i_dof->IsFree())
    i_dof->GetSolutionStepValue() =
      equation_system.GetResults()[i_dof->EquationId()];
}
```

In this case giving different template parameters to Dofs prevent us to use them in a normal array and using virtual functions is not allowed due inner loop usage of these methods in finite element code. So what to do? In Kratos an indexing mechanism is used to decide if the variable is a component or not and the some *traits* [99] are used to dispatch and select the proper procedure. This solution is not so clean and applying it is not so encapsulated yet. Further work to improve these tasks remains to be done in the future.

```
TDataType& GetReference(VariableData const& ThisVariable,
                        FixDataValueContainer& rData,
                        int ThisId)
{
  switch(ThisId)
  {
    KRATOS_DOF_TRAITS
  }
  KRATOS_ERROR(std::invalid_argument, "Not supported type for Dof" , "");
}
```

Chapter 7

# Data Structure

Data structure is one of the main parts of a finite element program. Many restrictions in functionality of finite element codes comes from their data structure design. The Kratos' data structure has to provide the high flexibility, necessary for dealing with multi-disciplinary problems.

In this chapter first, a brief description of different concepts in data structure programming is given. Then, some classical containers are explained and their advantages and disadvantages are discussed. It continues with design and implementation of new containers suitable for multi-disciplinary finite element programming. After that, some common organization of data in finite element programming is explained and finally the organization of the data structure in Kratos is described.

## 7.1 Concepts

In this section a brief description of common concepts in data structure programming is given. More information about the concepts described here and also other concepts in this field can be found in [55, 14, 46].

### 7.1.1 Container

*Container* is an object which stores another objects and gives some method to access these objects, add new objects, or remove some objects from it. Each object stored in a container is referred as an *element* of it. Container must provide some methods for creating and modifying it and also some access method to its element. Here is a list of some common methods:

**Access** Gives the element in given position. Depending on the container the access may be implemented via position or by some reference key. Usually the `[]` operator provides this interface for container.

**Insert** Inserts a given element after given position. Adding elements to a container increases its size.

**Append** Adds given element to the end. Adding elements to a container increases its size.

**Erase** Removes the element in given position. This operation reduce the size of container by the number of erased elements.

**Find** Searches for an element with given specification in container.

**Size** To get size of the container.

**Resize** Changes the size of container.

**Swap** Swaps the content of the container with a given one.

Some containers may not provide all these methods due to their structure and some other may provide some more methods for their specific uses. Also the performance of these operations depends highly on the internal structure of container. So before using a container it is very important to study its performance in term of the operations needed by algorithm.

### 7.1.2 Iterator

Usually a pointer referred as an *iterator* is used to access elements of a container. Here is an example of using a pointer as a C array iterator to print its contents:

```
double data[10];

// putting values in data
// ...

// now printing data using an iterator
double* data_end = data + 10;
for(double* i = data ; i != data_end ; i++)
    std::cout << *i << std::endl;
```

The *Iterator* pattern defines a generalized pointer to access elements of a container sequentially without exposing its internal structure. An iterator can be used to traverse element by element the container in a general way and without knowing how they are really stored in memory. Changing the C array of previous example to a container gives an example of using iterator in a general form:

```
ContainerType data;

// putting values in data
// ...

// now printing data using an iterator
typedef ContainerType::iterator iterator_type;
 for(iterator_type i = data.begin() ; i != data.end() ; i++)
    std::cout << *i << std::endl;
```

In this example any container which provides an iterator and two methods to indicate its begin and end position can be used to print its contents.

Iterator can be designed to traverse the container in different manner. For example a matrix can have different iterators:

`iterator` iterates over all members from $a_{11}$ to $a_{mn}$

`row_iterator` iterates over rows of the matrix.

`column_iterator` iterates over columns of the matrix.

`nonzero_iterator` iterates over all nonzero element.

This makes iterator a very powerful tool to access a container in a generic way. For algorithms using iterators there is no need to know about the container itself and this make them more generic.

Depending on the structure of container there are certain traversing is impossible or meaningless. So different containers use different category of iterators due to their restrictions. Here are the main categories of iterators:

**Forward Iterator** A simple iterator which allows just moving to the next element. This iterator cannot be used to go backward. For example to see the previous element of iterator position. Forward iterator can move only one element forward in each step and cannot be used to jump by a certain offset.

**Bidirectional Iterator** Unlike forward iterator this iterator can be used to traverse back the container. But still can move on step forward and backward each time.

**Random Access Iterator** This iterator can move freely forward and backward and also jump to any other position given by an offset.

### 7.1.3 List

List is a sequence of elements that can be linearly ordered according to their position on the list. A list is usually represented by a comma separated sequence of element:

$$a_1, a_2, ..., a_{i-1}, a_i, a_{i+1}, ..., a_n$$

What is important in a list is the relative position of elements. All the elements are in the same level and the only relation between them is to be the next element or previous one. For example the element $a_{i-1}$ is before $a_i$ and $a_{i+1}$ is the next one.

There are several structure representing a list. Each of them store its elements in a different way and provides different properties as we will see later.

### 7.1.4 Tree

Tree is a hierarchical collection of elements referred as *nodes*. Unlike the list the elements of the tree are not in the same level and some of them considered to be the parent of some other. In each tree there is a node called *root* which is the parent of whole tree.

### 7.1.5 Homogeneous Container

A container which is able to store only one type of elements is a *homogeneous container*. For example a C array of doubles is homogeneous while can store only double variables.

```
double homogeneous_array_of_doubles[3];
```

Even an integer must be converted first to double and then it can be stored in this array. The C++ standard containers are homogeneous and can store only one type of elements.

### 7.1.6 Heterogeneous Container

A container considered to be *heterogeneous* if is able to store different types of elements. Figure 7.1 shows an heterogeneous container in memory.

Usually a heterogeneous container is implemented to accept any type of data but in some cases its more convenient to implement a container which is also heterogeneous but can only store

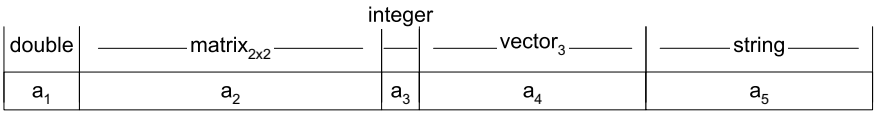



Figure 7.1: An heterogeneous container in memory storing a double, a matrix, an integer, a vector and also an string.

certain types of elements. In this work this type of containers will be referred as *quasi heterogeneous containers*.

## 7.2 Classical Data Containers

In this section some classical data containers which are usually used in finite element programs are briefly described and their advantage and disadvantages are also discussed. Further information about data containers and their implementation can be found in [55, 14, 46, 102].

### 7.2.1 Static Array

Static array is an implementation of a list which puts its elements sequentially in memory without any gap between them. Figure 7.2 shows an array in memory.

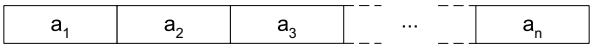



Figure 7.2: Array elements sequentially stored in memory

Static array can store certain number of element given in time of construction. For example the following array of doubles can hold up to 10 doubles:

```
double results[10];
```

This restriction makes static array unusable for variable size and growing containers.

#### Interface and Operations

The interface of an array is very simple due to its restricted functionality.

**Access** Usually, operator `[]` is used to access elements of an array. Accessing is very fast and is constant time respect to the position and also size of the array. This means that the time to access any element in an array is not dependent on its position and neither on the number of elements in container. Accessing an element is done by offsetting the base pointer by position index as shown in figure 7.3.

**Size** To provide size interface the array must store its size which implies an overhead specially in the case of small arrays. For some implementations this overhead can be eliminated as we will see later.

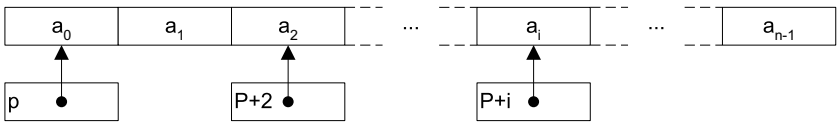



Figure 7.3: Array indexing

**Swap** In general swapping can be down element by element which causes swapping time to be linear respect to the size of the container. In some cases this can be down via pointers which make it a constant time operation because two arrays just change their pointing sequence and not all elements one by one.

As mentioned before the size of an array cannot be changed so there is no resize interface for the static array. For same reason there are no insert, append and erase interfaces while these interfaces will change the size of container.

### Advantages

- Very good use of system cache. The sequential nature of array makes it very cache efficient which can make a large increase in its performance.
- Iterating over an array is very fast. In any moment the next and previous elements are known and using cache memory makes iteration extremely fast.
- Each element is accessible very fast by its sequence number because knowing this number is enough to know its position without iterating over array.
- loop over elements of an static array can be optimized more for small arrays by unrolling the loop while the number of elements can be known in compilation time.
- No memory overhead per element. In some implementations even no memory overhead per container. This makes it a very good choice when a large amount of small containers is needed.

### Disadvantages

- There are no way to insert a new element or erase an exiting one. This makes it unsuitable for variable size sequences.
- The size of an array must be known in time of creation which makes it unusable for growing sequences of elements.

In general static is well suited for small and rigid containers. For example a 3 dimensional point can be implemented as an static array of 3 elements.

### Implementation

C (and consequently C++) provides an static array by itself. This array is the minimum implementation of an array which provides a pointer to iterate over it and a `[]` operator to access its data. C array do not provide size information which implies users to provide this additional information to any algorithms they pass a C array.

The simple C array can be improved using C++ templates as follows:

```
template<class TDataType, std::size_t TSize>
class array
{
    TDataType mElements[TSize];
  public:

    // type definitions
    typedef T              value_type;
    typedef T*             iterator;
    typedef const T*       const_iterator;
    typedef T&             reference;
    typedef const T&       const_reference;
    typedef std::size_t    size_type;
    typedef std::ptrdiff_t difference_type;

    // iterator support
    iterator begin() { return mElements; }
    const_iterator begin() const { return mElements; }
    iterator end() { return mElements+TSize; }
    const_iterator end() const { return mElements+TSize; }

    // access operator[]
    reference operator[](size_type i)
    { return mElements[i]; }
    reference operator[](size_type i) const
    { return mElements[i]; }

    static size_type size() { return TSize; }

    // element by element swap (linear complexity)
    void swap (array<T,TSize>& y)
    {
        std::swap_ranges(begin(),end(),y.begin());
    }

    // c array representation
    T* c_array() { return mElements; }
};
```

This implementation provides the size of array without any storage overhead. Also it is STL compatible and can be passed to STL algorithms which gives another added value to it. In this project the boost [3] implementation of the array is used which announced to be a part C++ standard draft.

### 7.2.2 Dynamic Array

There are many situations in which the exact size of array is not known but a maximum size can be given. This maximum size can be used as capacity of the array. So the array can be constructed with a given capacity and used to store any number of elements less than the capacity. Here an additional pointer, pointing to the tail of stored elements, is necessary to indicate the actual size of the array. Figure 7.4 shows this implementation of the array.

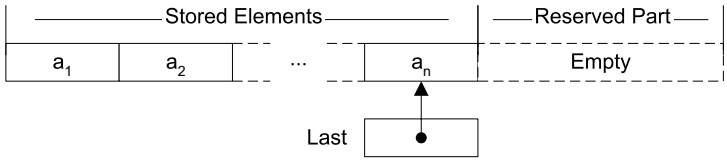



Figure 7.4: Array constructed with given capacity

### Interface and Operations

**Access** Usually, operator `[]` is used to access elements of an array. Like static array accessing is very fast and is constant in time. This means that the time to access any element in an array is not dependent on its position neither on the number of elements in the container. Accessing an element is done by offsetting the base pointer by position index starting from zero as shown in figure 7.5.

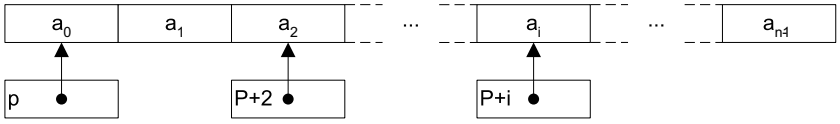



Figure 7.5: Array indexing

**Insert** Inserting an element in a position can be done by shifting all elements after that position to their next position in order to make room for a new element and then put the element in the prepared position. This operation is linear respect to the number of elements to be shifted. Figure 7.6 shows this procedure.

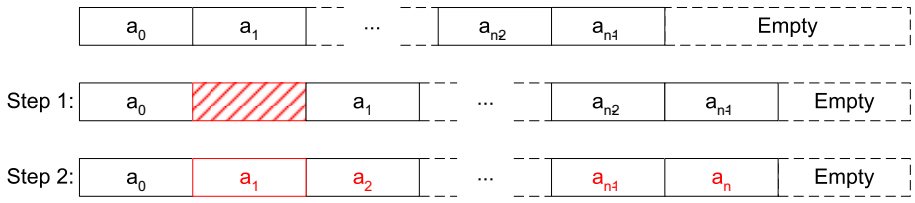



Figure 7.6: Adding an element in second position which causes all the rest of elements to move one step forward to make room for new one

Obviously this procedure is valid until the capacity of the array is more than its size. Otherwise a resize procedure is required.

**Append** Unlike inserting, appending an element to the end of array do not need any shifting and takes constant time independent of the size of array. Figure 7.7 shows this procedure. Again if the array is full append causes resizing with capacity changing which make it less efficient.

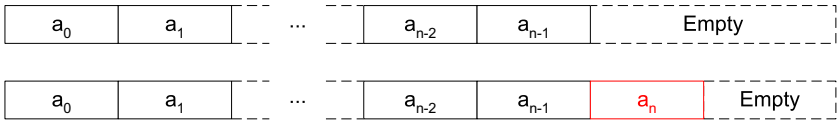



Figure 7.7: Appending an element to the end of array

**Erase** To erase an element in the middle of array again a shifting process is necessary to reconstruct the continuity of array. Erasing consist of removing the element and then shift the rest of element one step back to fill the gap. Figure 7.8 shows this procedure.

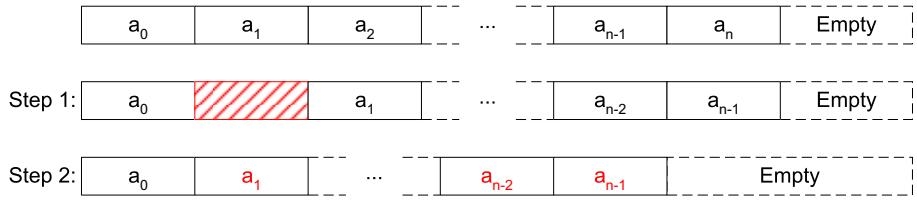



Figure 7.8: Erasing the second element of array

**Size** The size of the array can be calculated by the distance of begin and end of array or it can be stored depending on the implementation. In general it is a fast process which is constant in time respect to the size of container.

**Resize** Resizing without changing capacity is simple and efficient but changing the capacity is more complex and inefficient. Changing the capacity of an array implies reallocation of memory. If there is free memory after this array reallocation can be done without copying but if the memory is allocated then whole array must be copied to some other places with sufficient space as shown in figure 7.9. This causes the pointers to elements of array to be invalidated while they are still pointing to the previous position of the array in memory.

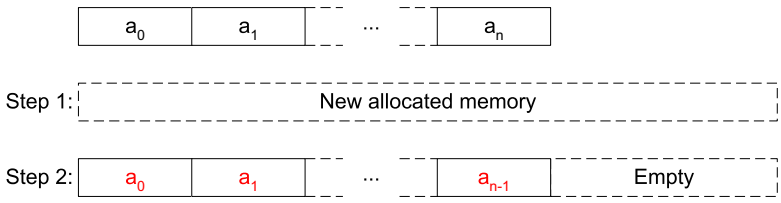



Figure 7.9: Resizing an array beyond its capacity when there is no space after array to grow

**Swap** In general swapping can be down element by element which causes swapping time to be linear respect to the size of the container. But in some cases this can be down via pointers which make it a constant time operation because two arrays just change their pointing sequence and not all elements one by one.

### Array Advantages

- Very good use of system cache. The sequential nature of array makes it very cache efficient which can make a large increase in its performance.
- Iterating over an array is very fast. In any moment the next and previous elements are known and using cache memory makes iteration extremely fast.
- Each element is accessible very fast by its sequence number because knowing this number is enough to know its position without iterating over array.
- No memory overhead per element. There is just an small overhead of storing to pointer for each container.

### Array Disadvantages

- Inserting elements in the middle of array is relatively slow due to the shifting procedure.
- Increasing the capacity of vector requires reallocating of memory which makes it slow. Reserving extra memory solves this problem with the cost of memory overhead.
- Shifting and reallocating invalidate pointers to the elements of an array, which makes element referencing a difficult task. Though using vector with sequence number instead of direct referencing can solve this problem.
- Array do not reduce the capacity automatically. For example removing elements from the array will not reduce the memory used by the array.

### Interface and Operations

**Access** Usually, the operator `[]` is used to access elements of an array. Accessing is very fast and is constant time respect to the position and also size of the array. This means that the time to access any element in an array is not dependent on its position neither to the number of elements in container. Accessing an element is done by offsetting the base pointer by position index starting from zero as shown in figure 7.10.

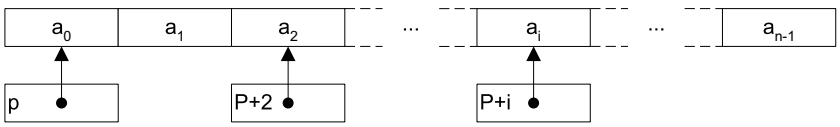



Figure 7.10: Array indexing

**Insert** Inserting an element in a position can be done by shifting all elements after that position to their next position in order to make room for new element and then put element in prepared position. This operation is linear respect to the number of elements to be shifted. Figure 7.11 shows this procedure.

Obviously this procedure is valid until capacity of array is more than the size. Otherwise a resize procedure is required.

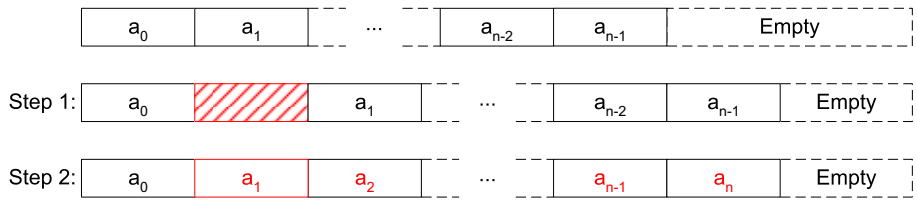



Figure 7.11: Adding an element in second position which causes all the rest of elements to move one step forward to make room for new one

**Append** Unlike inserting, appending an element to the end of array do not need any shifting and takes constant time independent of the size of array. Figure 7.12 shows this procedure. Again if the array is full, append causes resizing with capacity changing which make it less efficient.

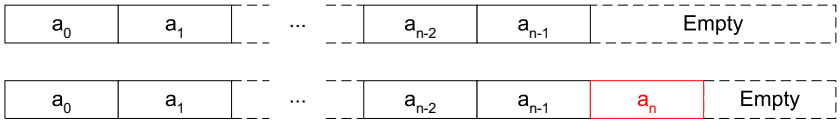



Figure 7.12: Appending an element to the end of array

**Erase** To erase an element in the middle of array again a shifting process is necessary to reconstruct the continuity of array. Erasing consist of removing the element and then shift the rest of element one step back to fill the gap. Figure 7.13 shows this procedure.

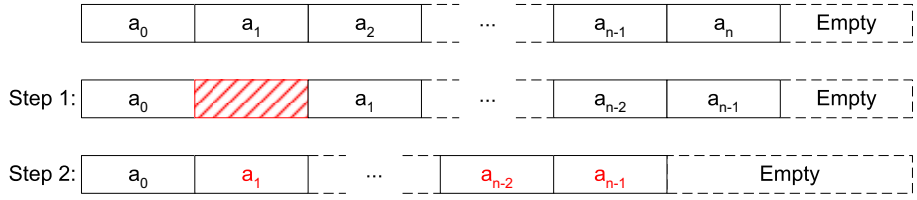



Figure 7.13: Erasing the second element of array

**Size** The size of array can be calculated by distance of begin and end of array or it can be stored depending on implementation. In general its a fast process with constant time respect to the size of container.

**Resize** Resizing without changing capacity is simple and efficient but changing the capacity is more complex and inefficient. Changing the capacity of an array implies reallocation of memory. If there is free memory after this array reallocation can be done without copying but if the memory is allocated then whole array must be copied to some other places with sufficient space as shown in figure 7.14. This causes the pointers to elements of array to be invalidated while they are still pointing to the previous position of the array in memory.

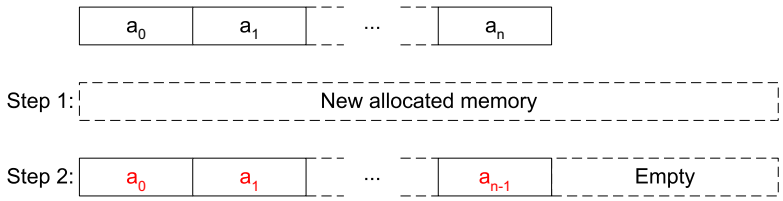



Figure 7.14: Resizing an array beyond its capacity when there is no space after array to grow

**Swap** In general swapping can be down element by element which causes swapping time to be linear respect to the size of the container. But in some cases this can be down via pointers which make it a constant time operation because two arrays just change their pointing sequence and not all elements one by one.

### Implementation

As mentioned before new elements can be appended to the tail of list very easily but inserting an element in the middle of the list requires all following elements to shift and make room for new one. Also removing elements from tail also is easy while removing from the middle is more complex due to the shifting procedure of following elements to close the gap. The shifting procedure causes inserting and removing elements in the middle of the list to be slower. It also invalidates all the pointers to the shifted element. This causes problem in accessing array elements via pointers. For example the following code for removing the negative values from given data will not work because removing each element changes the position of last element and so the end of sequence.

```
double* begin = data.begin();
double* end = data.end(); // Will be invalidated by remove!!

for(double* i = begin ; i != end ; i++)
   if(*i < 0.00)
       data.remove(i); // Invalidates the end pointer!!
```

Adding any element to an already full array requires increasing the capacity. This operation is slow and doing it for every added element over capacity, results poor performance. Reserving more to be used later is a common approach to this problem. Each time an increasing is needed an additional memory is allocated to avoid reallocation for next elements. Though this approach effectively solves the performance problem but applies a memory overhead. Larger buffer provides better performance and also larger memory overhead. All this can be avoided by creating an array with correct capacity.

`vector` class provides the array implementation in C++ standard library.

```
vector<class T, class Alloc = Allocator<T> >
```

The first template parameter `T` is the type of element to be store and second is the allocator which manages the memory used by vector. `vector` adjust its capacity automatically and also reserves an extra memory to improve the oversize inserting performance. The buffer size varies from one implementation to other but usually is 50% or 100% of vector size. This may cause unacceptable overhead for some cases and must be avoided by assigning correct capacity to the vector.

### 7.2.3 Singly Linked List

Singly linked list is a set of elements chained to each other by pointer. The main idea is giving each element a pointer to the next one. So having the first element the list can be traversed by going from each element to the next known position. Figure 7.15 shows a singly linked list in memory.

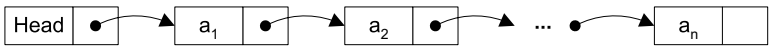



Figure 7.15: Singly linked list stored individually each element with a link to the next element

#### Interface and Operations

**Access** Accessing an element in singly linked list knowing its sequence number is not as easy as array. For accessing an element one must start from head of the list and go through the list to arrive to given position. For example to find fifth element of the list first the head must be used to find the first element, then first element have an associated pointer to the second one. Going to second one we can get the pointer to third element and from third to forth and finally to the fifth position. This searching nature makes indexing in a singly linked list a very slow procedure. Some implementations even do not provide access by position.

**Insert** Inserting a new element after given element is fast with no need to shifting. The inserting procedure consist of making new element pointing to the element the given one is already pointing and making the given element pointing to new one as its next element. Figure 7.16 shows this procedure.

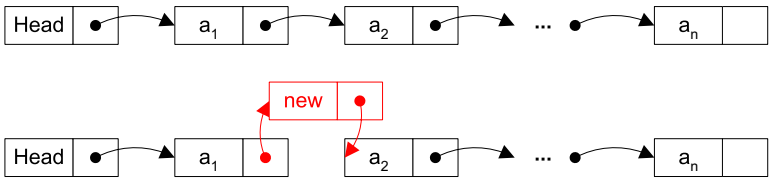



Figure 7.16: Inserting a new element after first element of the list

**Append** For a linked list append and insert has the same nature and can be done in the same way. The only difference is that there is no pointing element at the end so appending is to make the last element point to the new one as can be seen in figure 7.17.

**Erase** Like inserting a new element erasing an existing one is very simple and efficient. To erase an element after a given one is enough to make given element pointing to the element after the removing one and then delete it from memory as shown in figure 7.18. The time complexity of erasing an element is constant respect to its position and also to the size of array.

**Size** Getting the size of a linked list is not as easy as an array. In fact to find out how many element are connected together a complete traverse of list is necessary. So its convenient to accept the overhead and store the size of list for increasing its performance.

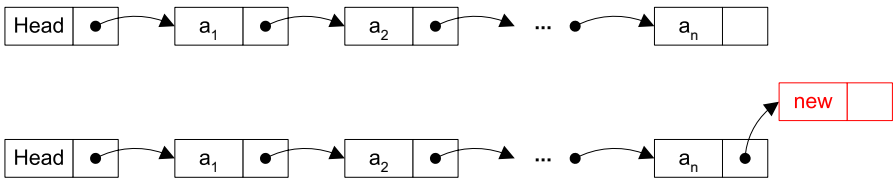



Figure 7.17: Appending a new element after last element of the list

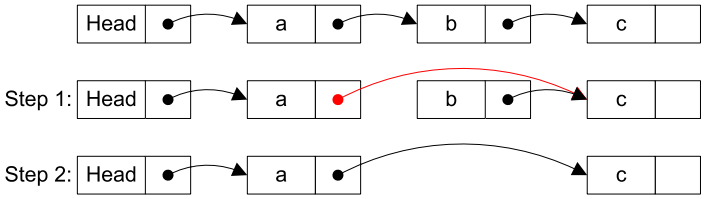



Figure 7.18: Erasing an element from list

**Resize** While a list do not have a maximum size or any restricted size, this process consist of adding new element or erase some others to achieve given size of the list.

**Swap** Swapping two lists is very simple and consists of swapping only the head pointer of the two lists. Figure 7.19 shows this procedure.

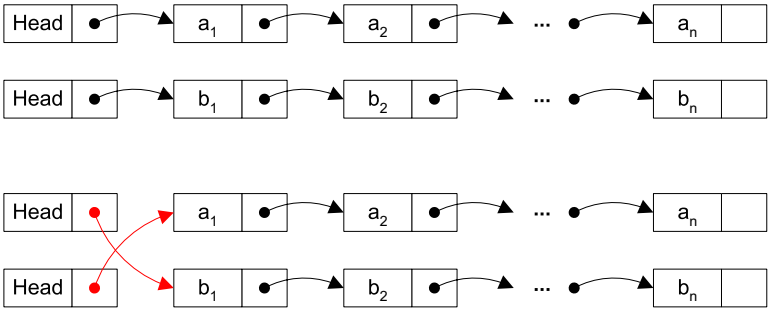



Figure 7.19: Swapping two list by changing their head pointer

### Singly Linked List Advantages

- Fast inserting a new element after a given one.
- Efficient removing of an element.
- A linked list can grow smoothly by adding new elements and without the need to resize it or creating a buffer like dynamic arrays.

- Removing elements from lists automatically reduces its size in memory which avoids the corresponding overhead or the manual controlling of its size.

**Singly Linked List Disadvantages**

- Singly linked list cannot be traversed in reverse direction because each element only knows its next neighbor but not the previous one. So this container cannot be used for algorithms which needs to traverse forward and backward the container.
- Accessing by position is a very slow task and has to be avoided. This accessing can be reduced by keeping the pointer to the necessary elements of the list for future use.
- Lists are used to be less cache efficient due to the fact that its elements can be arbitrary distributed in the memory. This makes them less attractive as fast iterating containers in practice.

A singly linked list is therefore suitable for highly variable data structure subjected to procedures with a lot of element inserting and deleting statements.

**Implementation**

Unfortunately C++ standard library do not support this container. There are some extensions to standard the library which provide `slist` [28, 8] as singly linked list variant of the standard list container.

In Kratos a modified version of a singly linked list is implemented in the `ProcessInfo` class to keep track of its history during a finite element procedure.

### 7.2.4 Doubly Linked List

As mentioned above a singly linked list has the drawback that cannot be traversed in reverse order. Doubly linked list solves this problem by adding another pointer to its elements which links them to their previous elements. In this way a doubly linked list can be traversed in both ways but causes an overhead of one pointer per each element. Figure 7.20 shows a doubly link list in memory.

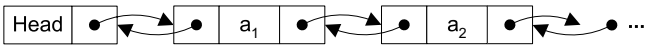



Figure 7.20: A doubly linked list in memory

**Interface and Operations**

**Access** Accessing an element in a doubly linked list knowing its sequence number requires the same procedure as for the singly linked list one. Again for accessing an element one must start from the head of the list and go through the list to arrive to the given position. For example to find the fifth element of the list first the head must be used to find the first element, then the first element has an associated pointer to the second one. Going to the second one we can get the pointer to the third element and from the third to the forth and finally to the fifth position. This searching nature makes indexing in a doubly linked list a very slow procedure. Some implementations even do not provide access by position.

**Insert** Inserting a new element after given element is fast with no need to shifting. It consists of linking the element in the given position to the new one and also the new one to the next element of the given position. Figure 7.21 shows this procedure.

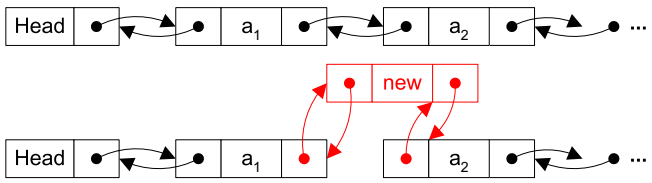



Figure 7.21: Inserting a new element after first element of the list

**Append** For a linked list append and insert has the same nature and can be done in the same way. The only difference is that there is no pointing element at the end so appending is just making the last element point to the new one as shown in figure 7.22.

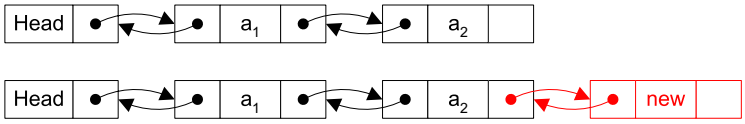



Figure 7.22: Appending a new element after last element of the list

**Erase** Like inserting a new element erasing an existing one is very simple and efficient. To erase an element after a given one is enough to make the given element pointing to the element after the removing one and then delete it from memory as shown in figure 7.23. The time complexity of erasing an element is constant respect to its position and also to the size of the array.

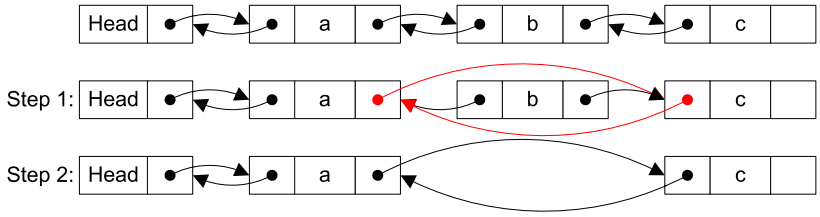



Figure 7.23: Erasing an element from the list

**Size** Getting the size of a linked list is not as easy as an array. In fact to find out how many elements are connected together a complete traverse of list is necessary. So its convenient to accept the overhead and store the size of the list for increasing its performance.

**Resize** While a list does not have a maximum size or any restricted size, this process consist of adding new elements or erase some others to achieve the given size of the list.

**Swap** Swapping two lists is very simple and consist of swapping only the head pointer of the two lists. Figure 7.24 shows this procedure.

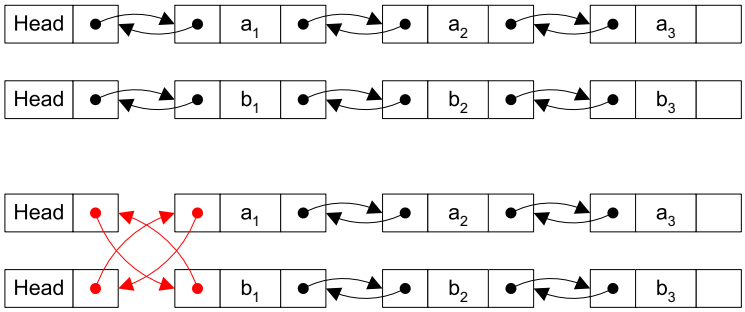



Figure 7.24: Swapping two list by changing their head pointer

### Doubly Linked List Advantages

- Fast inserting a new element after or before a given one.
- Efficient removing of an element.
- A linked list can grow smoothly by adding new elements and without the need to resize it or creating buffer like for dynamic arrays.
- Removing elements from lists automatically reduces its size in memory which avoid the corresponding overhead or manual controlling of its size.
- Doubly linked list can be traversed in both way which makes it usable for a wider range of algorithms than the singly linked list.

### Doubly Linked List Disadvantages

- Accessing by position is a very slow task and has to be avoided. This accessing can be reduced by keeping the pointer to the necessary elements of the list for future use.
- Lists are used to be less cache efficient due to the fact that its elements can be distributed arbitrary in the memory. This makes it less attractive as a fast iterating container in practice.
- The overhead of two pointers per element can be noticeable when elements are small. For example for a list of doubles these pointers can duplicate the memory in 32 bit compiling and even worse when compiling in 64 bit.

This container is suitable for storing data with high amount of insert and erase in the middle.

### Implementation

C++ standard library provides a list class to represent the doubly linked list. This class is parameterized by template to accept different types.

