Arch Comput Methods Eng (2010) 17: 253–297
DOI 10.1007/s11831-010-9045-2

ORIGINAL PAPER

An Object-oriented Environment for Developing Finite Element Codes for Multi-disciplinary Applications

Pooyan Dadvand · Riccardo Rossi · Eugenio Oñate

Received: 15 December 2009 / Accepted: 15 December 2009 / Published online: 22 July 2010

Abstract The objective of this work is to describe the design and implementation of a framework for building multi-disciplinary finite element programs. The main goals are generality, reusability, extendibility, good performance and memory efficiency. Another objective is preparing the code structure for team development to ensure the easy collaboration of experts in different fields in the development of multi-disciplinary applications.

Kratos, the framework described in this work, contains several tools for the easy implementation of finite element applications and also provides a common platform for the natural interaction of different applications. To achieve this, an innovative variable base interface is designed and implemented. This interface is used at different levels of abstraction and showed to be very clear and extendible. A very efficient and flexible data structure and an extensible IO are created to overcome difficulties in dealing with multi-disciplinary problems. Several other concepts in existing works are also collected and adapted to coupled problems. The use of an interpreter, of different data layouts and variable number of dofs per node are examples of such approach.

In order to minimize the possible conflicts arising in the development, a kernel and application approach is used. The code is structured in layers to reflect the working space of developers with different fields of expertise. Details are given on the approach chosen to increase performance and efficiency. Examples of application of Kratos to different multidisciplinary problems are presented in order to demonstrate the applicability and efficiency of the new object oriented environment.

P. Dadvand (✉) · R. Rossi · E. Oñate
International Center for Numerical Methods in Engineering (CIMNE), Technical University of Catalonia, Campus Norte UPC, 08034 Barcelona, Spain
e-mail: pooyan@cimne.upc.edu
url: http://www.cimne.com

1 Introduction

One of the open topics in the finite element method (FEM), is the combination of solvers in different fields (thermal, fluid dynamics, structural, etc.) in order to solve multi-disciplinary problems. The ideal approach to achieve this goal is to develop a unified software environment. This requires however a high investment in terms of software development.

Kratos, the object oriented framework described in current paper, has been designed from scratch to address such a challenge.

1.1 Problem

Conventional finite element codes encounter several difficulties in dealing with multi-disciplinary problems involving the coupled solution of two or more different fields. Many of these codes are designed and implemented for solving a certain type of problems, generally involving a single field. For example, to solve just structural, fluid mechanic, 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 monolithic coupling approaches.

Some newer implementations have used modern software engineering concepts in their design to make the program more extendible. They usually achieve extendibility to new algorithms in their programs. However, extensions to new fields is beyond their usual intent.

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

- Predefined set of dofs 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 (dofs) per node. For example, in a three dimensional (3D) structural program each node has six dofs $d_x, d_y, d_z, w_x, w_y, w_z$ where d is the nodal displacement and w the nodal rotation. Assuming all nodes to have just this set of dofs helps developers to optimize their codes and also simplifies their implementation tasks. This assumption, however, prevents the extension of the code to another field with a different set of dofs per node.

Usually the data structure of FEM programs is designed to hold certain variables and historical values. The main reasons for this are: easier implementation, better performance of data structure and less effort in maintenance. In spite of these advantages, using rigid data structures usually requires important changes revision to include new variables from other 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 the nodes in the model. In the Kratos 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 requires 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 usual to create domain specific interfaces in order to increase the code clarity. Let us consider as an example a thermal problem for this purpose. In a classical approach one would define a variable, possibly named `Conductivity` to allow the element to access its conductivity. This leads to following interface:

```
c = Conductivity()
```

Though this enhances code clarity, it is incompatible with extendibility to new fields. For example if we wish to add the "permeability", another method has to be defined for all element class hierarchy. This is a point of conflict when adding new formulations.

Another bottleneck in extending the program to new fields is IO. 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.

1.2 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 used to organize different parts of the code in a set of objects with clear interfaces. In this way the flexibility of the code increases, and reusing an object in some other places also becomes more practical.

Design and implementation of a multi-disciplinary conceptual interface is another solution for 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 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 dofs 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.

2 Background

The history of object-oriented design for finite element programs turns back to the early 1990's or even earlier. In 1990, Forde *et al.* [33] published one of the first detailed descriptions of applying object-oriented design to finite element programs. They used the essential components of FEM 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 focused on structural analysis only and the objects' interfaces reflect this fact.

In those years other authors started to write about the object-oriented paradigm and its applications in finite element programming. Filho and Devloo [32] introduced object oriented design in FEM codes and applied it to element design. Mackie [53] 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 a finite element program providing a data structure for entities and also introduced the use of iterators [54]. Pidaparti and Hudli [69] published a more detailed object oriented structure with objects for different algorithms in dynamic analysis of structures. Raphael and Krishnamoorthy [71] also used 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 FEM. 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 [58–60] 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.* [24, 25, 90] designed a structure for linear dynamic finite element analysis. Their structure consists of three categories of objects. First the FEM objects which are: `Node`, `Element`, `Load`, `LoadTimeFunction`, `Material`, `Dof`, `Domain`, and `LinearSolver`. The second includes some tools like `GaussPoint`, and `Polynomial`. The third category refers to 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++. The latter version provides a comparable performance versus 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 for each element. It also assembles its components in the global system of equations and updates itself after its solution. `Node` holds its position and manages dofs. It also computes and assembles the load vector and finally updates the time dependent data after the solution. `Dof` holds the unknown information and also the dofs values. 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 for components like nodes and elements and also manages the solution 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` [57].

Lu *et al.* [49, 50] 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 [8]. This library provides a high level interface using object-oriented language features.

Archer *et al.* [10, 11] presented another object-oriented structure for a finite element program dedicated to linear and

nonlinear analysis structures under static and dynamic loads. They reviewed features provided by 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, `Analysis` encapsulates the algorithms and `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. `ReorderHandler` optimizes the order of the unknowns. `MatrixHandler` provides different matrices and construct them over a 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, its position and its dofs. `ScalarDof` and `VectorDof` are derived from the `Dof` class and represent the different dof's types. `Element` uses the `ConstitutiveModel` and `ElementLoad` to calculate the stiffness matrix K^e, the mass matrix M^e and the damping matrix C^e in the local coordinate system. `LoadCase` includes loads, prescribed displacements and initial element state objects and calculates the load vector in the local coordinate system.

Cardona *et al.* [17, 42, 43] developed the Object Oriented Finite Elements method Led by Interactive Executor (OOFELIE) [66]. 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 `Fixations` and `Loads` handle the boundary conditions and the 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 for 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 dofs and also sorts them out.

Touzani [83] developed the Object Finite Element Library (OFELI) [82]. This library has an intuitive structure based on the FEM and can be used for developing finite element programs for different fields like heat transfer, fluid flow, solid mechanics and electromagnetics.

`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 to form a `FEEqua` class which implements formulations for different fields of analysis. It uses a static `Material` class in which each parameter is stored as a member variable. `Element` just provides the geometrical information and the finite element implementation is encapsulated via `FEEqua` classes. `Element` provides several features which makes it useful for even complex formulations but keeping all these members data makes it too heavy for industrial implementations.

Bangerth *et al.* [12, 14, 15] created a library for implementing adaptive meshing finite element solution of partial differential equations called Differential Equations Analysis Library (DEAL) II [13]. 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 the mesh. `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 dofs 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 the calculations and provides the local components to be assembled into the global solver.

Extensive use of templates and other advanced features of C++ 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.* [68] published an structure used in the Object Oriented Finite Element Modeling (OOFEM) [67] program. In this structure `Domain` contains the complete problem description which can be divided into several `Engineering Models` 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 finite element codes trying to increase their flexibility and reducing the maintenance cost. Two ways 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.* [24, 25, 90] is one of the classical approaches in designing the code structure on the basis of the 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.* to encapsulate the coordinate transformation in geometry is useful for relaxing the dependency of elemental formulation to an specific geometry.

Cardona *et al.* [17, 42, 43] 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 increases the maintenance cost of the interpreter itself and makes it difficult to using other libraries which may have interfaces to chosen script languages.

Touzani [83] designed an structure for multi-disciplinary finite element programs. His design is clear and easy to understand but uses field specific interfaces for its components 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.* [49, 50] is in line with designing a partial differential solver with emphasizes on numerical components. Archer *et al.* [10, 11] extended this approach to make it more flexible and extendible. More recently Bangerth *et al.* used the same approach for 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 represent 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 analysis perspective. There are also some new components like model part, process, kernel and application which are added to account for 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 has also been invested to design and implement a fast and very flexible data structure enabling 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.