```
list<class T, class Alloc = Allocator<T> >
```

## 7.2.5 Binary Tree

A binary tree is a tree in which every node has either no children, only a *left child*, only a *right child*, or both left child and right child. In the other words each node in a binary tree has two descending branches, left and right, which can be empty or not. Figure 7.25 shows a binary tree.

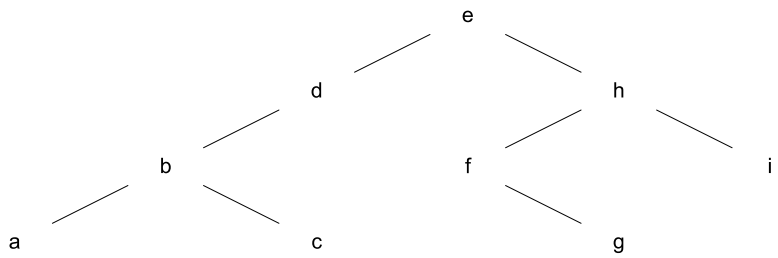



Figure 7.25: A binary tree

It is important to mention that the order of left and right child is part of the tree specification and swapping the left child and the right child of a node results in another binary tree deferent from the original one. For example trees in figure 7.26 are not equivalent due to the change of child positions.

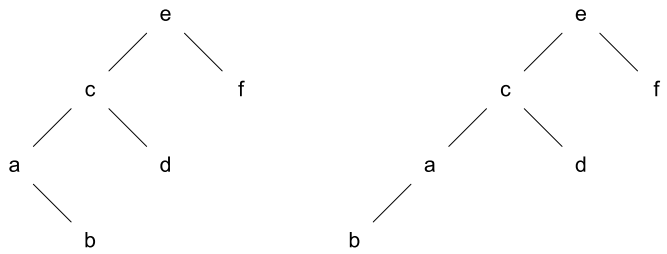



Figure 7.26: Two different binary trees because the position of node b is changed from being left child to be right child

Binary tree can be used for ordering data respect to a binary comparison operator which returns true or false. For example a less than operator $<$ can be used to order a set of numbers in a binary tree as can be seen in figure 7.27. In this tree each node is greater than all values in its left subtree and less or equal to all values in its right subtree. This ordering can be used later for searching a value in the tree.

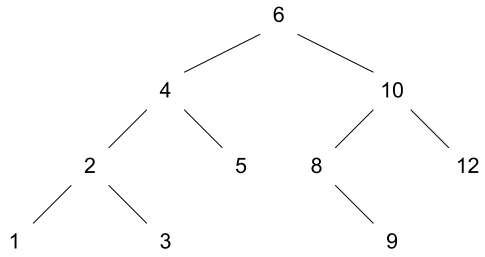



Figure 7.27: A binary tree containing a set of numbers ordered respect to the less than operator $<$

### Interface and Operations

**Access** In a binary tree its not usual to give access by position and access is usually done by finding a key in the tree.

**Insert** Inserting a new element in a tree consists of finding its place and then add it there. In some implementations a given position is also accepted but as a hint to find faster the correct places. This searching is necessary to guarantee the ordering of the tree. The procedure of finding the right place to insert starts from the root and compares the new element key with the root. If the result is true goes to the left child and if it is false goes to the right child and repeat the comparison. This procedure is repeated until an empty child is founded which is the position to insert the element. Having this position inserting is only pointing the parent to this new element like inserting in linked lists. Figure 7.28 shows this process for inserting the value 7 in the ordered tree of figure 7.27.

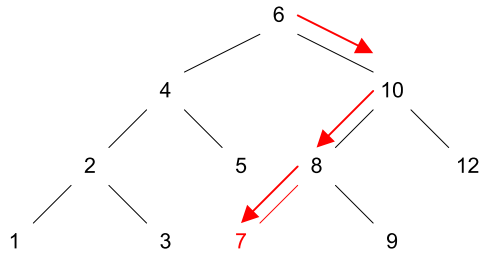



Figure 7.28: Inserting a new value in the tree.

**Find** To find a value in a tree one can began from root and compare the value with it. If comparison returns true means the value must be in left subtree and if it is false, means that the value must be in right subtree. Having the corresponding subtree the procedure can be repeated to see which subtree should has the value. This process will arrive either to our desirable variable or to an empty branch which indicates that the value does not exist. The ordering and hierarchial nature of the tree makes the finding algorithm very efficient. The number of comparisons required for this search is guaranteed to be $O(\ln N)$ which is much lower than the $O(N)$ comparison required for searching in an unordered normal array, specially for big number of entities $N$. Figure 7.29 shows this procedure in finding value 9 in the ordered tree

of figure 7.28.

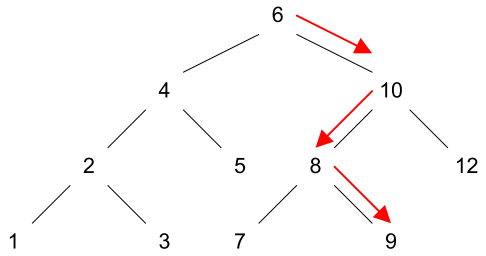



Figure 7.29: Finding a value in a binary tree

**Erase** Erasing an element of a binary tree requires some replacements of elements in order to keep the ordering of the tree. As mentioned earlier the ordering ensures that comparison of a node value with all the values in its left subtree is true and that with all its right subtree is false. In the case of using the less than operator $<$ for comparison it can be said that all values in left subtree are smaller than node value and all value in right subtree are greater than it. It can be seen that to keep this ordering we must take the minimum value, or the left most value of the right subtree and put it in the place of the removed node. For example removing 6 from the binary tree of figure 7.29 consists of first, finding the left most node of the right subtree which is 7 and put it in the place of 6. Then filling the empty place of the moved minimum node with its right child which is empty in this case. The resulting binary tree can be seen in figure 7.30.

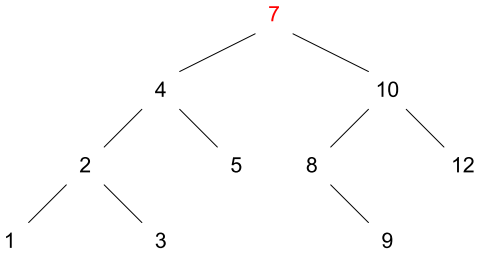



Figure 7.30: The binary tree after erasing an element

**Iterating** Unlike arrays and linked lists, iterating over an ordered binary tree is not so trivial and consist of ascending and descending the tree structure to keep track of the elements sequence. The idea is to take the start node and go into its right subtree, In each subtree the iterator finds the left most element and tries to go from left to right by going up and down in the tree layers. Figure 7.31 shows the iterator path from element 1 to element 12.

**Size** Like linked lists getting the size of a binary tree is not easy and needs a traverse over the tree. It is therefore convenient to accept the overhead and store the size of the tree in a member variable for increasing its performance.

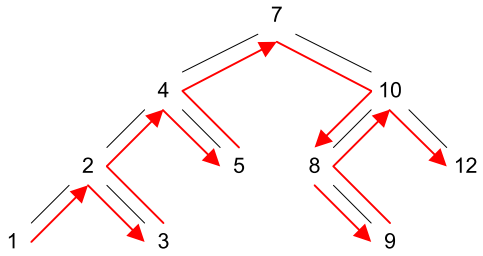



Figure 7.31: Iterator path from elements 1 to 12

**Swap** Swapping two binary trees is very simple and consist of swapping only the root pointer of two trees.

### Binary Tree Advantages

- Keeping elements in order all the time is an important feature specially for working with variable set of elements.
- Fast finding procedure. The hierarchical structure and order nature of binary trees enable us to make a very efficient binary search also over very large number of elements.
- Relatively fast inserting a new element while keeping the ordering of elements.
- Relatively efficient in removing elements again without altering the ordering of elements.

### Binary Tree Disadvantages

- There is no access by position in general. They are specialized for accessing by key and not by position.
- Binary lists are not cache efficient because their elements are not stored sequentially in memory.
- Iterating over a binary tree is a complex and consequently not efficient process.
- The overhead of three pointers per element can be noticeable when elements are small.

### Implementation

Fortunately the C++ standard library provides various classes representing binary trees in different ways. The first one is `set` which is an ordered binary tree of its template argument type. It also takes an optional comparison operator which make it more generic. `set` uses this comparison to understand the order of an element.

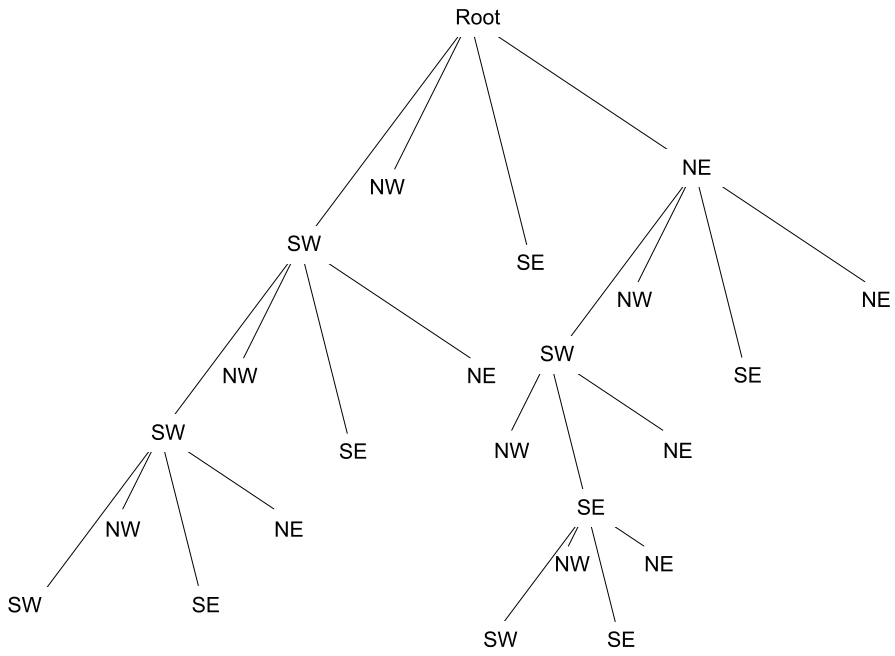



Figure 7.32: In a general quadtree each node has up to four children

### 7.2.6 Quadtree

Quadtree is a specific type of tree in which every node has zero to four children. Figure 7.32 shows a general quadtree.

This structure is very useful for organizing data in two dimensional spaces because it can be used to handle the partitioning information which defines the cells in space in hierarchial form. Figure 7.33 shows a domain divided by quadtree. For this reason the names of children in a quadtree node come from their relative positions in a two dimensional map: NW, NE, SW and SE.

As mentioned above, a binary tree can be used for organizing data in one dimension. Quadtree does the same in a two dimensional space. It can be used with two comparison operators to order points respecting their coordinates. For example using two less than operators $<$ results in an order quadtree in which the coordinate $x$ of each node is greater than all coordinates $x$ in its NW and SW subtrees and its $y$ coordinate is greater than all coordinates $y$ in SW and SE subtrees, as can be seen in figure 7.34.

#### Interface and Operations

**Access** In a quadtree accessing to an element is done by finding a pair keys, for example two coordinates of a point, in the tree.

**Insert** Inserting a new element in a quadtree consists of finding its place and then add it there. Sometimes a given position is accepted as a hint to find faster the correct position. The procedure is similar to binary tree but using two comparison to find the correct branch. It starts from root and goes through branches until a branch leads to an empty child. Having this position inserting is only pointing the parent to this new element.

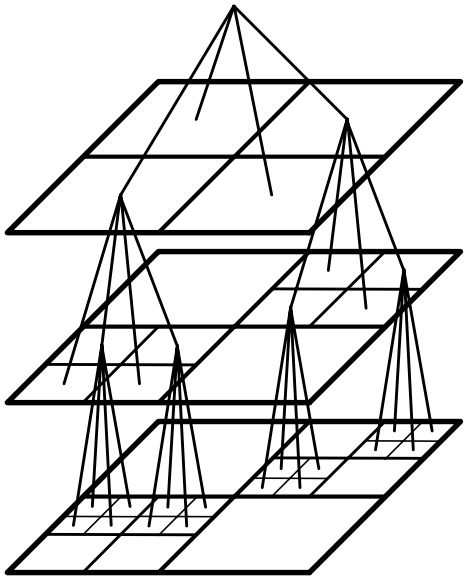


Figure 7.33: Quadtree can be used to partition a domain into layers of sub-domains.

**Find** Finding procedure starts from root and compares the keys, for example the point's coordinates, if matches to it the result is found and if not goes to the branch which is corresponding to the comparison result. The procedure is repeated for the node in the branch and keeps going until finding the entity or an empty leaf which indicates that the entity does not exist.

Different types of the quadtree and detail description of each type can be found in [91, 92]

### 7.2.7 Octree

Octree implements the same concept of quadtree but in a three dimensional space. In octree each node has 8 branches which may lead to a child or not. Considering the octree like the three dimensional extension of quadtree, all operations done by quadtree in two dimensions and using two comparison operators, now can be done by octree in three dimensions using three comparison operators.

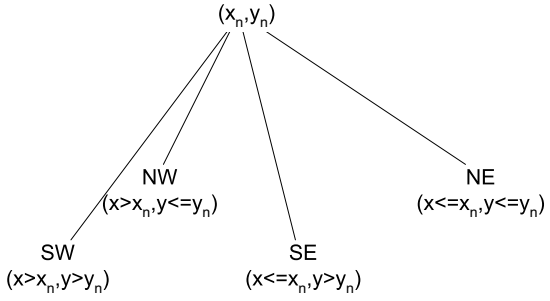



Figure 7.34: Ordering two dimensional points in quadtree.

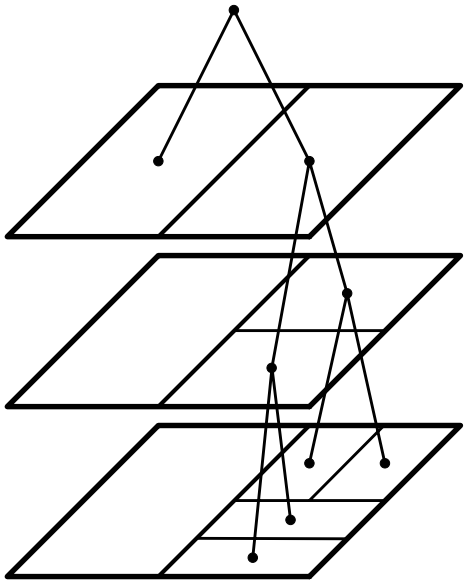


Figure 7.35: Partitioning a two dimensional domain using k-d tree.

### 7.2.8 k-d Tree

K-d tree is a generalized approach to deal with k dimensional spaces using only two-way branching at each node. Figure 7.35 shows a k-d tree used to handle the partitioning of a two dimensional domain.

### 7.2.9 Bins

A simple but effective data structure for storing and finding objects in two and three dimensional spaces is `Bins`. It divides the domain into a regular $n_x \times n_y \times n_z$ sub-domains and holds an array of buckets storing its elements. Figure 7.36 shows a two dimensional domain divided by the bins and figure 7.37 shows the structure of this bins.

This structure provides a fast spatial searching when entities are more or less uniformly distributed over the domain. The good performance for well distributed entities and simplicity make bins one of the popular data structure in different finite element applications [59].

### 7.2.10 Containers Performance Comparison

As mentioned before, each container respecting to its internal structure provides different performances in its operations and memory consuming. This difference has been described as advantage and disadvantage of each container. However this respective performance is also depends on the size of the container and can change radically in terms of the size. In this section a brief comparison between containers on certain operations is provided. This comparison gives some experimental results, indicating the behavior of different containers in practice.

It is important to mention that in these tests no manual optimization is performed and only the automatic optimization of compilers is set to its highest value.

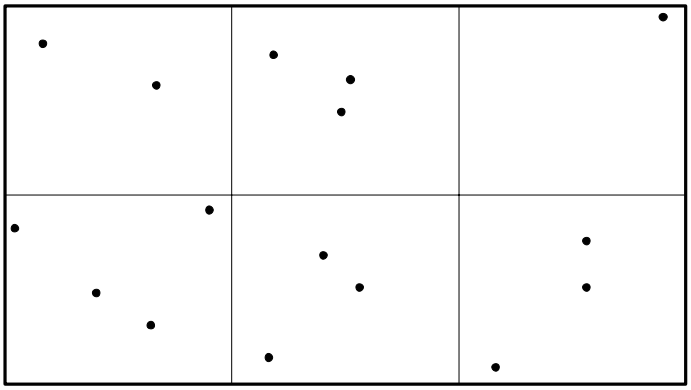


Figure 7.36: Partitioning a two dimensional domain using bins.

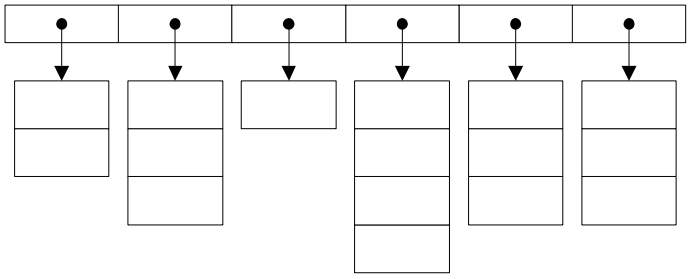


Figure 7.37: Bins structure.

### Memory usage of containers

In this comparison different containers are initialized to different sizes and the memory used by each of them is measured. Figure 7.38 shows the results of this comparison.

It can be seen that `vector` uses less memory than other contatiners. The reason is the memory used to store the pointers in elements.

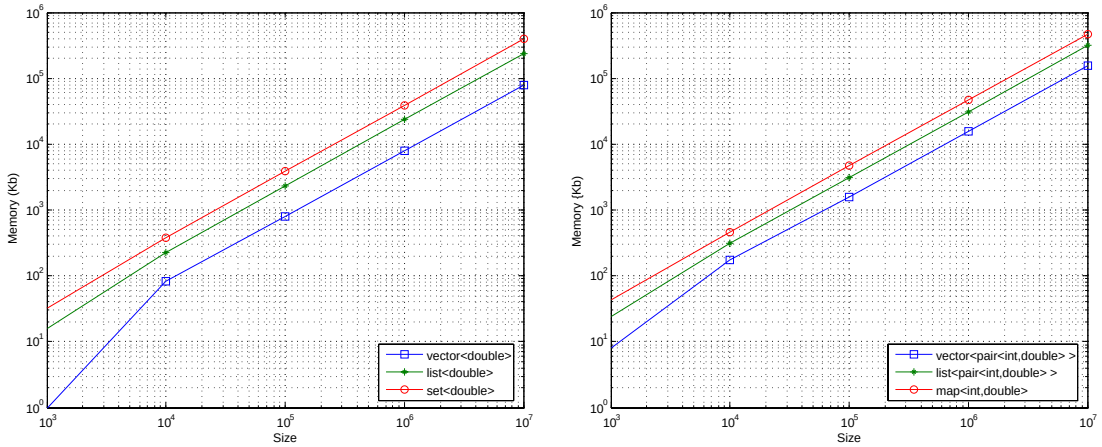
Another benchmark is done to see the memory used by different containers respecting to their sizes. For this reason an array of size $n = 100000$ is created with different containers and the memory used by each of them is shown in figure 7.39.

### Construction and Destruction

In this test the construction and destruction time of different containers are compared. Figure 7.40 shows the construction time comparison for different containers.

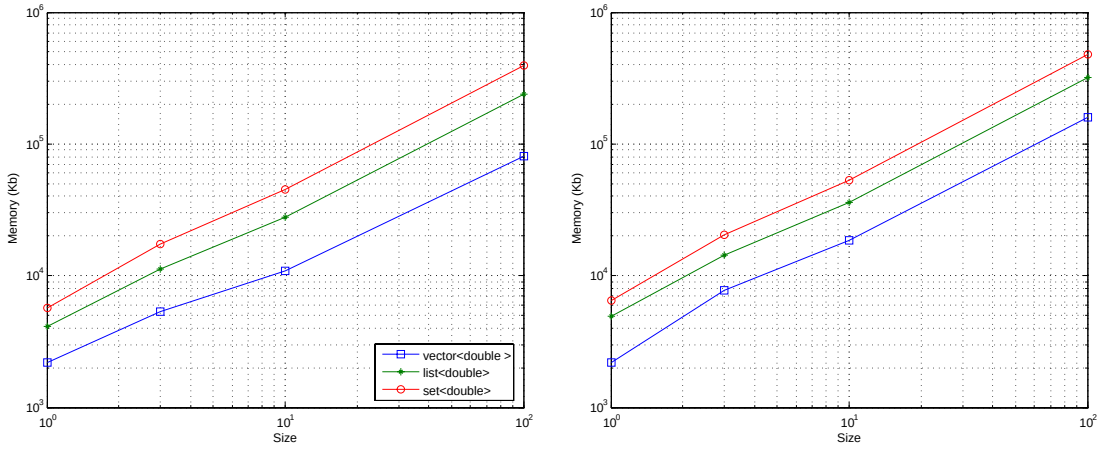
It can be seen that the constructing time for `vector` is far less that for other containers. The reason is its simpler internal structure than the `list` or `set`. Figure 7.41 shows the destruction time for different containers.

Again `vector` is far faster than other containers. This makes `vector` a good choice for situations



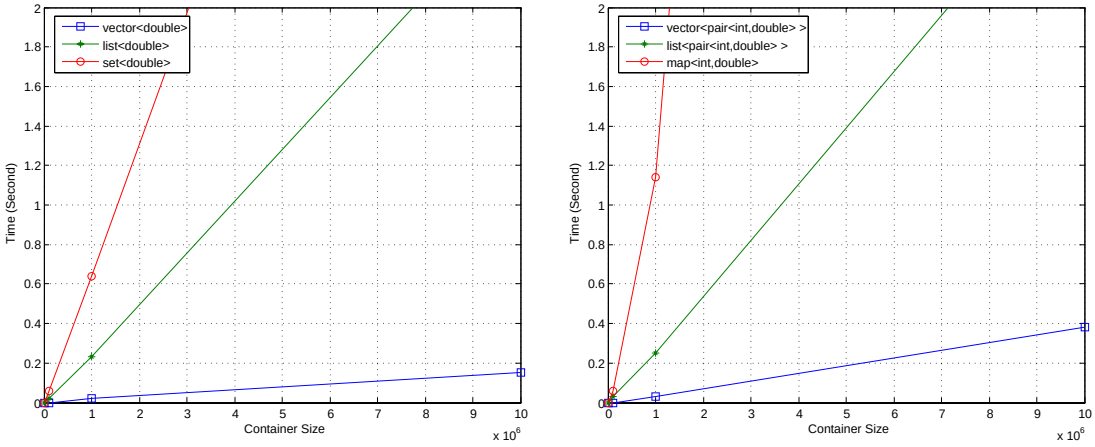

(a) (b)

Figure 7.38: Memory use comparison a) Comparing memory use of `vector<double>`, `list<double>` and `set<double>` b) Comparing memory use of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`



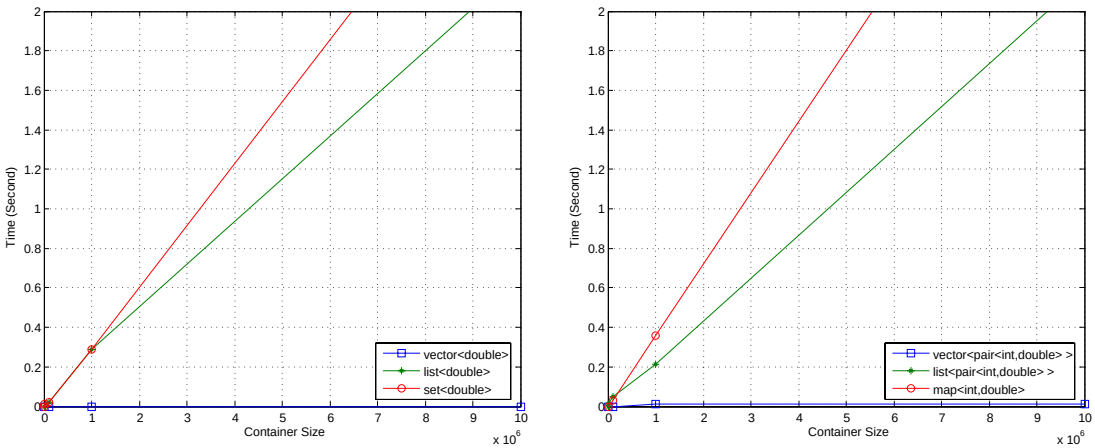

(a) (b)

Figure 7.39: Memory use comparison between arrays with size $n = 100000$ of different containers. a) Comparing memory use of `vector<double>`, `list<double>` and `set<double>` b) Comparing memory use of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`




(a) (b)

Figure 7.40: Construction time for different containers. a) Comparing construction time for `vector<double>`, `list<double>` and `set<double>` b) Comparing construction time for `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

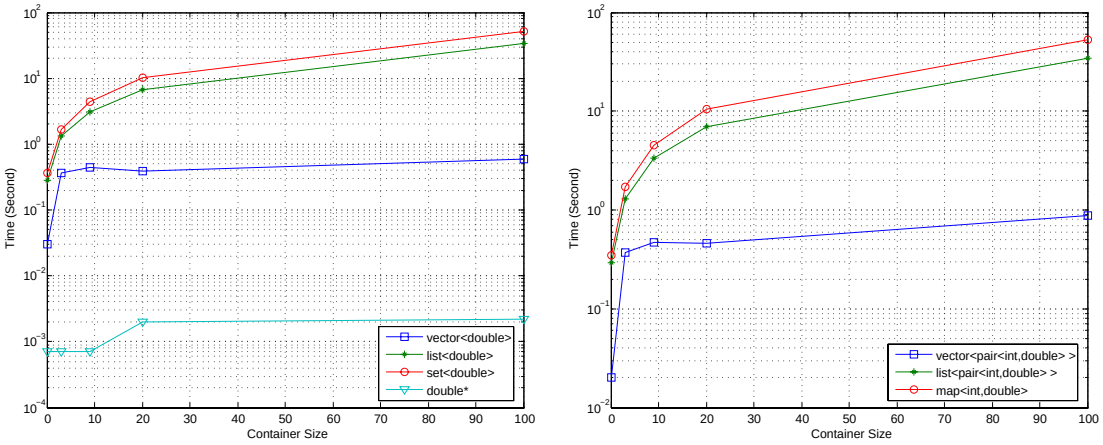



(a) (b)

Figure 7.41: Destruction time for different containers. a) Comparing construction time for `vector<double>`, `list<double>` and `set<double>` b) Comparing construction time for `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

that a container has to be created and deleted immediately.

Another test is to compare the construction and destruction time of containers as local variables allocated in stack memory. While it is usual to create containers as local variables in procedures it is important to see their time overhead for construction and destruction them each time the procedure is called. Figure 7.42 shows this comparison.




(a) (b)

Figure 7.42: Construction and destruction time of containers defined as local variables and allocated in stack. a) Comparing construction/destruction time for `double*`, `vector<double>`, `list<double>` and `set<double>` b) Comparing construction/destruction time for `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

This time the `C` static array shows far better performance even respecting to `vector`. The reason is the overhead of allocating dynamic memory for the `vector` while the `C` static array is allocated completely in stack. Consequently, implementing the small local containers in procedures as static arrays can significantly increase the performance of the code.

### Iterating

Many finite element algorithms are used to iterating over containers. For this reason good performance in iterating is an important factor in selecting a container. This benchmark consists of iteration over all elements of sample containers with different sizes. The time is measured for $10^9$ steps of iterations and the results are shown in figure 7.43. Each step consists of an access to the iterator content to be sure that the optimizer will not eliminate the loop.

This benchmark shows the bad performance of containers with tree structure like `set` and `map`. Surprisingly `vector` shows to be more robust in optimizing the iteration time than the `C` array and it seems that c array needs manual optimization to get its best performance.

### Inserting

The first benchmark shows the results of pushing back elements to different containers. The test consist of pushing back $10^4$ elements to each container several times and measuring the average

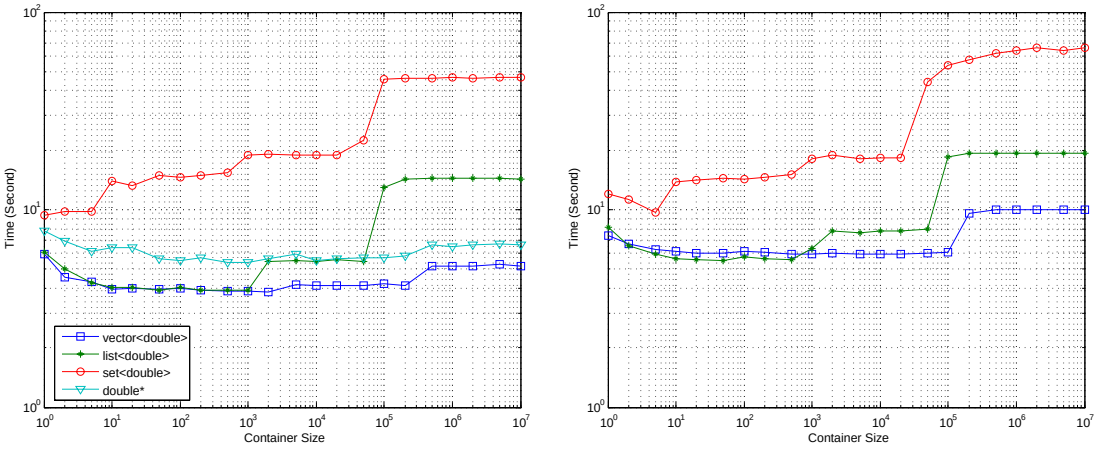



Figure 7.43: Iterating time for $10^9$ steps of iterations with different containers. a) Comparing performance of `double*`, `vector<double>`, `list<double>` and `set<double>` b) Comparing performance of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

time of this operation. Figure 7.44 shows the results of this benchmark.

The second benchmark shows the push front operations for vector and its linear complexity with respect to the size of the container. Other containers has the same performance as the push back operation and are significantly faster as expected. Figure 7.45 shows the result of $10^4$ pushfront to vectors with different sizes.

### Copying

Another important operation is the copying of containers. This benchmark shows the result time for copying different containers with different sizes. The test consists of calling the copy constructor of container several hundred times and calculating the elapsed time for 100 times calling the copy constructor. Figure 7.46 shows the results of this benchmark.

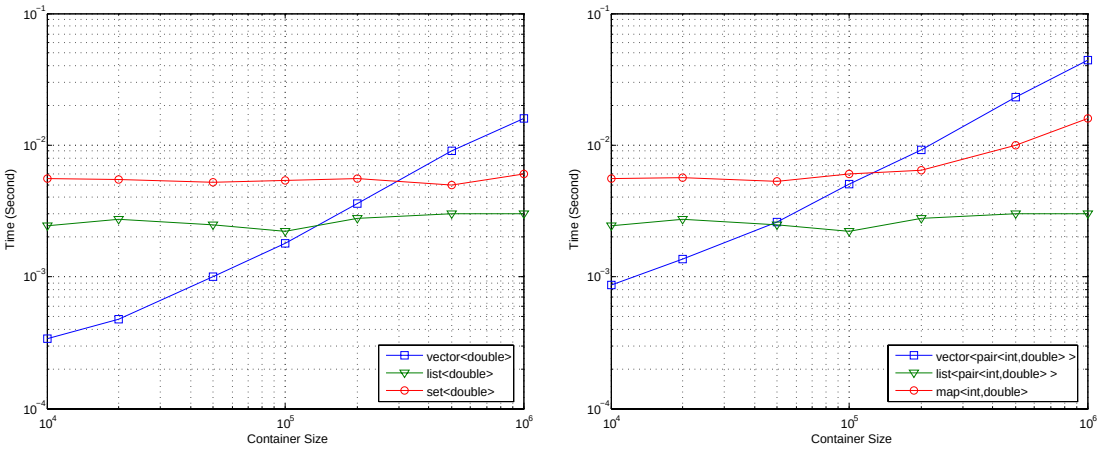
It can be seen that again `vector` has better performance to others due to its simple structure which leads to less overhead in time of copying than the others.

### Find

Finding an element in a container is another typical operation to be compared. The benchmark compares the performance of the brute-force search over unsorted containers with binary search over sorted containers. The first test is to find a value in container, using brute-force over an unsorted `vector` and an unsorted `list`, a binary search over a sorted `vector` and the tree search of `set`. The second test is an integer key finding using brute-force over an unsorted `vector` and an unsorted `list` and the tree search of `map`. Figure 7.47 shows the results of this comparison.

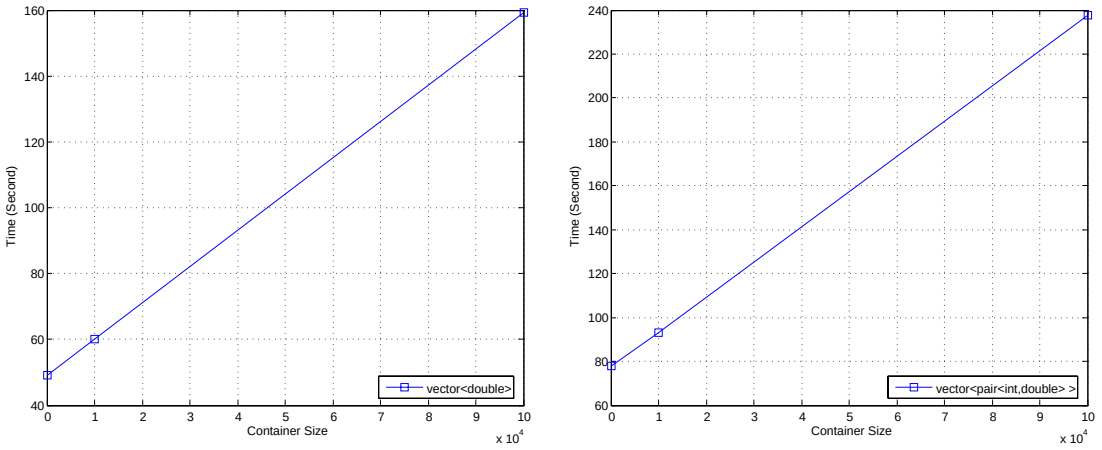
This benchmark shows how inefficient the brute-force algorithm becomes when the size of container increases. Also it can be seen that using a binary search over a sorted vector can lead to the same performance of searching in a tree like set.

Figure 7.48 shows the same results but focusing only on small containers. Here it can be seen



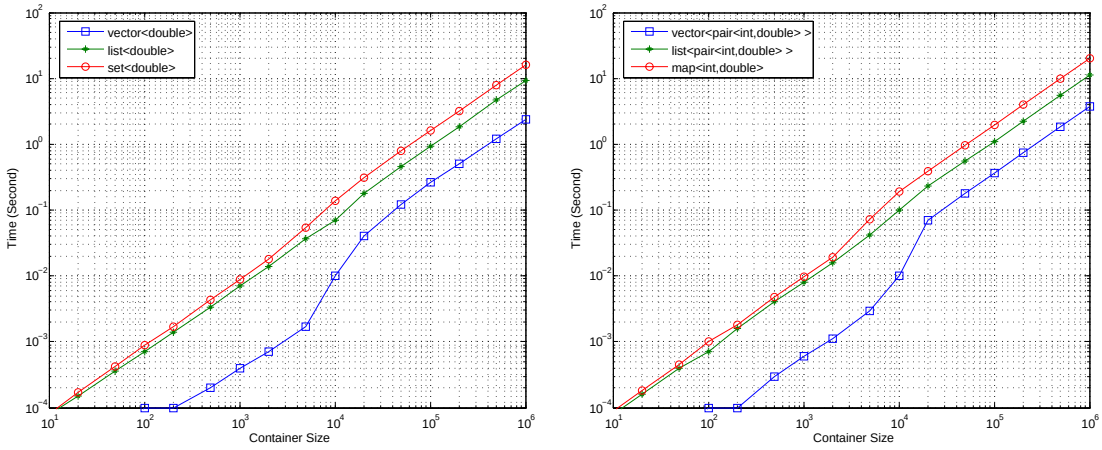

(a) (b)

Figure 7.44: Time comparison for $10^4$ elements pushback to different containers. a) Comparing performance of `vector<double>`, `list<double>` and `set<double>` b) Comparing performance of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`



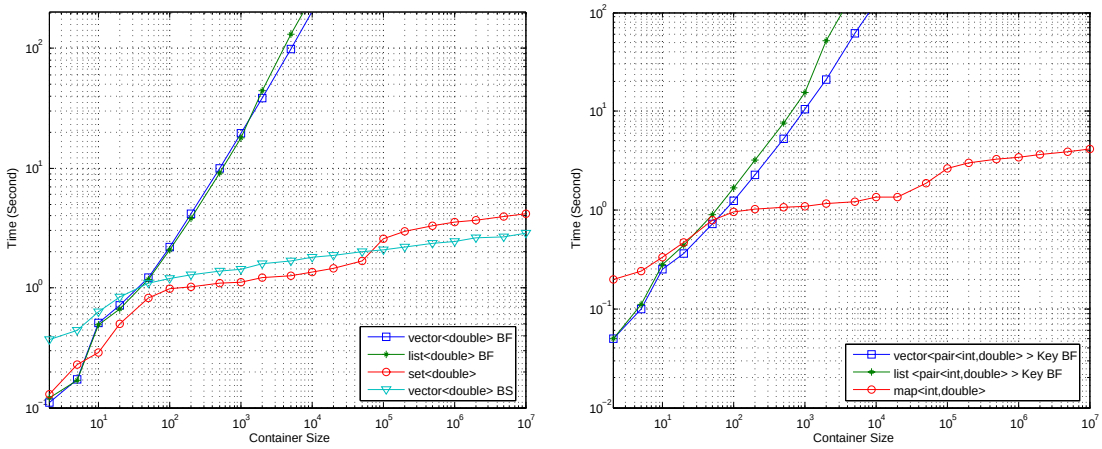

(a) (b)

Figure 7.45: Time comparison for $10^4$ pushfronts to vectors with different sizes. a) `vector<double>` b) `vector<pair<int,double> >`



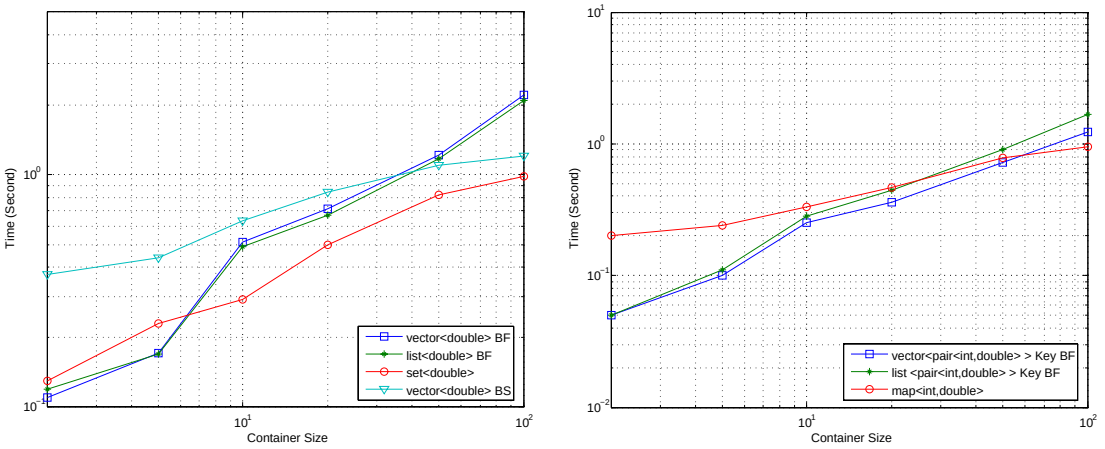

(a) (b)

Figure 7.46: Time comparison for 100 calls to copy constructor. a) Comparing performance of `vector<double>`, `list<double>` and `set<double>` b) Comparing performance of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`




(a) (b)

Figure 7.47: Time comparison for different searching algorithms over sorted and unsorted containers a) Comparing performance of `vector<double>` with brute-force, `list<double>` with brute-force, `set<double>` binary tree search and sorted `vector<double>` with binary search. b) Comparing performance of `vector<pair<int,double> >` with brute-force key finding, `list<pair<int, double> >` with brute-force key finding and `map<int,double>` binary tree search.




(a) (b)

Figure 7.48: Time comparison for different searching algorithms over sorted and unsorted small containers a) Comparing performance of `vector<double>` with brute-force, `list<double>` with brute-force, `set<double>` binary tree search and sorted `vector<double>` with binary search. b) Comparing performance of `vector<pair<int,double> >` with brute-force key finding, `list<pair<int, double> >` with brute-force key finding and `map<int,double>` binary tree search.

that using a brute-force algorithm for small containers can lead to better performance than other algorithms. This feature will be used later in finding values in small data containers.

## 7.3 Designing New Containers

It can be seen that standard C++ containers are homogeneous due to the static typing of C++ language. This means that using one of this containers to store doubles cannot be reused to store vectors and matrices without decomposing them to double values. This restriction makes developers to separate their containers for different types of data or to implement heterogeneous containers. In this part some new containers capable to store different types of data are introduced.

In previous chapter a Variable Base Interface (VBI) was introduced and its advantages were mentioned. In this part VBI is used to unify the interfaces of different containers and also to provide them with a generic interface capable to storing any new variable without rewriting them.

### 7.3.1 Combining Containers

Finite element developers usually work with integers, reals, vectors and matrices as their data. In some other fields like electro magnetic formulations complex numbers are also used. So making a new container capable to hold just these data types can cover a large part of finite element programming needs. A very fast an easy way to implement a quasi heterogeneous container holding above data types is to take different containers and put them together as a new container. Figure 7.49 shows an example of this container.

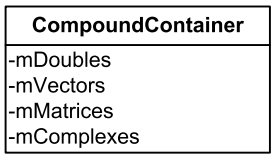



Figure 7.49: Combining containers for holding doubles, vectors, matrices and complex numbers.

### Interface and Operations

**Access** For accessing to an element this container first has to see what is the type of this element and then access in corresponding container. Usually this access consists of a find process for a given key. However access by position can be done by giving a pseudo order to the containers.

**Insert** Inserting a new element like accessing an existing one requires switching over element type. The process is simply finding the corresponding container and insert the new element in it.

**Find** Mainly depends on the sub-containers inside the compound one. Again this container only dispatches the request to the corresponding sub-container using type of data and the procedure of finding must be done by the underlying container.

**Erase** Simply removes an element from the container holding this type of element. Efficiency of this procedure also depends on the type of container holding this data.

**Iterating** The same iterating mechanism of sub-container can be reused to iterate over a certain type of data.

### Combining Containers Advantages

- Fast and easy implementation. Standard containers can be reused here and make the implementation task very easy.
- Very rigid and errorless structure. There is no chance in inserting wrong data type or getting data which is not of the expected type. Everything relies on C++ static type checking without dangerous type casting and raw pointer manipulation.
- Keeping separated different types of data lets more specialization for each case. For example in time of copying the containers for built in types can be copied directly in the memory and so on.
- Less searching time because the number of data in each container is less than the total number of data in container. In other words dividing container to sub-containers reduces the time for searching. Though the searching time is highly dependent on the type of sub-containers.

### Combining Containers Disadvantages

- Extra memory overhead is needed for supporting any new type. As mentioned in previous section, each container has a memory overhead. So using more containers to keep the same number of element increases the overhead per element. This factor may cause problems

for very small number of data per container. For example a container for holding doubles, vectors, matrices and complex numbers has a fix overhead of 8 pointers or more. So using this container to hold a double and a vector of 3 doubles causes at least 100% memory overhead. Supporting any new data type still increases more this overhead which may cause this container unusable for some problems.

- Adding new types needs modifying the container, though this modification is very small. A good implementation can minimize this modification but cannot make it automatic to accept any new type of data. This makes it unacceptable for a library with unspecified using field.

### Implementation

Combining containers is an easy task. Any classic container described in previous section can be use here to store one type of data. There is no restriction to have same containers for all types but usually this is the good choice while the nature of data and algorithms are not dependent on the type of element. However in some cases a type specific optimization can be implemented using different algorithms and containers for the data types.

As mentioned before a container provides interface for accessing, inserting, erasing and also iterating. Even though these are the common interfaces for standard containers, their type dependency given them a different nature. A simple way to overcome this problem is to implement separate methods for each type. Figure 7.50 shows an example of the accessing methods for the container of figure 7.49 using separate methods for each type.

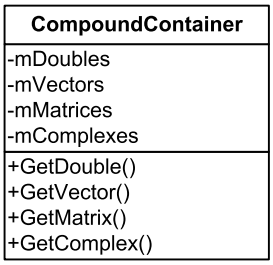



Figure 7.50: Implementing separate access interface for each type in a compound container

It can be seen that this interface design causes a big overhead in implementation cost. This can be avoided using a template access method with a dummy argument indicating the type of data. Here is an example of a template access method:

```
template<class TDataType>
  TDataType& GetValue(TDataType const& Dummy,
                      ...more data information)
  {
    return CorespoundingContainer.Get(...);
  }
```

Using this interface implies introducing a dummy variable just to help the container which is not elegant. In previous chapter the VBI was introduced and its generic way to deal with different data type was also discussed. Now it is time to use this concept to create a generic and extendible interface for our containers. VBI provides a uniform template model for accessing an element by

variable. This model uses the variable type parameter to distinguish the type of element without the need of dummy argument. In this manner the access method can be written in this new form:

```
template<TVariableType>
  typename TVariableType::Type& GetValue(TVariableType const&)
  {
    return CorespoundingContainer.Get(...);
  }
```

A variable not only gives information about the type of element but also gives information about how to find it by providing its unique index and also its name. This information can be used to find the element without passing extra arguments to the access method. So this access method can be written in the following way:

```
template<TVariableType>
  typename TVariableType::Type&
  GetValue(TVariableType const& ThisVariable)
  {
    return CorespoundingContainer.Get(ThisVariable.Key());
  }
```

This encapsulation of element information in a variable simplifies the interface and its usage by the users. Now a user can get a value only by giving the variable represent it as follows:

```
// Without VBI user must know exactly the type of
// data and also its index in container. Here is
// an example of accessing acceleration with index
// 3 in container
acceleration = mData.GetValue(array_1d<double, 3>(), 3);

// Using VBI, user just need to specify the previously
// defined variable
acceleration = mData.GetValue(ACCELERATION);
```

VBI also prevents users from making trivial errors by giving wrong type or information. For example:

```
// Error! acceleration type is array_1d<double, 3>
// but given type is std::vector
acceleration = mData.GetValue(std::vector(3),3);

// Error! displacement is in position 0 and not 3
displacement = mData.GetValue(array_1d<double, 3>(), 3);
```

Keeping this form a basic interface for our heterogeneous container can be implemented in the form below:

```
// Accessing to a value in container
template<TVariableType>
  typename TVariableType::Type&
  GetValue(TVariableType const& ThisVariable);

// Readonly access to the container
template<TVariableType>
  typename TVariableType::Type const&
```

```
  GetValue(TVariableType const& ThisVariable) const;

// Setting a value in container
template<TVariableType>
  void
  SetValue(TVariableType const& ThisVariable
           typename TVariableType::Type& Value);

// To see if the variable exist in container
template<TVariableType>
  bool
  Has(TVariableType const& ThisVariable);
```

This interface can be used in this form without problems but can be improved even more. In finite element applications there are many situations where there is a need for accessing a component of an array or matrix. For example to assign a value to a degree of freedom representing $Y$ component of displacement, a direct access to the component value in data structure is easier than extracting the whole displacement vector and assign to it manually, as shown in the following code:

```
void UpdataDofValueInContainer(Dof const& rThisDof, double Value)
{
    if(Dof.Variable() == DISPLACEMENT_X)
        mData(DISPLACEMENT)[0] = Value;
    else if(Dof.Variable() == DISPLACEMENT_Y)
        mData(DISPLACEMENT)[1] = Value;
    else if(Dof.Variable() == DISPLACEMENT_Z)
        mData(DISPLACEMENT)[2] = Value;
}
```

Having direct access to components makes above example as easy as follows:

```
void UpdataDofValueInContainer(Dof const& rThisDof, double Value)
{
    mData(Dof.Variable()) = Value;
}
```

It can be seen that the first form with manual accessing to components is not even general and works only for displacement components while the second form can be used for any defined component without problem. As a consequence we will implement the direct component access for our container.

Interface must be modified to distinguish a variable and a component of variable. In VBI this is an easy task while two different classes, `Variable` and `VariableComponent`, represent them. So interface can distinguish a variable from a component just by type of given argument. Overloading each method to accept either a `Variable` or a `VariableComponent` separates the implementation for this two concepts. It is very important to mention here that this separation is in compilation time and does not causes any cost due to type checking or other type recognizing mechanism. The first part of interface is related to get a variable of given type and process it. The type of variable is specified by a template parameter, to keep the generality of the interface.

```
// Accessing to a variable in container
template<TDataType>
  TDataType&
  GetValue(Variable<TDataType> const& );
```

```
// Readonly access to a variable in container
template<TDataType>
  TDataType const&
  GetValue(Variable<TDataType> const& ) const;

// Setting a variable in container
template<TDataType>
  void
  SetValue(Variable<TDataType> const&,
           TDataType& );

// To see if the variable exist in container
template<TDataType>
  bool
  Has(Variable<TDataType> const& );
```

The second part of interface consists of the same methods overloaded to accept a `VariableComponent` instead of a normal `Variable`.

```
// Accessing to component of a variable in container
template<TAdaptorType>
  typename TAdaptorType::Type&
  GetValue(VariableComponent<TAdaptorType> const& )

// Readonly access to component of a variable in container
template<TAdaptorType>
  typename TAdaptorType::Type const&
  GetValue(VariableComponent<TAdaptorType> const& ) const

// Setting a component of a variable in container
template<TAdaptorType>
  void
  SetValue(VariableComponent<TAdaptorType> const& ,
           typename TAdaptorType::Type& );

// Ask if container has this component which usually is
// equivalent to see if the variable holding this component
// is exist.
template<TDataType>
  bool
  Has(Variable<TDataType> const& );
```

Next we will implement the container. One way is implementing the container with sub-containers as its attributes. Figure 7.51 shows an example of this container with three sub-containers.

It can be seen that for each supported type an overloaded version of the interface methods is needed to call manually the corresponding container. For example the `GetValue` method must be overloaded for each type to call the `GetValue` method of sub-container which contains this type of data as follows:

```
// Accessing to a double in container
double GetValue(Variable<double> const& rThisVariable)
{
```

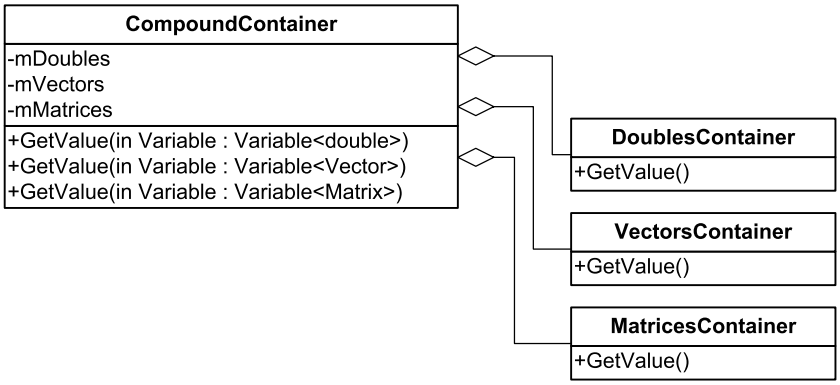



Figure 7.51: Container with three sub-containers as its attributes.

```
  return mDoubles[rThisVariable.Key()];
}

// Accessing to a Vector in container
Vector& GetValue(Variable<Vector> const& rThisVariable)
{
  return mVectors[rThisVariable.Key()];
}

// Accessing to a Matrix in container
Matrix& GetValue(Variable<Matrix> const& rThisVariable)
{
  return mMatrices[rThisVariable.Key()];
}
```

This manual switching must be done for all other methods working with different types of data like `GetValue const`, `SetValue`, `Has` and overloaded operators which makes this implementation strategy difficult to maintain.

Another approach is using multiple hierarchy to group different containers in a combine one. Figure 7.52 shows this approach for implementing the container of the previous example.

The big difference between this approach and the previous one is the container switching mechanism. In this way there is no need to manually overload each method for any acceptable type. Now a template method simply does the work. The mechanism is simple, each method is implemented as a template of the element type which accepts the corresponding variable and calls the related container automatically by calling the proper base class interface. Here is an example of this implementation:

```
class CombinedContainer : public BaseContainer<double>,
                                 BaseContainer<Vector>,
                                 BaseContainer<Matrix>
{
  public:

  /// Default constructor.
  CombinedContainer(){}
```

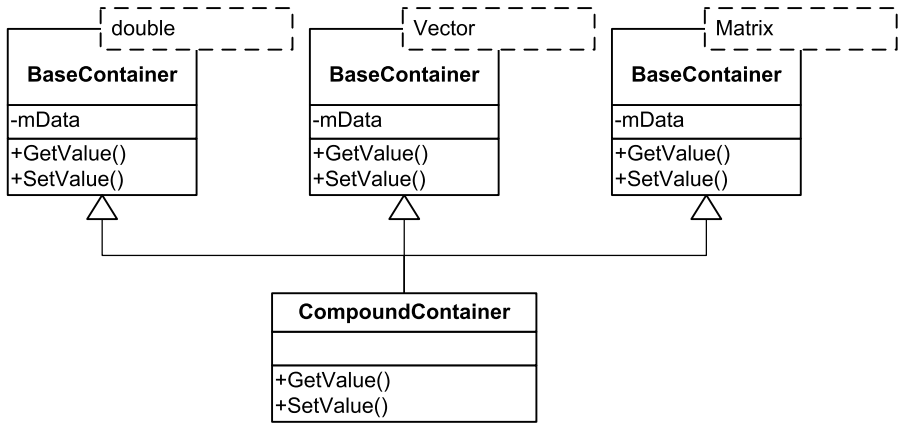



Figure 7.52: Combining different containers using multiple hierarchy.

```
  /// Copy constructor.
  CombinedContainer(const CombinedContainer& rOther) :
    BaseContainer<double>(rOther),
    BaseContainer<Vector<double> >(rOther),
    BaseContainer<Matrix<double> >(rOther)
  {
  }

  template<class TDataType>
  TDataType&
  GetValue(const Variable<TDataType>& rThisVariable)
  {
    return BaseContainer<TDataType>::GetValue(rThisVariable);
  }
};
```

Now supporting any new variables only needs another parent class to be added and modification in some other methods like copy constructor to incorporate the new base class.

Direct access to the components of variables can be done easily using tools provided by VBI. As mention in the previous chapter each component knows about its parent variable and also knows how to extract itself from it. So to extract a component is only necessary to access the parent variable value and give it to the component to extract itself form it. Here is an example of the component access method:

```
template<class TAdaptorType>
  typename TAdaptorType::Type&
  GetValue(const VariableComponent<TAdaptorType>& rThisVariable)
    {
      typedef typename TAdaptorType::SourceType source_type;

      return rThisVariable.GetValue(
                BaseContainer<source_type>::GetValue(
                        rThisVariable.GetSourceVariable()));
    }
```

It can be seen that all switching and accessing algorithms are implemented via templates, which make them very efficient. In time of compilation all variables and components types are known so the compiler can get the conversion algorithm provided by component and inline it inside the access code to eliminate the function call overhead. Finally optimizer will reduce all this code to the direct access method provided by adaptor which is equivalent to the hand written code.

Finding stored variables by their keys require searching in container. `map` provides a searching mechanism by given keys and can be used to find any variable key without any implementation cost. The `BaseContainer` can be implemented using `map` as follows:

```
template <class TDataType>
 class BaseContainer
 {
 public:
  typedef map<VariableData::KeyType, TDataType> ContainerType;

  TDataType&
  GetValue(const Variable<TDataType>& rThisVariable)
  {
    return mData[rThisVariable];
  }

 private:
  ContainerType mData;
 }
```

So a very first version of a container with VBI can be made by putting all above components together. This implementation was used in the first version of the Kratos code to develop a reliable container with minimum cost.

This container can be improved by changing the basic container from `map` to a `vector`. There are two major advantages in using `vector` instead of `map`:

- Less memory is needed to store a `vector` of indexed data than a `map`. In fact this container will be used to store nodal or elemental data, so any reduction in memory will deeply affect the overall memory used by the application. In the previous section the big difference in use of memory has been shown. For small containers `map` needs about 2 to 5 times more memory than `vector`. This overhead can be eliminated completely using vector.
- The `vector` searching time is also faster than `map` using even brute-force when the size of container is very small. Again respect to the fact that the amount of data to be stored for each `Node` or `Element` is small, so changing to vector can also make searching process faster.

The implementation is relatively easy. Each data will be stored in a `pair` combined with its key. So to find any data one must iterate over vector and compare the key with given one. A simple container called `VectorMap` can help us to encapsulate all these operations and reuse previous code by changing the standard `map` with it.

### 7.3.2 Data Value Container

A quasi heterogeneous container can be used successfully to hold data with small variety in their types. But to hold more different data types a heterogeneous container is more useful. Data value container is a heterogeneous container with a variable base interface designed to hold the value for any type of variable.

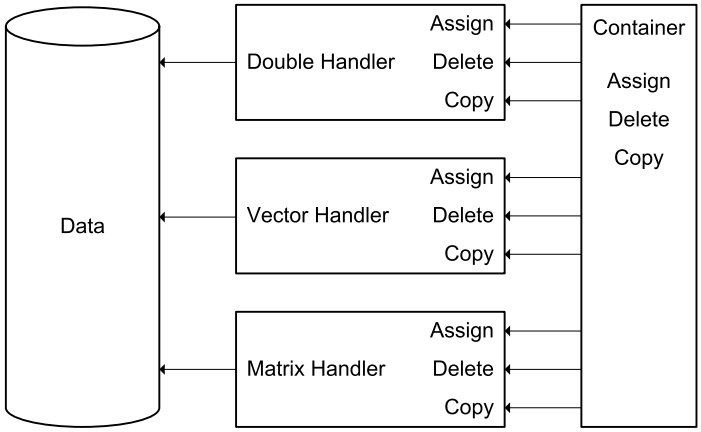



Figure 7.53: Heterogenous container uses different handlers to access data.

Usually a container needs to do some basic operations over its data. Creating, copying and deleting are examples of this operations. Unfortunately this operations may vary from one type to other. For example removing a double can be done by just removing it form container but removing a vector may consists of first freeing its memory and then removing it from container. So a heterogenous container needs a mechanism to handle each different type with its corresponding process. For example copy a double by copying its value and a vector by calling its copy constructor.

A common way to deal with this problem is to encapsulate all necessary operation into a handler object and associate it to its corresponding data. In this way the container only uses this handlers by their unique interface to do different tasks without any problem. Figure 7.53 shows this relation.

In our approach the variables are used as the handlers to help the container in its data operations. The data value container uses the `Variable` class not only to understand the type of data but also to operate over it via its raw pointer methods. Figure 7.54 shows the relationship of container and the `Variable` class.

### Interface and Operations

**Access** Container first uses the key of given variable and finds the location of its value in memory. Then return this position as a reference with the type of variable. It is important that the type recognition can be done in compilation time in order to eliminate its overhead in runtime.

**Insert** Inserting a new element consist of allocating memory and copying correctly the object. Finding the correct position and allocating strategy depends highly on the internal implementation of data. By knowing the type of data, copying can be done easily by calling its copy constructor.

**Find** Mainly depends on the internal structure of the container and consists of searching the key of variable in the container. Unlike the compound container, the finding process does not need the type switching and the type of data is important only in time of down casting of returned value.

**Erase** Removing data consists of two parts. First calling the destructor of the object using the

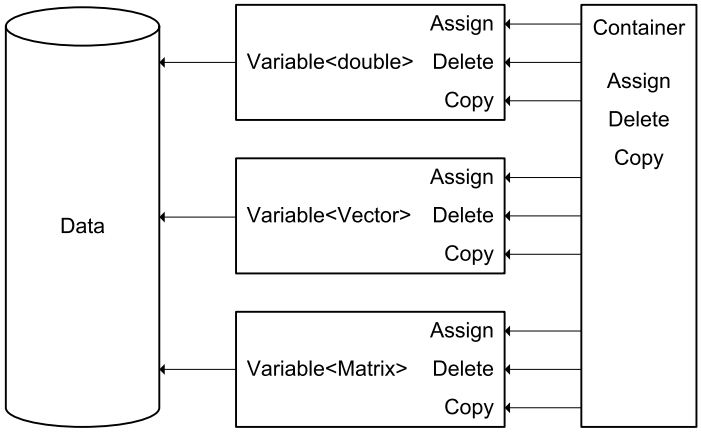



Figure 7.54: Data value container uses the `Variable` class to process its data.

type specification of the variable and then freeing the memory holding the value from internal data. The first part is an easy task as the type of variable is known at compilation time. The efficiency of the second part depends on the container's internal structure of data.

**Copy** Copying of this container is not a trivial task, as copying the memory block, or shallow copying, is not sufficient for copying some type of data. For example objects containing a pointer may need a deep copy of the pointed data and not the pointer itself. So the container goes element by element and uses the corresponding variable to copy the data. This procedure makes the process to be slower than for a homogeneous container.

**Clear** Clearing consists of first calling the destructor of each object and then freeing the memory. Container uses the variable to call the correct destructor of data and correctly remove it from the memory. This procedure is necessary because simply freeing the memory will not call the destructor of an object by itself. This causes problems specially when objects have some memory allocated internally and removing them without calling to destructor results in memory leaks in the system. Again this process is slower than for a homogeneous clearing due to the function calling overhead for each element.

### Data Value Container Advantages

- Extendibility to store any type of data without any implementation cost. Adding new types to data value containers is an automated task without changing the container or even reconfigure it. One can store virtually any type of data, from simple data like an integer to a complex one like a dynamic array of pointers to neighbor elements.
- Usually extensive use of void pointers and down casting make heterogeneous container open to type crashing. Using the variable base interface protects users from unwanted type conversion and guarantees the type-safety of this container.

### Data Value Container Disadvantages

- Heterogeneous containers are typically slower than homogeneous ones at least in some of their operations. The type recognition make them slower than for a homogeneous container.

### Implementation

The first step to implement this container is designing the structure of data in memory. One approach is to group each data with a reference to its variable and put them in a dynamic array as shown in figure 7.55.

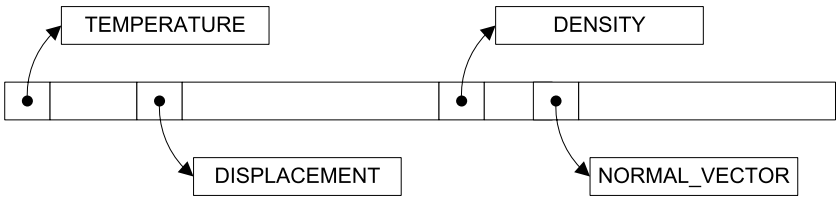



Figure 7.55: A data value container with continuous memory.

In this approach iterating over elements is somehow like a link list. For each element the variable knows the size of it and therefore the offset necessary for going to the next element. Having a pointer to each variable and not copying it is necessary to eliminate the unnecessary overhead of duplicated variables.

Another approach is to allocate each data separately and keep the pointer to its location in the container. Figure 7.56 shows a container with this structure in memory.

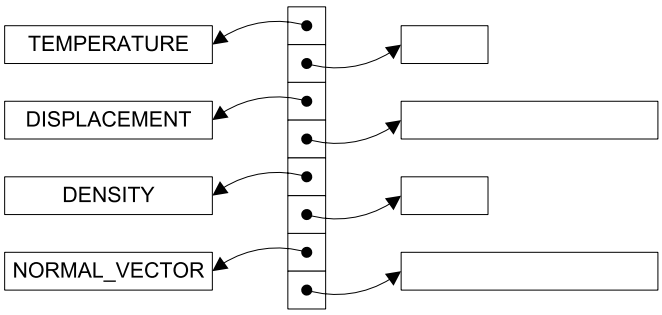



Figure 7.56: A data value container with discontinuous memory.

The first approach is typically more efficient in use of cache because accessing an element does not need a memory jump by a pointer. But adding new data to it may invalidate all references to its elements, as mentioned before for dynamic arrays. The second approach lets user to get a reference of its data once and use it several time without worrying about its validness. In this work the second approach is used because of its advantage in reducing the repeated accesses to the container which can increase significantly its overall performance in practice.

Using this memory structure the implementation is relatively easy. First a pair object is used to group the variable reference with a void pointer to its data:

```
// Grouping Variable reference with a pointer to its data
typedef std::pair<const VariableData*, void*> ValueType;
```

Now the internal data container can be implemented easily by putting these pairs in a vector:

```
// Type of the container used for variables
typedef std::vector<ValueType> ContainerType;
```

For inserting a new data in this container, first a new pair must be created which holds a reference to its variable and a pointer to the memory location holding the copy of the value. Adding this pair to the end of the vector finishes the inserting process:

```
mData.push_back(ValueType(&rThisVariable,new TDataType(rValue)));
```

Access methods use `Variable` or `VariableComponent` as data information as described for VBI. Each access consist of a find process for given variable key and then convert the data to the given variable type.

```
template<class TDataType>
  const TDataType&
  GetValue(const Variable<TDataType>& rThisVariable) const
{
  typename ContainerType::const_iterator i;

  i = std::find_if(mData.begin(),
                   mData.end(),
                   IndexCheck(rThisVariable.Key()))

  if (i != mData.end())
      return *static_cast<const TDataType*>(i->second);
}
```

The access method can be also configured to return zero if the given data does not exist yet in the container as follows:

```
template<class TDataType>
  const TDataType&
  GetValue(const Variable<TDataType>& rThisVariable) const
{
  typename ContainerType::const_iterator i;

  i = std::find_if(mData.begin(),
                   mData.end(),
                   IndexCheck(rThisVariable.Key()))

  if (i != mData.end())
      return *static_cast<const TDataType*>(i->second);

  return rThisVariable.Zero();
}
```

Its important to mention here that using VBI not only increases the readability of the code but also protects users from unwanted type crashing. To see the difference of these two approaches let us make an example of an access method without using VBI:

```
void*  GetValue(KeyType Key) const
{
  typename ContainerType::const_iterator i;
```

```
  i = std::find_if(mData.begin(), mData.end(), IndexCheck(Key))

  if (i != mData.end())
      return i->second;

  return &(rThisVariable.Zero());
}
```

Also considering the IO part of the application reads some data with different types from input and store them in the container as follows:

```
// Defining keys
  int acceleration_key = 0;
  int elasticity_key = 1;
  int conductivity_key = 2;

// Reading data
  array_1d<double> acceleration;
  matrix<double> conductivity;
  symmetric_matrix<double> elasticity;

  input >> acceleration >> elasticity >> conductivity;

// And store them in data value container
  data.SetValue(acceleration_key, acceleration);
  data.SetValue(conductivity_key, conductivity);
  data.SetValue(elasticity_key, elasticity);
```

In some other part of the code this variables are necessary and user will retrieve them from the container without specifying their types or with different types:

```
// The following innocent code simply will not compile!
// Error: There is no * operator which takes a double
//        and a void as its arguments
  Vector v = delta_time *  *data.GetValue(acceleration_key) + v0;

// The following erroneous code compiles without
// problem but crashes in runtime due to the type
// crashing of converting conductivity matrix to a
// double representing the conductivity coefficient!
  double k = *(double*)data.GetValue(conductivity_key);

// Again the following code will compile fine but crashes
// mysteriously in run time! Because the elasticity
// was stored as a symmetric matrix
  Matrix* d = (Matrix*)data.GetValue(elasticity_key);
```

The first statement calculates the velocity and store it as `Vector v`. This statement seems to be errorless, however simply will not compile because compiler has not any information about the type of acceleration.

The second statement is worse because it compiles without any problem but will not work as expected. So the application gives wrong results due to this type crashing and user has to debug it in order to find this simple error.

The third and more erroneous statement of taking a `symmetric_matrix` pointer as a pointer to

Matrix also will be compiled and even worse than second statement, may also work mysteriously or crash unfaithfully depending on the order of internal data in Matrix.

Now let see how using VBI protects user from simple errors by compilation time type checking:

```
// The following code works as it must while the return
// type of GetValue method is a reference to acceleration
// array. So the multiplication can be done correctly.
  Vector v = delta_time * data.GetValue(ACCELERATION) + v0;

// The following code will not compile due to the type
// mismatch.
  Matrix& d = data.GetValue(ELASTICITY);

// Again the following code will not compile due to the
// type mismatch.
  double k = data.GetValue(CONDUCTIVITY);
```

The first statement compiles without problem and also works as expected. A reference to the acceleration array is passed to the expression and the velocity vector will be calculated correctly.

Unlike the previous approach, the second statement will not compile because the compiler cannot convert a reference of symmetric_matrix type to a reference to Matrix type. This error in compiling protects users from unwanted type crashing. Also there is a possibility for users to copy correctly the elasticity symmetric matrix into a normal matrix for some subsequent operations.

Finally the third statement causes another compiling error and protects the user from erroneous type conversion.

Copying the container as mentioned before cannot be done by just copying the memory as this shallow copy results in the uncorrect copy of some objects, specially for ones with pointers to their individual data. For example let us consider a dynamic vector with the following implementation:

```
class Vector
{
    int mSize;
    double* mData;
public:
    // copy constructor
    Vector(Vector& Other)
    {
        mData = new double[Other.mSize];
        memcpy(mData,Other.mData,mSize);
    }

    // access
    double operator[](int i)
    {
        return mData[i];
    }
}
```

Shallow copying of this vector will result in the mData pointer taking the address in mData of the source vector and pointing to the source data as can be seen in figure 7.57. So any change in the elements of vector vc will change the elements of the source vector v!

But copying the same vector using its copy constructor will duplicate the allocated memory and safely uses the memcpy to copy the contents of the source vector to the copy one, as shown in

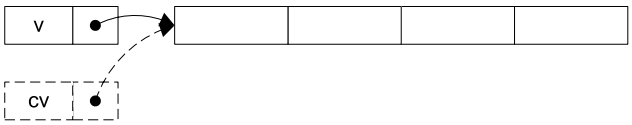



Figure 7.57: Shallow copying a pointer results in shared data for source and copied vectors.

figure 7.58.

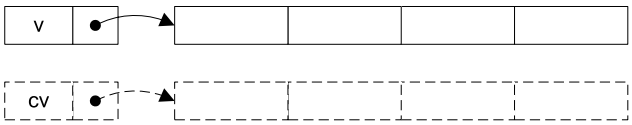



Figure 7.58: Deep copying results in an individual copy of source vector.

Data value container uses variables to call the copy constructor for each element at copying time in order to avoid errors produced by shallow copying. Here is the implementation for the copy constructor.