In the following sections we start with a brief description of FEM in Sect. 3 and continues in Sect. 4 with a short presentation of common techniques for the solution of multi-disciplinary problems. Section 5 gives an overview of the Kratos environment and its general structure. The new Variable Base Interface is described in Sect. 6. Section 7 is dedicated to the data structures. The next two sections describe the parts related to finite element method and the input output. Finally some validation examples of the possibility of Kratos for solving multidisciplinary are presented in Sect. 11.

3 Finite Element Method

3.1 Basic Concepts of FEM Discretization

The finite element method typically solves problems governed by differential equations with the appropriate boundary conditions. The more general setting can be expressed as:

– Solve the set of different equations

$$\mathbf{A}(\mathbf{u}) = 0 \quad \text{in } \Omega \tag{1}$$

subjected to the boundary conditions

$$\mathbf{B}(\mathbf{u}) = 0 \quad \text{in } \Gamma \tag{2}$$

where $\mathbf{A}$ and $\mathbf{B}$ are matrices of differential operators defined over the analysis domain Ω and its boundary Γ and $\mathbf{u}(\mathbf{x}, t)$ is the vector of problem unknowns which may depend on the

space dimension and the time. Indeed, for multidisciplinary problems **A** and **B** define the equations in the different interacting domains and their boundaries. Details of the form of the operators **A** and **B** for each particular problem can be found in standard FEM books [89].

The FEM solution is found by approximating the unknown polynomial expansions defined within each element discretizing the analysis domain Ω, i.e.

$$\mathbf{u} \simeq \mathbf{u}_h = \sum_{i=1}^{N} \mathbf{N}_i \mathbf{u}_i \tag{3}$$

where N is the number of nodes in the mesh, $\mathbf{N}_i$ is the shape functions matrix and $\mathbf{u}_i$ are the approximate values of $\mathbf{u}$ at each node. Typically, (3) is written on an element by element form.

The matrix system of equation is obtained by substituting (3) into (1) and (2) and using the Galerkin weighted residual form which can be written as

$$\int_{\Omega} \mathbf{N}_i \mathbf{r}_{\Omega} d\Omega + \int_{\Gamma} \mathbf{N}_i \mathbf{r}_{\Gamma} d\Gamma = 0, \quad i = 1, N \tag{4}$$

where

$$\mathbf{r}_{\Omega} = \mathbf{A}(\mathbf{u}_h), \qquad \mathbf{r}_{\Gamma} = \mathbf{B}(\mathbf{u}_h) \tag{5}$$

are the so called "residuals" of the governing equations over the domain and the boundary, respectively. Indeed (4) can be adequately transformed by integration by parts of many terms over the analysis domain, to give the so called weak integrals form which is more convenient for some problems [89]. In any case, the element structure of the FEM allows computing and assembling the integrals of (4) on an element by element manner. The resulting system of discretized equation can be written (for the static case) as

$$\mathbf{K}\mathbf{a} = \mathbf{f} \tag{6}$$

where

$$\mathbf{K} = \mathbb{A}\mathbf{K}^{(e)}, \qquad \mathbf{f} = \mathbb{A}\mathbf{f}^{(e)} \tag{7}$$

where $\mathbb{A}$ denotes the assembly operation which allows to obtaining the global stiffness matrix $\mathbf{K}$ and the equivalent nodal force vector $\mathbf{f}$ from the element contributions. In (7) $\mathbf{a}$ is the vector of unknowns which lists the values of $\mathbf{u}_i$ at all the nodes in the mesh.

The solution of the system of (6) requires prescribing the value of the unknown variables $\mathbf{u}_i$ at specific points (the so called Dirichlet boundary condition).

It is not the objective of this paper to explain the basis of the FEM which can be found in many text books. Instead we will focus in describing the details of the implementation of the FEM equations into an object-oriented environment with the aim of solving problems of multidisciplinary nature, where the differential operators **A** and **B** are defined over adjacent or overlapping domain, therefore leading to a coupled solution strategy. Typical examples of these situations are thermal-mechanical problems, fluid-structure interaction problems, electro-magnetic problems, aeroacoustic, etc.

3.2 Solution of FEM Equations

3.2.1 Calculating Components

This step consist of calculating the elemental integrals usually using the Gaussian Quadrature method. The integrand function contains the shape functions and their derivatives, which should be computed for each element. A common technique is to calculate these functions in local coordinates of the element and transform the result to global coordinates.

3.2.2 Creating the Global System

Once the elemental matrices and vectors $\mathbf{K}^e$ and $\mathbf{f}^e$ have been computed, they are assembled in the global system of equations. This requires finding the position of each elemental component in the global equations system and adding its value to its position.

This procedure first assigns a sequential numbering to all dofs. Sometimes is useful to separate the *prescribed* dofs, those with Dirichlet conditions, from the 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 assembly operator $\bigsqcup$, i.e.:

$$\mathbf{K}_{ij} \bigsqcup_{\mathbf{I}^e} \mathbf{K}^e_{ij} = \mathbf{K}_{\mathbf{I}^e_i \mathbf{I}^e_j} + \mathbf{K}^e_{ij} \tag{8}$$

$$\mathbf{f}_i \bigsqcup_{\mathbf{I}^e} \mathbf{f}^e_i = \mathbf{f}_{\mathbf{I}^e_i} + \mathbf{f}^e_i \tag{9}$$

where $\mathbf{I}^e$ is the vector containing the global position, which is the index of the corresponding dof, of each row or column.

Another task to be done when building the global system of equations is the application of essential boundary conditions by eliminating prescribed dofs or using a *penalty method* [74].

The global system matrix obtained in the FEM typically has many zeros in it. So it is convenient to use structures which hold a useful portion of the matrix to be solved like *compressed sparse row* (*CSR*) [80]. Another common structure is the *symmetric matrix* structure that uses the symmetry property of the matrix to hold approximately half of the elements.

3.2.3 Solving the Global System

There are two categories of solvers: *direct solvers* and *iterative solvers*.

Direct solvers solve the equation system by transforming the coefficient matrix in an upper triangular form, lower triangular, a diagonal form. Solvers like *Gaussian elimination* [70], *frontal solution* [16], *LU decomposition* [70] are examples of this category.

Iterative solvers start with some initial values for the unknown and find the correct solution by calculating the residual and minimize it over a number of iterations. *Conjugate gradient* (*CG*) [80], *Biconjugate gradient* (*BCG*) [80], *Generalized minimal residual method* (*GMRES*) [80] are examples of this category of solvers.

It is common to use a *reordering* procedure for reducing the bandwidth of the system matrix or make it more cache efficient. In practice these algorithms can be applied to renumber the dofs in an optimum way once they are created and then solve the system as usual. The *Cuthill McKee* [80] algorithm is a classical example of this type of algorithms.

Sometimes it is recommended to transform the system of equations to and equivalent but better conditioned one for using iterative solvers. This procedure is called *preconditioning*. *Diagonal* and *Incomplete LU* preconditioners [80] are examples of preconditioners. Unfortunately, finding the best combination of solver and preconditioner for a certain problem is a matter of experience and there is not a single best combination for all problems.

3.2.4 Calculating Additional Results

This step consist of calculating some additional results from the primary ones obtained by solving the global system. For example, calculating the stresses from the displacements in a structural problem.

These additional results can be discontinuous over the domain. This means that, for example, the stresses at a node are different for each of the element corrected to it. Different averaging methods are typically implemented to smooth the discontinuous results. An alternative is to use recovery methods which try to reproduce continuous gradient results with a better approximation [89].

3.2.5 Iterating

Several algorithms in FEM contain iterations in different ways. Some examples are *Newton method*, *Modified Newton method*, *Line search* [89], etc. for solving nonlinear problems.

4 Multi-disciplinary Problems

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 a 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.

4.1 Categories

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 Fig. 1.

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 subsystem depends on the other but this can be solved independently. Figure 2 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. Figure 3 shows this type of coupling.

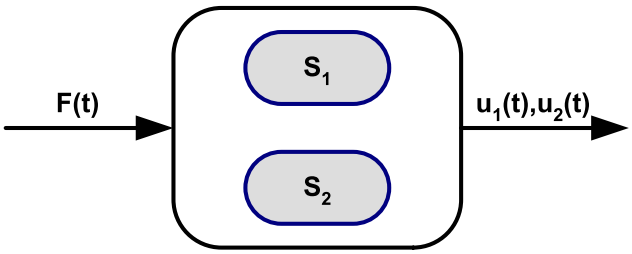


Fig. 1 A general multi-disciplinary problem with two subsystems

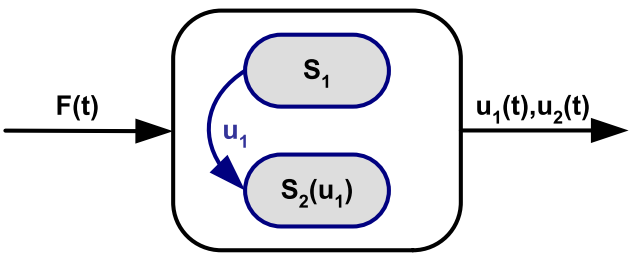


Fig. 2 A weak coupled system where subsystem S_2 depends on the solution of subsystem S_1