```
/// Copy constructor.
DataValueContainer(DataValueContainer const& rOther)
{
    for(ConstantIteratorType i = rOther.mData.begin() ;
        i != rOther.mData.end() ; ++i)
        mData.push_back(ValueType(i->first, i->first->Clone(i->second)));
}
```

Unfortunately the `Clone` method of the `Variable` class must be virtual and its function call overhead makes this operation slower than normal copying.

Destructing a data value container also need to be done carefully because freeing its memory can results memory leak for objects with internally allocated memory or result in unfinished jobs for some other objects. To avoid all these problems it is necessary to call the destructor of objects before removing them from memory. Data value container uses Delete method of Variable to call the destructor of each object in order to remove them correctly.

```
void Clear()
{
    for(ContainerType::iterator i = mData.begin() ;
        i != mData.end() ; i++)
        i->first->Delete(i->second);

      mData.clear();
}
```

This operation also consists of a function call in its loop which reduces its efficiency.

### 7.3.3 Variables List Container

In finite element programs it is common to store same set of data for all `Node`s of one domain. For example in a fluid domain each `Node` has to store velocity and pressure. Also each type of `Element` has to store an specific set of historical data at each integration point. The previous heterogeneous container can be used to store these data due to its flexibility to store any type of data. However the searching procedure in order to access data in this container makes it inefficient. To solve this problem another container is designed which stores only a specific set of data but with an efficient access mechanism.

The main idea is to use an *indirection* mechanism to access the elements of the container. A shared variable list gives the position of each variable in the containers sharing it. The mechanism is very simple. There is an array which stores the local offset for each variable in the container and assigning the value $-1$ for the rest of the variables. Offsets are stored in the position of variables key using a zero base indexing. In other words if the key of a variable is $k$, then its offset is stored as the $k+1$'th element of this array. This offset can be used to access the data in memory by offsetting the data pointer. For example to find temperature in this container, the key of the `TEMPERATURE` variable, in this example 2, indicates that the third element of the offset array contains the offset for temperature which is 1. Then this offset is used to get the value of temperature in the data array. Figure 7.59 shows this procedure.

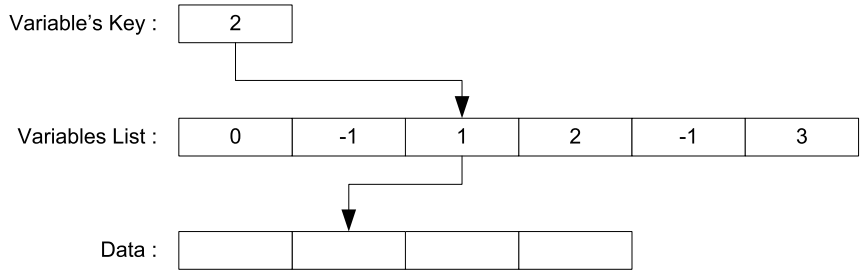



Figure 7.59: Accessing to a value in the variables list container.

#### Interface and Operations

**Access** This container first uses the key of a given variable and gets the necessary offset from the variables list. This offset is used to access its value in memory. Then return this position as a reference with the type of variable. Its important that the type recognition can be done in compilation time in order to eliminate its overhead in runtime. If the inserting in container is enabled, a control is necessary to see if the requested variable is really stored in the container or not.

**Insert** Inserting a new element consist of allocating memory and copying correctly the object and also adding it to the variables list. This means that adding a new variable to a container virtually adds it to all the other containers sharing the variable list with it. Using an array for data in this container implies that all references to elements of any container sharing the variables list can be invalidated by inserting a new element.

**Find** Finding is done via indirection and is a fast procedure. A variable key is enough to find it efficiently and only the type of data is needed for correctly casting the results of the search.

**Erase** Removing data is very complicated. It can be done by giving a removed tag to the variable in list and then each container has to update itself for new list. All these makes erasing practically unacceptable.

**Copy** Copying this container is similar to the `DataValueContainer`. Again the container goes element by element and uses the corresponding variable to copy the data. This process needs a virtual function call which reduces its performance.

**Clear** Clearing this container is also similar to the `DataValueContainer`. The container uses the variable to call the correct destructor of data and correctly remove it from memory. This procedure is necessary because simply freeing the memory will not call the destructor of object by itself. This causes problems specially when objects have some memory allocated internally and removing them without calling to destructor results in memory leaks in the system. Obviously this process is slower than an homogeneous clearing due to the function calling overhead for each element.

#### Variables List Container Advantages

- The accessing and finding processes are very efficient because only two indexing are necessary to find each value.
- Extendibility to store any type of data without any implementation cost. Adding new types to this container is an automated task without changing the container or even reconfigure it.
- Usually extensive use of void pointers and down casting make heterogeneous containers vulnerable to type crashing. Using the variable base interface protects users from unwanted type conversion and guarantees the type-safety of this container.

#### Variables List Container Disadvantages

- Having a shared variable list imposes an extra effort to group related containers and manage them in different groups. Practically this makes the object using this container less independent.
- Erasing a variable from this container is a complex and difficult task. For this reason variables list container cannot be used in problems with temporal variables.

#### Implementation

The first approach to implement this container is to put everything in the container and taking a simple list of variables to work. This approach looks attractive by encapsulating everything in the container and using an standard vector for the variable list. Unfortunately this design requires recalculation of the offset for each access which imposes an unacceptable overhead. So let us change the design to remove this unnecessary overhead.

Another approach is to divide the mechanism in two parts. One part for calculating the position and another for handling the memory. In this design the `VariablesList` class keeps the list of variables to be stored and also provides their local position by giving the necessary offset for each one. The container is in charge of allocating memory, copying itself and clearing the data in a correct way using a variables list. Figure 7.60 shows this structure.

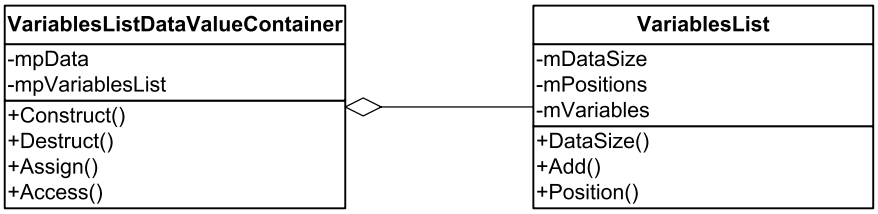



Figure 7.60: The `VariablesList` class provides the list of variables and their local positions for the `VariablesListDataValueContainer`.

An important decision here is to let container add new variables to the shared list or not?. How this feature affects our implementation? When each container is enable to add a new variable to the list of stored variables, this list also changes for all other containers sharing it. This change implies that for each access to a container, it must check if the list is changed or not and, in the case of new variables, update itself. This procedure introduces an overhead in all accesses to the container's elements. Also this updating invalidates all references to its elements which complicates more its use and increases the number of accesses to its data.

In Kratos the inserting was enabled to make this container compatible with previous ones. In practice the problem was not only the check and updating overhead. The updating makes debugging a difficult task. References are not reliable because there is no guarantee that some other container has not changed the list. This can be even worse in case of parallel computation because this change can happen in another thread just after a reference is taken. So finally the inserting feature was removed from this container to reduce problems using it.

Without the inserting ability the implementation of this container is very easy. Constructing is done by allocating the memory with data size provided by a given variables list. `VariablesList` is in charge of calculating the required memory to store its variables. To improve the portability of the code, the memory is divided into some blocks with a configurable specific size. `VariablesList` determines the number of blocks needed to store each variable and calculates the total size by the sum of the required blocks of memory. An additional step in the constructing process is assigning an initial value to elements in order to avoid uninitialized value problem. This can be done by using the `AssignZero` method of given variable which assigns its zero value to the allocated element in the container. The following list shows a default constructor for this type of container:

```
/// Default constructor.
VariablesListDataValueContainer()
  : mpData(0), mpVariablesList(&msVariablesList)
  {
    int size = mpVariablesList->DataSize()*sizeof(BlockType);

    // Allocating data using size provided by variables list.
    mpData = (BlockType*)malloc(size);

    // Initializing elements with zero value given by each
    // variable.
    VariablesList::const_iterator i_variable;

    for(i_variable = mpVariablesList->begin() ;
        i_variable != mpVariablesList->end() ;
```

```
        i_variable++)
        {
            std::size_t offset = mpVariablesList->Index(*i_variable);
            i_variable->AssignZero(mpData + offset);
        }
  }
```

The copying process for this container depends on its source. If the source is empty this just implies the clearing of the container. If it is not empty but sharing the same variables list, there is no need to perform the deallocation and allocation procedure of elements and the process consists of just assigning the values using the variables in the variables list. Finally for a source with different variables list it is necessary to clear the container and reallocating the memory for the new elements. The following code shows the assignment operator:

```
/// Assignment operator.
VariablesListDataValueContainer&
  operator=(const VariablesListDataValueContainer& rOther)
  {
    // if the source container is empty call clear.
    if(rOther.mpVariablesList == 0)
        Clear();
    // if other container uses the same variables list
    // assigns the container element by element.
    else if(mpVariablesList == rOther.mpVariablesList)
    {
        // Assigns other elements value using variable's assign
        // method.
        AssignElements(rOther);
    }
    else
    {
        // Destruct previous elements by calling their
        // destructors
        DestructElements();

        // Updates variables list
        mpVariablesList = rOther.mpVariablesList;

        // Reallocating the memory for new size
        int size = mpVariablesList->DataSize()*sizeof(BlockType);
        mpData = (BlockType*)realloc(mpData, size);

        // Copying other elements value to new allocated memory
        // using variable's copy method.
        CopyElements(rOther);
      }

      return *this;
    }
```

Like the previous container clearing the container consists of manually calling the destructor of each variable and then freeing the memory. Using the appropriate member of the variable class simplifies this procedure as seen before.

## 7.4 Common Organizations of Data

Different ways of distributing the data are used in finite element programs. Each of them has its advantages and disadvantages and can be useful for some cases while imposing difficulties to other problems. In this section some of the existing data distributions are explained and their properties are emphasized.

### 7.4.1 Classical Memory Block Approach

An old standard form of keeping data in memory is an indexed block memory container. Old fortran codes usually use this approach due to the restriction of old fortran compilers. Also some new codes are still using it because of its great performance.

In this approach each category of data is stored in a block of memory. The ordering of data in this block of memory depends on the algorithm uses the data. Some algorithms take one `Node` or `Element` and work over their data and then go to another one. In this case data of each `Node` and `Element` must be stored sequentially to minimize the cache miss while operating over a `Node` or an `Element`. Figure 7.61 shows this alignment of data.

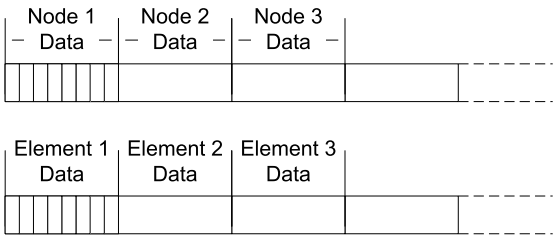



Figure 7.61: Grouping all variables of one `Node` or `Element` in order to reduce the cache miss in nodal and elemental operations.

Some other algorithms take one variable, for example the displacement, and then perform some operations over this variable for all `Node`s or `Element`s. For these algorithms an efficient alignment is to store the values of each variable in different `Node`s or `Element`s sequentially. In this way the cache missing is minimal and the vectorization of the process for parallel computing can be done more effectively. Figure 7.62 shows this alignment of data.

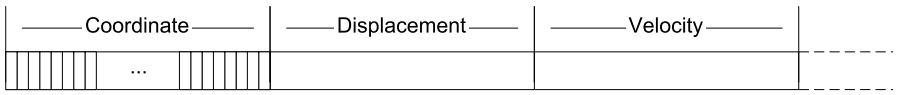



Figure 7.62: Storing values of each variable in different `Node`s or `Element`s sequentially to optimize the data structure for algorithms working with one variable over the domain.

An extension of this alignment is to group the variable components for algorithms working with components separately as can be seen in figure 7.63.

This structure is simple to implement but requires the programmer to know the exact number of variables per node or gauss points and also the size of the buffer which can be difficult to determine

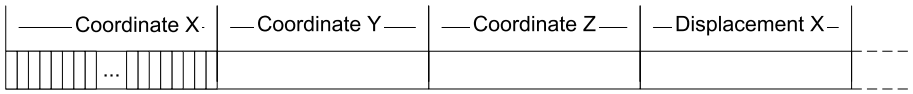



Figure 7.63: Separating the components of each variable for algorithms working with each component separately.

in some cases. By having all these information a block of memory for each group of entities can be allocated and all the data can be stored easily. Also a matrix indexing system can be established to help accessing a certain variable of an entity like a `Node` or an `Element`.

**Advantages**

- Very easy to program and to use. Normally indexing is used to access inside the data (via a global pointer) and there is no extra pointer to allocate and deallocate so it is easy to create and use and also to clear.
- Very fast and efficient in using system cache. Having all data together makes it easy to keep the system cache full in runtime. This advantage is more apparent when we have a loop over a data for all entities.
- Operation over these data structures can be easily parallelized for shared memory platforms.

**Disadvantages**

- The structure is rigid for adding new variables. Also having different variables in different `Node`s or `Element`s makes a big unnecessary overhead. For example when we attempt to introduce a new variable to some `Node`s we have to add a new row (or column) to this block. This means that we have to allocate memory equally for this variable in all `Node`s, wether they have that variable or not. This overhead in some codes is not big but for some other codes can be very important.
- This container is homogeneous and hence is not suitable for cases when we need a container to store different data types in it.
- Adding or removing `Node`s or `Element`s causes big reallocation in this data structure, this makes it less interesting for problems where number of `Node`s and `Element`s is changing continuously.

### 7.4.2 Variable Base Data Structure

This is an step forward from the previous data structure. In this approach the data related to each nodal or elemental variable are stored in a separate array. For example the structure has four arrays to store nodal coordinates, displacements, velocities and accelerations. So in this approach a variable can be added, allocated or removed from memory anytime its needed.

In this approach any algorithm operating over some entities not only has to take an array of entities as its arguments but also needs different arrays of data which are required for its operation. This makes the input and output of method more readable in the code but puts more restriction for some generic methods. However one can create a table of all variables and their names and pass them as an additional argument to guarantee the extendibility of the methods.

Many fortran codes as well as several C and C++ codes use this format for storing their data.

**Advantages**

This container has a many advantages which makes it popular in finite elements codes.

- Good performance in adding new variables or removing existing ones. In this approach creating new variables or removing some existing ones makes no changes in the global structure and will not affect other parts of the data structure.
- Very fast and efficient in using system cache for algorithms which are oriented to work with variables over the domain.
- Capable of adding new types. This structure can be used to store different types of data without any problem. Also adding new types of variables makes no difficulties because each variable's data are stored in a different array which can have any type of data.
- Easy to program and to use. Creating this data structure is relatively easy with less requirements than for the memory block approach. It is also easy to use as accessing is only via one indexing without any redirection or searching.
- Like the previous approach, the operation over these data structures can be easily parallelized for shared memory platforms.

**disadvantages**

- Fair performance in removing some entities' data from the container. For problems involving `Node` inserting and removing or `Element` inserting and removing this type of container can introduce a large overhead. To avoid this problem, one can assign a removed flag for entities and update the data structure once, however this method also has its own complexities.
- Not very efficient for using in algorithms working entity by entity. In this structure different data related to one entity can be stored in very far places from each other in memory. This results in cache misses which reduces the performance of the algorithm.

### 7.4.3 Entity Base Data Structure

In an abstract point of view we can assume this container as the transpose structure of variable base container. In this structure we put all the data related to one entity in the same group. For example all data related to a certain `Node` are stored together. But unlike the memory block there is no guarantee that the blocks of nodal data are sequentially stored in memory.

A rigid but very fast implementation is to make each variable of an entity a member of it. In this way the locality of data is guaranteed, which increases the performance of the application. The problem is the rigidness of this implementation. Each new variable must be added to the entity in programming time. For example `Node` has to have all variables of different problems that the application can solve. For this reason in multi-disciplinary codes this approach cannot be used.Figure 7.64 shows this structure.

Another approach is to allocate a block of memory for the data related to each entity and giving to the entity a reference to its data. A practical way is to define a generic container as the member of each entity where it can store its data. This implementation is more flexible but less efficient because separating the block of data from the entity produces a jump in memory for accessing it and produces cache missing. Figure 7.65 shows this implementation.

Using this structure, algorithms operating over entities just has to take the array of the entities. Because each entity knows how to access its data so algorithm can access its necessary data from

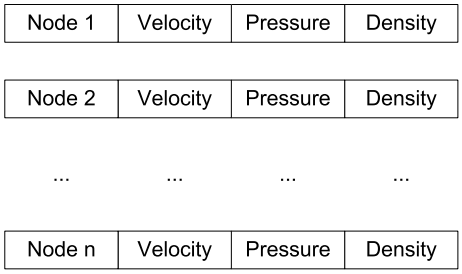



Figure 7.64: In an entity base data structure all data related to one entity can be stored with entity as its members.

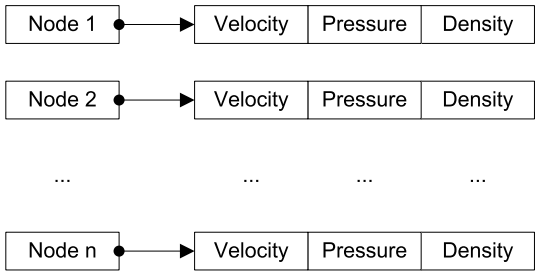



Figure 7.65: Each entity has a reference to its container or block of data.

the entity and there is no need to give its input and output as additional arguments. This reduces the number of arguments while maintaining the extendibility of the code.

### Advantages

- Good performance in adding an entity or removing an existing one. The data related to each entity are grouped together separately to other data, so any new group of data related to a new entity can be added without affecting other parts of the data structure. Also removing data related to one entity can be done independently without any problem.

- Separating data in this way makes parallelization of the application over distributed memory architectures easier. In this way there indexing is not necessary to find the data of an entity. This independency is handy for dividing data over machines.

- Good use of cache memory when used in algorithms which operate on entities. The algorithm can access several data of one entity to do its operation and then goes to the next entity. This structure has a good performance because keeps all the data in one block of memory and significantly reduces the cache misses.

**Disadvantages**

- Fair performance in adding new variables or removing existing ones. In this approach introducing new variables or removing some existing ones requires an entity by entity resizing of data, which makes it less efficient than the previous approach.
- Less efficient in using system cache for algorithms which are oriented to work with variables over the domain. Making a loop over a variable holds in entities produces jumps in memory when going from one entity to another. This make it less efficient in using the cache memory and therefore reduces the performance of the algorithm.

## 7.5 Organization of Data

In a finite element program several categories of data has to be stored. Nodal data, elemental data with their time histories and process data are examples of these categories. Also in a multi-disciplinary application, `Node`s and `Element`s can be stored in different categories representing domains or other model complexities. In this section the global distribution of data in Kratos will be discussed.

### 7.5.1 Global Organization of the Data Structure

In previous section some common ways to organize data in finite element application were described and their advantages and disadvantages were discussed. It was seen that both variable base and entity base structure offer good features but for two very different type of algorithms. The first structure is optimized for domain based algorithms and also when some variable has to be added or removed from data structure. While the second structure is better for entity based algorithms and is more flexible for adding or removing entities.

Kratos is designed to support an elemental-based formulation for multi-disciplinary finite element applications and also started with mesh adaptivity as one of its goals. So the entity based data structure becomes the best choice. First because elemental algorithms are usually entity based and can be optimized better using this type of structure. The second reason is the good performance and flexibility this structure offers, in order to add or remove `Node`s and `Element`s. Beside this entity base structure Kratos also offers different levels of containers to organize and group geometrical and analysis data. These containers are helpful in grouping all the data necessary to solve some problems and for simplifying the task of applying a proper algorithm to each part of the model in multi-disciplinary applications.

Nodal, elemental and conditional data containers are the basic units of this entity base structure. In Kratos each `Node` and `Element` has its own data. In this manner an `Element` can access easily the nodal information just by having a reference to its Node and without any complications. `Properties` also is a block of this structure as a shared data between `Element`s or `Condition`s. Figure 7.66 shows these basic units and their relations to different entities.

Separate containers for `Node`s, `Properties`, `Element`s and `Condition`s are the first level of containers defined in Kratos. These containers are just for grouping one type of entity without any additional data associated to them. These containers can be used not only to work over a group or entities but also to modify their data while each entity has access to its own data. These containers are useful when we want to select a set of entities and process them. For example giving a set of `Node`s to nodal data initialization procedure, sending a set of `Element`s to assembling functions, or getting a set of `Condition`s from a contact procedure. Figure 7.67 shows these containers and their accessible data.

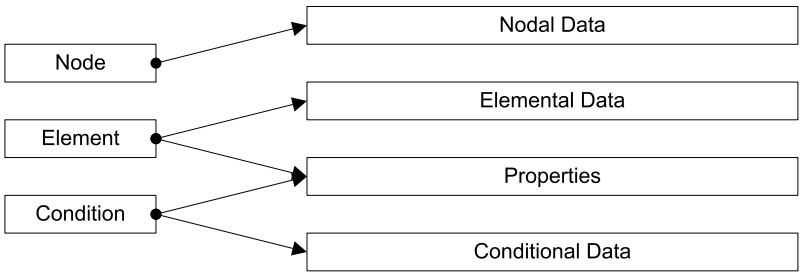



Figure 7.66: Nodal, elemental and conditional data containers with properties are the basic units of Kratos data structure.

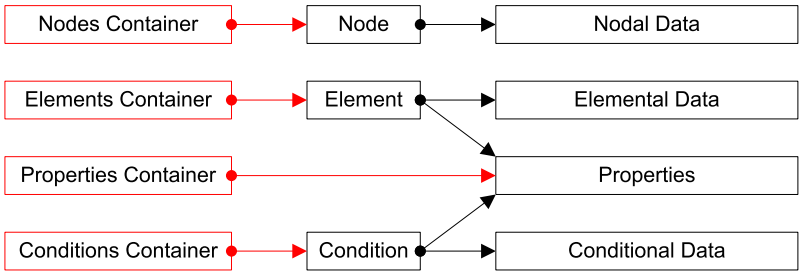



Figure 7.67: Separate containers for `Node`s, `Properties`, `Element`s and `Condition`s can be used to group each type of entities and then process themselves or their accessible data.

`Mesh` is the second level of abstraction in the data structure which hold `Node`s, `Element`s and `Condition`s and their `Properties`. In other word, `Mesh` is a complete pack of all type of entities without any additional data associated with them. So a set of `Element`s and `Condition`s with their `Node`s and `Properties` can be grouped together as a `Mesh` and send to procedures like mesh refinement, material optimization, mesh movement or any other procedure which works on entities without needing additional data for their processes. Figure 7.68 shows `Mesh` with its components.

The next container is `ModelPart` which is a complete set of all entities and all categories of data in the data structure. It holds `Mesh` with some additional data referred as `ProcessInfo`. Any global parameter related to this part of the model or data related to processes like time step, iteration number, current time, etc. can be stored in `ProcessInfo`. `ModelPart` also manages the variables to be hold in `Node`s, `Element`s and `Condition`s. For example all the `Node`s belonging to one `ModelPart` sharing the nodal variables list hold by it. From another point of view `ModelPart` is the nearest container to the domain concept in the multi-disciplinary finite element method. Figure 7.69 shows the `ModelPart` with its components.

In the first implementation, `ModelPart` was able to keep the history of data and also the `Mesh` if it is changing. But in practice this capability became the bottleneck of Kratos performance and was also considered to be unnecessary for our problems. So this feature was removed from `ModelPart`. However each `ModelPart` still can hold more than one `Mesh` which comes from the first implementation and can be used for representing the partitions in parallel computation.

Finally `Model` is a group of `ModelPart`'s and represents the finite element model to be analyzed.

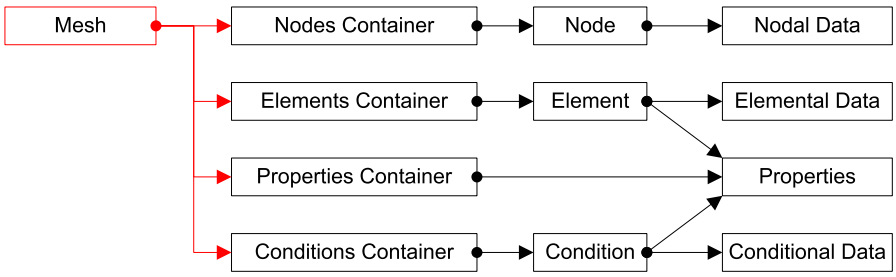



Figure 7.68: `Mesh` is a complete pack of all types of entities without any additional data associated with them.

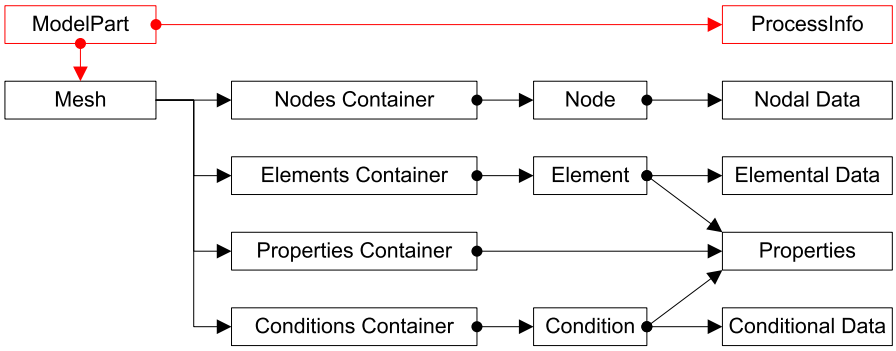



Figure 7.69: `ModelPart`holds `Mesh` with some additional data referred as `ProcessInfo`

It can be useful for some procedures that requires the whole data structure like saving and loading procedures. As processes in Kratos use `ModelPart` as their work domain, this container is not implemented yet but it is necessary to complete the data structure of Kratos.

Spatial containers are separated so can be used just when they are needed. This strategy also allows Kratos to use external libraries implementing general spatial containers like Approximate Nearest Neighbor (ANN) library [74].

### 7.5.2 Nodal Data

The first implementation of Kratos had a buffer of data value containers to hold all the nodal variables. This nodal container was very flexible but with considerable memory overhead for nonhistorical variables. For example for saving two time steps in history (a buffer with size 3) there were two redundant copies of all nonhistorical variables in memory as shown in figure 7.70. It also had fair access performance due to the searching process of the container. All these made us to redesign the way data are stored in `Nodes`.

The new structure is divided into two different containers: nodal data and solution step nodal data. Figure 7.71 shows this nodal data structure. A data value container is used for the nodal data (no historical data) and a variables list container is used for the solution step nodal data (historical data). In this way the memory overhead is eliminated because no redundant copy is

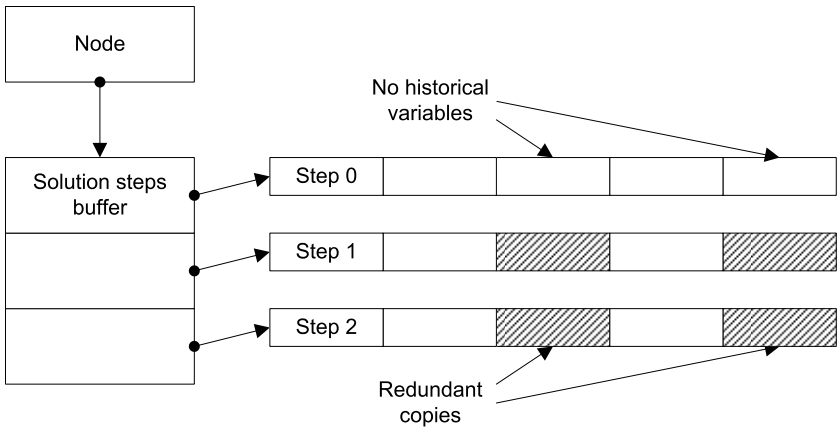



Figure 7.70: Using buffer for all variables results memory overhead due to redundant copies of no historical variables.

produced. Also accessing to historical variables is much faster than before due to the indirection process of accessing in the variables list container instead of the searching process in the data value container. This structure offers good performance and also is memory efficient but is slightly less robust and somehow less flexible to use.

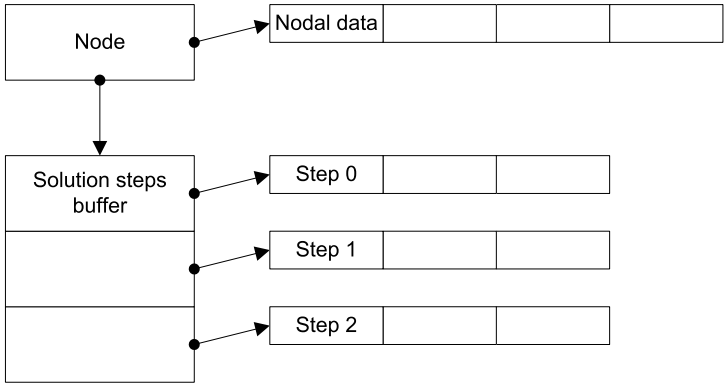



Figure 7.71: The first improvement is dividing nodal data structure into two different containers, nodal data (no historical data) and solution step nodal data (historical data).

Using the variables list container for historical variables, requires the user to define its historical variables in order to construct the nodal data container. For example a fluid application must define velocity and pressure as its historical variables at program startup. The rest of variables can be added any time during the program execution as a no historical variable, but not as a historical one.

The first implementation of this structure was done by creating a buffer of the variables list container to reduce the implementation task and test it in a real problem. After obtaining successful results it was the time to optimize it more. The buffer of variables list containers produces several

jumps in memory which reduces its cache efficiency. Also some mesh generators like TetGen [93, 7] require an array of nodal data as the argument to make the interpolation. For these reason in the current structure the buffer is moved inside the solution steps container. In this way the cache misses is reduced and the data array can be given to other application without any conversion. Figure 7.72 shows this structure.

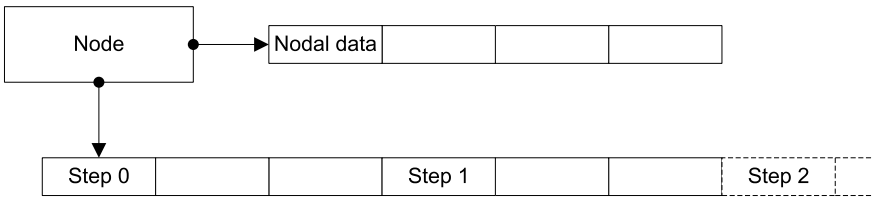



Figure 7.72: The current structure allocates all buffer data in a block of memory to reduce the cache misses produced by memory jumps and also to provide a compatible data with other libraries.

In the first structure all the data were stored in the same container. So there was one place to store them and also one place to recover them. Dividing the nodal data in two categories also changes the access interface to data. Now the user has to know were to put each variable, and more important, where to retrieve them afterward. A sophisticate interface is needed in order to provide a clear and complete control over these two categories of data. In general, three type of accessing methods are necessary:

- Methods for accessing only historical data. These methods guarantee to give the value of the variable if and only if it is defined as a historical variable and produce error if it is not defined. These methods are very fast because they do not need to search in the data value container and also give error for logical errors in the code.
- Methods for accessing only no historical data. Another set of methods are implemented to give access only to no historical data. Due to the flexibility of the data value container any variable can be added at any time as a nodal variable using these methods.
- Hybrid accessing also can be done using another set of methods. These methods try to find the variable in the solution steps container and if it does not exist they provide access to the nodal data container. These methods are helpful for accessing to some input variables that may come from input files as nodal data, or variables which are calculated in another domain and stored as a solution step variable. For example temperature for structural problem can be a parameter coming from the input data or calculated by the thermal elements and stored in `Node`s. This method guarantees the access to the proper temperature stored at each `Node`.

The following methods are implemented to provide above access ways to nodal data:

`GetSolutionStepValue` Takes a variable or a variable's component and the solution step index and returns its value in solution step data if exists, otherwise sends an error. Solution step index starts from 0 for the current step and then increases for past steps. For example 1 is for the previous step, 2 is for one step before the previous one, etc. An overloaded version accept only the variable or a variable's component and returns its current value. In all overloaded versions accessing to a variable which is not declared in the solution steps variables list produces an exception.

`GetValue` There are different overloaded versions of this method. One with the requested variable as its argument, gives an access to nodal data without looking to the solution steps data. Giving a solution steps index as an additional argument makes it look into nodal data and if it does not exist takes it from the solution steps container. It will find any existing value in the nodal data structure for given variable but searching in data makes it slow in accessing to the solution steps data.

### 7.5.3 Elemental Data

Another basic unit of Kratos data structure is the elemental data. Elemental data is divided into three different categories:

**properties** All parameters that can be shared between `Element`s. Usually material parameters are common for a set of `Element`s, so this category of data is referred as properties. But in general it can be any common parameter for a group of `Element`s. Sharing these data as properties reduces the memory used by the application and also helps updating them if necessary.

**data** All variables related to an `Element` and without history keeping. Analysis parameters and some inputs are elemental but there is no need to keep their history. These variables can be added any time during the analysis.

**historical data** All data stored with historical information which may be needed to be retrieved. Historical data in integration points are fall in this category. These data must be stored with a specific size buffer.

As mentioned above `Properties` are shared between `Element`s. For this reason the `Element` keeps a pointer to its `Properties`. This connection lets several `Element`s to use the same `Properties`.

A `DataValueContainer` holds no historical data in the `Element`. Using a `DataValueContainer` provides flexibility and robustness which is useful in transferring elemental data from one domain to another. It is important that these no historical data are not the most used during the analysis and flexibility here is more critical than performance. On the contrary, historical data are more used during the analysis and efficiency in accessing to them is more critical than their flexibility. These data are specified by formulation and other processes do not change them. For this reasons the `Element` do not provide any container for them and these containers can be implemented by element developers. In this way, the customized container will be more efficient and the overhead of any generic container will be eliminated.

### 7.5.4 Conditional Data

Conditional data is very similar to elemental data and is also divided into three different categories:

**properties** As for `Element`s, all parameters that can be shared between `Condition`s is referred as properties. Again sharing these data as properties reduces the memory used by the application and also helps updating them if it is necessary.

**data** All variables related to the `Condition` and without history keeping. Analysis parameters and some inputs are different for each `Condition` but there is no need to keep their history. This variables can be added any time during the analysis.

**historical data** All data stored with set of its history to be retrieved. Historical data in integration points are in this category. These data must be stored with a specific size buffer of their previous values to be used later.

`Condition` like `Element` keeps a pointer to its `Properties` which is shared by other `Conditions` or `Elements`.

The `Condition` has a `DataValueContainer` as its member to hold all data related to `Condition` without keeping its history. Any `Condition` derived from this class can use this container to hold its data without any additional implementation. This base class also provides an standard interface to these data which make it helpful for transferring some data from one `Condition` to another, for example in the interaction between two domains.

For `Conditions`, historical data is considered to be an internal data because is very related to its formulation and usually is used only by formulation inside and not from outside. So to minimize unnecessary overhead and also to increase the performance, no general container is provided for historical data and each `Condition` has to implement one for itself if necessary.

### 7.5.5 Properties

As mentioned before `Properties` is a shared data container between `Elements` or `Conditions`. In finite element problems there are several parameters which are the same for a set of `Elements` and `Conditions`. Thermal conductivity, elasticity of the material and viscosity of the fluid are examples of these parameters. `Properties` holds these data and is shared by `Elements` or `Conditions`. This eliminates memory overhead due to redundant copies of these data for each `Element` and `Condition` as can be seen in figure 7.73.

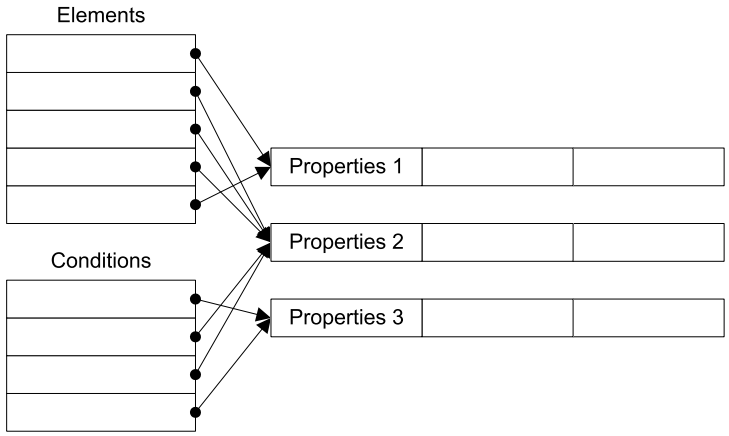



Figure 7.73: Different `Elements` or `Conditions` use `Properties` as their share data container. This avoids redundant copies of data in memory.

It can be seen that changing any data in `Properties` will affect all `Elements` or `Conditions` sharing it. This feature can also be useful in situations when a common parameter is changing during the analysis. The parameter can be changed only in `Properties` and each `Element` or `Condition` will get the new value by accessing to `Properties`. In this way there is no need to update all elemental and conditional value for this data each time its changing.

Properties also can be used to access nodal data if it is necessary. It is important to mention that accessing the nodal data via Properties is not the same as accessing it via Node. When user asks Properties for a variable data in a Node, the process starts with finding the variable in the Properties data container and if it does not exist then get it from Node. This means that the priority of data is with the one stored in Properties and then in Node. For example considering that TEMPERATURE is stored in Properties as a material temperature with value 24.4 and also there is a TEMPERATURE nodal data stored in Node with value 79.3. Now getting the value of TEMPERATURE variable from Properties gives 24.4 while getting it from Node gives 79.3

### 7.5.6 Entities Containers

Let us go one level higher in the Kratos data structure. The next level consists of four entities containers:

- Nodes Container
- Properties Container
- Elements Container
- Conditions Container

In a finite element program, there are several procedures which take a set of entities and operate over them or their data. These containers are created to help users in grouping a set of entities and work with them. For example to put all Nodes in the boundary in a Nodes container and change some of their data in each step. As mentioned before, each entity has access to its data, so having a set of entities in a container also gives access to their data which make these containers more useful in practice.

Another use of these containers is finding an entity by its index. Indexing is an standard way of identifying entities in finite element programs. For example Nodes have their indices to be identified by them. The index of each Node is given by the user as input and can be consecutive or not. A user will use these indices later to define the elemental connectivity. In time of creating Element, it is necessary to find the Node with a given index and give its pointer to the Element. Supporting the indexing system and providing a searching mechanism is very useful for implementing such a processes in a simple and efficient manner.

According to all uses mentioned before, a suitable container for holding entities must provide the following features:

**Sharing Entities** There are some situations when an entity may belong to more than one set of entities. For example a boundary Node belongs to the list of all Nodes and also to the list of boundary Nodes. So the Nodes container has to share some its data with other Nodes containers. In general, sharing entities is an important feature of these containers.

**Fast Iterating** As mentioned before, one important use of these containers is to collect some entities and pass them to some procedures. Usually these procedures have to make a loop over all the elements of a given container and use each element or its data in some algorithms. So these containers must provide a fast iterating mechanism in order to reduce the time of element by element iteration.

**Search by Index** Finding an entity by an index is a usual task in finite element programs. So entities containers must provide an efficient searching mechanism to reduce the time of these tasks.

| Method Name | Operation |
|---|---|
| NumberOfNodes | Returns the number of Nodes in the Mesh. |
| AddNode | Add given Node to its Nodes container. |
| pGetNode | Returns a pointer to the Node with the given identifier. |
| GetNode | Returns a reference to the Node with the given identifier. |
| RemoveNode | Removes the Node with given Id from the Mesh. |
| RemoveNode | Removes the given Node from the Mesh. |
| NodesBegin | Returns a Node iterator pointing to the beginning of the Nodes. |
| NodesEnd | Returns a Node iterator pointing to the end of the Nodes container. |
| Nodes | Returns the Nodes container. |
| pNodes | Returns a pointer to the Nodes container. |
| SetNodes | Sets the given container as is Nodes container. |
| NodesArray | Returns the internal array of Nodes. |

Table 7.1: Interface of Mesh for accessing Nodes

Sharing entities is the first feature to be provided by containers. Holding pointers to entities and not entities themselves can solve this problem. Different lists can point to the same entity without problem. Using an *smart pointer* [37] instead of normal pointer increases the robustness of the code. In this way entities which are not belong to any list anymore will be deallocated from memory automatically. So a container of smart pointers to entities is the best choice for this purpose.

As mentioned above arrays are very efficient in time of iterating. So using an array to hold pointers to entities can increase the iteration speed. In contrary, trees are very slow in time of iterating but efficient for searching by index hence using them can increase the searching performance of the code. A good solution to this conflict can be an ordered array. It is fast in iteration like an array and also fast in searching like a tree. Its only draw back is that its less robust and constructing it can take considerable time depending on the data. For example constructing an array of Nodes with Nodes given by inverse order can take a very long time. Fortunately most of the time the entities are given in the correct order and this eliminates the constructing time overhead for these containers. Also, for the worst cases a buffer of unordered data can significantly reduce the construction overhead. So an ordered array can fit properly into our problem.

PointerVectorSet is a template implementation of an ordered array of pointers to entities. This template is used to create different containers to hold different type of entities.

### 7.5.7 Mesh

The next level in Kratos' data structure is Mesh. It contains all entities containers mentioned before. This structure makes it a good argument for procedures that work with different entities and their data. For example an optimizer procedure can take a Mesh as its argument and change geometries, nodal data or properties. Mesh is a container of containers with a large interface that helps users to access each container separately.

First of all Mesh provides a separate interface for each type of entity it stores. Tables 7.1, 7.2, 7.3 and 7.4 show its interfaces for handling different components.

Mesh holds a pointer to its container. In this way several Meshes can share for example a Nodes or an Elements container. This helps in updating Meshes of different fields in multidisciplinary applications but over the same domain. Figure 7.74 shows this ability of sharing components between Meshes.

| Method Name | Operation |
|---|---|
| `NumberOfProperties` | Returns the number of properties in the `Mesh`. |
| `AddProperties` | Add given properties to its properties container. |
| `pGetProperties` | Returns a pointer to the properties with the given identifier. |
| `GetProperties` | Returns a reference to the properties with the given identifier. |
| `RemoveProperties` | Removes the properties with given Id from the `Mesh`. |
| `RemoveProperties` | Removes the given Properties from the `Mesh`. |
| `PropertiesBegin` | Returns the begin iterator of the properties container. |
| `PropertiesEnd` | Returns the end iterator of the properties container. |
| `Properties` | Returns the properties container. |
| `pProperties` | Returns a pointer to the properties container. |
| `SetProperties` | Sets the given container as is properties container. |
| `PropertiesArray` | Returns the internal array of properties. |

Table 7.2: Interface of `Mesh` for accessing properties

| Method Name | Operation |
|---|---|
| `NumberOfElements` | Returns the number of `Elements` in the `Mesh`. |
| `AddElement` | Add given `Element` to its `Elements` container. |
| `pGetElement` | Returns a pointer to the `Element` with the given identifier. |
| `GetElement` | Returns a reference to the `Element` with the given identifier. |
| `RemoveElement` | Removes the `Element` with given Id from the `Mesh`. |
| `RemoveElement` | Removes the given `Element` from the `Mesh`. |
| `ElementsBegin` | Returns the begin iterator of the `Elements` container. |
| `ElementsEnd` | Returns the end iterator of the `Elements` container. |
| `Elements` | Returns the `Elements` container. |
| `pElements` | Returns a pointer to the `Elements` container. |
| `SetElements` | Sets the given container as is `Elements` container. |
| `ElementsArray` | Returns the internal array of `Elements`. |

Table 7.3: Interface of `Mesh` for accessing `Elements`

| Method Name | Operation |
|---|---|
| `NumberOfConditions` | Returns the number of `Conditions` in the `Mesh`. |
| `AddCondition` | Add given `Condition` to its `Conditions` container. |
| `pGetCondition` | Returns a pointer to the `Condition` with the given identifier. |
| `GetCondition` | Returns a reference to the `Condition` with the given identifier. |
| `RemoveCondition` | Removes the `Condition` with given Id from the `Mesh`. |
| `RemoveCondition` | Removes the given `Condition` from the `Mesh`. |
| `ConditionsBegin` | Returns the begin iterator of the `Conditions` container. |
| `ConditionsEnd` | Returns the end iterator of the `Conditions` container. |
| `Conditions` | Returns the `Conditions` container. |
| `pConditions` | Returns a pointer to the `Conditions` container. |
| `SetConditions` | Sets the given container as is `Conditions` container. |
| `ConditionsArray` | Returns the internal array of `Conditions`. |

Table 7.4: Interface of `Mesh` for accessing `Conditions`

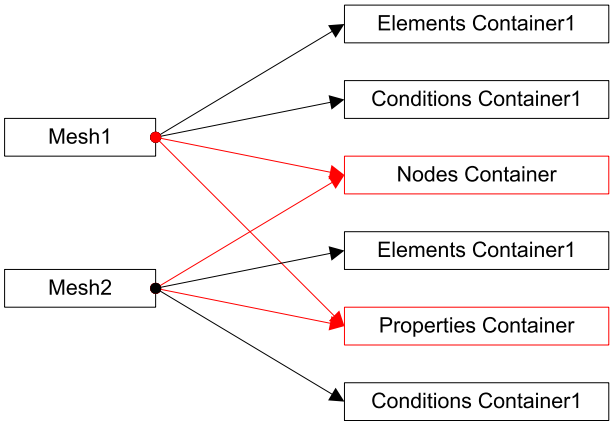



Figure 7.74: Different `Meshes` can share their entities' containers.

### 7.5.8 Model Part

`ModelPart` is created with two different tasks in mind. The first task is encapsulating all entities and data categories of Kratos which makes it useful as an argument of global procedures in Kratos. The second task is managing the variables lists of its components.

`ModelPart` can hold any category of data and all type of entities in Kratos. It can hold several `Meshes`. Usually just one `Mesh` is assigned to it and used in the computations, however this ability can effectively used for partitioning the model part and send it for example to the parallel processes. Beside holding different `Meshes`, it also stores the solution information encapsulated in the `ProcessInfo` object. Figure 7.75 shows the structure of `ModelPart`.

`ProcessInfo` holds not only the current value of different solution parameters but also stores their history. It can be used to keep variables like time, solution step, non linear step, or any other variable defined in Kratos. Its variable base interface provides a clear and flexible access to these data. `ProcessInfo` uses a linked list mechanism to hold its history as shown in figure 7.76.

`ModelPart` uses pointers to its `Meshes`. In this way it can share them with any other model parts if necessary. A typical use of this feature is defining two different domains over the same

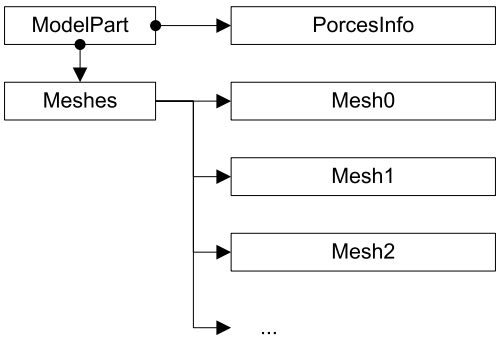



Figure 7.75: `ModelPart`'s structure.

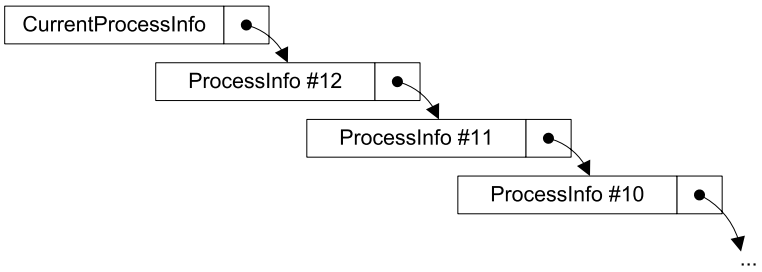



Figure 7.76: `ProcessInfo`'s linked list mechanism for holding history of solution.

`Meshes`. Figure 7.77 shows this sharing mechanism.

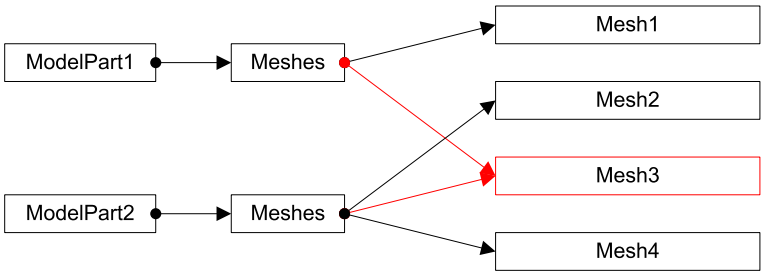



Figure 7.77: `ModelPart` can share its `Meshes` with other model parts.

`ModelPart` manages the variables lists of its components. In section 7.3.3 the mechanism of the variables list container has been described, where we also mentioned that a shared variables list specifies the data which can be stored in them. `ModelPart` holds this variables list for all its entities. In other words, all entities belonging to a model part sharing the same list of variables. For example all `Nodes` in `ModelPart` can store the same set of variables in their solution steps container. It is important to mention that this variables list is assigned to the entities which belong to the model part and is not changed when that model part share them with other model parts. Figure 7.78 shows this scheme.

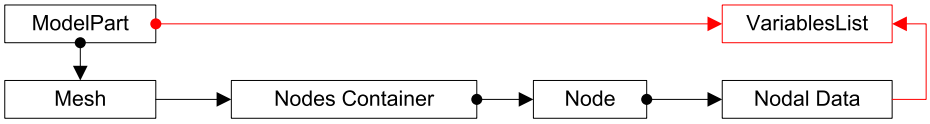



Figure 7.78: `ModelPart` manages the variables list for its entities.

### 7.5.9 Model

`Model` is the representation of whole physical model to be analyzed via the FEM. The main purpose of defining `Model` is to complete the levels of abstraction in the data structure and a place to gather

all data and also hold global information. This definition makes it useful for performing global operations like save and loading. It holds references to model parts and provides some global information like the total number of entities and so on. It can be seen that a `ModelPart` created over given model can do most of these operations by itself. This was the reason that `Model` itself has not been used yet in Kratos. Its practical use would be at the time of implementing the serialization process or other global operations.

Chapter 8

# Finite Element Implementation

## 8.1 Elements

`Element`s and `Condition`s are the main extension points of Kratos. New formulations can be introduced into Kratos by implementing a new `Element` and its corresponding `Condition`s. This makes the `Element` an special object in our design.

### 8.1.1 Element's Requirements

An `Element` is used to introduce a new formulation to Kratos. To guarantee the extendibility of Kratos, adding an `Element` must be an easy task and without large modifications in the code. Encapsulating all data and procedures necessary to calculate local matrices and results in one object and using a clear interface is required to achieve this objective.

A user may use different `Element`s for different parts of the model and then assemble them separately or together. For example in modeling a multi floor structure, the user would use beam elements for frames and shell elements for floors. So these `Element`s must be compatible to be solved together as a complex system. For this reason the ability to use any `Element` in any part, or even mix them, is another requirement to be considered in the Kratos design.

`Element` has to have a very flexible interface due to the wide variety of formulations and different requirement they have. For example, some formulations need to calculate the stiffness matrix and also the righthand side vector in each solution step. Some others just need to calculate the stiffness matrix once and right hand side for each step. Sometimes having the damping matrix separately is needed to handle different time dependent strategies. These formulations are not only different in their local matrices, but also they need different data for their calculations. For example some need time and time step, some other the number of nonlinear iterations, or other parameters depending on the strategy chosen for analysis.

Easy to implement is another requirement for an `Element`. Finite element developers are usually less familiar with advanced programming language features and they would like to focus more on their finite element developing tasks. For this reason the main intention in designing Kratos is to isolate this parts from working with memory or an excessive use of templates. The idea is to provide a clear and simple structure for an `Element` to be implemented by finite element developers wishing to introduce a new formulation to Kratos.

Performance is a very important point to be considered at the time of designing an `Element`. In a finite element code `Element`s methods are called in nearly most inner loops of code. This means that any small fault in `Element`'s performance can cause great overhead in the program execution time. It is obvious that performance of a new `Element` is highly depended on its implementations, but sometimes a weak design can lead to serious bottlenecks in the performance of all `Element`s. We will see later how an elegant but not optimized interface can highly decrease the performance of `Element`s.

Memory efficiency is also important for `Element`s. Modeling a real problem with the FEM usually needs a large number of `Element`s to be created. For this reason any unnecessary overhead in memory used by each `Element` can cause a significant overhead in the whole memory used by program. This overhead, by the way, can restrict the maximum size of the model that can be analyzed in a machine. So efficiency in memory is considered to be very important and less important features must be reduced to keep `Element` as small as possible.

### 8.1.2 Designing Element

After reviewing the `Element`'s requirements, the next step is to design it. In Kratos an `Element` is an object which holds its data and calculates elemental matrices and vectors to be assembled and also can be used to calculate local results after the analysis. For example a thermal element calculates the local stiffness matrix and the mass matrix (if necessary) and give it to Kratos for assembly process. Also it can be used to calculate thermal flow after solving the problem. This definition provides a good isolation for `Element` related to rest of the code which is helpful for the proper encapsulation of `Element`.

`Element`s must be designed to be implemented independently and added easily to Kratos in order to guarantee the extendibility of Kratos. Also they must be compatible with each other in order to let users interchange them or even mix them together in a complex model. According to these two requirements the strategy pattern described in section 3.4.1 is what we are looking for. Applying this pattern to our problem results in the `Element`s' structure shown in Figure 8.1.

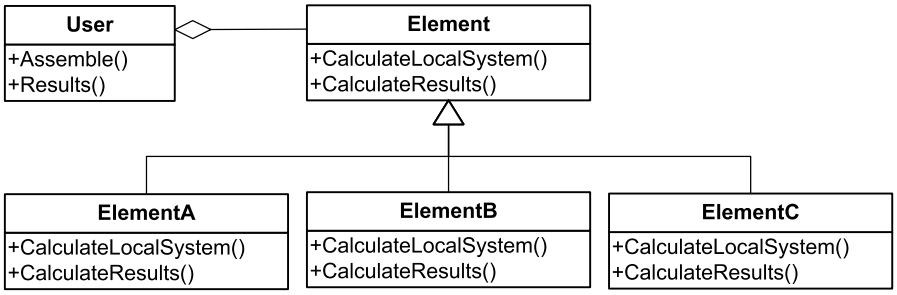



Figure 8.1: `Element`s' structure using strategy pattern.

Using this pattern each `Element` encapsulates one algorithm separately and also make them interchangeable as we want. User keeps a pointer to `Element` class which may point to any member of `Element`'s family and use the interface of `Element` to call different procedures.

The next concept in `Element`s design is its relation with the geometry. As described in section 5.3 a geometry holds a set points or `Node`s and provides a set of common operations to ease the implementation of `Element`s and `Condition`s. Each `Element` has to work with geometry and from

many points of view its an extended geometry with a finite element formulation as it is extension part. This relation can be translated in an object oriented philosophy as a parent and derived class relationship as can be seen in figure 8.2.

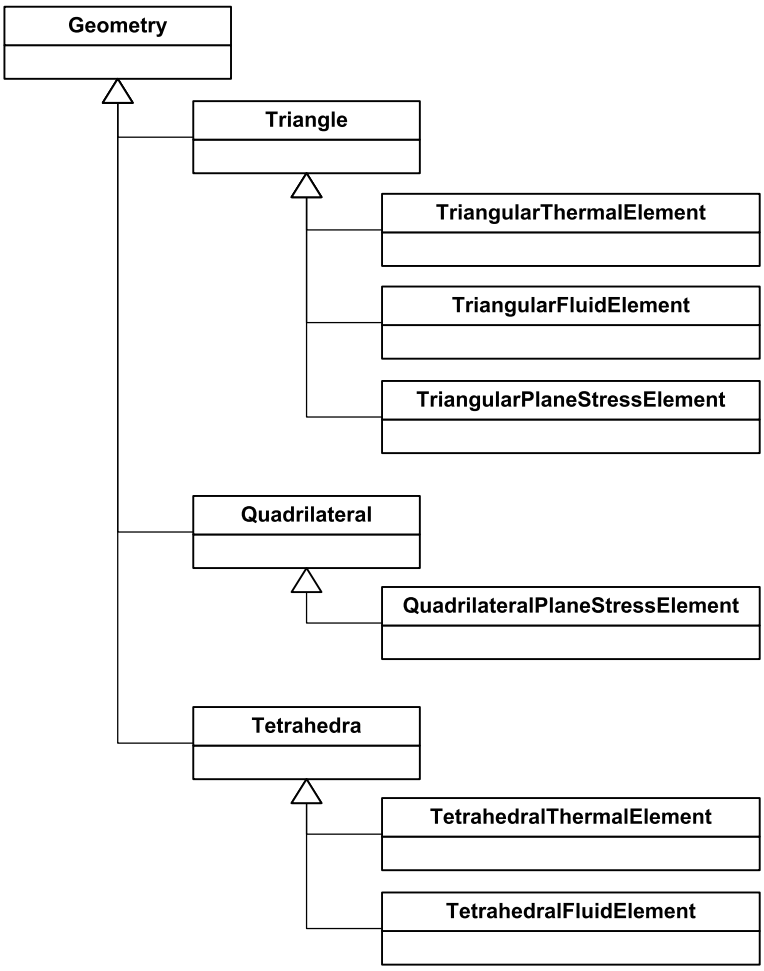



Figure 8.2: Deriving `Element` from geometry requires several `Element`s with same formulation but different geometries to be created.

This structure has the advantage that `Element`s access to geometry data is fast and increases the performance of elemental procedures. Beside this advantage there are two main disadvantages that make this structure unsuitable for our purpose. The first disadvantage is that applying a formulation to different geometries, requires several `Element`s to be implemented. The second drawback is that `Element`s with different formulation cannot share a geometry in memory.

In this structure each `Element` can be implemented to add a formulation to the geometry which is derived from. So different `Element`s must be implemented to extend a formulation to different geometries. For example a triangular plane stress element is derived from a triangle and has a plane stress formulation. Applying the same formulation to a quadrilateral requires another `Element`,

with nearly the same structure but derived from a quadrilateral to be written. This results in a significant overhead in the implementation and also in maintenance of `Element`s.

In a multi-disciplinary problem there are some situations when two interacting domains use the same mesh. In order to implement this possibility, different `Element`s should be able to share the same geometry. In this way the data transfer is minimized and the interpolation cost is eliminated. A simple example is a thermal and structural interaction. Mesh is used to create thermal elements and calculate the temperature over the domain. Then the same mesh is used to create structural elements and calculate the stresses and deformations in domain using the previously calculated temperature for temperature dependent materials. Without sharing geometries, a copy of all geometries must be created so that each set of geometries can be assigned to one domain. This increases the memory used by the application that can be avoided easily by sharing the geometries in mesh.

The alternative design is to use a bridge pattern. Introducing this pattern to our `Element`'s structure design results in the structure shown in Figure 8.3.

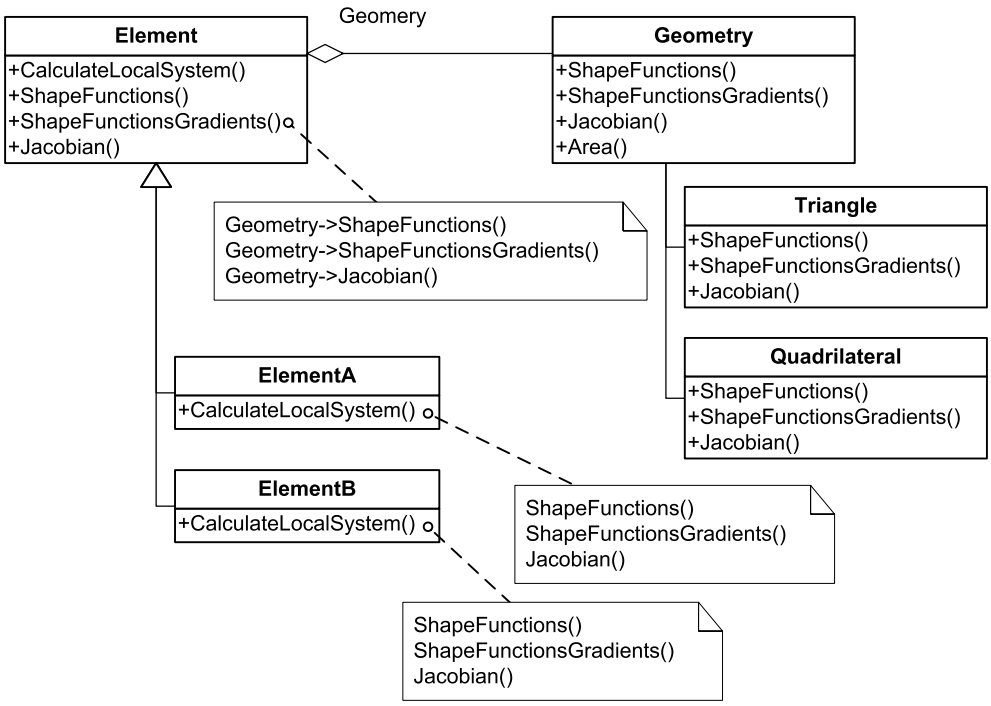



Figure 8.3: `Element`'s structure using the bridge pattern.

This pattern allows each `Element` to combine its formulation with any geometry. In this way less implementation is needed. Also having a pointer to geometry allows an `Element` to share its geometry with other ones. The only drawback of this structure is the time overhead comes from pointer redirection in memory. Having a pointer to geometry beside deriving from it creates a small overhead in accessing geometries' data respect to a direct derived `Element`. Though the efficiency in `Element` is crucial the complexity of the first approach imposes accepting the small different in performance and therefore we have implemented this second approach. A better solution is to

make `Element` a template of its geometry. Using templates provides good performance and also enough flexibility but it was considered to be too complex to be used by finite element users. As mentioned earlier an `Element` has to be easy to program with the less possible advanced feature of programming language. So finally the current structure with bridge pattern was selected.

There are some designs in which different `Element`s can be composed to create a more complex `Element` [70]. This approach can be simulated here using a *Composite* pattern. However this structure is not implemented yet in Kratos.

After designing the global structure now it is time to define interfaces. Here the finite element methodology helps in designing a generic interface. According to the finite element procedure, the strategy asks `Element` to provide its local matrices and vectors, its connectivity in form of equation id, and after solving also calls `Element` to calculate the elemental results. So `Element` has to provide three set of methods:

**Calculate Local System** The first set of methods are required to calculate local matrices and vectors.

**Assembling Information** These methods give information about the position of each row and column of the local system in the global system. This information comes from dof and `Element` provide it by giving its dofs or just their equation id.

**Calculate** It is used to calculate any variable related to an `Element` which usually are the results depending on gradients within the element.

An important issue here is the efficiency of these methods. An attractive form is to make these methods take their necessary parameters as their arguments and return their results as their return values:

```
Matrix CalculateLeftHandSide(ProcessInfo& rCurrentProcessInfo)
{
    //calculating stiffness matrix

    return stiffness_matrix;
}

// Assembling
for(int i = 0 ; i < number_of_elements ; i++)
    Assemble(elements[i].CalculateLeftHandSide(process_info));
```

It can be seen that this design is very natural and easy to use, but in practice produces a significant overhead in performance. Calling each method consists in creating a new matrix or vector, fill it an finally pass it by value as result. Creating a dynamic matrix or vector is a very slow process and passing them by value needs temporaries to be created which is time consuming. All these steps make this design very slow and therefore unacceptable. A better idea is passing the result matrix or vector by reference to these methods as additional arguments. In this way there are no temporaries for passing by value and there is no need to create a variable for the result inside each calculation method. A performance issue for this new design is the resizing of result matrices and vectors. In practice resizing dynamic matrices and vectors results to be very slow. A simple control of a given matrix or vector size before resize it can reduce the resizing overhead for cases that the given size is correct.

A set of methods are necessary for calculating local system matrices and vectors. Different procedures in finite element methods require different information in different analysis points from `Element`. For example a simple linear strategy requires the local matrices and vectors once to

assemble the global system. So a method which calculates local system components is enough to handle this strategy. But for a non-linear analysis, the strategy needs to get also the right hand side to calculate convergence. For these cases a method to calculate this right hand side component is necessary. Also there are some cases that the right hand side is not changing during the analysis and strategy only needs to update its left hand side component which requires a method for calculating only left hand side components.

Implementing only the first method to calculate local system components results in a calculation overhead for non-linear cases. For example calculating the convergence of the solution with a residual criteria, only needs the right hand side components to be updated and calculating all components can apply an unacceptable overhead to this procedure. Keeping only the interface for the left and right hand side components also produces calculation overhead. Usually for calculating each part of, local system the jacobian of the elements must be calculated which is a time consuming operation. Calculating right and left hand side components separately implies that jacobian must be calculated twice. So an optimum design is to keep both interfaces in parallel.

The drawback of this decision is the need for implementing duplicated methods. This problem can be solved by a more carefully implementation. One can create two private auxiliary methods: `LeftHandSide` and `RightHandSide` to calculate left hand side and right hand side matrices and vectors with the calculated jacobian as their input. Then `CalculateLocalSystem` can calculate the jacobian once and call these methods with this jacobian to calculate local system components. Also `CalculateLeftHandSide` and `CalculateRightHandSide` would calculate the jacobian and call their related method to calculate the local contribution. Here is an example of this implementation:

```
class MyElement
{
public:
  virtual void
  CalculateLocalSystem(MatrixType& rLeftHandSideMatrix,
                       VectorType& rRightHandSideVector,
                       ProcessInfo& rCurrentProcessInfo)
  {
    Matrix jacobian;

    Jacobian(jacobian);
    LeftHandSide(rLeftHandSideMatrix, jacobian,
                 rCurrentProcessInfo);
    RightHandSide(rRightHandSideVector, jacobian,
                 rCurrentProcessInfo);
  }

  virtual void
  CalculateLeftHandSide(MatrixType& rLeftHandSideMatrix,
                        ProcessInfo& rCurrentProcessInfo)
  {
    Matrix jacobian;

    Jacobian(jacobian);
    LeftHandSide(rLeftHandSideMatrix, jacobian,
                 rCurrentProcessInfo);
  }

  virtual void
```

```
  CalculateRightHandSide(VectorType& rRightHandSideVector,
                         ProcessInfo& rCurrentProcessInfo)
   {
    Matrix jacobian;

    Jacobian(jacobian);
    RightHandSide(rRightHandSideVector, jacobian,
                  rCurrentProcessInfo);
   }

  private:

   void LeftHandSide(MatrixType& rLeftHandSideMatrix,
                     Matrix& rJacobian,
                     ProcessInfo& rCurrentProcessInfo)
   {
       // Calculating left hand side matrix using given jacobian.
   }

   void RightHandSide(VectorType& rRightHandSideVector,
                     Matrix& rJacobian,
                     ProcessInfo& rCurrentProcessInfo)
   {
       // Calculating right hand side vector using given jacobian.
   }

};
```

Also it is important to mention that `Element`s not necessarily have to implement all these interfaces and they can be compatible with just one way and not providing the other. By the way, calling two separate methods in `CalculateLocalSystem` method of `Element` class can keep more compatible the `Element`s which are not providing the `CalculateLocalSystem` method and just provide `CalculateLeftHandSide` and `CalculateRightHandSide` methods.

Another issue is optimizing for symmetric or diagonal matrices. In Kratos local system matrices are defined as dense matrices in order to be more general. `Element`s with symmetric formulation also have to fill this dense matrix. The optimization can be done at the strategy level by assembling only half of this matrix in a symmetric global matrix to reduce memory usage and also assembling time. However the redundant time of filling all components of the dense matrix is unavoidable in order to keep `Element`s compatible with nonsymmetric strategies. Diagonal matrices can be treated as symmetric ones by keeping the optimization level in strategy and not in `Element`.

This interface is designed to be generic but its flexibility to support new algorithms also depends on its ability in passing different parameters necessary for different formulations. For this reason a variable base container is used to enable users pass any parameter to an `Element` using the VBI described before. `ProcessInfo` can be used to pass any parameter which is necessary for calculating local systems in an `Element`. The usual parameters are time, time increment, time step, non-linear iteration number, some global norms which are calculated over the domain, etc. Using `ProcessInfo` guarantees the flexibility which is necessary for the `Element` to be an extension point of Kratos.

According to the previous comments the following methods are designed:

`CalculateLocalSystem` This method calculates all local system components. It takes a left hand side matrix and a right hand side vector to put its result in them. `ProcessInfo` is passed to

provide the analysis parameters.

`CalculateLeftHandSide` This method calculates only the left hand side matrix. It takes a matrix to put its result in it. `ProcessInfo` is passed to provide the analysis parameters. `ProcessInfo` which provides analysis parameters.

`CalculateRightHandSide` Calculates right hand side component of local system. It takes a vector to put its result in it. `ProcessInfo` is passed to provide the analysis parameters.

`Element` also has to provide assembling information for `Strategy`. It has to provide the corresponding position of each local system row and column in the global equation system. `Strategy` then uses this information to properly assemble the local matrices and vectors in global equation system. This information comes from the `Dof` associated with each row or column of local system. `Strategy` by itself cannot find these equation because each `Element` may have different dofs and also may arrange them in different order. For example an structural element can define a local system with all displacement's components for the first `Node` then second `Node` and so on. Another structural element can arrange its local system by placing first the displacement's $x$ component of all `Node`s, then the $y$ components and their $z$ components. So an `Element`'s task is to give its local system arrangement to `Strategy`.

`Element` can give an array of `Dof`s with the same order that local system is constructed, or get their associated equation ids and give them to `Strategy` as an array of indices. `Strategy` uses these indices to assemble a given local system into the global equation system. This part of `Element`'s interface consists of two methods:

`EquationIdVector` This method is used to directly give the global equation id related to each row or column of local system matrices and vectors. For example giving a vector $i = \{24, 5, 9\}$ means that the first element of the right hand side vector must be added to the 24th row of global system's right hand side or component $k_{23}$ of the local stiffness matrix must be added to the component $K_{59}$ of the global left hand side matrix. A `ProcessInfo` is passed to this method to provide any addition parameter needs for this procedure.

`GetDofList` This method gives `Element`'s `Dof`s in the same order as local system is defined. `Strategy` can use this list to extract the equation id related to each local position and then use them to assemble the `Element`'s local system components correctly. Like the previous method, it takes a `ProcessInfo` object as its argument which can be used to pass any additional information needed for this procedure.

It can be seen that both methods take `ProcessInfo` as their argument. This argument seems to be redundant but in practice there are situations that is really necessary. For example in solving a fluid using a fractional steps method [26], `Element` must know which is the current fractional step for providing the corresponding list of dofs or equation ids. Passing a `ProcessInfo` to these methods provides these additional parameters and guarantees the generality of the design.

Here is an example of `EquationIdVector` implemented for a generic structural element which can be used with different geometries in 2D and 3D spaces:

```
virtual void EquationIdVector(EquationIdVectorType& rResult,
                              ProcessInfo& rCurrentProcessInfo)
{
  unsigned int number_of_nodes = GetGeometry().size();
  unsigned int dimension = GetGeometry().WorkingSpaceDimension();
```

```
  unsigned int number_of_dofs = number_of_nodes * dimension;

  if(rResult.size()!= number_of_dofs)
    rResult.resize(number_of_dofs);

  for (int i = 0 ; i < number_of_nodes ; i++)
  {
    unsigned int index = i * dimension;

    rResult[index] =
      GetGeometry()[i].GetDof(DISPLACEMENT_X).EquationId();
    rResult[index + 1] =
      GetGeometry()[i].GetDof(DISPLACEMENT_Y).EquationId();
    if(dim == 3)
      rResult[index + 2] =
        GetGeometry()[i].GetDof(DISPLACEMENT_Z).EquationId();
  }
}
```

The third category of methods are devoted to calculating elemental variable which are used mainly for calculating post-analysis results. A simple example is calculating stresses in structural elements after obtaining the displacements in the domain. Users can ask `Element` to calculate additional results using its internal information and solving results. A flexible interface here is very important and can increase the generality of the code. A VBI can be used to provide a clear but flexible interface for these methods. Element developers can define a set of methods to calculate variables related to its `Element` and users can use them for specifying the variable they wants to calculate. Similarly to methods for calculating the local system, the result is passed as an additional argument in order to increase the performance and eliminate the redundant time necessary to create temporaries. Two sets of methods are defined for this task:

`Calculate` Can be used to calculate elemental variables. These methods are overloaded to support different types of variables to be calculated. Element developer can override them to implement the procedure necessary to calculate each elemental variable. They take the variable which a user wants to be calculated as their argument. If the variable is supported by `Element` it will give the result and it will do nothing if the variable is not related to this `Element`. The result also is passed as an additional argument to increase the performance and eliminate the overhead produced by creating temporary objects. Passing `ProcessInfo` to these methods provides a generic way to pass additional calculation parameters.

`CalculateOnIntegrationPoints` This set of methods calculate variables not for the whole `Element` but specifically at each integration point. The interface is the same as for previous methods. The variable to be calculated is given as an argument and the results as another argument. `ProcessInfo` provides any additional information necessary for calculation procedure.

Providing an standard way to access neighbors of `Element`s can be very useful for some algorithms. The problem is that for the rest of algorithms keeping the list of neighbors results in large overhead in total memory used. Keeping in mind the importance of memory efficiency in `Element`s these features are considered to be optional. So the first solution was to have arrays for neighbor `Node`s and neighbor `Element`s which are empty and fill them when they are necessary. This implementation was good but still the empty containers was producing memory overhead for simple `Element`s. In the current implementation these containers are omitted and neighbor `Node`s

and Elements are stored inside the elemental data container. In this way the overhead of empty containers are eliminated and the existing container is reused to hold this information. This solution can be used for any other feature that must be provided optionally but without any overhead for other Elements.

## 8.2 Conditions

Condition is defined to represent the conditions applied to boundaries or to the domain itself. In many codes conditions or specially boundary conditions are represented by an element with a formulation modified for boundary conditions. In Kratos also Conditions are designed very similar to Elements. They interact with Strategy in the same way as Elements. Strategy ask their local system components and also information for assembling process. The reason of using a different type and not Element itself is to clarify the different purpose of these two objects. In a usual finite element model, there are much more Elements than Conditions. For this reason some features that are considered to be too expensive in performance or memory consuming for Elements can be used for Conditions. Making Element and Condition two independent types allows additional features to be added to Condition without affecting Element.

### 8.2.1 Condition's Requirements

Condition like an Element is used to introduce new formulations into Kratos. So adding a new Condition must be an easy task and without great modification in code. Encapsulating all data and procedures necessary to calculate local matrices and results in one object and using a clear interface is required to achieve this objective.

A complex model usually has different type of Conditions in its boundary or domain. This requires Conditions to be compatible with each other in order to be assembled and solved together in a complex system. Another design point is to let users change Conditions or mix them in the model without problem.

Like Element, Condition has to have a very flexible interface due to the wide variety of algorithms and their different requirement. For example, most Conditions are applied to the right hand side component of the system but in some cases, like thermal radiation, they affect also the left hand side matrix of equation system. For this reason creating the interface only for right hand side component results in sever restriction in adding some Conditions. Conditions are not only different in local components, but also they need different data for they calculations. Hence a generic interface is necessary to guarantee the flexibility required for implementing different Conditions.

Condition must be easy to implement. Finite element developers are usually less familiar with advance programming language features and they like to focus more on their finite element developing task. For this reason excessive use of templates or other difficult concepts of programming language cannot be used for the Condition's implementation. The idea is to provide a clear and simple structure for a Condition to be filled by finite element developers easily.

For Condition performance is important but not so crucial as for Element. Its performance is important because it is usually called in very inner loops of global procedure. So any small fault in Condition's performance can cause large overhead in the program execution time. However its less important than the performance of Element because there are less Conditions in the model and the global overhead is less. So in designing Condition the intention is to avoid features producing bottleneck in the performance.

Memory efficiency is another design point to keep in mind. As mentioned before `Element`s have to avoid any redundant memory usage due to their large quantity in a model. Number of `Condition`s in a model usually is far less than number of `Element`s. This lets `Condition`s to provide features that are considered too expensive for `Element`s. However abusing memory by `Condition` can also produce a large overhead in memory usage and has to be avoided.

### 8.2.2 Designing Condition

`Condition` is very similar to `Element`, and hence the same methodology is used to design it. `Condition` is defined as an object which holds its data and calculates its local matrices and vectors to be assembled and also can be used to calculate local results after analysis. Defining `Condition` in this way isolates it from the rest of the code and helps towards its encapsulation.

Like `Element`s, `Condition`s must be designed to be implemented independently and added easily to Kratos in order to guarantee the extendibility of Kratos. Also they must be compatible with each other in order to let users mix them together in a complex model. The same strategy pattern used for `Element` is reused here. The figure 8.4 shows the structure for `Condition` applying the strategy pattern.

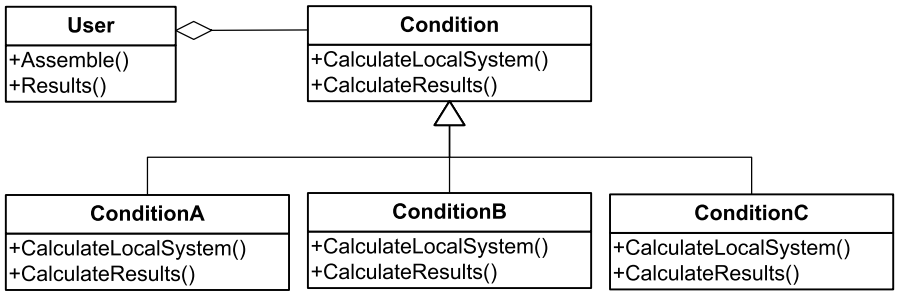



Figure 8.4: `Condition`'s structure using strategy pattern.

In this structure each `Condition` encapsulates one algorithm separately and also make them interchangeable as we want. The interface established by the `Condition` base class also make its derived class compatible with each other and enable user in mixing them together to model a multi-disciplinary problem.

`Condition` has a close relation to geometry. As explained for `Element`, deriving `Condition` from geometry, can increase the performance of geometries' data access but requires different `Condition`s to be implemented for a formulation applied to different geometries and also prevents `Condition` to share a geometry with `Element`s or other `Condition`s.

This structure reduces the flexibility of geometry and also produces unnecessary implementation overhead. So this structure is considered to be unsuitable because for `Condition` the flexibility is more important than a small increase in performance.

The alternative design is to use the bridge pattern. Introducing this pattern to our design, results in the structure shown in Figure 8.5.

This pattern allows each `Condition` to change its geometry and omits the strong relation of previous design. In this way less implementation is needed. Also having a pointer to geometry allows `Condition` to share its geometry with other `Condition`s or even with `Element`s without problem. The only drawback is the time overhead coming from pointer redirection in memory.

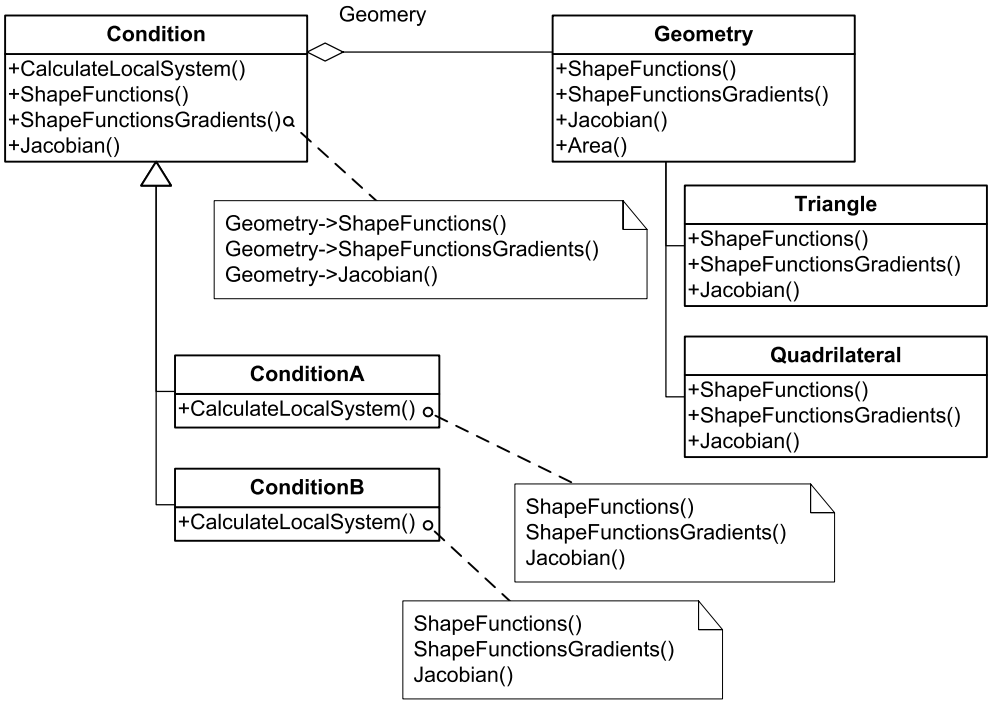



Figure 8.5: `Condition`'s structure using bridge pattern.

Having a pointer to geometry beside deriving from it produces an overhead in accessing geometries' data. There are other alternatives in designing `Condition`'s structure with enough flexibility and better performance but requires introducing advance features of programming language to `Condition` which are not acceptable in our design.

The interface of `Condition` is similar to the one designed for `Element`. Again here there are three categories of methods:

**Calculate Local System** The first set of methods are required to calculate local matrices and vectors.

**Assembling Information** These methods give information about the position of each row and column of the local system in the global system. This information comes from dof and `Condition` provide it by giving its dofs or just their equation id.

**Calculate** Is used to calculate any variable related to this `Condition` which usually are the results depending on gradients in the condition.

As described before in designing the `Element` interface, passing calculation parameters to `Condition` methods and getting the result as their return value produces a significant reduction the performance. To avoid this problem the result variable is also passed to each method. Passing the result vector or matrix by reference prevents the program from making temporaries and increases the performance. Also controlling the size of a given result matrix or vector and resize them if necessary can optimize the code performance.

`Condition` uses the same set of methods as `Element` to calculate the local system's matrices and vectors. `ProcessInfo` is used to pass any parameter which is necessary for calculating the local system in `Condition`. All methods are defined for working with dense matrix and strategies working with symmetric or other types of matrices must use a dense matrix to communicate with `Condition`. This part of the interface is defined by the following methods:

`CalculateLocalSystem` This method calculates all local system components. It takes a left hand side matrix and a right hand side vector to store the results and a `ProcessInfo` which provides the analysis parameters.

`CalculateLeftHandSide` This method calculates only the left hand side matrix. It takes a matrix to store the results and a `ProcessInfo` which provides the analysis parameters.

`CalculateRightHandSide` Calculates right hand side component of local system. It takes a vector to store the result and a `ProcessInfo` which provides the analysis parameters.

The second set of methods provide assembling information for `Strategy` which is the corresponding position of each local system row and column in global equation system. `Condition` can give an array of `Dof`s with the same order that local system is constructed, or get their associated equation ids and give them to strategy as an array of indices. Strategy uses these indices to assemble the local system into the global equation system. This part of `Condition`'s interface consists of two methods:

`EquationIdVector` This method is used to directly give the global equation id related to each row or column of local system matrices and vectors. A `ProcessInfo` is passed to this method to provides any additional parameter needs for this procedure.

`GetDofList` This method gives `Condition`'s Dofs in the same order as the local system is defined. Strategy can use this list to extract the equation id related to each local position and then use them to assemble the `Condition`'s local system components correctly. Like the previous method, it takes a `ProcessInfo` object as its argument which can be used to pass any additional information needed for this procedure.

Like `Element`, `Condition` uses its data container to store references to its neighbor `Node`s, `Element`s, or `Condition`s. This solution also can be extended to store the references to nearest `Element` or `Condition` in contact problems or other similar information.

## 8.3 Processes

Creating a finite element application consists of implementing several algorithms for solving different problems. In practice, each set of problems has their own solving algorithms. For example an steady state analysis algorithm is not the same as a transient algorithm. A one domain process is also different from a multi domain one and so on. While these algorithms are the heart of the code and flexibility and power of the code is depended on them, a good design to handle them in a generic way becomes very important.

A possible approach to handle algorithms in a finite element code is to provide some high level classes to handle different tasks in the code [33]. In Kratos, the `Process` class and its derived classes are defined to implement different algorithms and handle different tasks. Different processes may be used to handle a very small task like setting a nodal value to some complex one like solving a fluid structure interaction problem. Grouping some processes in a bigger one is also helpful specially to make a pack of small processes in order to handle a complex algorithm.

### 8.3.1 Designing Process

`Process` can be considered as a function class. `Process` is created and executed just like a function is called. The strategy pattern is used to design the family of processes. Figure 8.6 shows this pattern applied to `Process` structure.

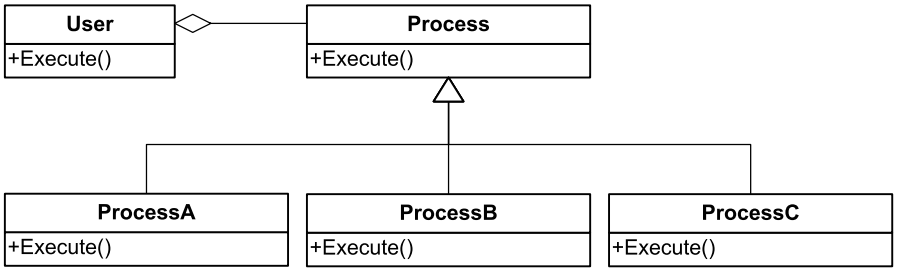



Figure 8.6: `Process` structure using strategy pattern.

Applying this pattern lets a `Process` to encapsulate an algorithm independently and also provide an standard interface which makes them to be replaceable with each other. Encapsulating each algorithm in one `Process` without modifying other parts of the code makes adding a new `Process` very easy and increases the extendibility of the library to new algorithms. The compatibility of processes with each other helps to customize the program flow and is useful in cases when user wants to interchange some algorithms.

Another feature to be provided by `Process` is the ability to combine different processes in one and use the resulting `Process` like a normal one. The composite pattern can be used to achieve this requirement. Applying this pattern to `Process` results in the extended structure shown in figure 8.7.

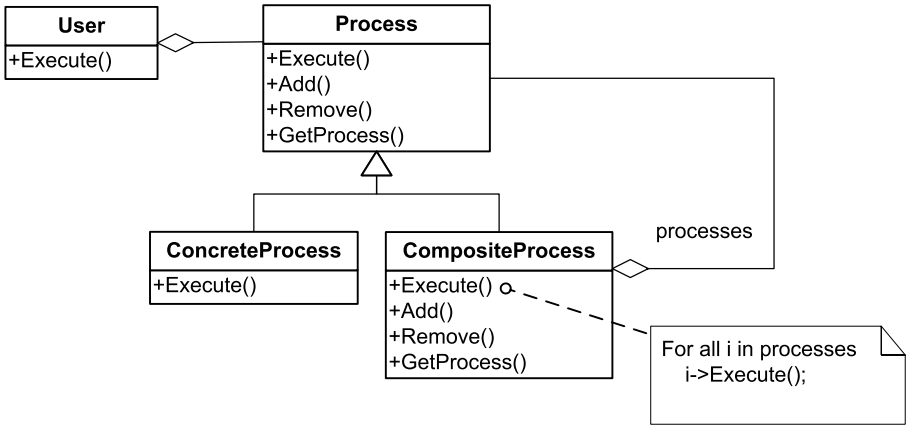



Figure 8.7: Applying composite pattern to the `Process` structure.

This structure allows users to merge different process in one and use it like an ordinary process.

In practice this structure is considered to be too sophisticated for our purpose. The composite pattern provides an interface for changing the children of each composite object. In order to simplify the implementation of new processes and the total implementation of the structure, the interface for changing sub-processes has been removed and `CompositeProcess` must get all its sub-processes with their other parameters at creation time. However this interface can be added in the future. Figure 8.8 shows the reduce structure.

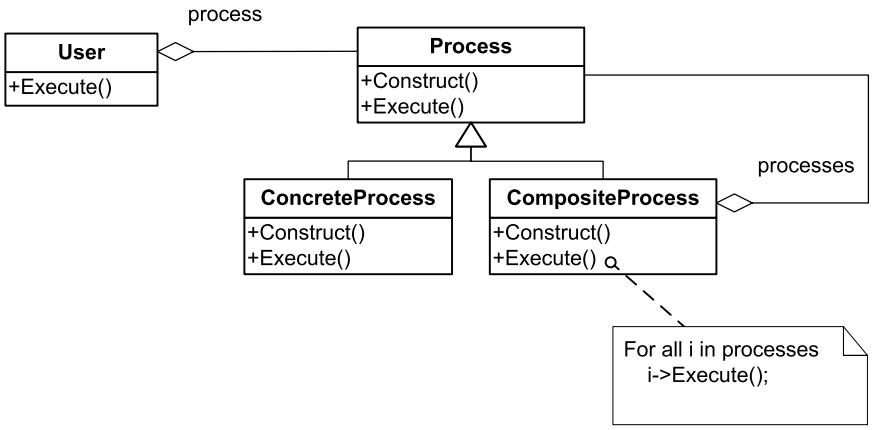



Figure 8.8: The reduced composite structure for `Process`.

The process interface is relatively simple. `Execute` method is used to execute the `Process` algorithms. While the parameters of this method can be very different from one `Process` to other there is no way to create enough overridden versions of it. For this reason this method takes no argument and all `Process` parameters must be passed at construction time. The reason is that each constructor can take different set of argument without any dependency to other processes or the base `Process` class.

## 8.4 Solving Strategies

After designing `Process` and its derived classes, we will focus in an important family of processes which are dedicated to manage the solving task in the program.

The `SolvingStrategy` is the object demanded to implement the "order of the calls" to the different solution phases. All the system matrices and vectors will be stored in the strategy, which allows to deal with multiple LHS and RHS. Trivial examples of these strategies are the linear strategy and the Newton Raphson strategy.

`SolvingStrategy` is derived from `Process` and use the same structure as shown in figure 8.9. Deriving `SolvingStrategy` from `Process` lets users to combine them with some other processes using composition in order to create a more complex `Process`. The strategy pattern used in this structure lets users to implement a new `Strategy` and add it to Kratos easily which increases the extendability of Kratos. Also lets them selecting an strategy and use it instead of another one in order to change the solving algorithm, which increases the flexibility of Kratos.

Composite pattern is used to let users combining different strategies in one. For example a fractional step strategy can be implemented by combining different strategies used for each step in

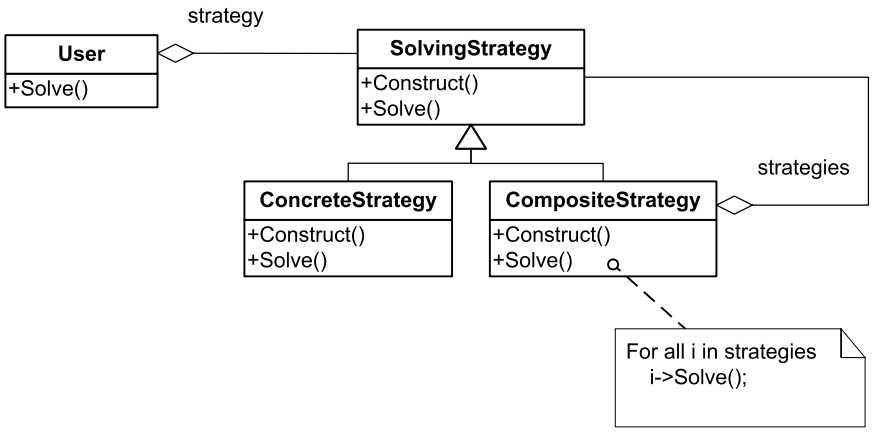



Figure 8.9: `SolvingStrategy` uses the structure designed for `Process`.

one composite strategy. Like for `Process`, the interface for changing the children of the composite strategy is considered to be too sophisticated and is removed from the `Strategy`. So a composite structure can be constructed by giving all its components at the constructing time and then it can be used but without changing its sub algorithms.

The interface of `SolvingStrategy` reflects the general steps in usual finite element algorithms like prediction, solving, convergence control and calculating results. This design results in the following interface:

`Predict` A method to predict the solution. If it is not called, a trivial predictor is used and the values of the solution step of interest are assumed equal to the old values.

`Solve` This method implements the solving procedure. This means building the equation system by assembling local components, solving them using a given linear solver and updating the results.

`IsConverged` It is a post-solution convergence check. It can be used for example in coupled problems to see if the solution is converged or not.

`CalculateOutputData` Calculates non trivial results like stresses in structural analysis.

Strategies sometimes are very different from each other but usually the global algorithm is the same and only some local steps are different. The template method pattern helps to implement these cases in a more reusable form. As mentioned before, this pattern defines the skeleton of an algorithm separately and defers some steps to subclasses. In this way the template method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. Applying this pattern to `SolvingStrategy` results in the structure shown in figure 8.10.

This structure is suitable when the algorithm is not changing at all but in our case the algorithm varies from one category of strategies to another. For this reason in order to reduce the dependency of the algorithm and its steps a modified form of the bridge pattern is applied to this structure. Different steps for solving template methods are deferred to two other objects which are not derived from `Strategy`: `BuilderAndSolver` and `Scheme`. Figure 8.11 shows this structure.

The main idea of using these two additional set of objects was to increase the reusability of the code and prevent users from implementing a new `Strategy` from scratch. In practice this

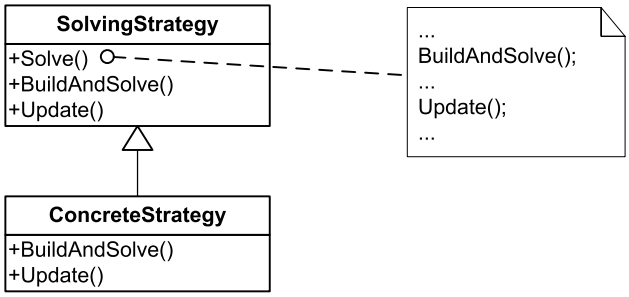



Figure 8.10: Template Method pattern applied to solving strategy.

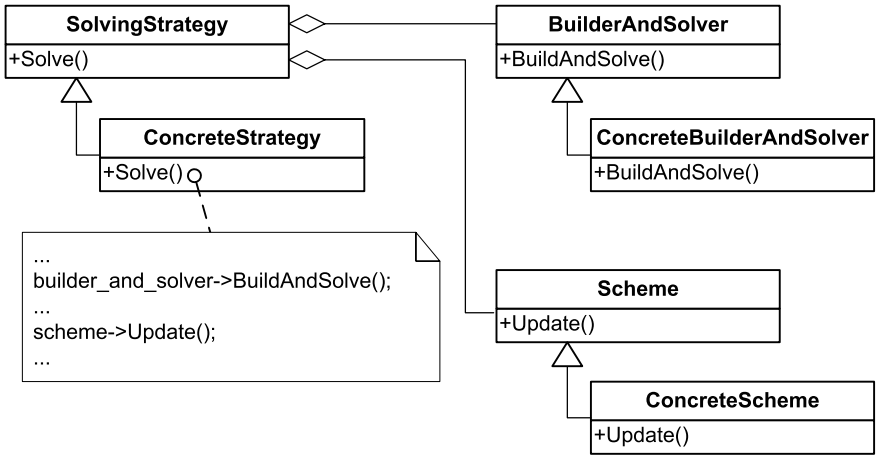



Figure 8.11: Deferring different parts of the algorithm to `BuilderAndSolver` and `Scheme`.

structure can support usual cases in finite element methodology but still advanced developers have to configure their own `Strategy` without using `BuilderAndSolver` or `Scheme`. For this reason in the current structure both approaches can be used to implement a solving algorithm.

### 8.4.1 BuilderAndSolver

The `BuilderAndSolver` is the object demanded to perform all of the building operations and the inversion of the resulting linear system of equations. The choice of grouping together the solution and the building step is not necessarily univocal. This choice was made in order to allow a future parallelization of the code, which should involve both the linear system solution and the Building Phase.

Due to its features `BuilderAndSolver` covers the most computational intensive phases of the overall solution process. This will clearly require low level tuning in order to ensure high performance. A typical user is not required to understand the implementation details for this class. Nevertheless the comprehension of the role of this object is necessary.

`BuilderAndSolver` needs a linear solver to solve its constructed equation system. In order to

give the possibility of assigning any linear solver to any `BuilderAndSolver` a bridge pattern is used to connect these two sets of classes. In this way `BuilderAndSolver` can use any linear solver available.

The interface of `BuilderAndSolver` provides a complete set of methods to build the global equation system or its components separately. It also provides methods for building the system and solving it or rebuilding just the left hand side or the right hand side and solve the updated equation system. This interface consists of the following methods:

`BuildLHS` Calculate the left hand side matrix of the global equation system.

`BuildLHS_CompleteOnFreeRows` Builds the rectangular matrix related to all free dofs, adding also the columns related to fixed dofs.

`BuildLHS_Complete` Gives the complete left hand side matrix regardless to fixed dofs.

`BuildRHS` Calculates and gives the right hand side vector of the global equation system. This method is useful in cases that left hand side matrix is the same for different solution steps but the right hand side is changing.

`Build` Builds the whole equation system. This method gives the possibility to calculate both sides at the same time and avoids duplicated calculation that must be done when calculating each component separately.

`ApplyDirichletConditions` In some strategies, for example for standard linear solutions, the Dirichlet condition can be applied efficiently by some operation over Dirichlet partition of the equations system. This can be done by this method.

`SystemSolve` Uses the linear solver to solve the prepared equation system.

`BuildAndSolve` Calling this method is equivalent to calling `Build` and then `SystemSolve` for most algorithms. It can be also used to implement algorithms that build the system while solving it, like the advancing front solution method.

`BuildRHSAndSolve` This methods is useful for updating just the right hand side and solve the equation system.

`CalculateReactions` Calculates the reaction at fixed degrees of freedom.

There are also several methods for initializing the internal system matrices and vectors and also to remove them from memory if it is necessary. `Strategy` can use this interface to implement its algorithm using any of the procedures defined above.

### 8.4.2 Scheme

`Scheme` is designed to be the configurable part of `Strategy`. It encapsulates all operations over the local system components before assembling and updating of results after solution. This definition is compatible with time integration schemes, so `Scheme` can be used for example to encapsulate the Newmark scheme. By the way definition is more general and can be used to encapsulate other similar operation over solution component.

According to the template method pattern the important steps of the solving procedure in usual finite element strategies is used to design the interface of scheme. Usually a finite element solving strategy consists of several steps like: initializing, initializing and finalizing solution steps, initializing and finalizing non linear iterations, prediction, update and calculating output data. Considering the steps mentioned before, the interface of `Scheme` is designed as follows:

`Initialize` This method is used for initializing `Scheme`. This method is intended to be called just once when `Strategy` is initializing.

`InitializeElements` Is used to initialize the `Element` by calling its `Initialize` method when `Strategy` is initializing.

`InitializeSolutionStep` `Strategy` calls this method at the beginning of each solution step. This method can be used to manage variables that are constant over time step. For example time-scheme constants depending on the actual time step.

`FinalizeSolutionStep` This method is called by `Strategy` at the end of a solution step.

`InitializeNonLinIteration` It is designed to be called at the beginning of each non linear iteration.

`FinalizeNonLinIteration` This method is called at the end of each non linear iteration.

`Predict` Performs the prediction of the solution.

`Update` Updates the results value in the data structure.

`CalculateOutputData` This method calculates the non trivial results.

## 8.5 Elemental Expressions

Finite element methodology usually consists of first converting the governing differential equation to its weak form, then its discretization over an appropriate approximation space, and finally the derivation of matrix forms as elemental contributions. Zimmermann and Eyheramendy [107, 39, 40, 38] have developed an environment for automatic symbolic derivation from the variational form to matrix form and integrate it into a unified environment with modeling tools [105]. Nowadays several computer algebra systems like *Matlab* [68], *Mathematica* [103], and *Maple* [66] can do this type of symbolic derivations. In Kratos the first part of changing the variational equation to weak form is dedicated to previous tools and only a set of tools is designed and implemented to help users converting their weak form to matrix form as elemental contributions.

Elemental expressions are designed and implemented to help users in writing their weak form expressions in `Element`. The main idea is to create a set of classes and overloaded operator to understand a weak form formulation and calculate the local matrices and vectors according to it.

For example in a simple heat conduction problem the governing equation is:

$$-\nabla^T \mathbf{k} \nabla T + Q = 0$$

where $T$ is the temperature over domain and $Q$ is the heat sources over domain. Converting this equation to its weak form results in the following equation [104]:

$$\mathbf{ST} + \mathbf{f} = \mathbf{0}$$

where the elemental matrix $\mathbf{S}$ is:

$$S_{ij} = \int_\Omega (\nabla N_i)^T \mathbf{k} \nabla N_j d\Omega$$

and the elemental right hand side vector $\mathbf{f}$ is defined as follows:

$$f_i = \int_\Omega N_i Q d\Omega + \int_{\Gamma_q} N_i \bar{q} d\Gamma$$

For an isotropic material the conductivity $k$ can be extracted from the integral and the resulting equation is:

$$S_{ij} = k \int_\Omega (\nabla N_i)^T \mathbf{I} \nabla N_j d\Omega$$

or:

$$S_{ij} = k(\nabla_i N_l, \nabla_j N_l)$$

This equation can be implemented in `Element` by the following code:

```
for(int i=0 ; i < nodes_number ; i++)
  for(int j=0 ; j < nodes_number ; j++)
    for(int l=0 ; l < integration_points_number ; l++)
    {
      Matrix const& g_n = shape_functions_gradients[l];
      for(int d=0 ; d < dimension ; d++)
        rLeftHandSideMatrix(i,j) += k * g_n(i,d)*g_n(j,d) * w_dj;
    }
```

Using elemental expressions the same formulation can be written in a simpler form as:

```
KRATOS_ELEMENTAL_GRAD_N(i,l) grad_Nil(expression_data);
KRATOS_ELEMENTAL_GRAD_N(j,l) grad_Njl(expression_data);

noalias(rLeftHandSideMatrix) = k * (grad_Nil, grad_Njl) * w_dj ;
```

It can be seen that the later form is conforming with the symbolic notation of equations which makes it much easier to implement. The overloading operators provided in C++ is the start point for implementing the code necessary to understand this notation, but simple overloading results, poor performance due to the redundant temporary objects that creates. Expression template technique described in section 3.4.2 can be used to convert above expression to previous hand written form automatically. Template metaprogramming described in section 3.4.2 also is used to impose the tensorial notation. All these techniques are used to evaluate the symbolic notation and generate an specialized code for each case. Here are examples of vector ”,” overloaded operators:

```
template<unsigned int TIndex1,
         unsigned int TIndex2,
         class TExpression1,
         class TExpression2,
         class TVectorType1,
         class TVectorType2>
typename result_type
  operator,(Elemental1DExpression<TIndex1,
                                  TExpression1,
                                  TVectorType1> const& rVector1,
            Elemental1DExpression<TIndex2,
                                  TExpression2,
                                  TVectorType2> const& rVector2)
  {
    return outer_prod(rVector1()(), rVector2()());
```

```
  }

template<unsigned int TIndex1,
         class TExpression1,
         class TExpression2,
         class TVectorType1,
         class TVectorType2>
double
  operator,(Elemental1DExpression<TIndex1,
                                  TExpression1,
                                  TVectorType1> const& rVector1,
            Elemental1DExpression<TIndex1,
                                  TExpression2,
                                  TVectorType2> const& rVector2)
  {
    return inner_prod(rVector1()(), rVector2()());
  }
```

The first overloaded version will be used in cases when two vectors have different indices and implements an outer product of these two vectors. While the second version will be used when two given expressions have a same index and implements an inner product of these two vector. It is important to mention that the first version returns the expression and not the calculated matrix and uses the expression template technique to optimize its efficiency.

There are similar operators implemented to handle different cases of matrix operations depending on their indices. Also the integration over domain is added for simplifying the elemental expressions even further.

In the current version of Kratos, elemental expressions are still in experimental phase. However some benchmarks have shown that their efficiency is comparable with hand coded `Elements` as supposed to be.

## 8.6 Formulations

Kratos was designed to support elemental approaches in finite element methods. For some problems elemental approach results to be less suitable than other approaches like nodal formulations. `Formulation` is defined as a place for implementing all these approaches. `Formulation` is not implemented yet, but is considered to be one of the future features of Kratos.

Chapter 9

# Input Output

In general most of the finite element applications have to communicate with pre and post processors ,except in some special cases in which the application generates its own input. This makes the input output (IO) an essential part of the application.

In this chapter, first different approaches in designing application's IO are discussed and a flexible and generic IO structure is presented. It follows a part dedicated to interpreter writing, which consists of small introduction to concepts and also brief explanation on the use of related tools and libraries. Next the use of Python as the interpreter is described and the reasons of using Python are explained. Finally a brief description of using boost python library is given.

## 9.1 Why an IO Module is Needed?

A typical finite element procedure consists of getting data from input sources, analyze it and send the result to an output. There are some applications which use the embedded IO structure. This means that they have their IO routines implemented in subroutines where an IO operation is needed. For example, element reads its properties directly from input file when they are needed and so on.

This approach is quite simple and easy to implement and in some cases eliminates some part of data storing overhead. However some cases exist in which an embedded IO approach introduces some difficulties in implementations or restrains the flexibility of the program.

In complex problems previous simple scheme changes to a repeating or multi input output scheme. These changes in strategy may introduce new IO statements, change some of them or invalidate some existing ones. This may result in many IO methods in different parts of the code which basically do the same things but with different objectives. Creating an IO module and use it in all of these statements help us to unify these efforts and simplify the maintenance of the code.

For a finite element program the possibility for connecting different programs is a great added value. The use of different pre and post processors or connecting different finite element applications are some typical examples. Connecting to each program means reading its output with given format, (in the case of pre processors) and generating their inputs also in their recognizable format (in the case of post-processors). It is obvious that each program may have an incompatible format with others. Using an embedded IO approach causes the program to be very rigid and difficult for extending to any new type of IO. In this approach a global revision of the code is needed to add any new interface and all the IO statements must be modified to include a new format. Placing

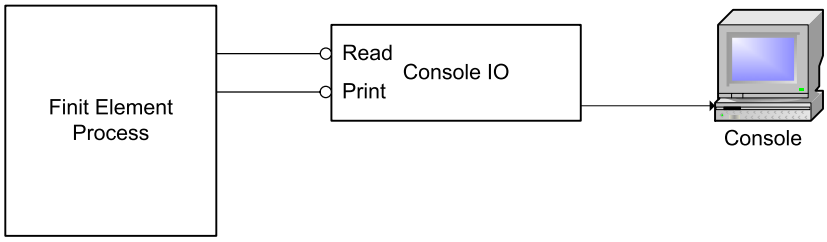



Figure 9.1: A Console IO interface

an IO module in the middle can solve this problem. Any request from inside or outside passed through IO, which is in charge of translate an outside format to inside one and vice versa. Now any new connection can be added by a new IO object.

Creating an IO module eases the team developing of an application. Without it each developer may put an IO statement in its part and cause conflicts in using files with others. In the same concept, easy file managing is another reason to use an IO module. IO module can open the file once, work with it and also close it at the end of work. If there is some problem with file opening it may report and try again. Finally handling multiple files is also simplified in this way.

As a conclusion it is useful to have a separate and robust IO module to handle the input and output tasks of the program for all different scheme with any format without problem. In this way flexibility guaranteed and extendibility can be achieved.

## 9.2 IO Features

Before starting with designing IO it is important to classify different aspects and features of IO for different programs. This helps to collect the IO features needed for a generic multi-disciplinary program.

### 9.2.1 IO Medium Type

People working in finite element analysis area are used to have files as input and output medium. This is correct for many cases but it is not always true. In general, an IO medium can be a file, some console stream, sockets for network communication or any other medium. Sometimes the difference in media comes from platforms and operating systems. For example opening a file and writing to it in Linux can be different from Windows. This make them virtually, more from implementation point of view, different media to interact.

This point of view to IO creates a set of open questions to be answered to before the designing phase. Does IO want to interact just with one type of medium or more? Is it supposed to be extendable in term of interacting with new media in future.

Our design can be depended on this questions. Working with just one type of media simplifies the IO interface and reduce implementation effort respect to the multiple media support. This difference can be large or small depending on the design, implementation and also to the type of supporting media and the way they are deferent. Let's make an example, considering a console IO as the first medium to interact. An interface to this medium just needs a `Read` and `Print` method to communicate. Assuming here that the console stream is always available. Figure 9.1 shows this simple interface.

Now let us add a file as a new medium. Here `Read` and `Write` methods provided for file accessing. Unfortunately a file is not always available to read or write as console is. We need to open it before any access and also close it at the end of procedure. This nature of file IO introduces two new methods to our interface, `Open` and `Close`. Figure 9.2 shows the interface for file IO.

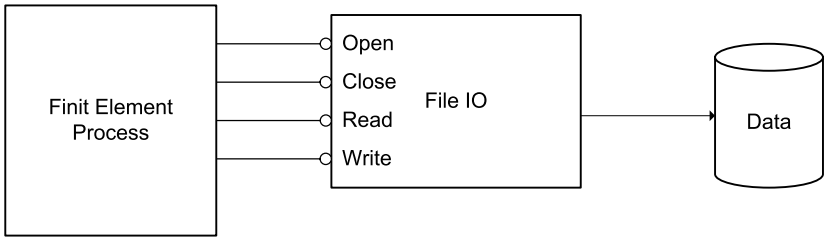



Figure 9.2: A file IO interface

Now, for having a multi-media IO a union of given interfaces is provided to interact with both media without problem.

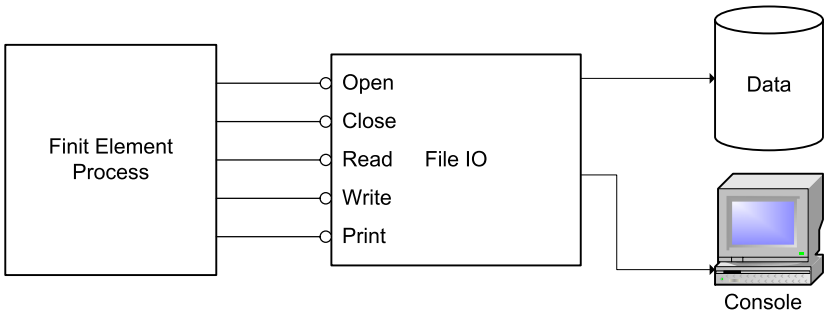



Figure 9.3: A multi-media interface

It is obvious that having the file interface, and unifying the `Print` and `Write` methods, no extra method is needed to handle the console IO. The intention of this example was to show the difference nature of each medium and not a serious design problem.

Handling each new medium may introduce the need for some new methods in interface, and making it extendable requires adding new layer to it. In other words, to do this another level of encapsulation is needed to separate different IO modules while keeping the established interface for all of them. Now, let's redesign our previous example and see how it can be organized to be extendible. Figure 9.4 shows the new structure.

Having above structure, any new medium can be supported using an IO interface via the new encapsulation level as shown in figure 9.5

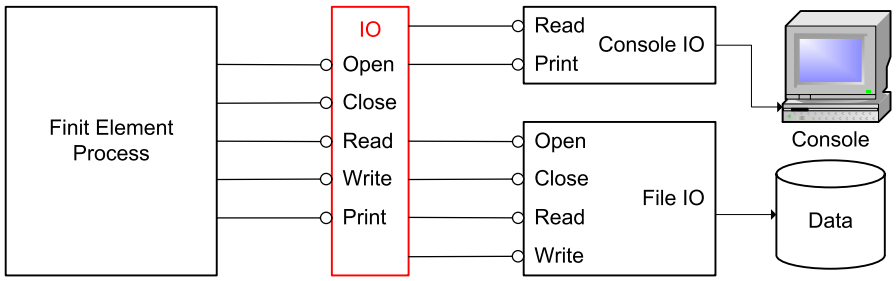



Figure 9.4: An Extendible IO interface

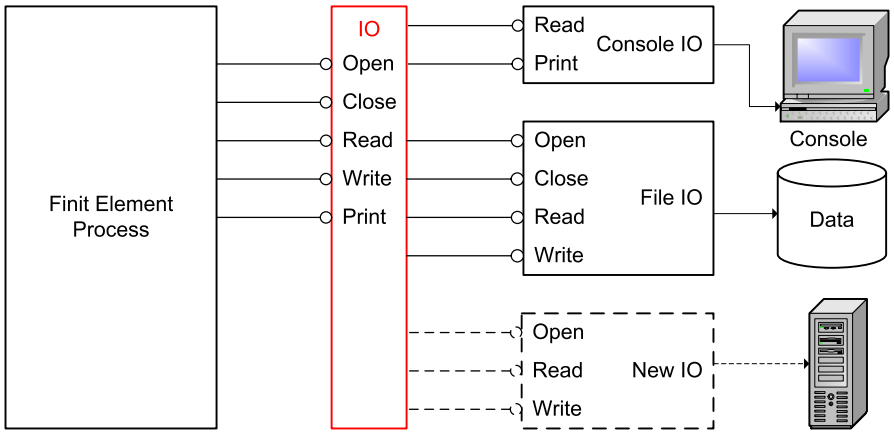



Figure 9.5: Extended IO

### 9.2.2 Single or Multi IO

A simple finite element procedure consists of getting data from an input source, analyze it and send the result to an output medium as shown in figure 9.6. A static structural analysis program is a good example of single IO.

This simple scheme also can be applied to some more complex problems in which there are various IO statements for certain points of program flow. Transient thermal problems, fluid dynamics and structural dynamics are typical examples of these type of solutions. In all of them program first reads the model data, then starts analyzing and meanwhile writes results for each time step. From the designing point of view this scheme is similar to the previous one as the IO statements are always the same. There are no changes due to the algorithm in reading or writing points.

In advanced problems there are some situations where IO statements change due to the nature of the algorithm. In other words the algorithm reads data or write them depending on the state of the problem. For example looking to some convergence criteria or displacement norms and not only in some certain points like the beginning of the analysis or the end of time steps. Optimization and interaction algorithms fall in this category. This situation implies more encapsulation of IO in time of designing. The reason is the extra flexibility needed to deal with these situations which

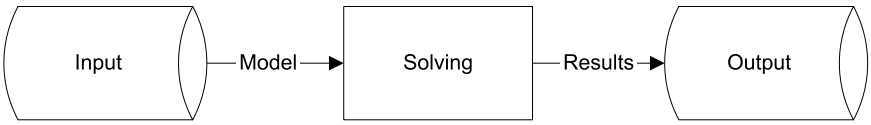



Figure 9.6: A single IO procedure

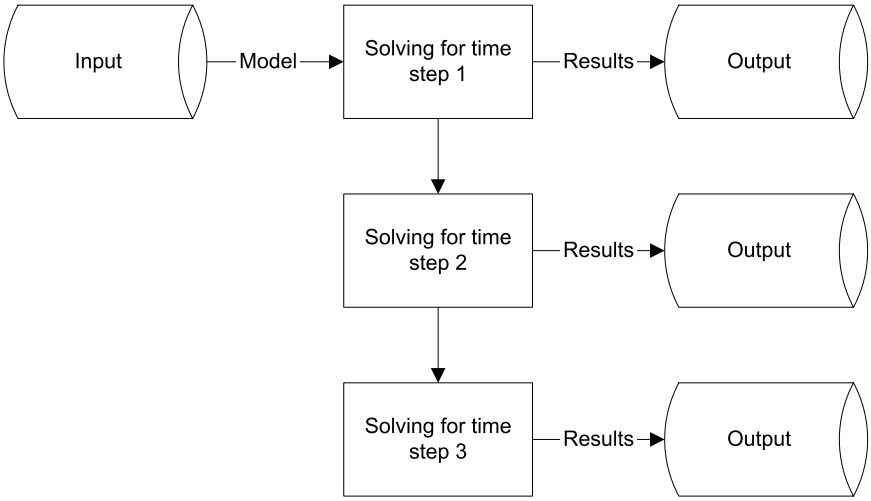



Figure 9.7: Repeating IO procedure

causes IO to be more independent.

### 9.2.3 Data Types and Concepts

The main task of IO is transferring data. A finite element program usually works with different types of data, like double, vector, matrix, etc. Also each data represents a concept like displacement, temperature, node's id and so on. Again there are decisions to be taken respect to data types and concepts to be supported. Which data types needed to be supported by IO? Is it needed to be extendible for new types? Does it supports new concepts?

These decisions are related to formats. Also they affect deeply the implementation of IO. From the designing point of view interfaces are affected by these decisions. Now, let us present some examples to explain these relations.

A simple thermal application is a good example of rigid IO. It has to read integers, doubles, vectors and matrices from input for concepts such as nodal information, elements information, properties like thermal conductivity and temperature and thermal flux as conditions and initial values. Output handles doubles and vectors needed to write temperature and fluxes. So in this case all the types are known and all the concepts are also defined as shown in table 9.1

It has to be mentioned that one can just handles doubles and treat all above types by their components as doubles and simplify more the IO. For the input part a column based format is well

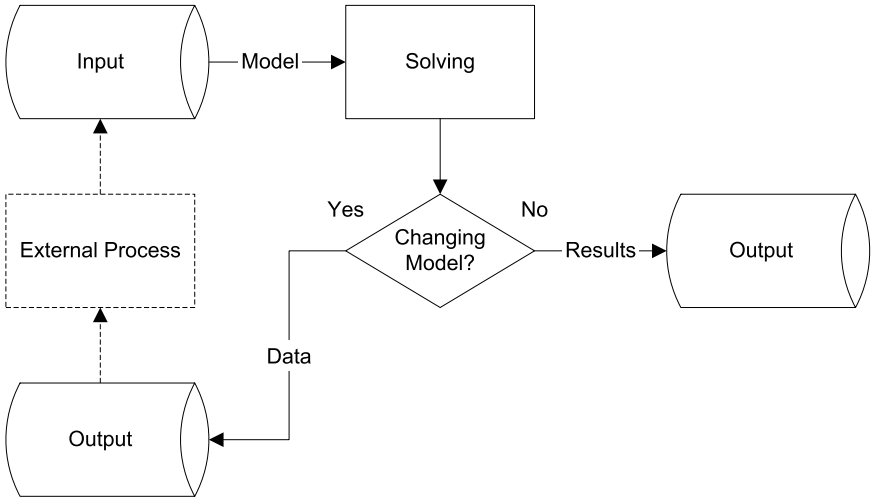



Figure 9.8: Complex IO procedure

| Types | Concepts |
|---|---|
| int | Id, Connectivity |
| double | Temperature |
| vector | Thermal flow |
| Matrix | Thermal conductivity |

Table 9.1: Thermal application types and concepts

suited and also makes implementation very simple.

```
# Format: NodeId X Y Z Temperature IsFixed
1 0.00 2.00 0.00 0.00 0
2 1.00 1.00 0.00 1.00 0
3 2.00 4.00 0.00 0.00 0
4 3.00 5.00 0.00 0.00 0
5 4.00 0.00 0.00 2.50 1
......
```

Implementation of IO for this example can be done easily and extremely efficiently by normal streaming or line by line scanning. Finally the interface can be defined very simply (although is very rigid) while all the types and concepts are known.

```
Read(NodesArray Nodes, ElementsArray Elements)

Write(vector Temperature, matrix ThermalFlows)
```

A more sophisticate interface can be designed to provide more control on IO sequences. for example

```
Read(NodesArray Nodes, ElementsArray Elements)
```

```
ReadNodes(NodesArray Nodes)

ReadElements(NodesArray Nodes, ElementsArray Elements)

Write(vector Temperature, matrix ThermalFlows)

WriteTemperature(vector Temperature)

WriteThermalFlow(matrix ThermalFlows)
```

Next example is a laplacian domain application. This refers to any problem govern by laplace equation like thermal, potential flow, low frequency electromagnetic, seepage problems or any combinations of them and solve it. In this example concepts are changing from one problem to other, or sometimes from one element to another, but the types of inputs are always the same. The input format gets more complex respect to previous example because concept definitions must be added to it. Here a tag mechanism helps to distinct the different concepts to be read.

```
# Format: NodeId X Y Z initialvalues IsFixed
1 0.00 2.00 0.00 VELOCITY_X 0.00 0
2 1.00 1.00 0.00 VELOCITY_X 1.00 1
3 2.00 4.00 0.00 TEMPERATURE 0.00 0
4 3.00 5.00 0.00 TEMPERATURE 0.00 0
5 4.00 0.00 0.00 TEMPERATURE 2.50 1
......
```

Flexibility in concepts introduces an extra cost in the implementation. A lookup table is needed to handle reading different variables via the names given and assign them internally. The lookup table to be used here can be a simple one with each variable name and their unique indices as shown in table 9.2.

| Name | Index |
|---|---|
| TEMPERATURE | 0 |
| VELOCITY_X | 1 |
| VELOCITY_Y | 2 |
| THERMAL_FLOW | 3 |
| THERMAL_CONDUCTIVITY | 4 |

Table 9.2: A sample part of lookup table

The interface design also must be changed due to these new features. First an initialize method is needed to pass the lookup table and setting the IO. Then `Write` method must be modified to get the variable to write as its argument. This is necessary because in this example the variable is not always Temperature and its Flow and the name of method cannot depend on it.

```
Initialize(Table LookupTable)

Read(NodesArray Nodes, ElementsArray Elements)

Write(string VariableName, double Value)
```

```
Write(string VariableName, vector Value)

Write(string VariableName, matrix Value)
```

Or the more complete version:

```
Initialize(Table LookupTable)

Read(NodesArray Nodes, ElementsArray Elements)

ReadNodes(NodesArray Nodes)

ReadElements(NodesArray Nodes, ElementsArray Elements)

Write(string VariableName, double Value)

Write(string VariableName, vector Value)

Write(string VariableName, matrix Value)
```

It can be seen that in this interface `Write` is overloaded for all different types handled by the output.

The final example is a generic IO module which the concepts and types both are extendible to new ones. In this case the type information and some basic operations must be included in the lookup table. This complicates even more the situation. To see this let us add a type extendibility to the previous example and use it for introducing a complex number to IO.

IO to handle new types needs to know how to perform some basic operations over them. For example, how to convert and string to them and viceversa, and also how to create them. So for each new types is necessary to give it some helper functions. These functions will help IO to manipulate the new types. These functions releasing IO from knowing anything about the new types. One can group all these helper functions in a TypeHandler class and pass it once to IO. Figure 9.9 shows this structure.

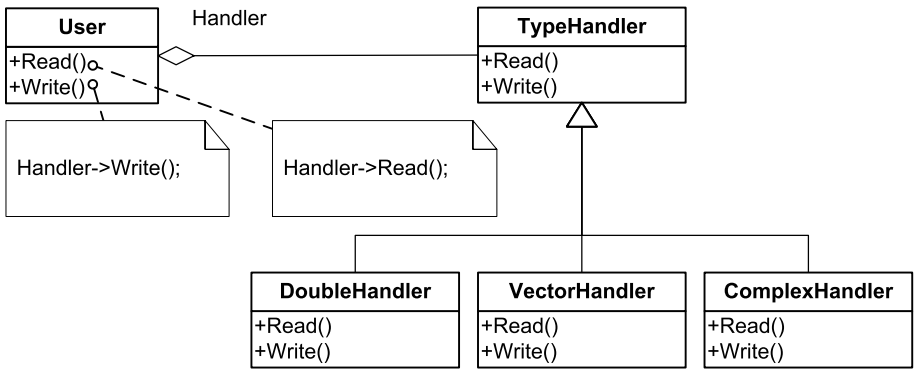



Figure 9.9: Using handlers to extend type supporting

The previous interface also changes due to this new mechanism. For example writing an interface can be changed to this new form:

```
Write(string VariableName, void* Value,
      TypeHandler& ThisTypeHandler)
```

Now, the `Write` method does not know about the type of the variable and just gets a void pointer and then handle it by a given type handler. This solution will work, but it is not type safe and leaves a possibility for serious errors. An example is to pass an integer value and a matrix type handler. This make `Write` method to write garbage and `Read` method may cause the system to crash by violating the memory. There is a more elegant way to handle this problem using templates. In this way the casting to void pointer is not necessary and the type of given data will be checked in compiling time.

```
template<class TTypeHandler>
void Write(string VariableName,
       typename TTypeHandler::DataType& Value,
       TTypeHandler const& ThisTypeHandler)
```

Going one step further, this TypeHandlers can be added to the lookup table for making this mechanism more automatic. In this manner, IO for each concept checks the lookup table as before and gets all the tools to handle this concept, even if is a new type of information.

It is important to mention that a variable base interface greatly helps in overcoming all these complexities and in designing a generic interface maintaining clarity and flexibility as explained in a later section.

### 9.2.4 Text and Binary Format

A text file is a sequence of characters stored as a file which depend on its content is human readable. Generally these files contains ASCII characters, where ASCII (American Standard Code for Information Interchange) is a character encoding based on English alphabet.

Writing strings to a text file is simple because there is no conversion needed. But for writing a number, first it has to be converted to a string format and then it can be written to the file. For example to write an integer variable which stores `1234567890` as its value, first it has to be converted from its binary form in memory which is `0100 1001 1001 0110 0000 0010 1101 0010` to its representative sting which is `"1234567890"` and then stored in file (Figure 9.10).

In time of reading again strings are read directly but numerical values must be converted from the string (Figure 9.11).

Binary format on the other hand, works with the binary value of each object and not with its representative string like a text file. In this manner a string or a number dump in the same way to the file. For example for the same above integer variable it is just enough to write the binary value 0100 1001 1001 0110 0000 0010 1101 0010 to the file directly (Figure 9.12).

Similarly for the reading just the sequence of bytes are read and put directly in the variable (Figure 9.13).

The advantage of the text format is its human readability. This makes it a good choice for small and academic applications because the aspect of input data can be verified and even changed manually and the output can be checked without any post-processor. For example to see if all results are zero. Disadvantage of this format is its latency time in read or write numbers due to the conversion time from and to its representative string. Another disadvantage is the large overhead in storing data size for numerical data. This comes from the fact that in most cases the representative string of a number occupies more memory than the number. Looking to our previous example, the integer `1234567890` occupies 4 bytes in memory like any other integer. Converting this to a string takes 10 bytes, one byte per character, without null at the end which is 150% overhead. However for small numbers, less than 4 digits, the representative string is smaller but

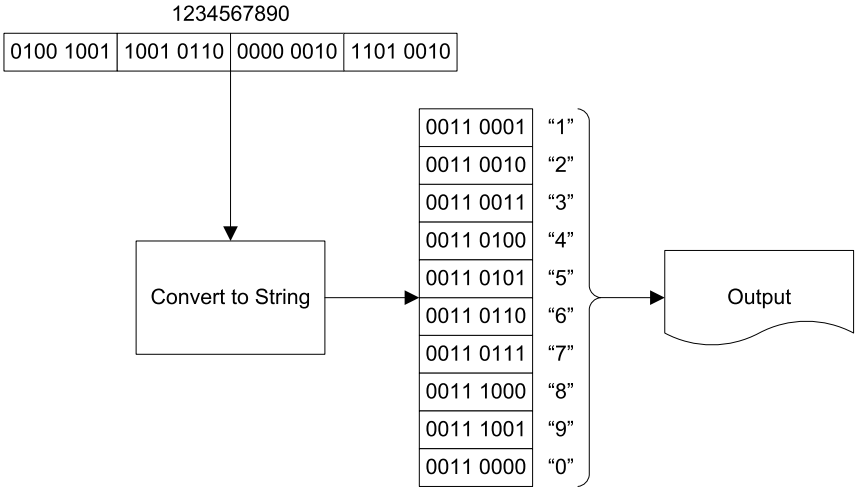



Figure 9.10: Writing a number to text format

in general finite element data usually consist of longer numerical data. These two problems make difficult the use of text format for very large problems in finite element applications.

The first advantage of using binary format is the direct read and write of data which increases the performance respect to text file format. Having no overhead in stored data respect to their real size is another advantage of using the binary format. These two advantages make this format a good choice for large problems data storing. Binary format is not human readable and using it as an input our output format complicates the debugging process and prevents users from manual checking of data for any obvious error, like having all coordinates equal to zero or so on. Finally, because there is no interest to open and read binary files, they can be compressed to occupy even less space than they normally need.

Using Ascii or binary format affect more the implementation details than its design. The interface and structure can be the same and the implementation details can be hiden via IO module. So using one or the other will not affect the implementation costs and the choice depends only on which is more suited to our needs.

### 9.2.5 Different Format Supporting

The previous section started with formatting concepts by describing two different category of formats, text and binary. In this section the concept is viewed from the format supporting point of view.

Format is the way data is represented and organized in a medium. Any variation in the data representation way or its organization creates a new format of data. As described in the previous section text and binary formats are different in the way they put their data. In the same manner, changing data organization in each one causes new format either binary or text.

Nowadays, a large number of data formats are defined and used by programs in different areas. Some of them are popular while many of them are just for specific programs. Also each different format has some advantages and disadvantages as seen before for the text and binary formats. All these aspects makes it necessary to take some decisions before starting the implementation. Using an existing format or creating a new one? Supporting an additional format or one is just enough?

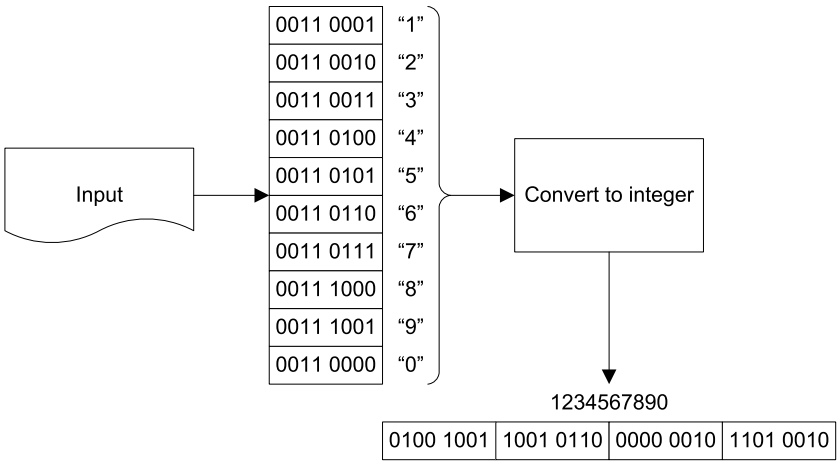



Figure 9.11: Reading a number from text format

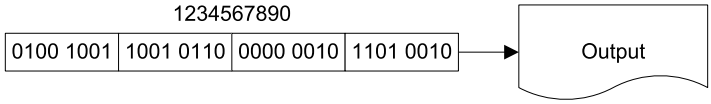



Figure 9.12: Writing a number as binary format

Can IO be extendible to support a new format in the future or not?

Most these decisions affect the design and implementation in the same way as for supporting media. The first choice is to support only one format. Choosing this option makes both design and implementation very simple. The interface for this case can be very simple without any argument or method to specify the format. Implementation also is simple while there are neither switch and case to handle a given format and there is not a duplicated method for different formats. Figure 9.14 shows an example of interface for an IO which supports only text formats.

It is important to mentioned that the simplicity of implementation mentioned above is respect to global design aspects and not the cost of handling each specific format which may cost a complete parser to be implemented.

To take advantage of more than one format, the previous design must be modified to handle each format separately. Interface must provide some way for users to set their format. A simple approach for small amount of formats to handle is to create separate methods. This leads a very clear interface and in some cases eases the implementation. This design from one side keeps each format support implementation separate and hence it increases the maintainability of the code. However it makes it more rigid and static. Figure 9.15 shows an example of this interface for a dual format supporting. The clarity and readability can be seen, while it is obvious that for supporting some more formats this approach is not suitable.

There is a more elegant approach which just passes a flag through interface to identify which format is being used. In this manner the interface is clear and adding a new format is easy while using it via hierarchy is not very helpful. Figure 9.16 shows the previous example with a new

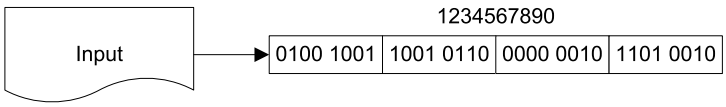



Figure 9.13: Reading a number from Binary format

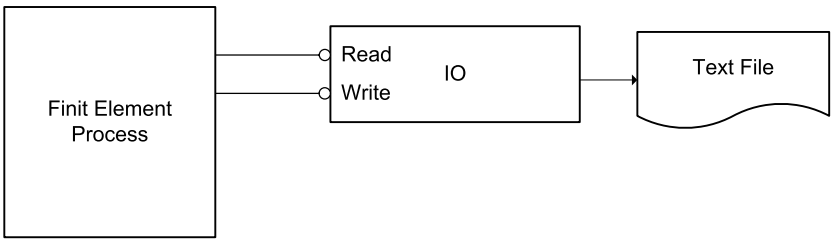



Figure 9.14: IO single format example

interface. Now the interface is simple and more dynamic than before.

Let us go now one step further and give flexibility to add new formats without changing previous ones. This can be done with the same methodology as before for the extendibility of media. A new layer of encapsulation takes place in the IO structure to unify the interface and the support for each format is encapsulated separately. This design provides a flexible structure which adding new format will not change any other part of the code and grantees the extendibility. This design imposes that all supported formats must use a unique interface. So to complete this structure a generic interface needs to handle all formats in the same way without critical restrictions. Reusing previous examples for applying the new design, results in a more flexible structure. Figure 9.17 shows this new structure.

Now supporting a new format implies adding another part without any modification of other the IO parts. Figure 9.18 shows the extended version.

### 9.2.6 Data IO and Process IO

It is usual for single purpose finite element applications to get data from the input, process it and write the results on an output. These applications always use their own implemented algorithm to solve the problem. They also give some options to the user to modify some aspects of algorithms via input data but not the algorithm itself. A typical input for these programs contains nodes, elements, conditions, properties and some algorithm parameter like convergence criteria, maximum number of iterations and so on. Some other codes also give the possibility to choose some parts of the algorithm by some given options, like changing the solver, static or dynamic strategies, etc.

For more complex problems and for multi-disciplinary programs the previous approach is too rigid to be applicable to all problems. For example for a thermo-mechanical application sometimes the mechanical solution is wanted not from the beginning, but from a certain time. So a mechanism is necessary to start solving the mechanical part from a certain time to avoid unnecessary calculation overhead. Another example is a fluid structure interaction problem, which may cause the program to change its global strategy depending on the type of structure and fluid interacting

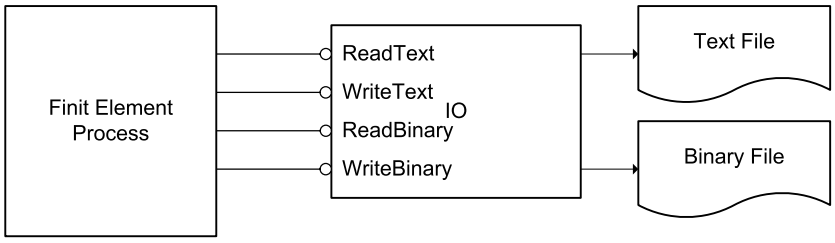



Figure 9.15: IO dual format example with specialized methods

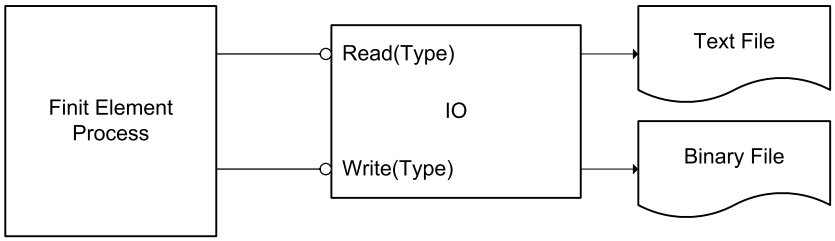



Figure 9.16: IO dual format example with passing type as argument

with each other. All these makes developers to implement their codes to get not only data from users but also the process necessary for the solution.

The first approach is to create a semi-language like input data which controls the order of execution of code blocks and also gives option to add processes in specific statements of algorithms. In this manner the user can define the global layout of the program flow by choosing when to read, write, solve, etc. via the input file. Also it can put some extra process in certain places like at the beginning of each time step and so on. Limiting the flexibility of modifying a program's algorithm imposed by this approach comes from two facts. First the lack of a good interpreter in the input part makes it impossible to introduce a whole algorithm. Second is the internal design of the code which is not well split and can limit the way one can put different algorithms together in order to create a new algorithm.

Usually codes written in fortran use previous approaches to handle different algorithms due to their limited facilities to write an interpreter.

Another approach is to implement a high level language interpreter as input in order to manage the global flow of the program. This gives a great flexibility to the code and new algorithms can be easily implemented. A working example of this approach is the Object Oriented Finite Elements method Led by Interactive Executor (OOFELIE) [77] developed by Cardona, *et al.* [25, 53, 54] which has its own command interpreter to deal with different algorithms in multi-disciplinary problems.The advantage is the flexibility in input language syntax. One can define a very minimal language which adopts well to its needs. This can give a clear and powerful input format for certain type of applications. The backward of this approach is the implementation cost. Writing a good interpreter is hard work and maintaining it is even worse. C++ programmer are more motivated to write their own interpreters because there are a lot of available libraries which can help them

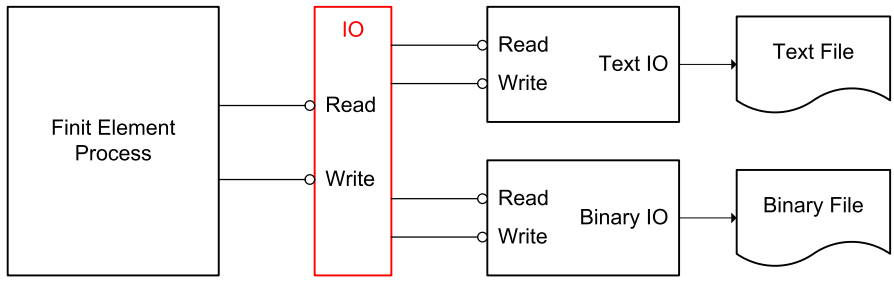



Figure 9.17: Multi format extendible IO example

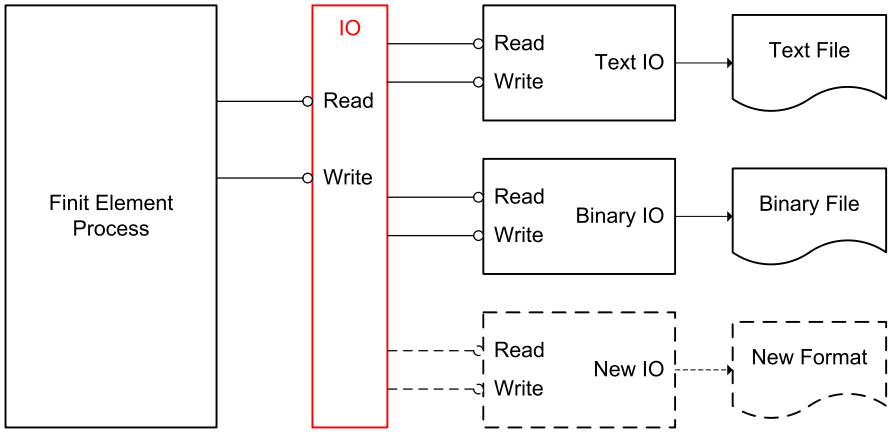



Figure 9.18: Extended IO to a new format

[67, 6, 2, 5]. Even though using libraries and compilers creating tools, implementing an interpreter for a finite element program can introduce a great overhead to its cost. Also many finite element programmer do not have compiler writing knowledge in their background and learning or hiring some specialist causes again extra cost for the program.

The last option is to use an already implemented interpreter and not to write a new one. The first advantage of this approach is omitting cost and time overhead needs to develop a new interpreter. Another good news is the freedom from knowing how to write a compiler. Also an existing language has its documentations like tutorials, reference manuals and so on. This reduces the program IO documentations just to documenting the extended part of the language and its specific commands. However this approach has its backwards too. First is the effort to connect the program to the interpreter can be significant depending on the programming language of the FE code and the interpreter chosen. Second disadvantage is the generic concept of popular languages which may prevent them from fitting well into finite element concepts. Another backward is their overhead in memory and performance. After all there are good news again. A good selection of language and its interpreter can reduce or effectively remove these disadvantages. There are many industrial applications using this approach, for example ABAQUS [10, 11] uses Python [89, 63] or

ANSYS [1] uses TCL/TK.

### 9.2.7 Robust Save and Load

Programs manipulating any kind of document usually have to provide some mechanism to save and load again their documents. In computer science an object which keep its state after the execution of program is referred as a *persistence* object. Also persistence is known as the program ability to store and retrieve its data. Finite element applications usually do not need to save their model in the same way they read it. On the contrary, they just read the model and write the results. Pre and post processors are responsible to store the model and processed results for future use. Still there are some situations that save and load features are needed for a finite element solver. For example to analyze big problems is sometimes useful to stop the process and resume it sometimes later due to some resources schedule. A typical reason is working with computers that are available at night. This makes necessary for the code to store its state before ending the process and retrieve it next time it is executed to be started from the last state.

There are different ways to implement persistency. Two typical approaches are object persistency and global or database persistency.

Object persistency follows the idea that each object knows how to store and retrieve itself using certain interface. This approach is also known as *serialization*. This comes from the fact that data is stored and retrieved serially in this mechanism. To implemented a serialization mechanism first an interface is introduced to all objects that provide a unified form of save and load their state. For example including a `Serialize` method for each of them. Then each object implements its manner of storing and retrieving its data. For objects consisting of simple data this can be done simply by storing the data sequentially and read it back also sequentially in the same order. Save and load for a complex object of some other object and also some simple data, consist of putting simple data as they are and calling `Serialize` method of all member objects sequentially and then load them again in the same order also by calling the members `Serialize` methods. The important point is calling the `Serialize` method of each component, which cause each part to serialize its data and call again `Serialize` of its components. In this way starting from top of the data pyramid and start to save or load, system will traverse to the bottom automatically and cover all data from top to bottom.

An example will this method. Let us implement a serialization mechanism for a simple `Mesh` class containing `Node`s and `Element`s and each `Element` has its `Properties` as shown in Figure 9.19.

The first step is to create the necessary interface by adding a `Serialize` method to all objects. Figure 9.20 shows this interface.

Now let us implement the `Serialize` methods. While the `Mesh` does not have any simple data by itself it just call the `Serialize` methods of all `Node`s and `Element`s and also some additional information for retrieving itself.

```
Serialize(File, State) {
  if(State == IsRead){
    File >> NumberOfNodes;
    File >> NumberOfElements;
    CreateNodesArray(NumberOfNodes);
    CreateElementsArray(NumberOfElements);
  }
  else{
    File << NumberOfNodes;
    File << NumberOfElements;
```

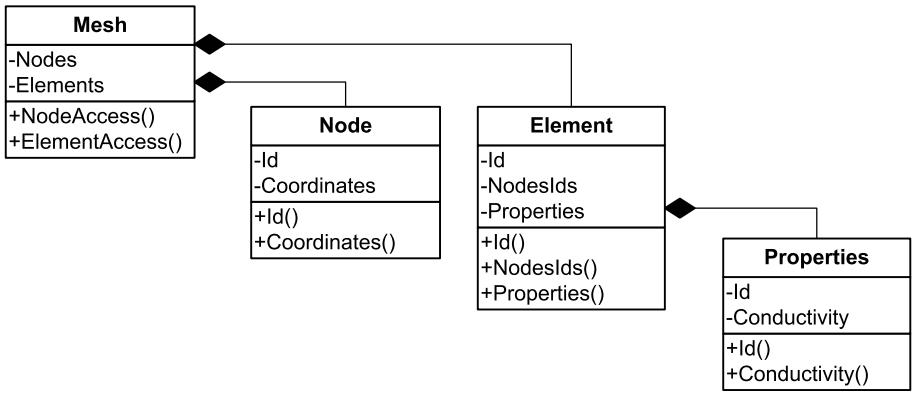



Figure 9.19: A simple `Mesh` class and its components

```
  }
  for(int i = 0 ; i < NumbreOfNodes ; i++)
    NodesArray[i].Serialize(File, State);
  for(int i = 0 ; i < NumbreOfElements ; i++)
    ElementsArray[i].Serialize(File, State);
}
```

`Node`'s `Serialize` method is very simple and just transfer its two simple data, `Id` and `Coordinates`.

```
Serialize(File, State) {
  if(State == IsRead){
    File >> Id;
    File >> Coordinates[0] >> Coordinates[1] >> Coordinates[2];
  }
  else{
```

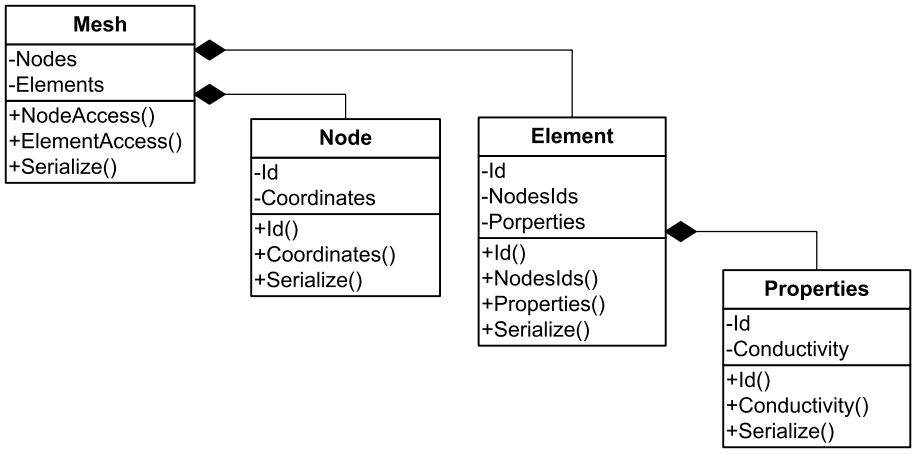



Figure 9.20: Adding the `Serialize` method to create necessary interface

```
    File << Id;
    File << Coordinates[0] << Coordinates[1] << Coordinates[2];
  }
}
```

But `Element` has simple data like its `Id` and also a component which is the `Properties`. To handle them the `Serialize` method will be something like:

```
Serialize(File, State) {
  if(State == IsRead){
    File >> Id;
    File >> NodesIds[0] >> NodesIds[1] >> NodesIds[2];
  }
  else{
    File << Id;
    File << NodesIds[0] << NodesIds[1] << NodesIds[2];
  }
   Properties.Serialize(File, State);
}
```

And finally the `Serialize` method of `Properties` finishes by adding its data to the file.

```
Serialize(File, State) {
  if(State == IsRead){
    File >> Id;
    File >> Conductivity;
  }
  else{
    File << Id;
    File << Conductivity;
  }
}
```

Figure 9.21 Shows the global scheme of the implemented example.

Now to save the `Mesh` we just need to call its `Serialize` method with a writing flag:

```
mesh.Serialize(File, IsWrite);
```

This causes all `Nodes` and `Elements` write themselves in the output file and also each `Element` causes its `Properties` to write itself too. Figure 9.22 shows the sequence of data written in the output file.

To retrieve data from the file its just necessary to call again the `Serialize` method of the `Mesh` but this time with the reading flag:

```
mesh.Serialize(File, IsRead);
```

This method takes first `Mesh` information and then begin to load `Nodes` and after that `Elements`, as shown in the above pseudo code. Looking to the written data's sequence shown in Figure 9.22, it can be easily verified keeping the same order for writing and reading, results in the same structure.

In this simple example the `Mesh` creates the object by knowing there are `Node` or `Element`. In practice there are many cases which the type of object is not known before reading. For example a real mesh can have various type of elements. So how can we create them before calling there `Serialize` method? There are two ways to solve this problem. The first way is using C++ *Run Time Type Identification (RTTI)* to store the object name and create an object using this name. The second way is to name each object manually and store it as a reference name in order to create

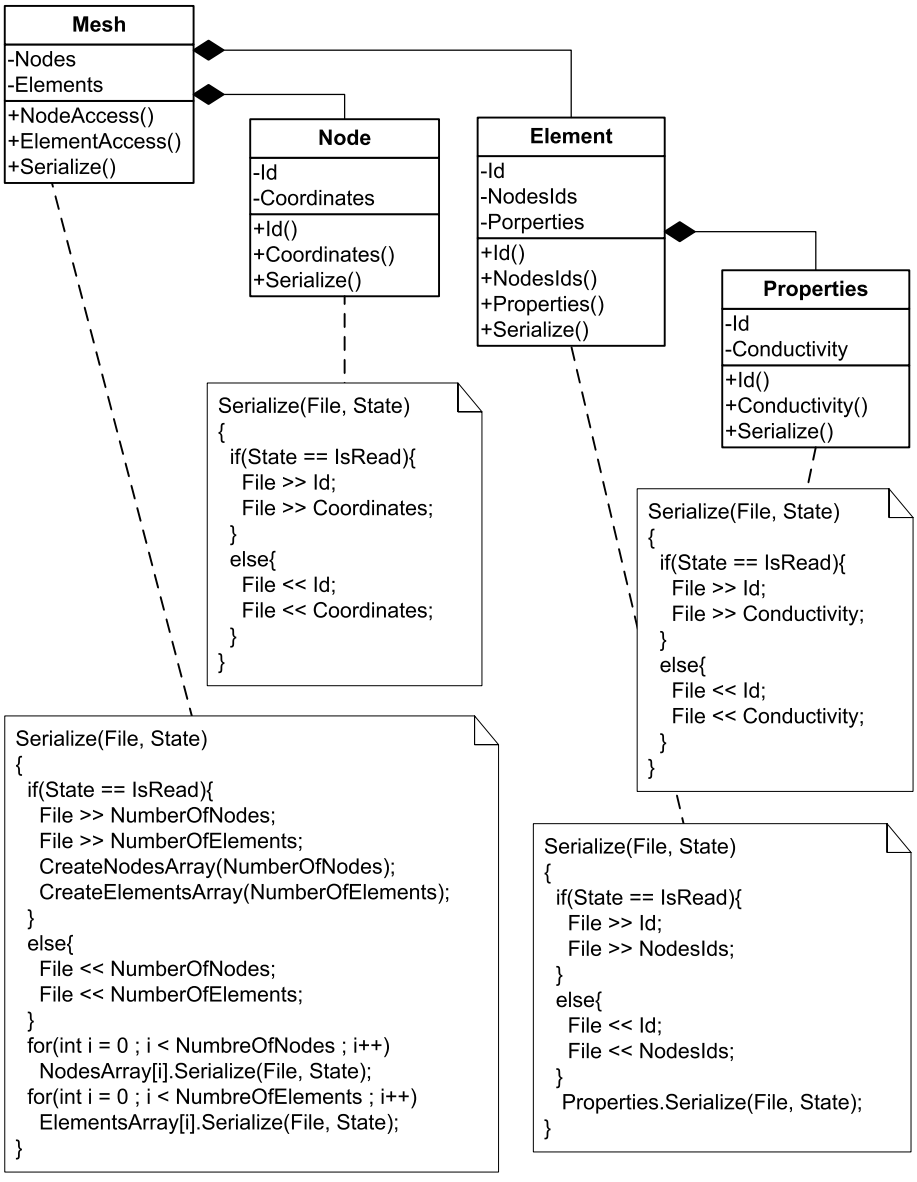



Figure 9.21: Pseudo implementation of `Serialize` methods.

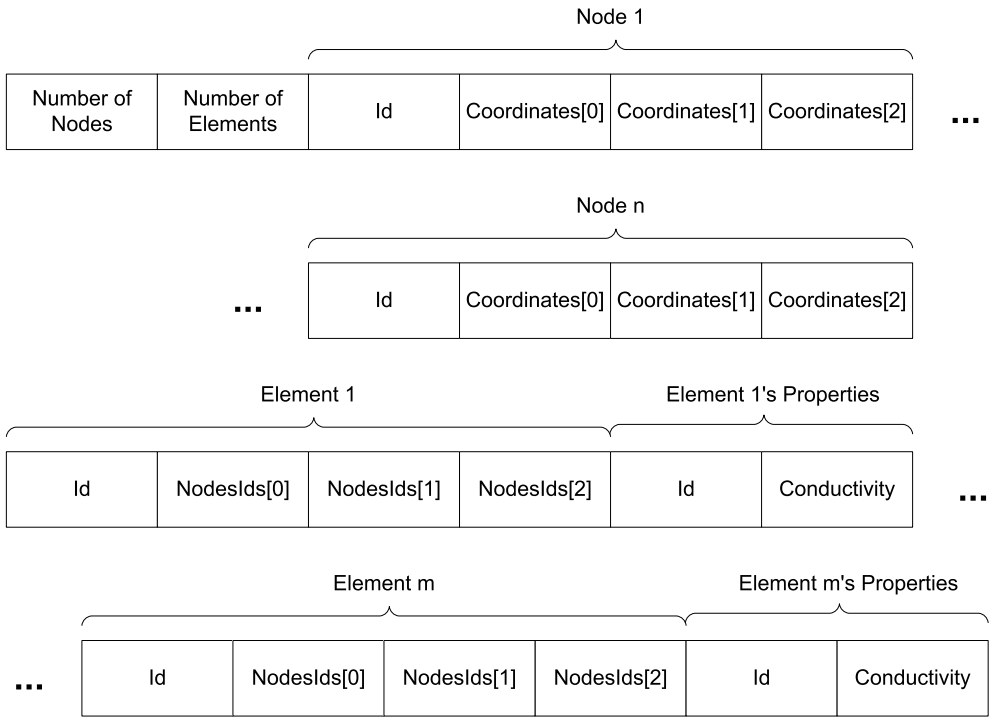



Figure 9.22: Data sequence in output file resulting from the `Mesh` serialization example

it again. The first method is very robust and easy to use. It is applicable to any type of classes and is very extendible. The second one is less robust because names are given manually and any coincidence may cause a problem but it is more tunable because different names can be given to different states of a class. Both approaches need a registry database which for any given object name indicates the registered way of create it.

Now let us take a look at pointer management and serializing pointers. Considering an element class which keeps pointers to its nodes instead of their indices. Applying our standard method will save the memory address of nodes stored in pointers. While the objects after restoring can be in other position of memory, this stored address is completely useless to recover the node. A more careful implementation can save a copy of the pointed node, but this results duplicated nodes and still it is not what we want. The solution is to give a unique index to each object and store and retrieve pointed objects by these indices. In our finite element case nodes and elements' indices fit well for this purpose.

Serialization is very much dependent on the internal structure of the program. Any time a component of objects changes the previous stored data will be invalidated. Using versions and retrieve data by their version solves this problem and it helps to use it in practice.

Serialization in general is very extendible and well suited as a generic solution in saving and loading tasks. There are libraries which provides a good implementation of serializing libraries and can be used in other codes to give persistency with minimal effort. XParam [9] and Boost Serialization library [4] are examples of these libraries. Microsoft Foundation Class (MFC) library also includes a serialization mechanism but using it introduces a platform dependency on the code and makes it not portable to other platforms.

Database persistency on the other hand, comes from the idea that a database must know how to save and load its data. This approach is less extendible but it works well for working with databases. Any new object which can be stored in a given database even by its component can be stored and restored via existing IO without modification. This is the great advantage of this approach.

### 9.2.8 Generic Multi-Disciplinary IO

After a short introduction to each different features in IO, now let us see which is necessary and which can be useful for a generic multi-disciplinary IO.

**Supported Media** Supported media can be considered as an optional feature. However in order to add portability to our code or to make it a usable library in some cases it can become necessary. Treating media like formats can give an effective solution to maintain the extendibility to new media without putting extra effort on it.

**Single or Multi IO** It is obvious that at the time of designing a generic IO there is no clue about how it will be used. This increases the amount of encapsulation in our design which is also the key point to make it usable in multi IO schemes. Also many multi-disciplinary algorithms are working with multi IO schemes which makes it an essential part of our design.

**Data Types and Concepts** Making IO extendible to new concepts is essential for a generic library and also for a multi-disciplinary one. Extendibility to new types is essential for generic library to guarantee the ease of using it.

**Different Format Supporting** Though supporting different formats is not a necessary feature, it is a great added value for any generic library. To develop a generic library having an extendible interface to new formats increases the use of the generated library because any new user can make it work with its already existing format. All these makes format extendibility a part of our IO features to be implemented.

**Text or Binary Format** Selecting one format depends on the use of the program and not on the multi-disciplinary nature of it. Deciding to have a extendible multi-format IO opens the way to implement text or binary formats when they are necessary.

**Data or Process IO** Process IO is one of the essential features for a multi-disciplinary IO. The variety of algorithms in solving different problems is source of constant changes in the code without process IO. Giving the user the ability to change the algorithms or introduce new algorithms without modifying the code dramatically increases the extendibility and reusability of the library.

**Serialization** As mentioned earlier, serialization can help to store the state of process to resume it later. This is a useful feature but in many cases is not necessary. The intention is to use an existing library (like Boost Serialization) without altering the IO design respect to it. After all the database approach is always available to add save and load features to the program.

## 9.3 Designing the IO

In previous section a quick review of differen IO features was presented. Also a brief description of their influence in design and implementation was given and finally a list of the selected features was prepared. The next step is to design an IO module considering all those features.

### 9.3.1 Multi Formats Support

Let us take the multi formats supporting feature to start the design process. The first goal is to make an extendible multi format IO. As mentioned before a proper design is to establish an interface and encapsulate each format support in a separate class. The strategy pattern is what we are looking for. As mentioned in section 3.4.1, this pattern defines a family of algorithms and make them interchangeable via encapsulating each one separately, which is what we are looking for. Applying this pattern to our problem results in the IO structure shown in Figure 9.23.

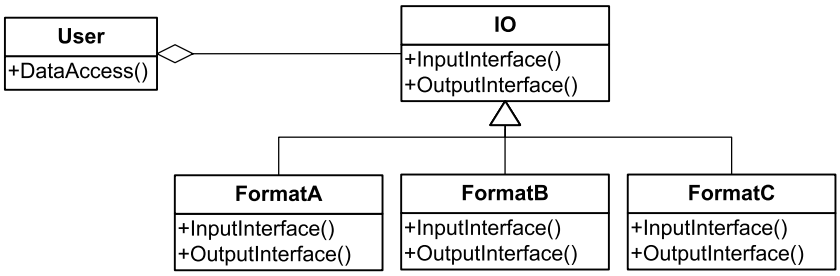



Figure 9.23: Multi format IO structure

### 9.3.2 Multi Media Support

Multi media support can be added as the second component for designing IO. The first approach is to assume that each medium can be used to store any format without dependency. In this case the bridge pattern can be used to decouple these two concepts and cut their dependencies. Introducing this pattern to the previous design results in the structure shown in Figure 9.24.

The other approach is treating media like formats and put them in the same structure. This approach simplifies very much the structure and its implementation. Beside this advantage it makes strong relation between formats and media and having the same format in two media may duplicate the code. Figure 9.25 shows this simple modification.

This seems to be too rigid but it can be improved to be more flexible with some minor changes. Also the independency of medium and format interfaces is guaranteed by the bridge pattern is not highly necessary in our case and comes at the cost of complicating the structure. In practice big variety of finite element formats are used to create files, and other types of media are used to work with other class of formats. So a simplified approach is to work with the second simpler structure and enrich it via templates or multiple hierarchy as shown in figure 9.26.

### 9.3.3 Multi IO support

This structure by itself provides multi IO support. In any places where an IO statement is needed, one can create any IO object and perform its IO without duplicating code. Also in time of implementation providing some controls can reduce conflicts. Only exception, which is related to the layering nature of Kratos, is that IO statements cannot be in `Node`, `Element` or `Conditions` because it violates the layer order as it requires to place upper layer objects in lower layer ones.

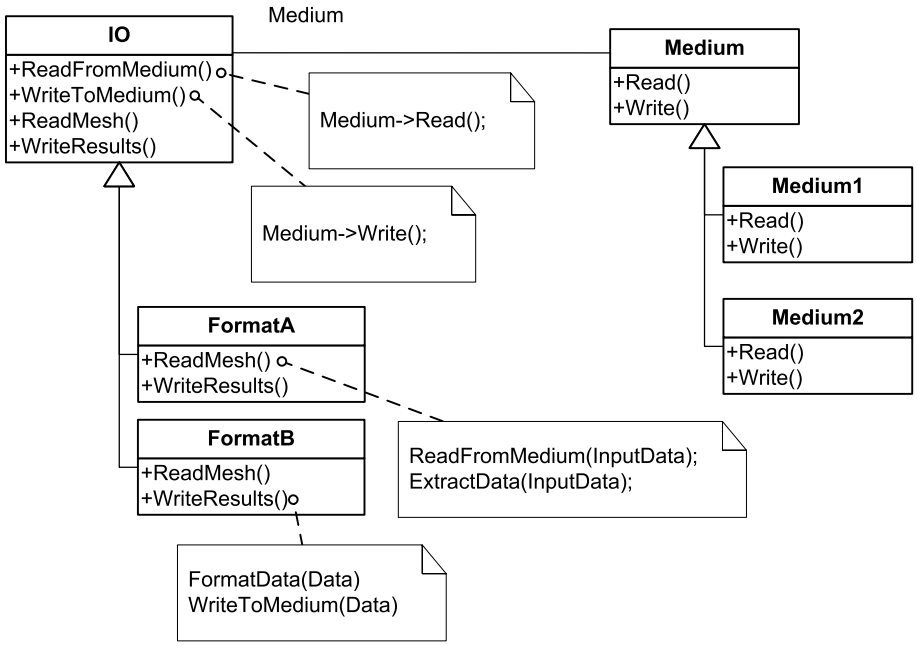



Figure 9.24: Multi format and multi media IO structure

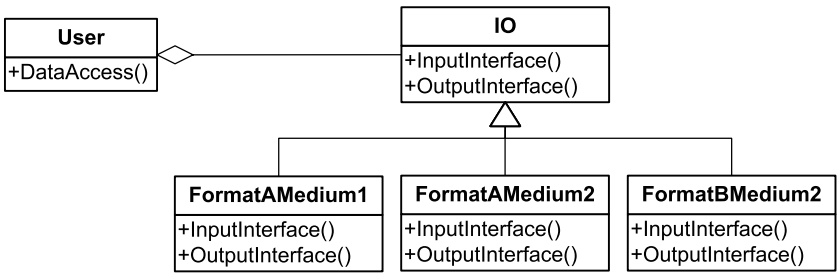



Figure 9.25: Multi format and Medium IO structure

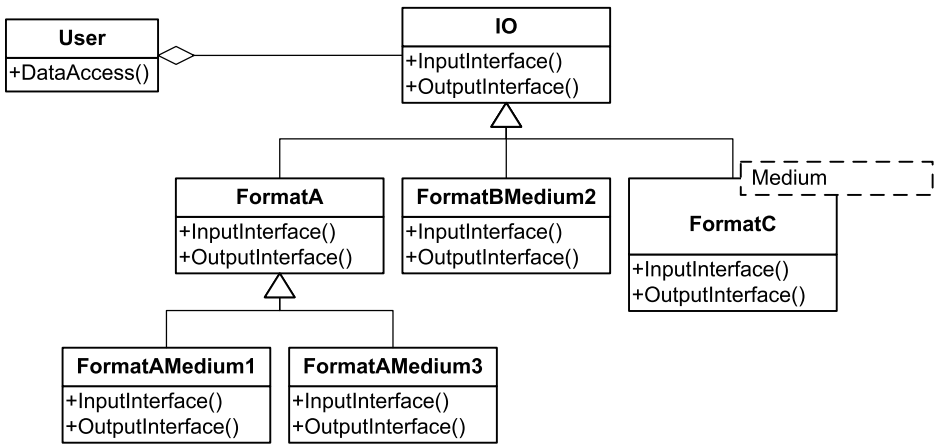



Figure 9.26: Extended Multi format and Medium IO structure

### 9.3.4 Multi Types and Concepts

The next feature to add is the extendibility to new types and concepts. As mentioned before this is an essential feature for a generic multi-disciplinary library. Here the variable base interface comes handy and it is used to generalize the IO.

The first step is providing IO extendibility to new concepts. As described earlier this can be done by introducing a lookup table which relates the concept names and their internal handlers. A simple list of `Variable`s is enough for IO to take as its lookup table. Each `Variable` knows its name and also its reference number. In time of reading IO reads a tag and searches in the list for the `Variable` whose name coincides with the tag. Then use the variable to store the tagged value in the data structure. For example, when IO reads a `"TEMPERATURE=418.651"` statement from input, takes the `"TEMPERATURE"` tag and searches in the list to find the `TEMPERATURE` variable. Having this variable is enough to use the variable base interface of the data structure to store the value in it. For writing results there is no need to search in the table. IO can use the variable to get its value in the database and use its name as the tag. Here is an example of `WriteNodalResults` method.

```
template<class TDataType>
 void WriteNodalResults(Variable<TDataType> const& rVariable,
                        NodesContainerType& rNodes,
                        std::size_t SolutionStepNumber)
 {
    for (NodesContainerType::iterator i_node = rNodes.begin();
         i_node != rNodes.end() ; ++i_node)
                Output << rVariable.Name()
                       << "="
                       << i_node->GetSolutionStepValue(rVariable,
                                    SolutionStepNumber) )
                       << std::endl;
 }
```

Now the IO is extendible to new concepts, but what about new `Element`s and `Condition`s? Each new `Element` or `Condition` also is a new type which implies that the IO does not know how

to create it. Again the idea of lookup tables comes handy. Here we need a table for each `Element` or `Condition` and their relative factory method. Prototype pattern helps to manage this situation in a generic way. Here we reuse each object as its prototype by adding a `Clone` method to it. Figure 9.27 shows the `Element` prototyping mechanism used by IO.

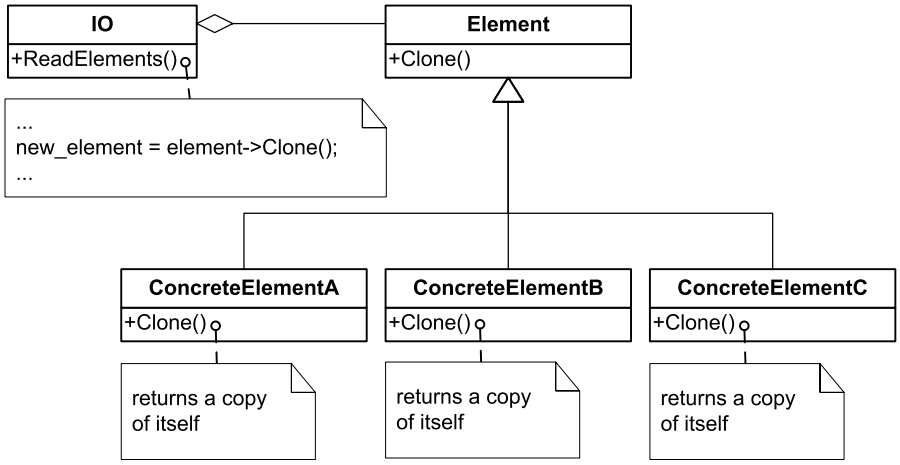



Figure 9.27: Using prototype pattern in IO for creating `Element`s.

IO uses lookup table to find the object prototype for any component name. This table consists of representative names and their corresponding prototype. Encapsulating this table introduces the new `KratosComponents` class to our structure. `KratosComponents` class encapsulates a lookup table for a family of classes in a generic way. Prototypes must be added to this table by unique names to be accessible by IO. These names can be created automatically using C++ RTTI or given manually for each component. In this design the manual approach is chosen, so shorter and more clear names can be given to each component and also there is a flexibility to give different names to different states of an object and create them via different prototypes. For example having, `TriangularThermalElement` and `QuadrilateralThermalElement` both as different instances of `2DThermalElement` initializing with a Triangle or a Quadrilateral.

This structure allows us to create any registered object just by knowing its representative name. But sometimes it is useful to know the family which an object belongs to. For example at the time of reading `Element`s there is no need to search in `Variable`s and `Condition`s and put all of them together can slow down the parsing process unnecessarily. Dividing the lookup table to three family of classes: `Variable`s, `Element`s and `Condition`s helps to distinguish them in the time of search. Doing this also eliminates unnecessary type casting and makes the implementation easier and clearer. Figure 9.28 shows the resulting structure.

Unfortunately storing `Element`s and `Condition`s using their prototyped name requires each `Element` and `Condition` to store their names. This introduces an unnecessary overhead and it is better to use the RTTI and not manual prototyping.

Conflict arises between accepting new types and multi format supporting. The restriction comes from format which not only have to support the type but also to indicate the type to IO. In fact `Element`s and `Condition`s provide a uniform initializing interface which is not possible for different data values. So `Variable` can create data or clone it from some existing sources but the IO to set the value of a given `Variable` needs its type information. However in finite element

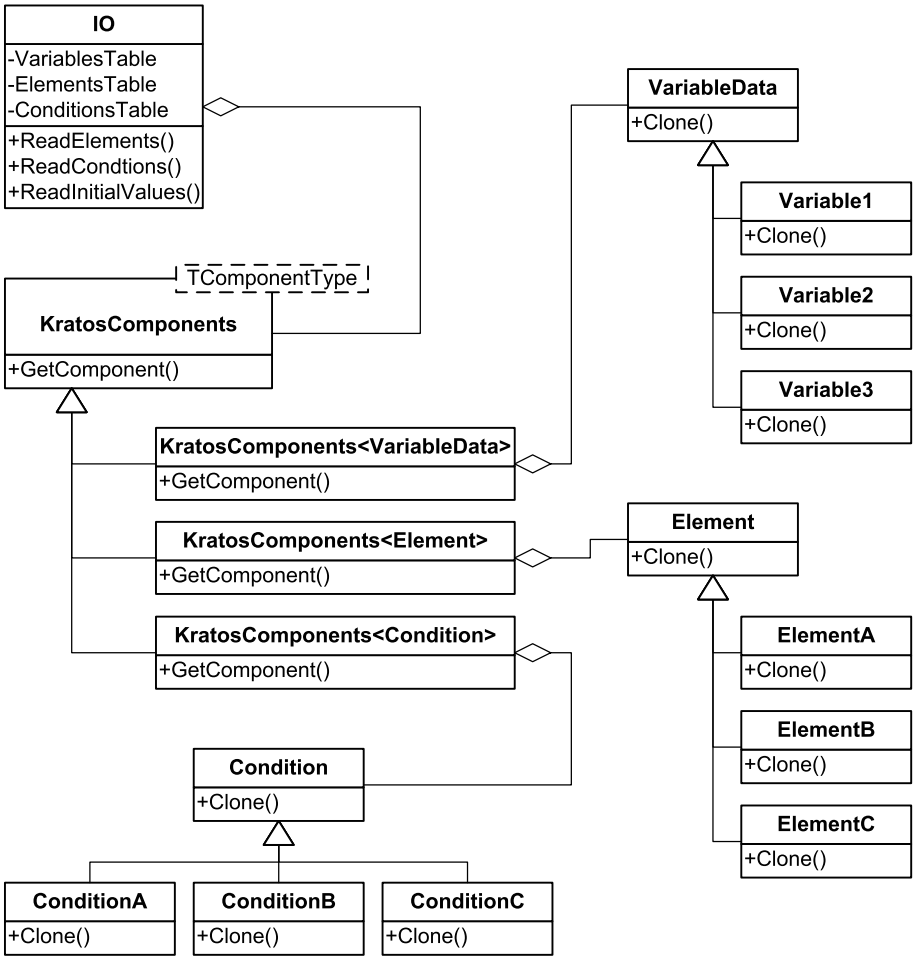



Figure 9.28: Using `KratosComponents` in IO

formats usually few number of types are available and one can handle them separately at the time of implementing the input.

### 9.3.5 Serialization Support

As mentioned before this feature gives new abilities to the program but it is not a very essential part of IO in our design. Serialization can be implemented separately and parallel to our design. So it is possible to use an external serialization library without changing our design as shown in figure 9.29.

### 9.3.6 Process IO

Understanding a given process by IO needs an interpreter which constructs dynamically the algorithm statements and executes them. The interpreter concept and its design will be described

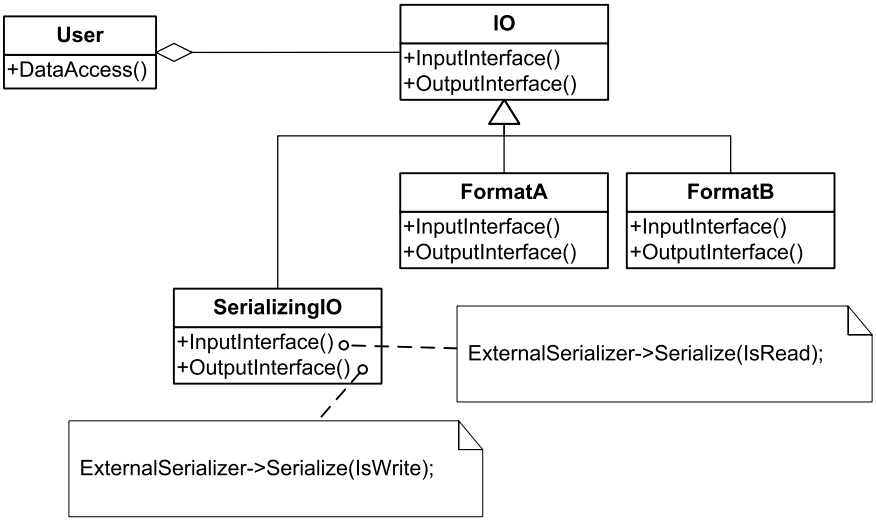



Figure 9.29: Using an external serialization library

later. But the important point to mention here is the difference of interpreting inside or outside the program.

One can implement an interpreter inside any IO subclass and use it in concepts of the IO module. In this way any application which uses this library can use the interpreter and accepts scripts without calling any other executable. Also encapsulating the interpreter in IO allows to implement and use more than one interpreter simultaneously. This allows the user to use subprograms written in languages different than the main script itself.

Another approach is to implement or use an interpreter like an application with close binding to our library. In this approach more complex interpreters can be used easier while there is even no need to compile them. This makes the library more portable and easier to compile.

Finally in practice there is no such a difference between the two approaches. The implementation cost in both cases is more or less the same. Also modern interpreters can run another interpreter which make it indifferent to put them inside IO or not. Usually it is better to put the small interpreters inside the IO and binding to the complex ones as an external application. In Kratos both approaches are used however this can be changed easily in future.

After designing the main structure of IO, now it is time to go through more details. The first part will be input interface designing and following that will be the output interface description.

### 9.3.7 Designing an Input Interface

Looking to the global scheme of a finite element application it can be seen that `Node`s, `Properties`, `Element`s, `Condition`s and initial values are the common input data. So a straightforward design for input interface consist of methods to extract these objects from an input source. However, providing these methods is sufficient to start working but still the interface can be extended more to ease its use. In a finite element program reading data usually consists of filling some finite element container like `Mesh` or `Modelpart`. So it is useful to include also some methods to handle directly this containers which eases the use of IO and in some cases increases the performance.

As mentioned before the `IO` class represents the interface to be implemented in derived classes. The input part of its interface is defined as shown below:

**ReadNode** Retrieves the next `Node` from the input and stop reading. This is useful in manually reading of especial `Node`s. This method gets a reference to the `Node` to be read and set it by input information.

```
ReadNode(NodeType& rThisNode)
```

**ReadNodes** Reads all the `Node`s in a source until the end of sequence is reached. Normal reading of `Node`s array can be performed using this method. It gets a `Node`s array and puts all the input `Node`s inside it.

```
ReadNodes(NodesContainerType& rThisNodes)
```

**ReadProperties** Reads the next `Properties` from the input and stops reading. It is useful to update a `Properties`. For example to read a modified `Properties` in an optimization procedure. It takes a `Properties` and put given information inside it.

```
ReadProperties(Properties& rThisProperties)
```

**ReadProperties** Reading different `Properties` from input can be done by this method. This method reads all the `Properties` until the end of sequence. It takes an array of `Properties` and put all the given `Properties` inside it.

```
ReadProperties(PropertiesContainerType& rThisProperties)
```

**ReadElement** Reads the next `Element` in the input. This method recognizes any `Element` registered in Kratos and reads the first one given as input. This method needs an array of `Node`s and Properties to extract the necessary information to create an `Element`. It also takes an `Element` pointer and assigns it to the new created `Element`.

```
ReadElement(NodesContainerType& rThisNodes,
            PropertiesContainerType& rThisProperties,
            Element::Pointer pThisElement)
```

**ReadElements** `Element`s reading method. This method recognizes any `Element` registered in Kratos and reads all the `Element`s given in the input. This method needs an array of `Node`s and `Properties` to extract necessary information to create `Element`. It also takes an `Element`s array to put created `Element`s in it.

```
ReadElements(NodesContainerType& rThisNodes,
             PropertiesContainerType& rThisProperties,
             ElementsContainerType& rThisElements)
```

**ReadCondition** Reads the next `Condition` in the input sequence. Like the `ReadElement` method, needs an array of `Node`s and `Properties` to extract the necessary information to create a `Condition`. It also takes a `Condition` pointer and if the read `Condition` is not Dirichlet type, it assigns it to the new created `Condition`.

```
ReadCondition(NodesContainerType& rThisNodes,
              PropertiesContainerType& rThisProperties,
              Condition::Pointer pThisCondition)
```

**ReadConditions** Conditions reading method. This is very similar to ReadElements but look for Conditions instead of Elements. This method also reads Dirichlet type conditions given as input. Also like ReadElements it takes arrays of Nodes and Properties to create Conditions. The generated Conditions are placed in the given Conditions array.

```
ReadConditions(NodesContainerType& rThisNodes,
               PropertiesContainerType& rThisProperties,
               ConditionsContainerType& rThisConditions)
```

**ReadInitialValue** Reads the next initial value from the input and puts it in data structure. It takes Nodes, Elements and Conditions and assign them their initial values.

```
ReadInitialValues(NodesContainerType& rThisNodes,
                  ElementsContainerType& rThisElements,
                  ConditionsContainerType& rThisConditions)
```

**ReadInitialValues** Reads all initial values and put them in the data structure. It takes Nodes, Elements and Conditions to assign their initial values given by input.

```
ReadInitialValues(NodesContainerType& rThisNodes,
                  ElementsContainerType& rThisElements,
                  ConditionsContainerType& rThisConditions)
```

**ReadMesh** Reads all Nodes, Properties, Elements and Conditions and store them in the given Mesh. This method is the easiest way to fill the Mesh at the beginning of program.

```
ReadMesh(MeshType & rThisMesh)
```

**ReadModelPart** Reads all Nodes, Properties, Elements and Conditions and store them in the given ModelPart. It also initialize the ModelPart name and its attributes.

```
ReadModelPart(ModelPart & rThisModelPart)
```

It is important to mention that above methods are just the interface and each derive class may implement all or just a set of them. So to avoid unexpected problems it is useful to put an error message in all IO methods and inform the user about calling some unimplemented feature in the derived class.

```
virtual void ReadModelPart(ModelPart & rThisModelPart)
{
  KRATOS_ERROR(std::logic_error,
    "Calling base class member. Please check the ...", "");
}
```

### 9.3.8 Designing the Output Interface

The output interface is defined by IO and output format is encapsulated in IO derived classes. The first part of the interface is devoted to writing basic objects. These methods are helpful for example to write a restart file. Also methods to write finite element containers are provided. These methods are useful in term of exporting the model or giving additional information to thepost processors. Finally there is a part of the interface dedicated to writing results. Here is the list of available output methods:

**WriteNode** Writes given `Node` into the output. This method may also write nodal data if the output format require it.

```
WriteNode(NodeType const& rThisNode)
```

**WriteNodes** This method writes an array of `Node`s into the output. It can be used also to create the restart file. Its argument is simply a `Node` container.

```
WriteNodes(NodesContainerType const& rThisNodes)
```

**WriteProperties** Takes a `Properties` and writes it into the output. This method is useful to communicate some `Properties` or communicate it via an interface.

```
WriteProperties(Properties const& rThisProperties)
```

**WriteProperties** Writes all given `Properties` to the output. Takes a `Properties` container and write all of them. This method can be used to create restart file by giving writing all `Properties` in it.

```
WriteProperties(PropertiesContainerType const& rThisProperties)
```

**WriteElement** Writes an `Element` into the output. It takes an `Element` as its argument. It is useful to do a selective writing between `Element`s.

```
WriteElement(Element const& rThisElement)
```

**WriteElements** This method takes a container of `Element`s and write all of them into the output. So it is a useful method to write a restart file by passing all `Element`s to it.

```
WriteElements(ElementsContainerType const& rThisElements)
```

**WriteCondition** Writes the given `Condition` into the output.

```
WriteCondition(Condition const& rThisCondition)
```

**WriteConditions** Takes a container of `Node`s and writes the Dirichlet conditions assigned to each `Node`. This method can be used to write the restart file.

```
WriteConditions(NodesContainerType const& rThisNodes)
```

**WriteConditions** An overload of previous method which takes a container of `Condition`s and write them into the output. This method also is useful to write the restart file.

```
WriteConditions(ConditionsContainerType const& rThisConditions)
```

**WriteMesh** This method writes all the `Node`s, `Properties`, `Element`s and `Condition`s in the given `Mesh` to the output. This makes it one of the best methods to write the restart file or it can be also used to create additional information for post processing task.

```
WriteMesh(MeshType const& rThisMesh)
```

**WriteModelPart** Writes all the `Node`s, `Properties`, `Element`s and `Condition`s for a given `ModelPart` into the output. This method also puts the `ModelPart` attributes into the output.

```
WriteModelPart(ModelPart const& rThisModelPart)
```

**WriteNodalResults** This method writes the nodal value of a given `Variable` for all given `Nodes` into the output. This method can be used to write nodal results for post processing. It takes the variable to be written and a container of `Nodes` which their values must be written. Also it takes a `ProcessInfo` which gives extra information for writing results, like the time step.

```
WriteNodalResults(VariableData const& rVariable,
                  NodesContainerType& rNodes,
                  ProcessInfo & rCurrentProcessInfo)
```

**WriteElementalResults** This method writes the elemental value of given variable into the output. This method can be use to write elemental results for post processing. It takes the variable to be write and a container of `Elements` which their values must be written. Also it takes a `ProcessInfo` which gives extra information for writing results, like the time step.

```
WriteElementalResults(VariableData const& rVariable,
                      ElementsContainerType& rElements,
                      ProcessInfo & rCurrentProcessInfo)
```

**WriteIntegrationPointsResults** This method writes the result calculated on integration points of given `Elements` into the output. It takes the variable to be write and a container of `Elements` which their integration points values must be written. Also it takes a `ProcessInfo` which gives extra information for writing results like time step.

```
WriteIntegrationPointsResults(VariableData const& rVariable,
                              NodesContainerType& rNodes,
                              ProcessInfo & rCurrentProcessInfo)
```

There is an important detail to be mentioned here. It can be seen that the last three methods are taking `VariableData` as their argument and not a typed variable or components. In C++ template function cannot be defined as virtual. This prevent us to make these methods a template. So to have it still extendible to new type the interface provide just a very general interface and each implementation has to check the type of variable via RTTI and then dispatch it to the proper implementation.

## 9.4 Writing an Interpreter

This section describes a step by step approach to write an interpreter. First the global structure and its components will be commented. Then creating an interpreter by using two different tools is described.

### 9.4.1 Global Structure of an Interpreter

An interpreter usually is divided into two parts as shown in figure 9.30.

The first part is the *lexical analyzer*. This part reads the input characters, for example a source file or given command line statement, and converts it to a sequence of *tokens*, like digits, names, keywords, etc., usable for parser. Since white spaces (like blank, tab and newline characters) and comments are not used in parsing the code, there is a secondary task for a lexical analyzer to

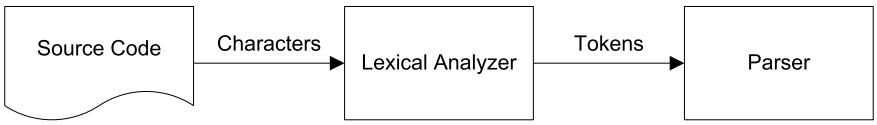



Figure 9.30: Global structure of an interpreter

eliminate all white spaces and also comments during the analysis. The lexical analyzer may also provide additional information for the parser to produce more descriptive error messages.

A *token* is a sequence of characters having a logical meaning or making a unit in our grammar. One can divide tokens in different categories depending on the grammar working with. Table 9.3 shows some examples of tokens.

| Token's Type | Examples |
|---|---|
| INTEGER | 1 34 610 |
| REAL | 0.23 4.57 2e-4 |
| IF | if |
| FOR | for |
| LPARENTHESES | ( |
| RPARENTHESES | ) |

Table 9.3: Some typical tokens with example of matching sequences

If more than one sequence of input characters matches to a token then this token must provide a value field to store the input data represented by it. To clarify the concept of token and lexical analyzer let make an example. Considering a simple C `if` statement:

```
if(x > 0.00)
  return 0;
```

passing it to a C lexical analyzer would result in the sequence of tokens:

| | | | | |
|---|---|---|---|---|
| IF | LPARENTHESES | ID(x) | GREATER | REAL(0.) |
| RPARENTHESES | | RETURN | INTEGER(0) | SEMICOLON |

The second part is the *parser*. Parser does the syntax analysis. Takes the tokens from lexical analyzer and tries to find their relation and meaning due to its grammar. Parser produces a parse tree by putting together recognized statements and their sub-operations. This tree can be used to execute the given source. For example passing above sequence of tokens to a C parser would result in a parse tree as presented in figure 9.31.

There are several reasons to separate the lexical analyzer from the parser. The first one is simplifying the design and reducing parser complexity. Creating a parser over separated tokens without any comments or whitespaces is easier than a parser over input characters. The second reason is improving the performance of the interpreter. Large amount of interpreter time spent to do the lexical analysis, separating it and using techniques like buffering can increase significantly the performance. Another reason is related to portability and reuse ability of the interpreter. Any problem due to the different character maps in different devices can be encapsulated in the lexical analyzer without changing the parser.

In some cases, the lexical analyzer is also divided into two phases. A scanner which does simple tasks like removing comments and whitespaces, and analyzer which does the real complex job.

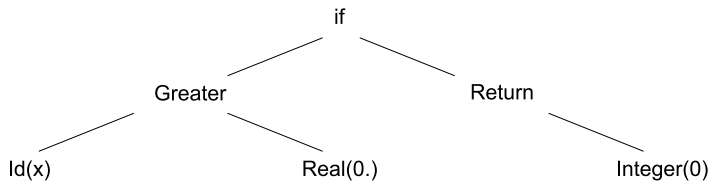



Figure 9.31: Parse tree for `if` statement example

Now let us go inside the each part and see how to implement it.

### 9.4.2 Inside a Lexical Analyzer

A lexical analyzer uses some patterns to find tokens in a given sequence of characters. Usually these patterns are specified by *regular expressions*. So before starting with the lexical analyzer pattern a brief description of regular expressions is necessary.

#### Regular expressions

A language can be considered as a set of strings with some special properties. So it is important to describe exactly these properties to define a language. Regular expressions are the common notation to define these concepts in a generic way.

This notation consist of some rules which let regular expression $r$ denoting the language $L(r)$ and also some other rules to combine different expressions together in different ways. This combination rules makes it very powerful in term of dividing complex definition to simpler ones or to reuse some expressions to make a more complex one.

Regular expressions rules consist of symbol collective rules from a set of symbols called *alphabet*, which regular expression is defined over it. Here are the rules which define the regular expression over alphabet $\Sigma$:

**Epsilon** $\epsilon$ is a regular expression to denote the language $\{\epsilon\}$ containing just the empty string .

**Symbol** For a symbol $a$ in $\Sigma$, the regular expression $a$ denotes the language $\{a\}$, whose only string is `a`.

**Alternative** Alternative operator written with a vertical bar | combines two regular expressions $r$ and $s$ into new regular expression $r|s$. This expression represent the language $L(r) \bigcup L(s)$ that contains all strings contained by either $r$ or $s$. For example $a|b$ represents the language $\{a, b\}$.

**Concatenation** Two regular expressions $r$ and $s$ can be combined into a new regular expression $r \cdot s$ using the concatenation operator $\cdot$. This new expression denotes the language $L(r)L(s)$ that contains all strings $\alpha\beta$ for all $\alpha \in L(r)$ and $\beta \in L(s)$. For example $a \cdot b$ denotes the language $\{ab\}$. In some notations the $\cdot$ symbol is simply omitted and putting two expressions simply one after other means concatenation, $a \cdot b \equiv ab$.

**Repetition** Repetition operator $^*$, or *Kleene star*, applied to a regular expression $r$ represents the language $L(r)^*$ that contains strings with zero or more instances of any symbol in $L(r)$. For example, $(a|b)^*$ represents the language $\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, ...\}$.

**Parentheses** Parentheses can be use to group an operation and applying it to a single rule will not change the expression rule.

There is also some conventions in precedence of operators which helps to reduce ambiguities:

- Kleene star has the highest precedence.
- Concatenation has the second precedence.
- Alternative operator has the lowest precedence.
- All these operators are left associative.

Finally there are some abbreviations for frequent expressions which eases its use in practice:

**Positive closure** The positive closure operator $^+$ is left associative and applied to the regular expression $r$ denotes the language $L(r)^+$ which contains one or more instances of any symbol in $L(r)$. This operator has the same precedence as the repetition operator $^*$. $r^* = r^+|\epsilon$ and $r^+ = rr^*$ equations describes the relation between the positive closure operator and the repetition operator. For example, $(a|b)^+$ represents the language $\{a, b, aa, ab, ba, bb, ...\}$.

**Zero or one instances** Applying the unary operator ? to the regular expression $r$ denotes the language $L(r) \bigcup \{\epsilon\}$ which contains zero or one instance of any symbol in $L(r)$. This operator is the abbreviation of $r|\epsilon$. For example, $(a|b)$? represents the language $\{\epsilon, a, b\}$.

**Character classes** For alphabet symbols $a$,$b$,...,$z$ The notation $[abc]$ is equivalent to $a|b|c$ and the $[a-z]$ is equivalent to $a|b|...|z$. This two abbreviations can also be combined. For example a C identifier can be any letter followed by zero or more letters or digits. This definition can be written using regular expression $[A-Za-z][A-Za-z0-9]^*$.

After a brief description of regular expressions, now it is time to take a look at the lexical analyzer itself.

### Lexical Analyzer

As mentioned before a lexical analyzer is in charge of converting a sequence of characters as its input to a sequence of tokens as its output using some patterns. Usually regular expressions are used to specify these patterns. For example to read the following node statements:

```
// A typical Node Statement
Node(1,1.00,0.00,0.00)
```

The lexical analyzer need to understand the `Node` keyword, integer, real parenthesis and also comments. Here are the regular expressions defining legal tokens and their corresponding actions for this case:

```
Node                                        {return NODE_TOKEN}
[0-9]+                                      {return INTEGER_TOKEN}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)         {return REAL_TOKEN} "//"("
"|[A-Za-z0-9])*"\n"                  {/* It is just comment*/} "
"|"\n"|"\t"                               {/* white spaces */}
```

These rules are somehow ambiguous. For example reading the x coordinate `1.00` can create a problem while the first `1` can be recognized as an integer and the `.00` as a double after it! To reduce this ambiguities lexical analyzers use two additional rules:

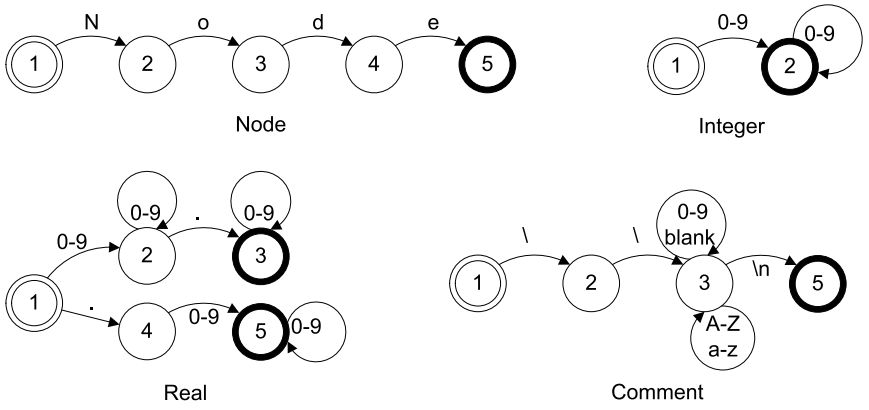



Figure 9.32: Finite automata example

**Longest match** Analyzer tries to find the longest sequence of input characters that can match to one of the patterns.

**Pattern priority** For a particular longest string which can much, the first regular expression which matches to it has priority to determine its type.

The idea here is to create a mechanism to convert a given regular expressions to a lexical analyzer automatically. The method is to convert given regular expressions to a graph which can be implemented easily.

First the regular expression has to be converted to *finite automata*. The finite automata is a graph consisting of finite set of *states* connected by *edges* and each edge is labeled with a symbol. This graph shows the way a sequence of character can traverse all states to be matched to a regular expression. Finite automata is *deterministic* if all its states have a maximum of one edge going out for a symbol and any state with more than one outward edge for a symbol converts it to a *nondeterministic finite automata*.Figure 9.32 shows a finite automata of the regular expressions in previous example.

Now let us combine all these separate finite automata in a single one which represents our lexical analyzer. For this small example this can be down manually but in practice manual combination can be impossible. Fortunately there are some ways to do this automatically. These methods are consist of converting the regular expressions to nondeterministic finite automata (NFA) and then reduce them to a deterministic finite automata (DFA) plus some optimizations. Detail information about automatic converting procedure of regular expressions into deterministic finite automata can be found in [12, 13, 16]. Here the manual combination is used to create the complete graph shown in figure 9.33.

Finally this combined graph can be converted to a *transition table*. This table consists of several rows, one for each state and also several columns, one for each symbol. Each field contains the target state number for its row state giving its column symbol.

Having a transition table, analyzing an algorithm is very easy to implement. The program first reads a character and goes to the field corresponding to the state 1's row and to this symbol's column. This field contains the target state which is for example state 7. Then it reads the next character and goes to state 7's row and to this new symbol's column to find the next step and so on. An auxiliary array which maps the state numbers to actions is also necessary to verify if a

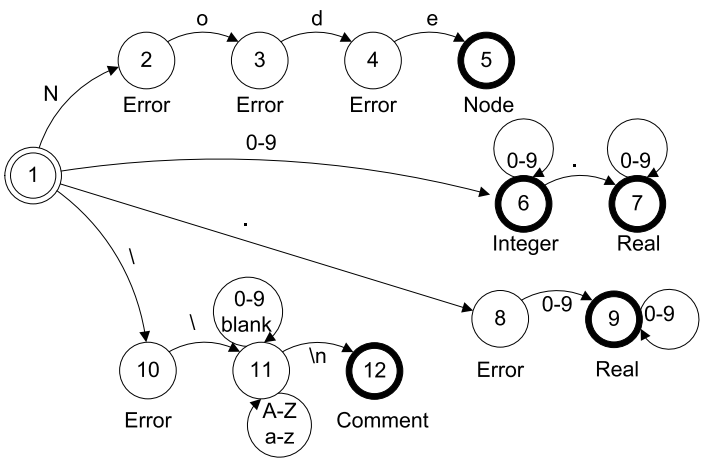



Figure 9.33: Combined finite automata example

pattern is matched what is the action to perform. To find the longest match is just need to store the last match found and update it each time a new match is found. This procedure continues until a dead state is reached. In this time the last match case, if there is any, is returned or if there is no match an error can be send.

### 9.4.3 Parser

A lexical analyzer prepares tokens to be analyzed by the parser and create the parse tree to be executed. The parser tries to match a given sequence of tokens to a pattern. Here a context free grammar is used to specify the patterns. So let see what it is before continuing with the parser.

#### Context Free Grammar

The idea of this notation is to group basic parts in some more complex ones like regular expressions and also to make a recursive use of symbols to enable recursively construction of complex statements.

A context free grammar consists of terminals, nonterminals, productions and also a start symbol.

**Terminal** Terminals are the basic units of grammar. In grammars for programming languages terminals are equivalent to tokens. In our case each token is a terminal in the context free grammar.

**Nonterminal** Nonterminals are syntatic variables that represents a group of terminals or nonterminals as a part of a language syntax.

**Production** Production is the manner of defining a nonterminal by combination of terminals and nonterminals. Production has a general form of:

$$nonterminal \rightarrow sting\ of\ terminals\ and\ nonterminals$$

In some notations symbol ::= is used instead of the arrow.

**Start Symbol** It is a nonterminal that is selected as a start symbol and denotes the language defined by the grammar.

To make this definition clear let is create a grammar to define the previous node input file language:

- *statement* → *statement* | *node_creator*
- *node_creator* → **Node** ( **integer** , *coordinates*)
- *coordinates* → **real** , **real** , **real**

The third line creates a pattern of coordinates with three reals separated by a comma. The second line gives the pattern for construction a `Node`. Finally the first line define the statement which is constructing the `Node` or itself. This leads parser to recurse and reads all the node statements comes statements comes after.

It can be verified that every regular expression set is a context free grammar by itself. This may introduce a question, Why use regular expression to define lexical patterns and not the context free grammar by itself. Here are some reasons:

- Regular expressions are enough to define the syntax and there is no need to deal with the complexity of context free grammars.
- Regular expressions are easier to understand and they define a token more clear.
- Lexical analyzer can be automatically optimized to be more efficient when constructed over regular expression rather than a generic grammar.

#### Designing a Parser

To design a parser the *Interpreter* pattern is what we are looking for. A context free structure can be represented by this structure and then used to create the parse tree to execute the source code.

In general there are various ways to implement a parser in the literature [12, 13, 16]. Also there are some tools to generate parser. Here we will use this tools without going into the parser implementation details.

### 9.4.4 Using Lex and Yacc, a Classical approach

There are several tools to create a parser. Among these, *Lex* and *Yacc* are two classic ones.

Lex is a lexical analyzer generator. It takes a *lexical specification* as input and produces a lexical analyzer written in C. Lexical specification contains regular expression defining each token and also an action that will be executed when an input sequence is matched to a regular expression. These actions are normal C statements and usually returning the token for their case. The backward of Lex is its performance. *Fast lexical analyzer generator (Flex)* is significantly faster than Lex [16] and solves the problem of performance, But still it generates a C code. Though there is no problem in binding a C code to a C++ program, this still pollutes the scope by using several global variables and functions. This also make it difficult to maintain more than one lexical analyzer in the same code. In this work a C++ variation of Flex named *Flex++* is used to generate the lexical analyzer.

*Yet Another Compiler-Compiler (Yacc)* is a classic and widely used parser generator. There are several compilers implemented using Yacc. *Bison* is another parser generator which is more

modern than Yacc but is upward compatible with it. Both generate parsers which are implemented in C language. In this work *Bison++* is used which is a variety of Bison that generates parsers implemented in C++.

### Using Flex

Flex++ like Lex uses lexical specifications to generate the lexical analyzer. This specifications must be prepared by creating an input file usually with extension "`l`", like "`input.l`", and consists of three parts separated by `%%`:

```
%{
C++ Definitions
%}
Lex Definitions

%%

Regular expressions and their actions

%%

Auxiliary procedures
```

The first part is for definitions which is also divided in two parts, C++ definitions part which is enclosed between `%{ %}` symbols and Lex definitions parts. FLex++ simply puts the C++ declaration part above the generated output file. This part usually contains the necessary include files and some macro definitions to customize the generated code. In our case the C++ declaration part is:

```
%{
#define YY_DECL \
  int Kratos::InputCScanner::yylex(yy_InputCParser_stype &yylval)

#include <iostream>
#include <malloc.h>
#include "includes/input_c_parser.h"
#include "includes/input_c_scanner.h"

int yyFlexLexer::yylex(){ return 0; }
%}
```

The lex definition part is to put macros for lex or conditions to be used afterwards. Here is an example of macro definition:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

In this case only a condition for comments is defined here:

```
%x COMMENT
```

The second part is to define rules. Each rule is a regular expression and its corresponding action. In Kratos four groups of rules are defined:

- Rules to handle comments. C++ comments are accepted in this format. One point here is to keep line numbering inside the comments. Another point is to give a warning in case of comments in comments.

```
\n                  { ++mNumberOfLines; }

"//".*$             /* remove one line coments */

"/*"                { BEGIN COMMENT;  }
<COMMENT>"/*"       { Warning("Comment in comment."); }
<COMMENT>\n         { ++mNumberOfLines; }
<COMMENT>[^*/\n]*   ;
<COMMENT>"*"+[^*/\n]*   ;
<COMMENT>"/"+[^*/\n]*   ;
<COMMENT>"*"+"/"    { BEGIN INITIAL; }
```

- The second group are symbols. Most of them are passed as they are. In more formal way each symbol must be passed by a representative token which is not done here for simplicity.

```
\[|\]|"("|")"|"{"|"}" { return yytext[0]; }

"="                   { return yytext[0]; }

","|";"|"."|":"       { return yytext[0]; }

"<"|">"               { return yytext[0]; }

"=="                  { return InputCParser::EQUAL_TOKEN; }

"!="                  { return InputCParser::NOT_EQUAL_TOKEN; }

"<="                  { return InputCParser::LESS_EQUAL_TOKEN; }

">="                  { return InputCParser::GREATER_EQUAL_TOKEN;}

"+"|"-"               { return yytext[0]; }
```

- After symbols there are rules to define Kratos variables. Here are some examples of these rules:

```
TEMPERATURE  {
                yylval.double_variable = &TEMPERATURE;
                return InputCParser::DOUBLE_VARIABLE_TOKEN;
             }

VELOCITY     {
                yylval.vector_double_variable = &VELOCITY;
                return InputCParser::VECTOR_DOUBLE_VARIABLE_TOKEN;
             }
```

- And finally rules for keywords:

```
NODES            { return InputCParser::NODES_TOKEN; }

PROPERTIES       { return InputCParser::PROPERTIES_TOKEN; }

Node             { return InputCParser::NODE_TOKEN; }
```

```
Fix             { return InputCParser::FIX_TOKEN; }

ElementsGroup   { return InputCParser::ELEMENTS_GROUP_TOKEN; }

ELEMENTS        { return InputCParser::ELEMENTS_TOKEN; }

LinearSolver    { return InputCParser::LINEAR_SOLVER_TOKEN; }

SolvingStrategy { return InputCParser::SOLVING_STRATEGY_TOKEN; }

Smooth          { return InputCParser::SMOOTH_TOKEN; }

Map             { return InputCParser::MAP_TOKEN; }

DataMapper      { return InputCParser::MAPPER_TOKEN; }

for             { return InputCParser::FOR_TOKEN; }

Solve           { return InputCParser::SOLVE_TOKEN; }

MoveMesh        { return InputCParser::MOVE_MESH_TOKEN; }

Print           { return InputCParser::PRINT_TOKEN; }

Execute         { return InputCParser::EXECUTE_TOKEN; }

CreateSolutionStep {return InputCParser::CREATE_SOLUTION_STEP_TOKEN; }

CreateTimeStep  { return InputCParser::CREATE_TIME_STEP_TOKEN; }
```

There are also some other rules defines to scape white spaces our returning not matches words.

The third part of Flex input is to put the auxiliary procedures which in this case is left empty.

Flex++ takes these information to generate the transition tables and also a generic code to be called by Bison++ later.

### Using Bison++

Like Flex++, specifications for Bison++ must be prepared in a file but usually with extension "`y`", like "`input.y`". This file again consists of three parts separated by `%%`:

```
%{
C++ Definitions
%}
Parser Definitions

%%

grammar rules

 %%

Auxiliary codes
```

The C++ definition here is important because all definitions of the abstract tree classes goes there:

```
%{
  Kratos::String block_name;

  class Statement{
  public:
    Statement(){}
    virtual void Execute(Kratos::Kernel* pKernel){}
    virtual int Value(Kratos::Kernel* pKernel){return 0;}
  };

  class BlockStatement : public Statement{...};

  class GenerateNodeStatement : public Statement{...};

  template<class TDataType>
  class GeneratePropertiesStatement : public Statement{...};

  class SetElementsGroup : public Statement{...};

  class GenerateElementStatement : public Statement{...};

  class FixDofStatement : public Statement{...};

  template<class TDataType>
  class SetSourceStatement : public Statement{...};


  class GenerateLinearSolverStatement : public Statement{...};

  class GenerateSolvingStrategyStatement : public Statement{...};

  class GenerateMapperStatement : public Statement{...};

  class SolveStatement : public Statement{...};

  class MoveMeshStatement : public Statement{...};

  class ProcessStatement : public Statement{...};

  class ForStatement : public Statement{
    Statement* mpFirst;
    Statement* mpSecond;
    Statement* mpThird;
    Statement* mpForth;
  public:
    ForStatement(Statement* pFirst, Statement* pSecond,
                 Statement* pThird, Statement* pForth) :
      mpFirst(pFirst), mpSecond(pSecond),
      mpThird(pThird), mpForth(pForth) {}
```

```
    ~ForStatement(){
        delete mpFirst; delete mpSecond;
        delete mpThird; delete mpForth;
    }

    void Execute(Kratos::Kernel* pKernel){
        for(mpFirst->Execute(pKernel); mpSecond->Value(pKernel) ;
                                       mpThird->Execute(pKernel))
            mpForth->Execute(pKernel);
    }
};

template<class TDataType>
class PrintStatement : public Statement{...};

template<class TDataType>
class PrintOnGaussPointsStatement : public Statement{...};

template<class TDataType>
class SmoothStatement : public Statement{...};

template<class TDataType>
class MapStatement : public Statement{...};

class CreateTimeStepStatement : public Statement{...};

class CreateSolutionStepStatement : public Statement{...};

template<class TFunction, class TDataType>
class LogicalStatement : public Statement{...};

template<class TDataType>
class ValueStatement{...};

template<class TDataType>
class ConstantValueStatement : public ValueStatement<TDataType> {...};

template<class TDataType>
class VariableStatement : public ValueStatement<TDataType> {...};

template<class TFunction, class TDataType>
class BinaryVariableStatement : public ValueStatement<TDataType> {...};

template<class TDataType>
class AssigningVariableStatement : public Statement {...};

template<class TDataType>
class AssigningNodalVariableStatement : public Statement {...};
%}
```

The class implementations are removed to reduce the size of the document. The second division is for parser definitions. Macro definitions with all tokens and nonterminal declarations are placed here:

```
%define CONSTRUCTOR_PARAM Kratos::Kernel* ...
%define CONSTRUCTOR_INIT : mInputCScanner(pNewInput, ...
%define YY_InputGid_CONSTRUCTOR_CODE
%define LEX_BODY { return mInputCScanner.yylex(yylval); }
%define MEMBERS  Kratos::InputCScanner mInputCScanner;...

%token <integer_value> INTEGER_TOKEN
%token <double_value> DOUBLE_TOKEN
%token <string_value> WORD_TOKEN
%token <double_variable> DOUBLE_VARIABLE_TOKEN
%token <vector_double_variable> VECTOR_DOUBLE_VARIABLE_TOKEN
%token <matrix_variable> MATRIX_VARIABLE_TOKEN
%token OPEN_BRACKET_TOKEN
%token CLOSE_BRACKET_TOKEN
%token NODES_TOKEN
%token NODE_TOKEN
%token PROPERTIES_TOKEN
%token ELEMENTS_GROUP_TOKEN
%token ELEMENTS_TOKEN
%token ELEMENT_TOKEN
%token BEGIN_TOKEN
%token END_TOKEN
%token FIX_TOKEN
%token SOURCES_TOKEN
%token FOR_TOKEN
%token SOLVE_TOKEN
%token MOVE_MESH_TOKEN
%token PRINT_TOKEN
%token PRINT_ON_GAUSS_POINTS_TOKEN
%token EXECUTE_TOKEN
%token TRANSIENT_TOKEN
%token ALPHA_TOKEN
%token SMOOTH_TOKEN
%token MAP_TOKEN
%token MAPPER_TOKEN
%token LINEAR_SOLVER_TOKEN
%token SOLVING_STRATEGY_TOKEN
%token EQUAL_TOKEN
%token NOT_EQUAL_TOKEN
%token LESS_EQUAL_TOKEN
%token GREATER_EQUAL_TOKEN
%token CREATE_SOLUTION_STEP_TOKEN
%token CREATE_TIME_STEP_TOKEN

%type <statement_handler> statement
%type <statement_handler> expresion
%type <statement_handler> node_generating
%type <statement_handler> properties_adding
%type <statement_handler> set_elements_group
%type <statement_handler> element_generating
%type <statement_handler> nodes_variables_fixing
%type <statement_handler> nodes_variables_setting
%type <statement_handler> elements_variables_fixing
```

```
%type <statement_handler> linear_solver_generating
%type <statement_handler> solving_strategy_generating
%type <statement_handler> mapper_generating
%type <statement_handler> solve_process
%type <statement_handler> move_mesh_process
%type <statement_handler> execute_process
%type <statement_handler> print_process
%type <statement_handler> print_on_gauss_points_process
%type <statement_handler> smooth_process
%type <statement_handler> map_process
%type <statement_handler> step_creating
%type <statement_handler> time_step_creating
%type <statement_handler> for_loop
%type <statement_handler> assignment_expresion
%type <statement_handler> logical_expresion
%type <block_statement_handler> block_generating
%type <block_statement_handler> block
%type <double_value_statement_handler> double_variable_expresion
%type <integer_value> integer_expresion
%type <integer_value> integer_index
%type <integer_value> nodes_array
%type <integer_value> elements_array
%type <integer_value> properties_array
%type <double_value> double_expresion
%type <vector_integer_value> integer_sequence
%type <vector_integer_value> vector_integer
%type <vector_double_value> vector_double
%type <matrix_double_value> matrix
%type <vector_double_value> double_sequence
%type <vector_vector_double_value> vector_double_sequence
```

The second part holds grammar definitions in context free format and their corresponding semantic actions. Here is the organization grammar for input:

```
statement               :   expresion ';' {$$ = $1}
                        |   block {$$ = $1;}
                        |   for_loop {$$ = $1;}
                         ;

expresion               :   assignment_expresion {$$ = $1;}
                        |   logical_expresion {$$ = $1;}
                        |   node_generating {$$ = $1}
                        |   properties_adding {$$ = $1;}
                        |   set_elements_group {$$ = $1;}
                        |   element_generating {$$ = $1;}
                        |   nodes_variables_fixing {$$ = $1;}
                        |   nodes_variables_setting {$$ = $1;}
                        |   elements_variables_fixing {$$ =$1;}
                        |   solve_process {$$ = $1;}
                        |   move_mesh_process {$$ = $1;}
                        |   execute_process {$$ = $1;}
                        |   print_process {$$ = $1;}
                        |   print_on_gauss_points_process {$$ = $1;}
                        |   smooth_process {$$ = $1;}
```

```
                        |   map_process {$$ = $1;}
                        |   linear_solver_generating {$$ =$1;}
                        |   solving_strategy_generating {$$ = $1;}
                        |   mapper_generating {$$ = $1;}
                        |   step_creating {$$ = $1;}
                        |   time_step_creating {$$ = $1;}
                        ;

block                   :   block_generating '}'
                            {
                                $$ = $1;
                            }

block_generating        :   '{' statement
                            {
                                $$ = new BlockStatement();
                                $$->AddStatement($2);
                            }
                        |   block_generating statement
                            {
                                $1->AddStatement($2);
                                $$ = $1;
                             }
```

The rest of definitions have not brought here to reduce the size of listing. The third part is for the auxiliary function. In our case an error reporting function is defined here is:

```
using Kratos::Exception;

void InputCParser::yyerror( char *s) {
  Kratos::String buffer;
  buffer << "Pars error in line no "
         << mInputCScanner.rNumberOfLines()
         << ", last token was : "
         << mInputCScanner.GetYYText();
  KRATOS_ERROR(std::runtime_error, "Reading Input", buffer, "");
}
```

### 9.4.5 Creating a new Interpreter Using Spirit

Flex++ and Bison++ are great tools to generate an interpreter but they have been put aside from this work for mainly two reasons. The first was changes in the strategy of code development form creating a sophisticated interpreter to using an existing one. The idea was to have a simple but flexible data IO and using an existing interpreter for process IO. By the way for reading data still a simple parser was necessary, but these tools were too heavy to be used for this case. The other reason was portability and maintaining issues.Any small changes in grammar, for example introducing a new variable name, requires to regenerate the parser and then recompile the program. In practice this yields portability problems specially in Windows, due to the incompatibilities between Linux and Windows compilations of these tools.

*Spirit* is an object-oriented parser generator framework implemented in C++ using template meta-programming techniques. Spirit enable us to write the context free grammar directly in C++ code and compile it with a C++ code to generate the parser. In this way the translation step from

context free grammar to a parser implemented in C++ by an external tool is omitted. There are several small parsers already defined in Spirit which help users to implement its parsers easier.

Parsing in Spirit can be done using one of the overloaded parse functions. Some of these overloads work in character level while some others work in phrase level and take and skip parser. This skip parser is helpful for example in skipping white spaces and comments. The `parse` function is also overloaded respected to its input. There are overloads that accept first and last operator like STL algorithms, and others which accept a null terminating string. Phrase level parse functions with first and last iterator as input are used in this work:

```
template <typename IteratorT, typename ParserT, typename SkipT>
parse_info<IteratorT> parse
    (
        IteratorT const&         first,
        IteratorT const&         last,
        parser<ParserT> const&   p,    // Grammar rules
        parser<SkipT> const&     skip // Skip rules
    );
```

Now let us start to create a parser for reading our node statement example. This helps to see how a simple parser can be generated using Spirit. Here is the statement to be parsed:

```
// a node statement:
// Node(Index,X,Y,Z);
Node(1, 0.02, 1.00, 0.00);
```

First we need to use Spirit grammar components to define our grammar. Table 9.4 shows valid operators in Spirit. These operators are defined to be as close as possible, respecting to the C++ language restrictions, to their corresponding regular expression operators. So understanding regular expressions helps to use Spirit as well.

| Operator | Description |
|---|---|
| `!P` | Zero or One P |
| `*P` | Zero or more P |
| `+P` | One or more P |
| `~P` | Anything except P |
| `P1 % P2` | One or more P1 separated by P2 |
| `P1 - P2` | P1 but not P2 |
| `P1 >> P2` | P1 followed by P2 |
| `P1 & P2` | P1 and P2 |
| `P1 ^ P2` | P1 or P2, but not both |
| `P1 \| P2` | P1 or P2 |
| `P1 && P2` | Synonym for P1 >> P2 |
| `P1 \|\| P2` | P1 or P2 or P1 >> P2 |

Table 9.4: Spirit's operators

Using this operators together with some predefined Spirit parsers creates the necessary grammar to read the node statements:

```
(
  str_p("Node")
  >> '('
```

```
    >> uint_p
    >> ','
    >> real_p
    >> ','
    >> real_p
    >> ','
    >> real_p
    >> ')'
  )
```

str_p is a predefined parser which matches input with given string which is "Node". uint_p is another parser which matches to an unsigned integer in input and used to read the index of the Node. real_p used to read coordinates. It matches to a real number from input. Finally any character matches to itself. This grammar can be used to parse the statement but still there is no action to do when input matches each part. So we have to add some actions to it:

```
  (
    str_p("Node")
    >> '('
    >> uint_p[assign_a(node.Id())]
    >> ','
    >> real_p[assign_a(node.X())][assign_a(node.X0())]
    >> ','
    >> real_p[assign_a(node.Y())][assign_a(node.Y0())]
    >> ','
    >> real_p[assign_a(node.Z())][assign_a(node.Z0())]
    >> ')'
  )
```

Here assign_a is an action which retrieves the value read by parser and assign it to its argument. Also it can be seen that actions can be cascaded when there are more than one action has to be performed for a rule. For handling comments and white spaces we need a skip parser. This parser can be written easily as below:

```
(space_p | comment_p("//") | comment_p("/*", "*/"))
```

space_p is a parser already defined in Spirit which matches to any white space character. There is another predefined parser in sprit which handles comment patterns. comment_p with just one argument matches to a sequence starting with given string and finished by end of line, like C++ comments. comment_p with two arguments matches with a sequence starting from the first argument and finished with second one, like C comments.

Now let us put every things together and create our ReadNode method:

```
void ReadNode(NodeType& rNode,
              IteratorType First,
              IteratorType Last)
{
   using namespace boost::spirit;
   parse(First, Last,
        // Begin grammar
        (
            str_p("Node")
            >> '('
            >> uint_p[assign_a(rNode.Id())]
```

```
            >> ','
            >> real_p[assign_a(rNode.X())][assign_a(rNode.X0())]
            >> ','
            >> real_p[assign_a(rNode.Y())][assign_a(rNode.Y0())]
            >> ','
            >> real_p[assign_a(rNode.Z())][assign_a(rNode.Z0())]
            >> ')'
        )
        // End grammar
        ,
        // Skip grammar
        (space_p | comment_p("//") | comment_p("/*", "*/")));
}
```

That's it! This function can parse any sequence of characters given by iterators `First` and `Last`. But we may need to parse an input data file. This can also be done by slightly modifying above function. Spirit provides a file iterator which can be used in the same way that normal the iterators are used. Here is the new version of `ReadNode` which is able to read a `Node` from an input file:

```
void ReadNode(NodeType& rNode,
              std::string Filename)
{
    using namespace boost::spirit;

    FileIterator first(Filename);
    FileIterator last= First.make_end();

    parse(first, last,
        // Begin grammar
        (
            str_p("Node")
            >> '('
            >> uint_p[assign_a(rNode.Id())]
            >> ','
            >> real_p[assign_a(rNode.X())][assign_a(rNode.X0())]
            >> ','
            >> real_p[assign_a(rNode.Y())][assign_a(rNode.Y0())]
            >> ','
            >> real_p[assign_a(rNode.Z())][assign_a(rNode.Z0())]
            >> ')'
        )
        // End grammar
        ,
        // Skip grammar
        (space_p | comment_p("//") | comment_p("/*", "*/")));
}
```

In this IO we also want to parse new components, for example new variables, `Elements`, or `Conditions`, without changing and updating the parser each time. Spirit provides a `symbols` class which fits well to our purpose. This class is a parser associated with a symbol table. This table can be initialized with different symbols and parser matches to any symbol stored in its table. So adding KratosComponents elements to a symbol class provides a parser to match these components.

```
template<class TComponentType>
  class  ComponentParser :
    public symbols<reference_wrapper<TComponentType const> >
  {
  public:
    ComponentParser()
    {
      typedef KratosComponents<TComponentType> ComponentsType;
      typedef ComponentsType::ComponentsContainerType ContainerType;
      typedef ContainerType::const_iterator iterator_type;

      iterator_type i_component;

      for(i_component = ComponentsType::GetComponents().begin() ;
          i_component != ComponentsType::GetComponents().end() ;
          i_component++)
            add(i_component->first.c_str(), i_component->second);
      }

  };
```

`ComponentParser` derived from `symbols` class and in its constructing time adds automatically all `TComponentsType` components to the symbol table. This make it a parser that match to any components stored in `KratosComponents` list of that type.

Finally all these concepts and tools are used together to create a flexible and extendible IO for Kratos.

## 9.5 Using Python as Interpreter

As we mentioned before during the evolution of Kratos, the strategy for implementing a complete interpreter for the IO part changed to use an existing one. This change in the development strategy, lead finally to using the Python interpreter.

### 9.5.1 Why another interpreter

In time of decisions about what to include in the IO features, a process IO was chosen as a way to introduce new algorithms without changing the code and recompile it. To be able to do this in practice IO has to provide a complete set of language flow control commands.

Implementing an interpreter is a hard work. From the management point of view, it introduces a significant overhead to project cost and time. Maintaining it is even harder and results in more overhead in the cost of the code. Also implementing an interpreter requires knowledge in different concepts like grammar notations, some tools and libraries usage and compiler implementation techniques. Finite element programmers usually are not familiar with most of these concepts and in many cases even do not like to deal with them. So in developing a finite element program it is not easy to share the implementation and maintenance of an interpreter with others due to this lack of experience. In practice, this can lead to longer implementation times and extra cost.

During the implementation of Kratos all these facts affected the implementation of an interpreter and made its development more and more difficult to the point that it became one of the bottlenecks of the project. Consequently the strategy changed by stop writing an interpreter and looking for an existing one.

Binding Kratos to Python makes it extremely flexible and extendible. New algorithms can be implemented using existing Kratos tools and methods without even recompiling the code. Interaction between domains and planning different staggered strategies to solve a coupled problem also can be perform easily at this level without changing the Kratos or its applications. Also it can be used as a small lab for testing new algorithms and formulations before programming it into the Kratos. The list of added features is unlimited and many complex tasks can be done easily using this interface.

### 9.5.2 Why Python

Selecting a script languages between the large amount of available script language is not an easy task. Each different language has its own advantages and disadvantages which can be useful for some cases but leaves uncovered some objectives. So an intention is to understand first the requirements and then find a language that fits more naturally to these tasks.

In this case there are different features required to be supported by the language:

- First of all any interpreter to be used has to have an interface to C++ or at least to C. It really makes no sense to choose an interpreter which cannot be bounded to our code.
- Portability is another essential point to be considered. Kratos is written in standard C++ and uses also portable libraries to enhance its portability. So the interpreter must continue this idea.
- To be object-oriented is an important feature. Though high level commands can be expressed by traditional languages but an object-oriented language can represent much better internal classes and data structures.
- Language syntaxes and its readability is another important feature. This depends on personal tastes but a self descriptive syntax considered to be clearer than the highly symbolized one.
- Ease of learning also is considered as an important point. A complex and difficult to learn language reduces the number of users who wants to use these IO features.
- Flexibility and extendibility are other features to be considered. Adding new types of data and also using them via the interface is essential in order to add finite element data and containers.
- An active language in sense of development will be preferred to an old but not active one due to the risk of new unresolved problems.
- Finally a popular language is preferable because the larger community of programmers helps a lot in dealing with day to day problems and questions.

From the long list of languages first some more popular ones were selected to be verified:

- *Lisp* is one of the widely used script languages. However its full of parenthesis syntax and lack of other features causes to be eliminated quickly from our list. Here is a square generator sample in Lisp:

```
(defun print-squares (upto)
    ( loop for i from 1 to upto
        do (format t "~A^2 = ~A~%" i (* i i))))
```

- *Perl* (Practical Extraction and Report Language) is a mature language with a fast interpreter. Also it is used to interpret data files with its built-in regular expressions, which is another added value for our case. Perl has a modular structure with some object-oriented features added to it later and is single thread without supporting multi-thread. Here is the print squares code in Perl:

```
for($i = 1 ; $i <= 10 ; ++$i) {
    print $i*$i, ' ';
}
```

- *Ruby* is a young but attractive full object oriented language including many features of Perl, with an intuitive syntax. It also supports multi-threading at the interpreting level which enables it to be used also in single thread operating systems.

```
for i in 1..10 do
    print i*i, " "
end
```

- *Tcl* is a widely used script language. It is portable and can be easily integrated with C. It is a modular language without object-oriented features. Here is the print squares example in Tcl:

```
set i 1
while {$i <= 10} {puts [expr $i*$i];set i [expr $i+1]}
```

- *Python* is a general purpose script language with object oriented features and very easy and intuitive syntax. It also support multi-threading. Here is the print squares example in Python:

```
for i in range(1,11): print i*i,
```

Finally Python was selected due to some details and also our background. Tcl was a good choice as it is already used in GiD and a large amount of experience was available. Lack of object-oriented features however prevented this choice. Perl also was considered for its maturity and performance, but again it is less object-oriented than Python and does not support multi-threading. Ruby at that time was completely new to us and also was considered to be somehow young.

There were some other reasons to chose Python. An existing well designed interface to Python was one important reason to be selected. Also there was some practical use of Python in finite element applications [10, 11].

### 9.5.3 Binding to Python

Boost Python library is used for binding Kratos to Python. This library provides an easy but powerful way to connect a C++ code to Python.

Now let us create a `Node` interface to Python and use it as a step by step introduction in using this library. The first step is to introduce the `Node` class itself. The following code introduces the `Node<3>` class as a `Node` identifier in Python:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer>("Node");
```

`class_` template introduces a class to Python. The first template parameter is the class to be imported. The second one is the handler in Python for objects of this type. In this case is the `Node<>::Pointer` which is a reference counted pointer. Boost Python library handles properly this type of pointer and binds it correctly to the memory manager of Python. The argument of constructor is the representative name of this class in Python. Python does not have templates so different instances of a template can be exposed to Python and not the template itself. For example here the instance `Node<3>` is exposed.

In Kratos `Node<>` is derived from `Point<>` and `IndexedObject`. We can keep this hierarchy by introducing `Node`'s bases in its definition:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType, IndexedObject > >("Node");
```

By introducing its bases, `Node` automatically inherites all imported methods of `Point<>` and `IndexedObject`. This class is exposed now and can be constructed by its default constructor. But we prefer to construct it by its id and coordinates. `init` provides a way to introduce a constructor and can be passed as another argument to the `class_` constructor:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>());
```

Another constructors can be added using `def` method of `class_`. For example a constructor by id and point can be exposed in the following way:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    ;
```

`def` uses a visitor pattern which makes it extendible to new concepts. Here `init` uses this pattern to expose a new constructor. Now we can create the `Node` in Python and calling its inherited methods from `Point<3>` and `IndexedObject`, but we cannot print it yet. To make Node printable, another statement must be added:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    .def(self_ns::str(self))
    ;
```

This statement uses the << operator already defined for Node to print its information. def can be used also to define member functions. For example Node.IsFixed(Variable) can be imported to Python as follow:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    .def("IsFixed", &NodeType::IsFixed)
    .def(self_ns::str(self))
    ;
```

To import Kratos as a module, the following statements are necessary:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(Kratos) {
  // Module components will be defined here.
}
```

Adding the IndexedObject, the Point<3> and the Node<3> interfaces results in the first Kratos module:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(Kratos) {

AddPointsToPython();

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    .def("IsFixed", &NodeType::IsFixed)
    .def(self_ns::str(self))
    ;

}
```

The Point interface is placed in another function and is just called here. This makes the code easier to read and also reduces the required memory to compile this file.

Compilation consists of compiling Boost Python library itself and above codes as a dynamic link library, then we must put them into Python dynamic link libraries folder.

In Python one must import this module to use all its components or just import the require components. Here is an example of using this module in Python:

```
from Kratos import *

# Constructing a node:
node = Node(1,2.00, 10.00, 0.00)
```

```
# Using X property inherited from Point:
node.X = 4.5

# Calling exposed IsFixed method
if(node.IsFix(TEMPERATURE)) :
    # Using << operator of Node
    print node
```

This simple case was just an introduction. Many other concepts from the Boost Python library are necessary to implement a real interface. Call policies, virtual and overloaded methods, exception handling and so on are examples of these concepts which are described in the library documentations.

Chapter 10

# Validation Examples

In this chapter some applications developed using Kratos are presented to test the desired flexibility and extensibility of the framework.

## 10.1 Incompressible Fluid Solver

In this example an incompressible fluid application implemented in Kratos is described. It shows the ability of Kratos to handle a standard finite element formulation efficiently while achieving performance comparable to single purpose codes.

### 10.1.1 Methodology Used

An Arbitrary Lagrangian Eulerian (ALE) formulation is used to solve the fluid problem [32]. The solver is based on fractional step method [26] using equal order pressure-velocity elements stabilized by Orthogonal subscales (OSS) [27]. This application uses an element based approach optimized for simplex elements.

### 10.1.2 Implementation in Kratos

The fractional step method chosen consists of four solution steps, of which the first one involves a nonlinear loop for solving the nonlinearity in the convection term, while the rest are linear and the third one involves the explicit calculation of projection terms.

This method can be implemented effectively by creating a new `SolvingStrategy` combining existing ones for different steps. The strategy is hard coded in C++, however the implementation is such that all the different steps can be solved separately. In this way a flexible Python interface allowing direct interaction with the solver can be defined. With this interface, different parts of the algorithm can be changed reusing existing strategies with minimal performance loss. This flexibility provides major advantages in defining new fluid structure interaction coupling strategies based in existing ones.

For the present application a combined `Strategy` is created to handle the solution process. The following code shows the `Solve` method of this `Strategy` which calls other methods for implementing different steps:

```
double Solve()
{
  // assign the correct fractional step coefficients
  InitializeFractionalStep(this->m_step, this->mtime_order);
  double Dp_norm;

  //predicting the velocity
  PredictVelocity(this->m_step,this->mprediction_order);

  //initialize projections at the first steps
  InitializeProjections(this->m_step);

  //Assign Velocity To Fract Step Velocity and Node Area to Zero
  AssignInitialStepValues();

  if(this->m_step <= this->mtime_order)
      Dp_norm = IterativeSolve();
  else
  {
      if(this->mpredictor_corrector == false) //standard fractional step
          Dp_norm = FracStepSolution();
      else  //iterative solution
          Dp_norm = IterativeSolve();
  }

  if(this->mReformDofAtEachIteration == true )
      this->Clear();

  this->m_step += 1;
  this->mOldDt =  BaseType::GetModelPart().GetProcessInfo()[DELTA_TIME];

  return Dp_norm;
}


double FracStepSolution()
{
  //setting the fractional velocity to the value of the velocity
  AssignInitialStepValues();

  //solve first step for fractional step velocities
  this->SolveStep1(this->mvelocity_toll, this->mMaxVelIterations);

  //solve for pressures (and recalculate the nodal area)
  double Dp_norm = this->SolveStep2();

  this->ActOnLonelyNodes();

  //calculate projection terms
  this->SolveStep3();

  //correct velocities
  this->SolveStep4();
```

```
    return Dp_norm;
}
```

This strategy can be exported to Python without loss of performance, by providing access to the methods implementing each step in above strategy class. We can write in Python an equivalent solution step as follows:

```
def SolutionStep1(self):
    normDx = Array3(); normDx[0] = 0.00; normDx[1] = 0.00; normDx[2] = 0.00;
    is_converged = False
    iteration = 0

    while(  is_converged == False and iteration < self.max_vel_its  ):
    (self.solver).FractionalVelocityIteration(normDx);
        is_converged = (self.solver).ConvergenceCheck(normDx,self.vel_toll);
    print iteration,normDx
        iteration = iteration + 1


def Solve(self):
    if(self.ReformDofAtEachIteration == True):
        (self.neighbour_search).Execute()

    (self.solver).InitializeFractionalStep(self.step, self.time_order);
    (self.solver).InitializeProjections(self.step);
    (self.solver).AssignInitialStepValues();


    self.SolutionStep1()

    (self.solver).SolveStep2();
    (self.solver).ActOnLonelyNodes();
    (self.solver).SolveStep3();
    (self.solver).SolveStep4();

    self.step = self.step + 1

    if( self.ReformDofAtEachIteration == True):
        (self.solver).Clear()
```

As can be seen the Python code is self explanatory and simple. Providing this interface also has the great advantage of allowing users to customize the global algorithm without accessing the internal implementation in Kratos.

In order to implement the elemental formulation a new `Element` has to be created. The `Element` should provide different contributions for each solution step. This is achieved by passing the current fractional step number as a variable of the `ProcessInfo` to the calculation method of the `Element`. Interestingly, this can be done without any modification of the standard elemental interface. This is one of the cases where the generality of the interface helps to integrate new types of formulations. The following code shows the structure of the calculation method for this `Element`:

```
void Fluid3D::CalculateLocalSystem(MatrixType& rLeftHandSideMatrix,
                                   VectorType& rRightHandSideVector,
                                   ProcessInfo& rCurrentProcessInfo)
```

```
{
  KRATOS_TRY

  int FractionalStepNumber = rCurrentProcessInfo[FRACTIONAL_STEP];

  if(FractionalStepNumber <= 3)
  {
      int ComponentIndex = FractionalStepNumber - 1;
      Stage1(rLeftHandSideMatrix,
             rRightHandSideVector,
             rCurrentProcessInfo,
             ComponentIndex);
  }
  else if (FractionalStepNumber == 4)
  {
      Stage2(rLeftHandSideMatrix,
             rRightHandSideVector,
             rCurrentProcessInfo);
  }

  KRATOS_CATCH("")
}
```

Where `Stage1` and `Stage2` are private methods.

### 10.1.3 Benchmark

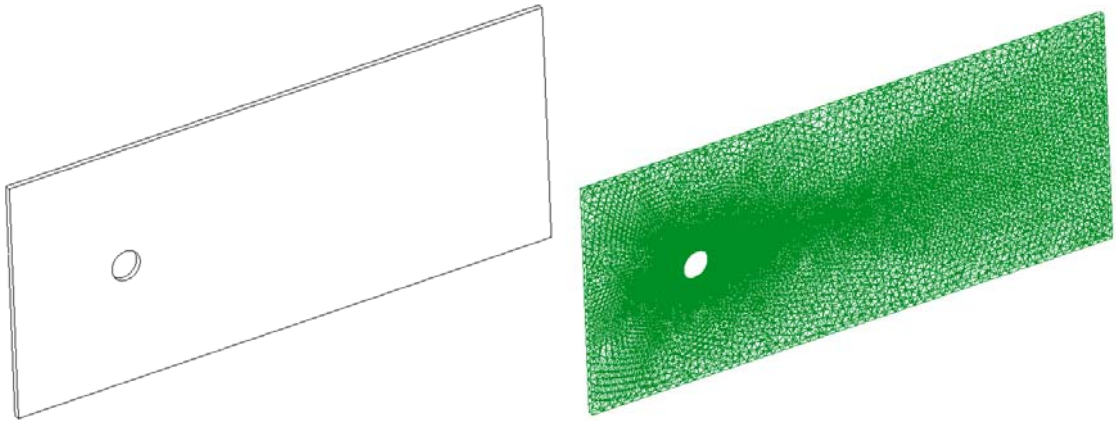
Kratos is a general purpose code. Therefore, it is expected to show a lower performance than codes optimized for a single purpose. A well optimized implementation can reduce the performance overhead to the amount introduced by Kratos. An effort was made to optimize the implementation mentioned above, so it is interesting to compare its performance against existing fluid solvers in order to estimate the order of performance overhead introduced by Kratos.

As usual it is not trivial to perform a good benchmark as each program implements a slightly different formulation. Nevertheless comparison was possible with the code Zephyr, an in house program in UPC, and with FEFLO a highly optimized fluid solver developed at the Laboratory for Computational Physics and Fluid Dynamics (LCPFD) in George Washington University at Washington, DC [56]. For the first case the formulation is exactly the same with only minor differences in the implementation. The second solver is an edge based formulation and the only possible comparison was with a predictor corrector scheme.

The benchmark represents the analysis of a three dimensional cylinder at Reynolds number $Re = 190$. Figure 10.1 shows the model used. The no slip boundary condition is used at the walls of the cylinder, while slip conditions are used everywhere else. The inflow velocity is set to $1m/s$. The mesh provided by Prof. R. Löhner with resolved boundary layer and contains 30000 nodes and $108k$ elements. Figure 10.2 shows the mesh used for this test. The values computed were the lift and drag history for the cylinder.

Figures 10.3 and 10.4 show the pressure and velocity calculated by Kratos. The results showed an excellent agreement with the values calculated by FEFLO both in term of peak values and of shedding frequency, as can be seen in figures 10.5 and 10.6.

The timing results are interesting. FEFLO appeared to be 50% faster than Kratos. This is considered a good result taking in account that FEFLO features a highly optimized edge based data structure while Kratos is purely element based.



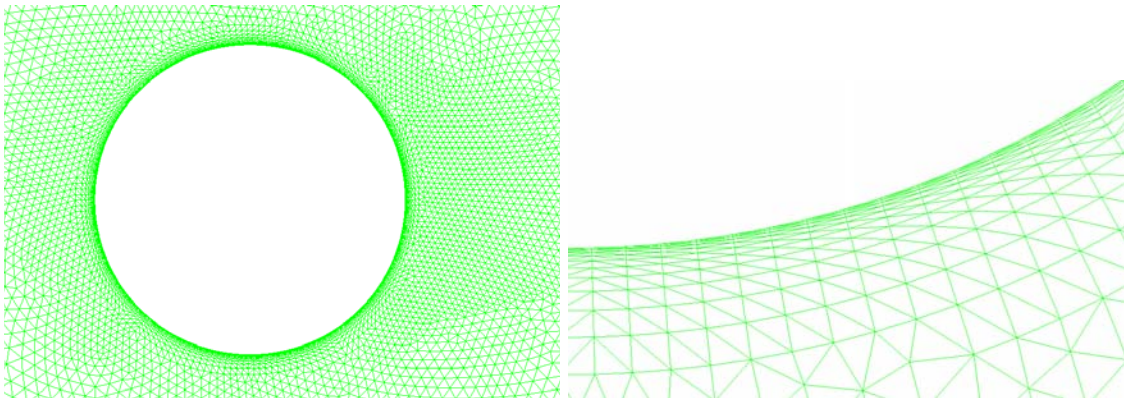
(a) (b)

Figure 10.1: a) Geometry of cylinder example, domain dimension $19.0 \times 8.0 \times 0.2$ and $R_c = 0.5$. b) the mesh used.

On the other hand, Zephyr features a element based formulation and implements the same fractional step. The main difference was the treatment of the projection terms and the use of four integration points for the calculation of the element contributions in Zephyr. The results showed that Kratos is about 100% faster with a solution time around one half of the solution time of Zephyr.

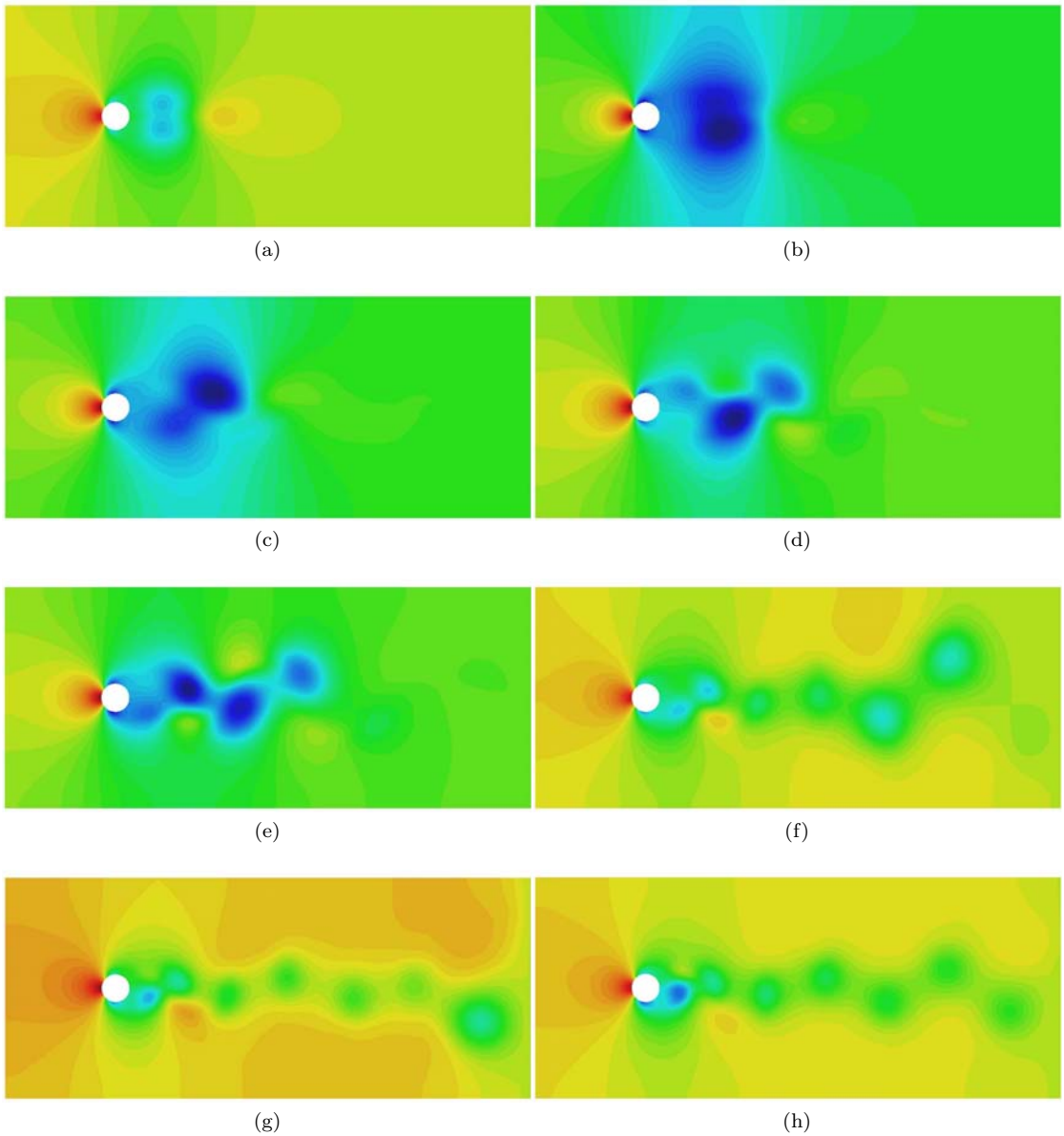
## 10.2 Fluid-Structure Interaction

Coupled problems can be naturally implemented inside Kratos taking advantage of the Python interface. The fluid solver and the structural solver can be implemented separately and coupled using this interface without any problem. The first action required to solve a fluid structure interaction (FSI) problem is to load the different applications involved. The following code shows



(a) (b)

Figure 10.2: Detail of the mesh used for the cylinder example.



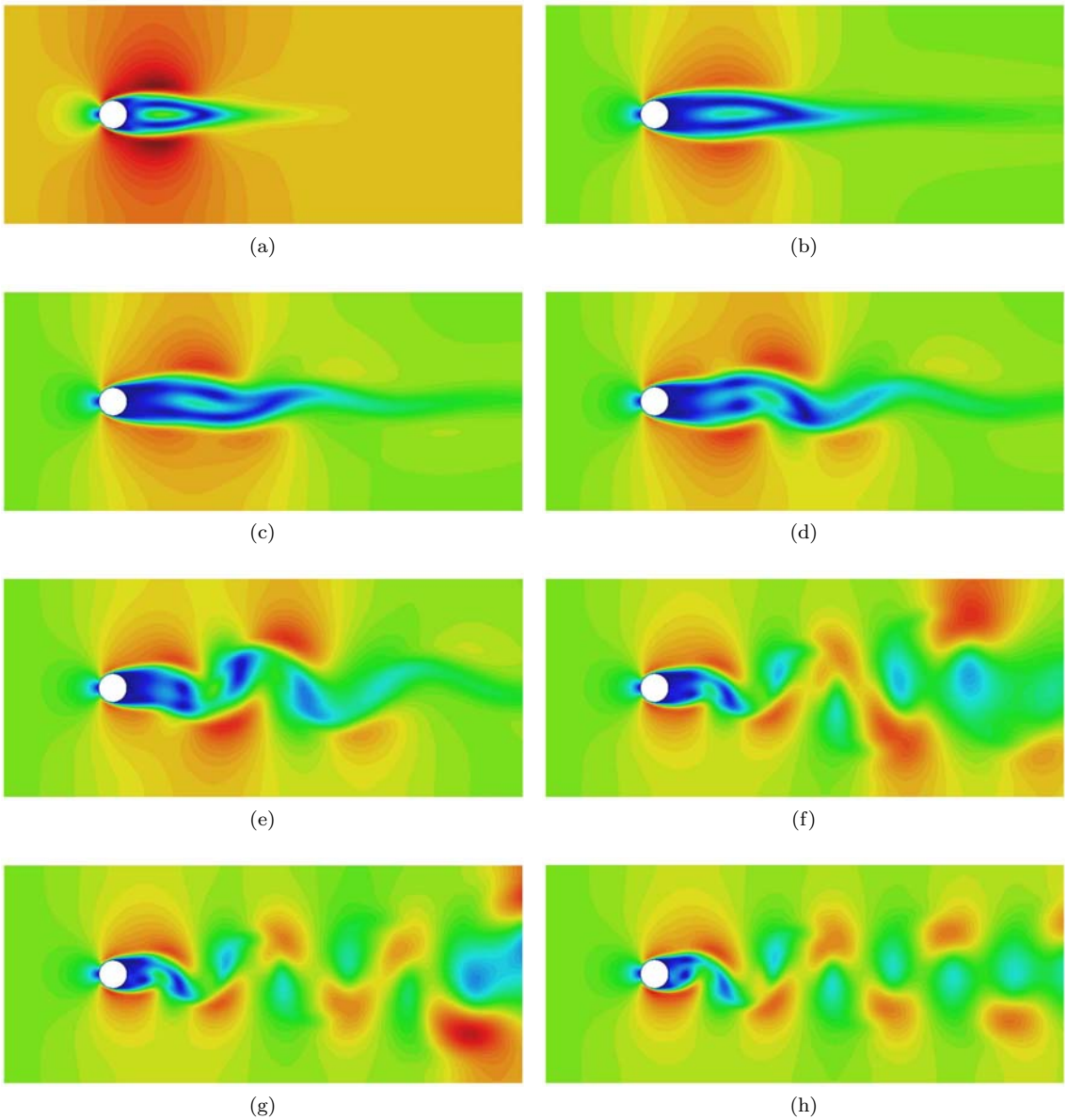
(a) (b)

(c) (d)

(e) (f)

(g) (h)

Figure 10.3: Pressure at different time steps.



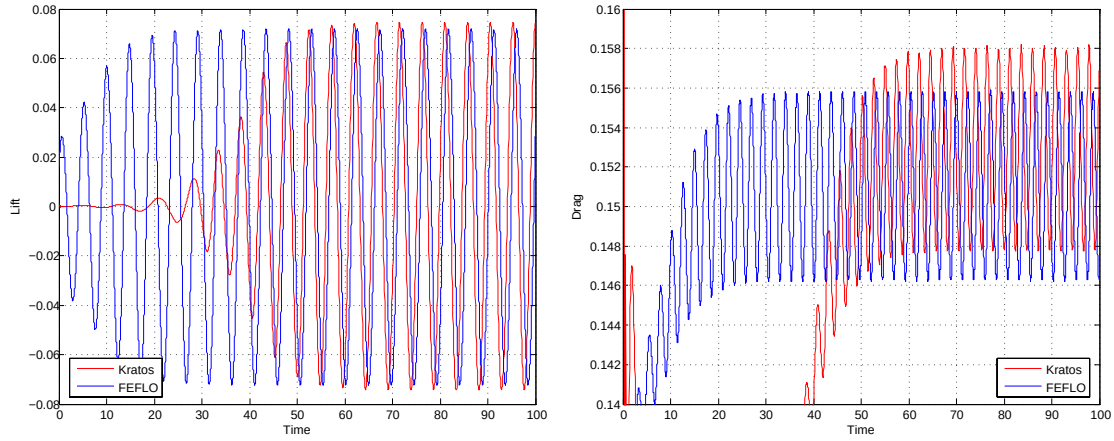
(a) (b)

(c) (d)

(e) (f)

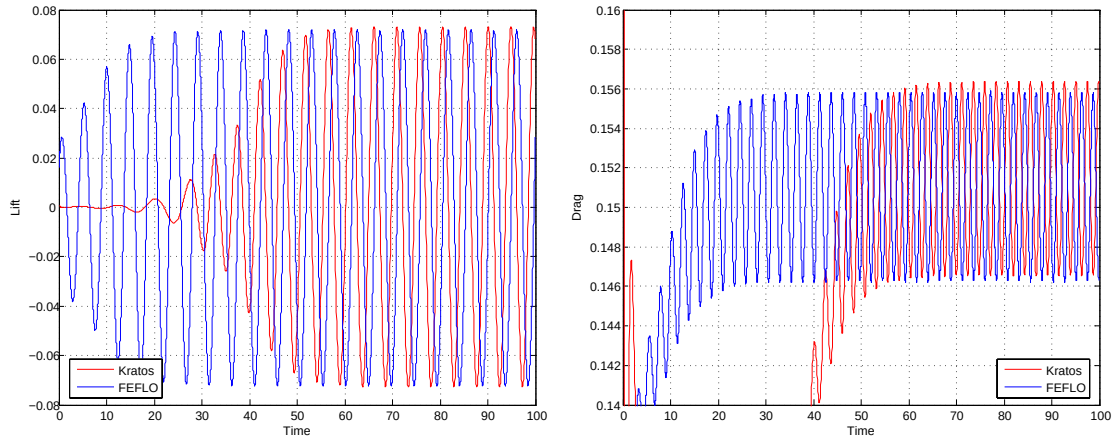
(g) (h)

Figure 10.4: Velocity at different time steps.




(a) (b)

Figure 10.5: The comparison of results obtained by Kratos (red line) using a fractional step algorithm and FEFLO (blue line). a) Lift calculated for the cylinder b) Drag calculated for the cylinder.




(a) (b)

Figure 10.6: The comparison of results obtained by Kratos (red line) using a predictor corrector scheme and FEFLO (blue line). a) Lift calculated for the cylinder b) Drag calculated for the cylinder.

this step in Python:

```
#including kratos path
kratos_libs_path = 'kratos/libs/'
kratos_applications_path = 'kratos/applications/'
import sys
sys.path.append(kratos_libs_path)
sys.path.append(kratos_applications_path)

#importing Kratos main library
from Kratos import *
kernel = Kernel()   #defining kernel

#importing applications
import applications_interface
applications_interface.Import_ALEApplication = True
applications_interface.Import_IncompressibleFluidApplication = True
applications_interface.Import_StructuralApplication = True
applications_interface.Import_FSIApplication = True
applications_interface.ImportApplications(kernel, kratos_applications_path)
```

Then a very simple explicit coupling procedure can be expressed as:

```
class ExplicitCoupling:

  def Solve(self):

    # solve the structure (prediction)
    (self.structural_solver).Solve()

    ## map displacements to the structure
    (self.mapper).StructureToFluid_VectorMap(DISPLACEMENT,DISPLACEMENT)

    ## move the mesh
    (self.mesh_solver).Solve()

    ## set the fluid velocity at the interface to
    ## be equal to the corresponding mesh velocity
    self.CopyVectorVar(MESH_VELOCITY,VELOCITY,self.interface_fluid_nodes);

    ## solve the fluid
    (self.fluid_solver).Solve()

    ## map displacements to the structure
    (self.mapper).FluidToStructure_ScalarMap(PRESSURE,POSITIVE_FACE_PRESSURE)

    # solve the structure (correction)
    (self.structural_solver).Solve()
```

Of course, many different coupling schemes can be implemented without making changes to the single field solvers. An example of fluid-structure interaction is given in figure 10.7.

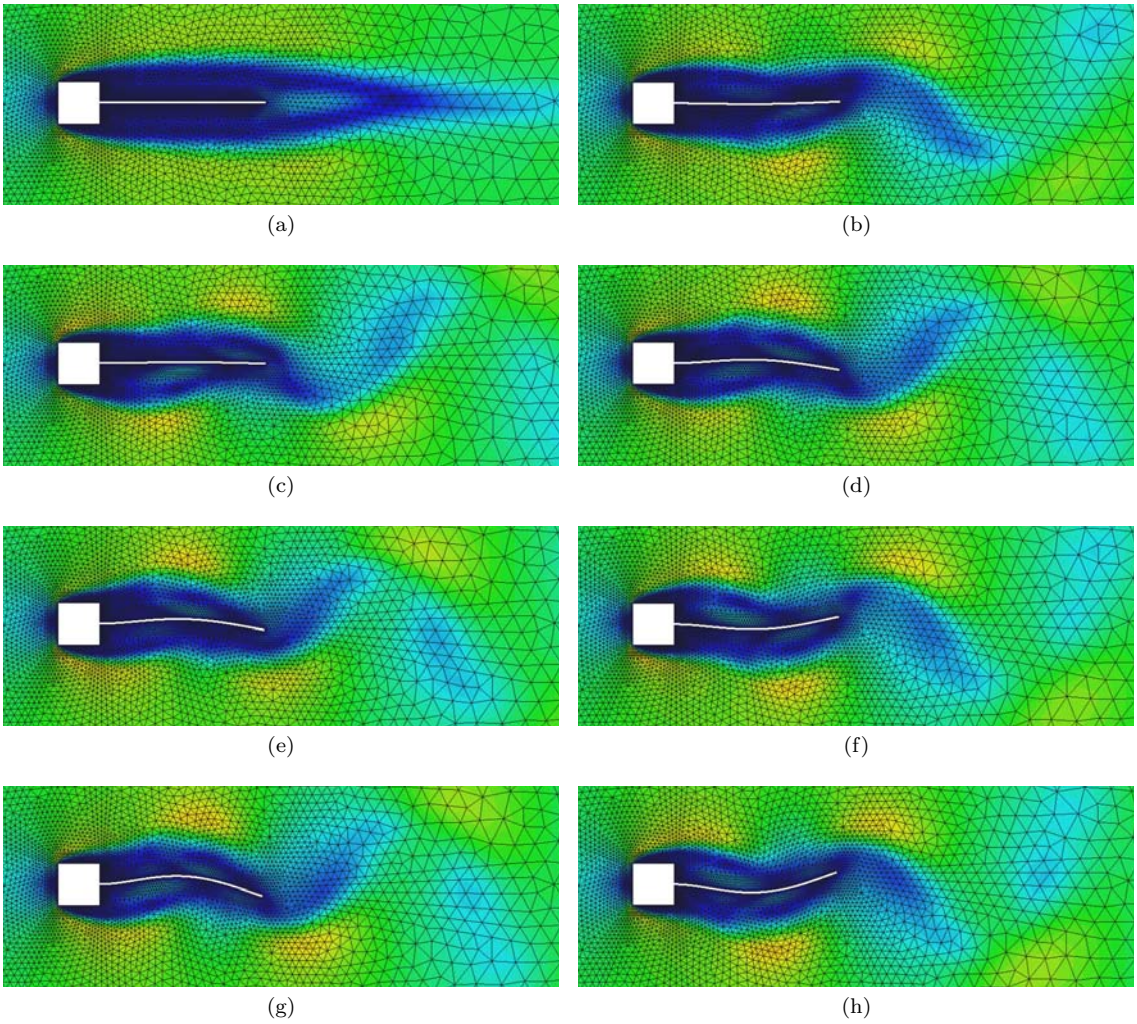


Figure 10.7: Flag flatter simulation using fluid structure interaction with mesh movement.

## 10.3 Particle Finite Element Method

The Particle Finite Element Method (PFEM) [76, 51, 50, 49] is a method for the solution of fluid problems on arbitrarily varying domains. The basic concept is that each particle is followed in a lagrangian way and the mesh is regenerated at each time step.

The main computational challenges faced to are the efficient regeneration of the mesh and the optimized recalculation of all elemental contributions. Good performance is achieved by linking with an external mesh generation library and by using the optimized Kratos fluid solver. The solution sequence is controlled by the Python interface. Here a part of the Python script is given:

```
def Solve(self,time,gid_io):

    self.PredictionStep(time)
```

```
    self.FluidSolver.Solve()

def PredictionStep(self,time):
    domain_size = self.domain_size

    # performing a first order prediction of the fluid displacement
    (self.PfemUtils).Predict(self.model_part)
    self.LagrangianCorrection()
    (self.MeshMover).Execute();

    (self.PfemUtils).MoveLonelyNodes(self.model_part)
    (self.MeshMover).Execute();

    ## ensure that no node gets too close to the walls
    (self.ActOnWalls).Execute();

    ## move the mesh
    (self.MeshMover).Execute();

    ## smooth the final position of the nodes to
    ## homogenize the mesh distribution
    (self.CoordinateSmoother).Execute();

    ## move the mesh
    (self.MeshMover).Execute();

    # regenerate the mesh
    self.Remesh()
```

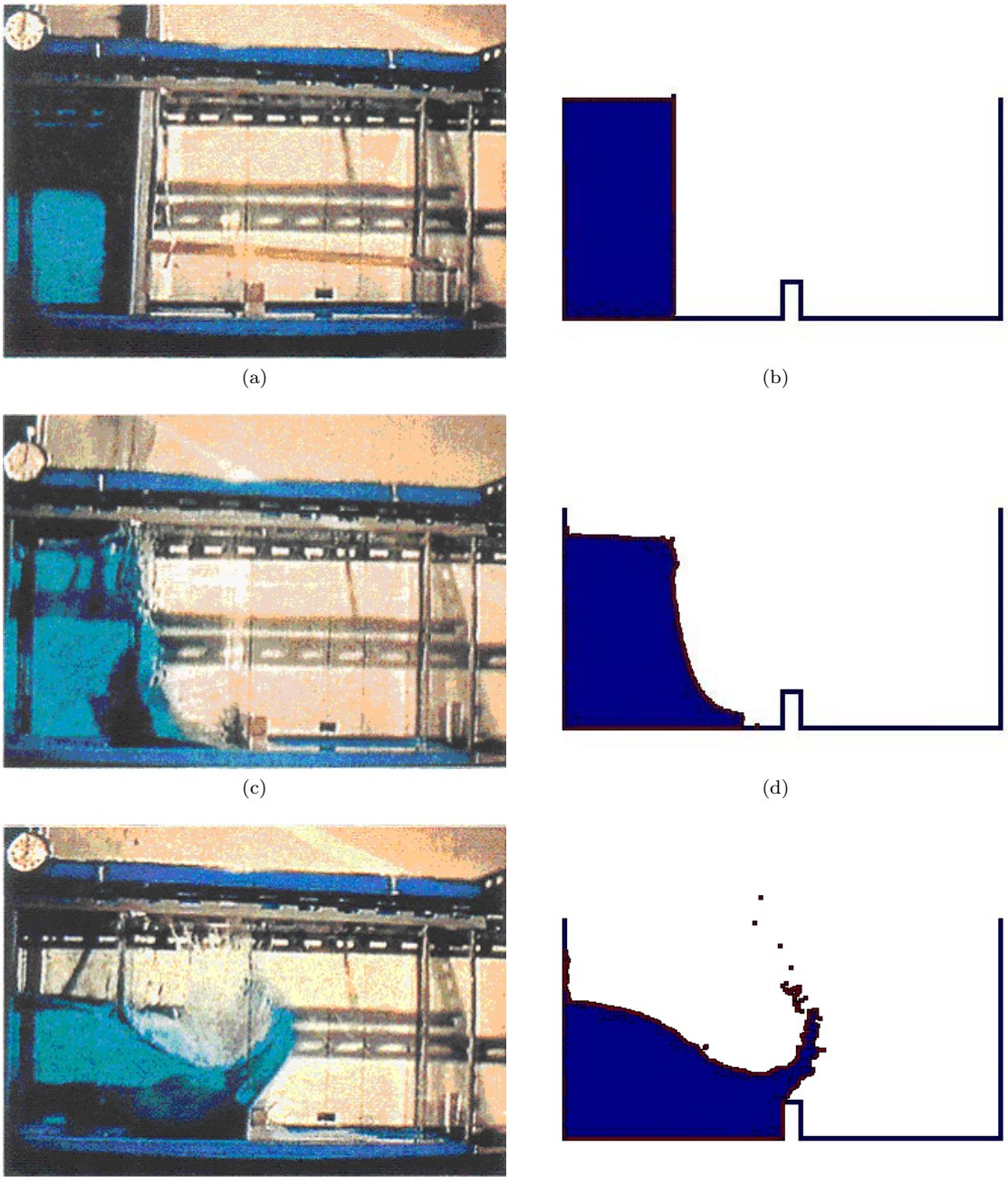
This example shows how a previously implemented fluid solver can be reused when implementing a new algorithm. This reusability allows the fast development of new formulations which can be tested by solving large scale real-life problems. Figure 10.8 shows a dam break simulation done with the PFEM application implemented in Kratos [57].

## 10.4 Thermal Inverse Problem

Inverse problems are found in many areas of science and engineering. They can be described as being opposite to direct problems. In a direct problem the cause is given, and the effect is determined. In an inverse problem the effect is given, and the cause must be estimated [52]. There are two main types of inverse problems: input estimation problems, in which the system properties and output are known and the input is to be estimated; and properties estimation problems, in which the the system input and output are known and the properties are to be estimated [52].

Mathematically, inverse problems fall into the more general class of variational problems. The aim of a variational problem is to find a function which minimizes the value of a specified functional. By a functional, we mean a correspondence which assigns a number to each function belonging to some class. Also, inverse problems might be *ill-posed* in which case the solution might not meet existence, uniqueness or stability requirements.

While some simple inverse problems can be solved analytically, the only practical technique for general problems is to approximate the solution using direct methods. The fundamental idea underlying the so called direct methods is to consider the variational problem as a limit problem

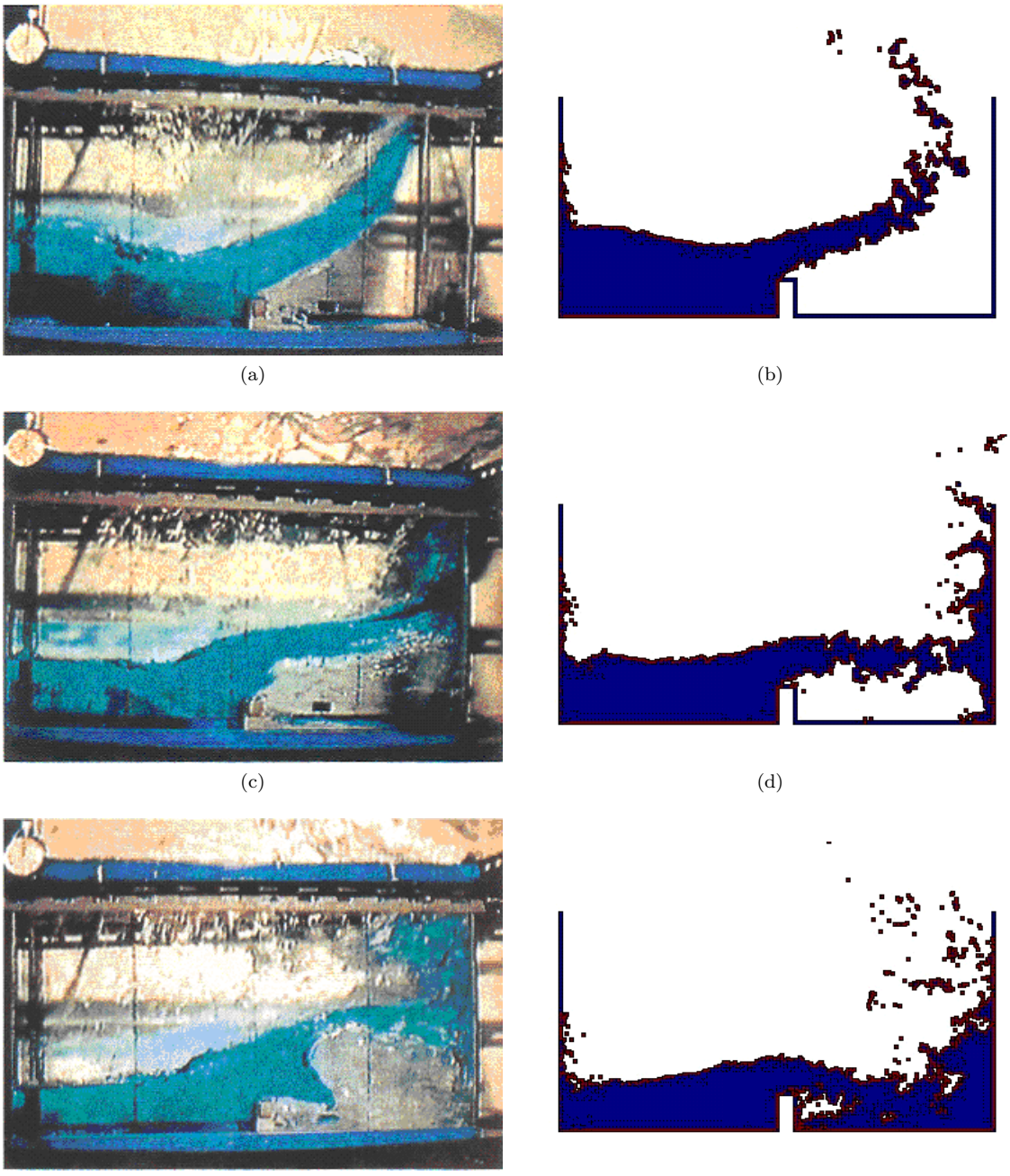


(a) (b)

(c) (d)

(e) (f)

Figure 10.8: A Dam break test and its simulation by the PFEM implementation in Kratos. [57]



(a) (b)

(c) (d)

(e) (f)

Figure 10.9: Dam break, continued. [57]

for some function optimization problem in many dimensions. Unfortunately, due to both their variational and ill-posed nature, inverse problems are difficult to solve.

*Neural networks* is one of the main fields of artificial intelligence [47]. There are many different types of neural networks, of which the *multilayer perceptron* is an important one [94]. Neural networks provide a direct method for the solution of general variational problems and consequently inverse problems [30].

In this example neural networks are used to solve thermal inverse problems. In order to solve this problem we need to solve the heat transfer equation. The *Flood* library [60] developed at CIMNE is an open source neural networks library written in C++ and used to create the neural network necessary for solving this problem. While this library does not include utilities for solving partial differential equations, it uses Kratos and its thermal application to solve the thermal problem. This example validates the integrability of Kratos as a library in another project. It also demonstrates its robustness due to the fact that this algorithm runs Kratos to analyze the same model several times. In this situation any small problem (for example, in memory management) might cause an execution error.

### 10.4.1 Methodology

The general solution of variational problems using Neural networks consists of three steps [**?**]:

- Definition of the functional space. The solution here is to be represented by a multilayer perceptron.
- Formulation of the variational problem. For their effect a performance functional $F(y(x, a))$ must be defined. In order to evaluate that functional we need to solve a partial differential equation which is done using the FEM within Kratos.
- Solution of the reduced function optimization problem f(a). This is achieved by the training algorithm. The training algorithm will evaluate the performance function $f(a)$ many times.

Figure 10.10 shows these three steps.

This algorithm provides an example of how Kratos can be embedded into an optimization application in which different steps of finite element analysis are necessary to achieve the solution.

Kratos has been embedded inside the Flood library as its solving engine in order to calculate the solution of partial differential equations.

Here the above methodology is applied to solve two different thermal inverse problems.

### 10.4.2 Implementation

The Flood library uses Kratos to solve a thermal problem several times with different properties and boundary conditions. In order to do this it has to access to internal data of Kratos and change the boundary conditions assigned to the different `Node`s. This is done without any file interface which would dramatically reduce performance. The first part is the interface for the direct solution using Kratos. The following code shows the main part of this interface:

```
// Initializing Kratos kernel
Kratos::Kernel kernel; kernel.Initialize();

// Initializing Kratos thermal application
Kratos::KratosThermalApplication kratosThermalApplication;
kernel.AddApplication(kratosThermalApplication);
```

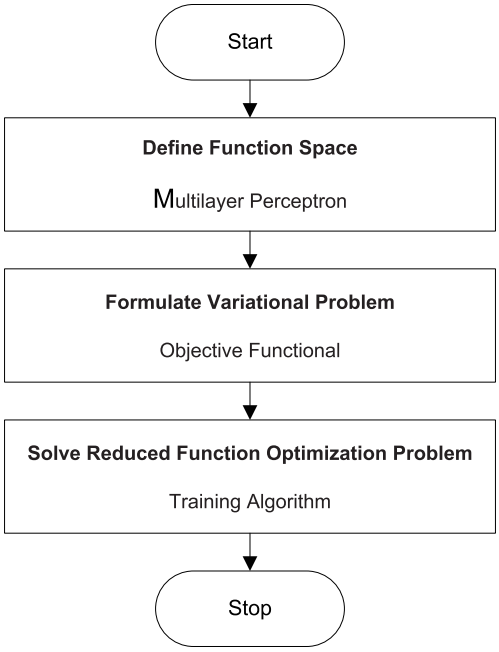



Figure 10.10: General solution of variational problems using neural networks consists of three main steps.

```
// Read mesh
Kratos::GidIO gidIO("thermal_problem"); gidIO >> mesh;

// Set properties
mesh.GetProperties(1)[DENSITY] = density;
mesh.GetProperties(1)[SPECIFIC_HEAT_RATIO] = specificHeat;
mesh.GetProperties(1)[THERMAL_CONDUCTIVITY] = thermalConductivity;

// Assign initial temperature
for(MeshType::NodeIterator i_node = mesh.NodesBegin();
    i_node != mesh.NodesEnd(); i_node++)
  if(!(i_node->IsFixed(TEMPERATURE)))
    i_node->GetSolutionStepValue(TEMPERATURE) = initialTemperature;

// Creating solver
// ...

// Main loop
for(int i = 1; i < numberOfTimeSteps; i++) {
   // Obtain time
   time[i] = time[i-1] + deltaTime;

   // Obtain boundary temperature
   // Gaussian function
```

```
    double mu = 0.5;
    double sigma = 0.05;

    double numerator = exp(-pow(time[i]-mu,2)/(2.0*pow(sigma,2)));
    double denominator = 8*sigma*sqrt(2.0*pi);

    boundaryTemperature[i] = numerator/denominator;

    // Assign boundary temperature
    for(MeshType::NodeIterator i_node = mesh.NodesBegin() ;
        i_node != mesh.NodesEnd() ; i_node++)
      if(i_node->IsFixed(TEMPERATURE))
        i_node->GetSolutionStepValue(TEMPERATURE) =
                                        boundaryTemperature[i];

    // Solving using thermal solver
    Solve();

    // Now updating the nodal temperature values
    // by result of solved equation system.
    Update();

    // Obtain node temperature
    nodeTemperature[i] =
      mesh.GetNode(nodeIndex).GetSolutionStepValue(TEMPERATURE);

    equation_system.ClearData();
}
```

The part initializing the solver has been removed to make the sample code shorter and only the parts that Flood uses to interact with Kratos are kept. This code shows the flexible but clear and intuitive interface that Kratos provides for other applications to communicate with it. First, application changes the `Element` properties to its prescribed values. Then, it changes the temperature value for all `Nodes` to some initial value. Afterwards it tries to solve using different boundary conditions assigning fixed values of temperature to `Nodes`. Finally it takes the temperature at a specific `Node`. It can be seen that some steps are directly inside the time loop. This restrict us from solving the problem using usual time processes.

The inverse problem also needs similar steps but in the form of performance functions of the Flood library. In this case Kratos is adapted to the working methodology of Flood without problems.

### 10.4.3 The Boundary Temperature Estimation Problem

For the boundary temperature estimation problem, consider the heat equation in the square domain $\Omega = \{(x, y) : |x| \leq 0.5, |y| \leq 0.5\}$ with boundary $\Gamma = \{(x, y) : |x| = 0.5, |y| = 0.5\}$,

$$\nabla^2 u(x, y; t) = \frac{\partial u(x, y; t)}{\partial t} \quad in \quad \Omega, \tag{10.1}$$

for $t \in [0, 1]$, with the initial condition $u(x, y; 0) = 0$ in $\Omega$. The problem is to estimate the boundary

temperature $y(t)$ on $\Gamma$ and for $t \in [0,1]$, from measurements of the temperature at the center of the square $u(0,0;t)$ for $t \in [0,1]$,

$$\begin{matrix} t_1 & u_1(0,0;t_1) \\ t_2 & u_2(0,0;t_2) \\ \vdots & \vdots \\ t_P & u_P(0,0;t_P) \end{matrix}$$

where $P$ is the number of time steps considered. For this problem we use 101 time steps.

The first stage in solving this problem is to choose a network architecture to represent the boundary temperature $y(t)$ for $t \in [0,1]$. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used [47]. This neural network is a class of universal approximator [48]. The network must have one input and one output neuron. We guess a good number of neurons in the hidden layer to be six. This neural network spans a family $V$ of parameterized functions $y(t;\underline{\alpha})$ of dimension $s = 19$, which is the number of free parameters in the network.

The second stage is to derive a performance functional in order to formulate the variational problem. This is to be the mean squared error between the computed temperature at the center of the square for a given boundary temperature and the measured temperature at the center of the square,

$$F[y(t;\underline{\alpha})] \quad = \quad \frac{1}{P}\sum_{i=1}^{P}\left(\hat{u}_{y(t;\underline{\alpha})}(0,0;t_i) - u_i(0,0;t_i)\right)^2. \tag{10.2}$$

Please note that evaluation of the performance functional (10.2) requires a numerical method for solving partial differential equations. Kratos is used here to solve this problem.

The boundary temperature estimation problem for the multilayer perceptron can then be formulated as follows:

> *Let $V$ be the space consisting of all functions $y(t;\underline{\alpha})$ spanned by a multilayer perceptron with 1 input, 6 sigmoid neurons in the hidden layer and 1 linear output neuron. The dimension of $V$ is 19. Find a vector of free parameters $\underline{\alpha}^* \in \mathbf{R}^{19}$ that addresses a function $y^*(t;\underline{\alpha}^*) \in V$ for which the functional (10.2), defined on $V$, takes on a minimum value.*

The third stage in solving this problem is to choose a suitable training algorithm. Here we use a conjugate gradient with Polak-Ribiere train direction and Brent optimal train rate [24]. The tolerance in the Brent's method is set to $10^{-6}$. Training of the neural network with the conjugate gradient algorithm requires the evaluation of the performance function gradient vector $\nabla f(\underline{\alpha})$. This is carried out by means of numerical differentiation [24]. In particular, we use the symmetrical central differences method [24] with an $\epsilon$ value of $10^{-6}$.

In this example, we set the training algorithm to stop when the norm of the performance function gradient $\nabla f(\underline{\alpha})$ falls below $10^{-6}$. That means that the necessary condition for a local minimum has been satisfied. The neural network is initialized with a vector of free parameters chosen at random in the interval $[-1,1]$. During the training process the performance function decreases until the stopping criterium is satisfied. Table 10.1 shows the training results for this problem. Here $N$ denotes the number of epochs, $M$ the number of performance evaluations, $F[y^*(t;\underline{\alpha}^*)]$ the final performance, and $\nabla f(\underline{\alpha}^*)$ the final performance function gradient norm.

Figure 10.11 shows the actual boundary temperature, the boundary temperature estimated by the neural network, and the measured temperature at the center of the square for this problem.

| $N$ | $M$ | $F[y^*(t;\underline{\alpha}^*)]$ | $\nabla f(\underline{\alpha}^*)$ |
|---|---|---|---|
| 421 | 25352 | $2.456 \cdot 10^{-4}$ | $8.251 \cdot 10^{-7}$ |

Table 10.1: Training results for the boundary temperature estimation problem.

The solution here is good, since the estimated boundary temperature matches the actual boundary temperature.

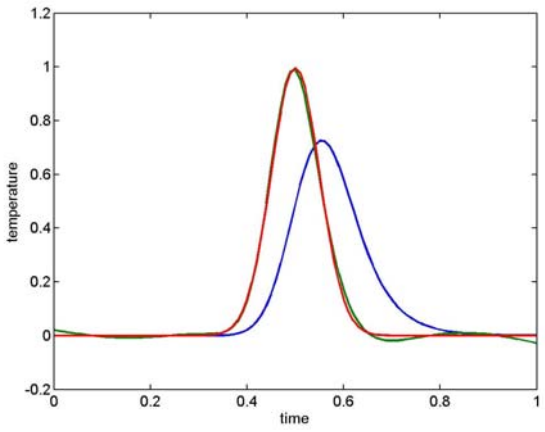



Figure 10.11: Actual boundary temperature (red), estimated boundary temperature (green) and measured temperature at the center of the square (blue) for the boundary temperature estimation problem.

### 10.4.4 The Diffusion Coefficient Estimation Problem

For the diffusion coefficient estimation problem, consider the equation of diffusion in an inhomogeneous medium in the square domain $\Omega = \{(x,y) : |x| \leq 0.5, |y| \leq 0.5\}$ with boundary $\Gamma = \{(x,y) : |x| = 0.5, |y| = 0.5\}$,

$$\nabla\left(\kappa(x,y)\nabla u(x,y;t)\right) = \frac{\partial u(x,y;t)}{\partial t} \quad in \quad \Omega, \tag{10.3}$$

for $t \in [0,1]$ and where $\kappa(x,y)$ is called the diffusion coefficient, with boundary condition $u(x,y;t) = 0$ on $\Gamma$ and for $t \in [0,1]$, and initial condition $u(x,y;0) = 1$ in $\Omega$. The problem is to estimate the diffusion coefficient $\kappa(x,y)$ in $\Omega$, from measurements of the temperature at different points on the square $u(x,y;t)$ in $\Omega$ and for $t \in [0,1]$,

$$\begin{array}{cccccc} t_1 & u_{11}(x_1,y_1;t_1) & u_{12}(x_2,y_2;t_1) & \dots & u_{1Q}(x_Q,y_Q;t_1) \\ t_2 & u_{21}(x_1,y_1;t_2) & u_{22}(x_2,y_2;t_2) & \dots & u_{2Q}(x_Q,y_Q;t_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ t_P & u_{P1}(x_1,y_1;t_P) & u_{P2}(x_2,y_2;t_P) & \dots & u_{PQ}(x_Q,y_Q;t_P) \end{array}$$

where $P$ and $Q$ are the number of points and time steps considered, respectively. For this problem we use 485 points and 11 time steps.

The first stage in solving this problem is to choose a network architecture to represent the diffusion coefficient $\kappa(x, y)$ in $\Omega$. Here a multilayer perceptron with a sigmoid hidden layer and a linear output layer is used. This neural network is a class of universal approximator [48]. The neural network must have two inputs and one output neuron. We guess a good number of neurons in the hidden layer to be six. This neural network is denoted as a $2 : 6 : 1$ multilayer perceptron. It spans a family $V$ of parameterized functions $\kappa(x, y; \underline{\alpha})$ of dimension $s = 25$, which is the number of free parameters in the network.

The second stage is to derive a performance functional for the diffusion coefficient estimation problem. The performance functional for this problem is to be the mean squared error between the computed temperature for a given diffusion coefficient and the measured temperature,

$$E[\kappa(x, y; \underline{\alpha})] = \frac{1}{PQ} \sum_{i=1}^{P} \left( \sum_{j=1}^{Q} \left( \hat{u}_{\kappa(x,y;\underline{\alpha})}(x_j, y_j; t_i) - u_{ij}(x_j, y_j; t_i) \right)^2 \right), \tag{10.4}$$

The diffusion coefficient estimation problem for the multilayer perceptron can then be formulated as follows:

> *Let $V$ be the space consisting of all functions $\kappa(x, y; \underline{\alpha})$ spanned by a multilayer perceptron with 2 inputs, 6 sigmoid neurons in the hidden layer and 1 linear output neuron. The dimension of $V$ is 25. Find a vector of free parameters $\underline{\alpha}^* \in \mathbf{R}^{25}$ that addresses a function $\kappa^*(x, y; \underline{\alpha}^*) \in V$ for which the functional (10.4), defined on $V$, takes on a minimum value.*

Evaluation of the performance functional (10.4) requires a numerical method for integration of partial differential equations. Here we choose the Finite Element Method [104]. For this problem we use a triangular mesh with 888 elements and 485 nodes.

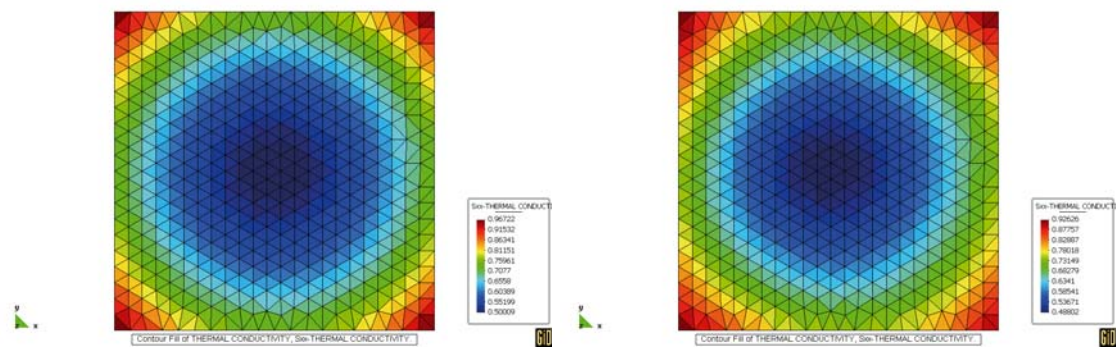
The third stage is to choose a suitable algorithm for training. Here we use a conjugate gradient with Polak-Ribiere train direction and Brent optimal train rate methods for training [24]. The tolerance in the Brent's method is set to $10^{-6}$. Training of the neural network with the conjugate gradient algorithm requires the evaluation of the performance function gradient vector $\nabla f(\underline{\alpha})$ [24]. This is carried out by means of numerical differentiation. In particular, we use the symmetrical central differences method [24] with $\epsilon = 10^{-6}$.

In this example, we set the training algorithm to stop when the norm of the performance function gradient falls below $10^{-6}$. That means that the necessary condition for a local minimum has been satisfied. The neural network is initialized with a vector of free parameters chosen at random in the interval $[-1, 1]$. Table 10.1 shows the training results for this problem. Here $N$ denotes the number of epochs, $M$ the number of performance evaluations, $F[\kappa^*(x, y; \underline{\alpha}^*)]$ the final performance, and $\nabla f(\underline{\alpha}^*)$ the final performance function gradient norm.

| $N$ | $M$ | $F[\kappa^*(x, y; \underline{\alpha}^*)]$ | $\nabla f(\underline{\alpha}^*)$ |
|---|---|---|---|
| 312 | 22652 | $1.562 \cdot 10^{-5}$ | $7.156 \cdot 10^{-7}$ |

Table 10.2: Training results for the diffusion coefficient estimation problem.

Figure 10.12 shows the actual diffusion coefficient and the diffusion coefficient estimated by the neural network for this problem. The solution here is good, since the estimated diffusion coefficient matches the actual diffusion coefficient.




(a) (b)

Figure 10.12: Actual diffusion coefficient a) and estimated diffusion coefficient b) for the diffusion coefficient estimation problem.

Further applications and other class of problems can be found in [73, 85, 86, 88, 87, 29].

Chapter 11

# Conclusions and Future Work

## 11.1 Conclusions

Kratos, a framework for developing multi-disciplinary programs has been designed and implemented. It helps developers in implementing applications for different fields of analysis by providing input-output, data structures, solvers, basic tools, and standard algorithms. The applications implemented in this framework can be used for solving multi-disciplinary problems using any master and slave strategies or even by solving simultaneously. At this moment several solvers (incompressible fluid, structural, thermal, and electromagnetic) are implemented in Kratos. Combination of these applications are also used to solve different multi-disciplinary problems, specially fluid-structure interaction and thermal-structural problems.

This framework provides a high level of flexibility and generality which is required for dealing with multi-disciplinary problems. Developers in different areas can configure Kratos for their needs without altering the standard interface used to communicate with other fields in coupled analysis. Different applications like: the particle finite element method and explicit compressible fluid are implemented in Kratos which helped for validating its flexibility in handling different algorithms. Finally its python interface gives extra flexibility in handling nonstandard algorithms.

Several reusable components are provided to help developers allowing easier and faster implementation of their applications. Data structure, IO, linear solvers, geometries, quadrature tools, and different strategies are examples of these reusable components. Use of these components makes application development not only faster, but also ensures compatibility with other tools for solving multi-disciplinary problems.

Kratos is also very extensible at different levels of implementation. Each application can add its variables, degrees of freedom, `Properties`, `Element`s, `Condition`s, and solution algorithms to Kratos. The object-oriented structure and appropriate patterns used in its design make these extensions easy while reducing the need for modifications. The extensibility is also validated at all levels by implementing different applications varying from standard finite element applications to optimization procedures using Kratos and its applications.

Last but not least, the performance of Kratos is comparable even to single purpose programs and different benchmarks show this in practice. This makes Kratos a practical tool for solving industrial multi-disciplinary problem.

### 11.1.1 General Structure

An object-oriented structure has been designed to maximize the reusability and extensibility of the code. This structure is based on finite element methodology and many objects are designed to represent the basic finite element concepts. In this way the structure becomes easily understandable for developers with a finite element method background.

In this design `Vector`, `Matrix`, and `Quadrature` representing the basic numerical concepts. `Node`, `Element`, `Condition`, and `Dof` are defined directly form finite element concepts. `Model`, `Mesh`, and `Properties` are from the practical methodology used in finite element modeling complemented by `ModelPart`, and `SpatialContainer`, for organizing better all data necessary for analysis. `IO`, `LinearSolver`, `Process`, and `Strategy` represent the different steps of a finite element program flow. Finally `Kernel` and `Application` are defined for library management and its interface definition.

Kratos uses a multi-layer approach in its design which reduces the dependency between different parts of program. It helps in maintenance of the code and also helps developers in understanding the code. These layers are defined in a way such as each user has to work in the smallest number of layers as possible. In this way the amount of code that each users has to be familiar with is minimized and the chance of conflict between users of different categories is reduced. The implementation difficulties needed for each layer is also tuned for the knowledge of users working in it. For example the finite element layer uses only basic to average features of C++ programming but the main developer layer use advanced language features in order to provide desired performance.

### 11.1.2 Basic Tools

Different reusable tools have been implemented to help developers in writing their applications in Kratos. Several geometries and different quadrature methods are provided and their performances are optimized. Their flexible design and general interface make them suitable for use in different applications. Their optimized performance make them appropriate not only for academic applications but also for real industrial simulations.

An extensible structure for linear solvers has been designed and different common solvers have been implemented. In this design the solver encapsulates only the solving algorithms and all operations over vectors and matrices are encapsulated in space classes. In this way solvers become independent of the type of mathematical containers and can be used to solve completely different types of equations systems like symmetric, skyline, etc. This structure also allows highly optimized solvers (for just one type of matrices or vectors) to be implemented without any problem.

### 11.1.3 Variable Base Interface

A new variable base interface has been designed and implemented. All information about a concept or variable to be passed through this interface is encapsulated in the `Variable` class. The information about components of a variable also is encapsulated in the `VariableComponent` class which gives an additional flexibility to this interface. This interface is used at different levels of abstraction and proved to be very clear, flexible, and extensible.

`Variable` provides the type of data statically and objects can use it to configure their operations for a given type of data via template implementation. This type information also prevents the use of variables in procedures that cannot handle their type of data. Each variable has a unique key which can be used as the reference key in data structures. The name of variable as an string helps routines like IO to read and write them without requiring additional parameters. Finally it provides a zero value which can be used for initializing data independent of its type in generic algorithms.

Beside this information, variable provides different methods for raw memory manipulations. These methods are excellent tools for low level generic programming, specially for writing heterogeneous containers.

This interface has been used successfully in different parts of Kratos. Its flexibility and extendibility is demonstrated in practice and its evident contribution to readability of the code is shown. This interface played a great role in uniforming different concepts coming from different area of analysis.

### 11.1.4 Data Structure

New heterogeneous containers have been implemented in order to hold different types of data without any modifications. The `DataValueContainer` can be used to store variables of any type without even explicitly defining the list of them. This container is very flexible but uses a search mechanism to retrieve given variable's data. The `VariablesListContainer` only stores the variables defined in its variables list which can be have any type but its advantage is its fast indirection mechanism for finding the variables data. In Kratos these two containers are used alternatively in places where performance or flexibility are more important. Being able to store even the list of neighbor `Node`s or `Element`s shows their flexibility in practice.

An entity base data structure has been developed in Kratos. This approach gives more freedom in partitioning the domain or in creating and removing `Node`s and `Element`s, for example in adaptive meshing. Several levels of abstraction are provided to help users group model and data information in different ways. In Kratos the `Model` contains the whole model, divided to different `ModelPart`s. Each model part can have different `Mesh`es which hold a complete set of entities in Kratos. These objects are effectively used for separating domain information or sending a single part to some process.

### 11.1.5 Finite Element Implementation

The `Element` and `Condition` classes are designed as the extension points of Kratos. Their generic interfaces provide all information necessary for calculating their local components and also are flexible enough for handling new arguments in the future.

Several processes and strategies has been developed to handle standard procedures in finite element programming. These components increase the reusability of the code and decrease the effort needed to implement new finite element application using Kratos.

Some experimental work has been done to handle elemental expression using a higher level of abstraction. In this way elemental expressions can be written in C++ but with a meta language very similar to mathematical notations and then can be compiled with the rest of the code using the C++ compiler. These expressions have been successfully tested and their performance is comparable to manually implemented codes.

Finally the `Formulation` is designed to handle nodal, edge based, or even elemental formulations in different forms of implementations. However these capabilities have not been implemented yet due to the lack of interest from developers.

### 11.1.6 Input-Output

A flexible and extensible IO module for finite element programs has been developed. It can handle new concepts very easily while Kratos automatically adds variables to its components and IO uses these components as its list of concepts. Therefore any application built with Kratos can use IO

for reading and writing its own concepts without making any change to it. However, more effort is required to extend this IO system to handle new types of data.

This IO is multi-format. It can support new formats just by adding a new IO derived class an without changing any other part of IO. For example a binary format IO can be added using this feature.

An interpreter is also implemented to handle Kratos data files. Its format is relatively intuitive and similar to Python scripts. The major interpreting task is given to the Python interpreter. This flexible interpreter with its object-oriented high level language can be used to implement and execute new algorithms using Kratos. In this way the implementation and maintenance cost of a new sophisticated interpreter is eliminated.

## 11.2 Future Work

This work can be continued in different ways. First of all different extensions to the existing framework can increase the number of useful features provided by it. Besides these extensions, parallelization of the code is the next task to perform in order to guarantee its success in solving future large scale problems.

### 11.2.1 Extensions

Here is a list of suggested extensions to this work.

#### Basic Tools

New solvers and preconditioners should be added to extend the solving abilities of Kratos. Also a new type of linear solvers for very small system of equations should be implemented. They can be used for solving efficiently the small equations appearing in some algorithms like the patch recovery method [104].

#### Variable Base Interface

As mentioned before, there are some complexities related to incompatibility of variables and their components in this interface. This also is an open door for further improvements. A solution could be deriving the `VariableComponent` from `variable` and using some indexing mechanism to distinguish them. This can be completed by some traits to avoid virtual function calling in cases where a good performance is also needed. This is something to be implemented and tested carefully.

#### Finite Element Implementation

Creating new processes and strategies can increase the reusability of the code and also the completeness of Kratos. This can be done also by revising the processes and strategies implemented in different applications and adding a generic version of them to Kratos which could be usable for a wider set of applications.

As mentioned in section 8.5, an experimental elemental expression has been developed and tested. These part needs more components to be useable in a wide variety of formulations. Implementing missing components and practically use them can help the fast development of finite element formulations in Kratos, At the same time, it can be used to optimize the new formulations or even transform them automatically to parallel codes.

Formulations are another part of Kratos to explore. Adding nodal or edge based formulations to Kratos can be a good way to refine its design in practice.

**Input-Output**

Serialization is not implemented yet but is considered to be useful for automatization of problem loading and saving. Adding this feature to Kratos would help users to run longer problems and pause them whenever they want. Using an external library is considered to be a better solution than implementing it.

Supporting binary format for input can reduce significantly the data reading time. The multi-format feature of Kratos reduces a lot the effort necessary to implement it.

### 11.2.2 Parallelization

Beside the extensions mentioned before, parallelization of Kratos framework is the main task to be undertaken in the future. The growing size of problems and the increase of available parallel computing machines (even in the personal computers sector), stress the importance for parallelization of numerical codes. For this reason a big effort should be invested to parallelize Kratos for shared memory and distributed memory architectures.

Fortunately, several aspects of Kratos become useful in this process. Its entity based data structure makes the distribution of data over processors easier. Also, several layers of abstraction in the data structure will help the partitioning task which are needed for division of the model over processors. Finally the `Strategy` is designed to be parallelized with a very small effort.

## 11.3 Acknowledgments

This work is dedicated to Hengameh who left her family, friends and also her work to come to Barcelona with me. Her support and patience during these long years have been priceless.

I would like to thank Professor Eugenio Oñate for his support which made this project possible and to the whole Kratos team for their contributions and valuable feedback that was decisive for the evolution of Kratos.

I would also like to thank specially my good friend Dr. Riccardo Rossi for his great effort in developing Kratos as a very good engineer and programmer. He helped me to overcome several difficulties arising during this work not only with his practical solutions but also with his high motivation and belief in the project.

This work has been done in the memory of Francisco Javier Royo who started it but his short life did not let him finish it...

# References


[1] Ansys. http://www.ansys.com/default.asp.

[2] Bison. http://www.gnu.org/software/bison/.

[3] Boost library. http://www.boost.org/.

[4] Boost serialization library. http://www.boost.org/libs/serialization/doc/index.html.

[5] Boost spirit library. http://www.boost.org/libs/spirit/index.html.

[6] Flex. http://flex.sourceforge.net/.

[7] A quality tetrahedral mesh generator and three-dimensional delaunay triangulator (tetgen) programming interface. http://tetgen.berlios.de/library.html.

[8] Stlport. http://www.stlport.org/.

[9] Xparam library. http://xparam.sourceforge.net/.

[10] ABAQUS Inc. *ABAQUS Scripting Reference Manual*.

[11] ABAQUS Inc. *ABAQUS Scripting User's Manual*.

[12] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Priciples, Techniques, and Tools*. Addison-Wesley, 1986.

[13] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1978.

[14] A. V. Aho, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 1995.

[16] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1999.

[17] G. C. Archer. *Object-Oriented Finite Element Analysis*. PhD thesis, University of California at Berkeley, 1996.

[18] G. C. Archer, G. Fenves, and C. Thewalt. A new object-oriented finite element analysis program architecture. *Computers & Structures*, 70(1):63–75, 1999.

[19] W. Bangerth. Using modern features of C++ for adaptive finite element methods: Dimension-independent programming in deal.II. In M. Deville and R. Owens, editors, *Proceedings of the 16th IMACS World Congress 2000, Lausanne, Switzerland, 2000*, 2000. Document Sessions/118-1.

[20] W. Bangerth, R. Hartmann, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org`.

[21] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. Technical Report ISC-06-02-MATH, Institute for Scientific Computation, Texas A&M University, 2006.

[22] W. Bangerth and G. Kanschat. Concepts for object-oriented finite element software – the `deal.II` library. Preprint 99-43 (SFB 359), IWR Heidelberg, Oct. 1999.

[23] K.-J. Bathe. *Finite element procedures*. Prentice-Hall, 1996.

[24] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[25] A. Cardona, I. Klapka, and M. Geradin. Design of a new finite element programming environment. *Engineering Computations*, 11(4):365–381, 1994.

[26] R. Codina. Pressure stability in fractional step finite element methods for incompressible flows. *Journal of Computational Physics*, 170:112140, 2001.

[27] R. Codina. Stabilized finite element approximation of transient incompressible flows using orthogonal subscales. *Computer Methods in Applied Mechanics and Engineering*, 191(39):4295–4321, 2002.

[28] H.-P. Company. Sgi standard template library (stl). http://www.sgi.com/tech/stl/.

[29] Coupled Problems 2007. *Updated lagrangian formulation of a quasi-incompressible fluid element*, Ibiza, Spain, 2007.

[30] P. Dadvand, R. Lopez, and E. Oñate. Artificial neural networks for the solution of inverse problems. In *ERCOFTAC*, 2006.

[31] P. Dadvand, J. Mora, C. González, A. Arraez, P. Ubach, and E. Oñate. Kratos: An object-oriented environment for development of multi-physics analysis software. In *Proceedings of the WCCM V Fifth World Congress on Computational Mechanics*. WCCM V Fifth World Congress on Computational Mechanics, July 2002.

[32] J. Donéa and A. Huerta. *Finite Element Methods for Flow Problems*. John Wiley and Sons, 2003.

[33] Y. Dubois-Pélerin and P. Pegon. Improving modularity in object-oriented finite element programming. *Communications in Numerical Methods in Engineering.*, 13:193–198, 1997.

[34] Y. Dubois-Pèlerin and T. Zimmermann. Object-oriented finite element programming: Iii. an efficient implementation in c++. *Comput. Methods Appl. Mech. Eng.*, 108(1-2):165–183, 1993.

[35] Y. Dubois-Pèlerin, T. Zimmermann, and P. Bomme. Object-oriented finite element in programming: Ii. a prototype program in smalltalk. *Comput. Methods Appl. Mech. Eng.*, 98(3):361–397, 1992.

[36] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Clarendon Press, New York, NY, USA, 1989.

[37] D. R. Edelson. Smart pointers: They're smart, but they're not pointers. Technical report, Santa Cruz, CA, USA, 1992.

[38] D. Eyheramendy and T. Zimmermann. Object-oriented finite element programming: an interactive environment for symbolic derivations, application to an initial boundary value problem. *Adv. Eng. Softw.*, 27(1-2):3–10, 1996.

[39] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements ii. a symbolic environment for automatic programming. *Comput. Methods Appl. Mech. Eng.*, 132(3):277–304(28), June 1996.

[40] D. Eyheramendy and T. Zimmermann. Object-oriented finite elements iii. theory and application of automatic programming. *Comput. Methods Appl. Mech. Eng.*, 154(1):41–68(28), February 1998.

[41] C. A. Felippa and T. L. Geers. Partitioned analysis of coupled mechanical systems. *Engrg. Comput.*, 5:123–133, 1988.

[42] C. A. Felippa, K. C. Park, and C. Farhat. Partitioned analysis of coupled mechanical systems. *Computer Methods in Applied Mechanics and Engineering*, 190:3247–3270, 2001.

[43] J. R. A. Filho and P. R. B. Devloo. Object oriented programming in scientific computations : the beginning of a new era. *Engineering computations*, 8(1):81–87, 1991.

[44] B. W. R. Forde, R. O. Foschi, and S. F. Stiemer. Object-oriented finite element analysis. *Comp. Struct.*, 34(3):355–374, 1990.

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

[46] G. H. Gonnet and R. Baeza-Yates. *Handbook of algorithms and data structures: in Pascal and C (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

[47] S. Haykin. *Neural Networks: A Comprehensive Fundation*. Prentice Hall, 1994.

[48] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, 1989.

[49] S. R. Idelson and E. Oñate. To mesh or not to mesh. that is the question. *Computer methods in applied mechanics and engineering*, 195:4681–4696, 2006.

[50] S. R. Idelson, E. Oñate, N. Calvo, and F. D. Pin. The meshless finite element method. *International Journal for Numerical Methods in Engineering*, 58:893–912, 2003.

[51] S. R. Idelson, E. Oñate, and F. D. Pin. The particle finite element method: a powerful tool to solve incompressible flows with free-surfaces and breaking waves. *International Journal for Numerical Methods in Engineering*, 61:964 – 989, 2004.

[52] A. Kirsch. *An Introduction to the Mathematical Theory of Inverse Problems*. Springer, 1996.

[53] I. Klapka, A. Cardona, and M. Geradin. An object oriented implementation of the finite element method for coupled problems. *Revue Europenne des lments Finis*, 7(5):469–504, 1998.

[54] I. Klapka, A. Cardona, and M. Geradin. Interpreter oofelie for pdes. In *European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2000)*, 2000.

[55] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1998.

[56] Laboratory for Computational Physics and Fluid Dynamics (LCP&FD). *FEFLO Project*.

[57] A. Larese, R. Rossi, E. Oñate, and S. R. Idelson. Validation of the particle finite element method (pfem) for simulation of free surface flows. *Submitted for publication to Engineering and Computation*, 2007.

[58] J. Lindemann, O. Dahlblom, and G. Sandberg. Using corba middleware in finite element software. *Future Generation Comp. Syst.*, 22(1-2):158–193, 2006.

[59] R. Löhner. *Applied CFD Techniques: An Introduction Based on Finite Element Methods*. Wiley, 2001.

[60] R. Lopez. *Flood: An Open Source Neural Networks C++ Library*. CIMNE.

[61] J. Lu, D. White, and W.-F. Chen. Applying object-oriented design to finite element programming. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 424–429, New York, NY, USA, 1993. ACM Press.

[62] J. Lu, D. W. White, W.-F. Chen, and H. E. Dunsmore. A matrix class library in c++ for structural engineering computing. *Coputers & Structures*, 55(1):95–111, 1995.

[63] F. Lundh. *Python Standard Library*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[64] R. I. Mackie. Object oriented programming of the finite element method. *International journal for numerical methods in engineering*, 35(2):425–436, 1992.

[65] R. I. Mackie. Using objects to handle complexity in finite element software. *Engineering with Computers*, 13(2):99–111, 1997.

[66] Maplesoft. *Maple's Documentation*.

[67] T. Mason and D. Brown. *Lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1990.

[68] MathWorks. *Matlab's Documentation*.

[69] P. Menetrey and T. Zimmermann. Object-oriented non-linear finite element analysis: application to j2 plasticity. *Computers & Structures*, 49(5):767–773, 1993.

[70] G. R. Miller. An object-oriented approach to structural analysis and design. *Coputers & Structures*, 40(1):75–82, 1991.

[71] G. R. Miller. Coordinate-free isoparametric elements. *Coputers & Structures*, 49(6):1027–1035, 1994.

[72] G. R. Miller, S. Banerjee, and K. Sribalaskandarajah. A framework for interactive computational analysis in geomechanics. *Computers and Geotechnics*, 17(1):17–37, 1995.

[73] J. Mora, R. Otín, P. Dadvand, E. Escolano, M. A. Pasenau, and E. Oñate. Open tools for electromagnetic simulation programs. *COMPEL*, 25(3):551–564, 2006.

[74] D. Mount and S. Arya. Ann: A library for approximate nearest neighbor searching, 1997.

[75] E. Oñate. *Cálculo de Estructuras por el Método de Elementos Finitos*. CIMNE, 2nd edition, 1995.

[76] E. Oñate, S. Idelson, F. D. Pin, and R. Aubry. The particle finite element method. an overview. *International Journal of Computational Methods*, 1(2):267–307, 2004.

[77] Open Engineering. *OOFELIE*.

[78] B. Patzák. *OOFEM Documentation*. Czech Technical University, Faculty of Civil Engineering, Department of Structural Mechanics.

[79] B. Patzák and Z. Bittnar. Object oriented finite element modeling. *Acta Polytechnica*, 39(2):99–113, 1999.

[80] R. M. V. Pidaparti and A. V. Hudli. Dynamic analysis of structures using object-oriented techniques. *Computers & Structures*, 49(1):149–156, 1993.

[81] W. H. Press, W. T. Vetterling, S. A. Teukolsky, and B. P. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2002.

[82] B. Raphael and C. S. Krishnamoorthy. Automating finite element development using object oriented techniques. *Engineering computations*, 10(3):267–278, 1993.

[83] R. Ribó, M. Pasenau, E. Escolano, and J. S. P. Ronda. *GiD User Manual*. International Center for Numerical Methods in Engineering (CIMNE), Edificio C1, Campus Norte UPC, Gran Capitán s/n, 08034 Barcelona, Spain.

[84] R. Ribó, M. Pasenau, E. Escolano, J. S. P. Ronda, and L. F. González. *GiD Reference Manual*. International Center for Numerical Methods in Engineering (CIMNE), Edificio C1, Campus Norte UPC, Gran Capitán s/n, 08034 Barcelona, Spain.

[85] R. Rossi. *Light weight Structures: Structural Analysis and Coupling Issues*. PhD thesis, University of Bologna, 2005.

[86] R. Rossi, S. R. Idelson, and E. Oñate. On the possibilities and validation of the particle finite element method (pfem) for complex engineering fluid flow problems. In *Proceedings of ECCOMAS CFD 2006*, The Netherlands, 2006.

[87] R. Rossi and R. Vitaliani. Numerical coupled analysis of flexible structures subjected to the fluid action. In *5th PhD symposium*, Delft, 2004.

[88] R. Rossi, R. Vitaliani, and E. Oñate. Validation of a fsi simulation procedure - bridge aerodynamics model problem. In *Coupled Problems 2005*, Santorini, Italy, 2005.

[89] G. V. Rossum. *The Python Language Reference Manual*. Network Theory Ltd., 2003.

[90] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[91] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[92] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[93] H. Si. *A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator (TetGen) User's Manual*.

[94] J. Šíma and P. Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, 15(12):2727–2778, December 2003.

[95] R. Touzani. *OFELI Documentation*.

[96] R. Touzani. An object oriented finite element toolkit. In *Proceedings of the Fifth World Congress on Computational Mechanics (WCCM V)*, 2002.

[97] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[98] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[99] T. L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18, 1999.

[100] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Berlin, Heidelberg, New York, Tokyo, 1997. Springer-Verlag.

[101] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[102] M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 3rd edition, 2006.

[103] Wolfram Research. *Mathematica's Documentation*.

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

[105] T. Zimmermann, P. Bomme, D. Eyheramendy, L. Vernier, and S. Commend. Aspects of an object-oriented finite element environment. *Computers and Structures*, 68(1):1–16, 1998.

[106] T. Zimmermann, Y. Dubois-Pèlerin, and P. Bomme. Object-oriented finite element programming: I: Governing principles. *Comput. Methods Appl. Mech. Eng.*, 98(2):291–303, 1992.

[107] T. Zimmermann and D. Eyheramendy. Object-oriented finite elements i. principles of symbolic derivations and automatic programming. *Comput. Methods Appl. Mech. Eng.*, 132(3):259–276(18), June 1996.