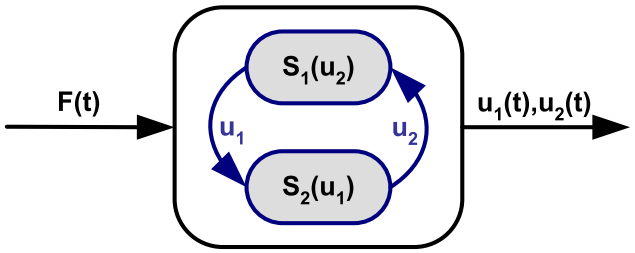


Fig. 3 A strong coupled system where not only subsystem S_2 depends on the solution of subsystem S_1 but also subsystem S_1 depends on S_2

Another classification can be done by looking not on how the subsystem interact but where they interact with each other. There are two categories of multi-disciplinary problems within this criteria [89]:

Class I In this category the interaction occurs at the boundary of the domains.

Class II This category include problems where domains can overlap totally or partially.

4.2 Solution Methods

The solution of one-way coupled problems is straightforward. Considering the problem of Fig. 2 with two subsystems S_1 and S_2, where S_2 depends on u_1 (the solution of S_1). This problem can be solved by solving S_1 first and using its solution u_1 for solving S_2.

Monolithic and *staggered* methods can be used for solving two-way coupled problems. The monolithic approach consists of modeling the interacting fields together in one global system of equations and solve them together. Consider the problem with strong coupling in Fig. 3 where not only S_2 depends on S_1, but also S_1 depends on S_2. Using the monolithic approach results in the following system of equations:

$$\begin{bmatrix} \mathbf{K}_1 & \mathbf{H}_1 \\ \mathbf{H}_2 & \mathbf{K}_2 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1(t) \\ \mathbf{f}_2(t) \end{bmatrix} \tag{10}$$

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 the interaction variables. The monolithic approach is robust and stable but introduces a difficulty of the formulation, larger size matrices and an increase in the bandwidth of the global system.

Staggered approach aim to solving each field separately and deal with the interaction by applying different techniques for transferring variables from one field to another. Some common techniques used for staggered methods are described below:

Prediction Predicting the value of the dependent variables in the next step.

Advancing Calculating the next time step of a subsystem using the calculated or predicted solution of the other subsystem.

Substitution Using the calculated value of one field in another field for solving it separately.

Correction Substituting the obtained result in place of the predicted value and solve it again for obtaining a better result. This procedure can be repeated several times.

More information about staggered methods and their techniques can be found in [30, 31]. Staggered approaches often use less resources than the monolithic approaches because in each step solve only one part of problem. Another advantage is the possibility of reusing existing single field codes for solving multi-disciplinary problems using master and slave technique. This approach also enables the use of different discretizations for each field. Beside all these advantages this approach requires a careful formulation to avoid instability and obtain an accurate solution. In general, staggered methods are less robust than monolithic one and need more attention and time for modeling and solving.

5 Programming Concepts

This section describes different software engineering solutions and programming techniques useful for designing a finite element program.

There are several classical problems that appear during the design and can be solved easily by applying existing *Design Patterns* [34]. Several patterns are used in this work in order to increase the quality of the design. Examples are *Strategy Pattern*, *Bridge pattern*, *Composite Pattern*, *Template Method Pattern*, *Prototype Pattern*, *Interpreter Pattern*, *Visitor Pattern* and *Curiously Recursive Template Pattern*. Description of other patterns and also more detailed explanation of the patterns mentioned above can be found in [34].

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 as Fortran implementations [86] and that usually the inefficiency of C++ codes comes from the developer's misunderstanding of the language [43]. For this reason we briefly review some techniques used for implementing high performance and efficient numerical algorithms in C++. These techniques are used extensively in the Kratos to improve its efficiency while providing a clear and easy-to-use interface.

Expression Templates is a technique used to transfer expressions to a function argument in a very efficient way [84]. For example passing a function to an integration module for its integration. The idea is to create a template object for each operator and constructing the whole expression by

combining these templates and their relative variables. This technique is also used in high performance linear algebra libraries to evaluate vectorial expressions [85]. In this way expressions consisting of operations over matrices and vectors can be evaluated without creating any temporary object and in a single loop.

Templates were added to C++ for a better alternative to existing macros in old C. Eventually they did not eliminate the use of macros but performed much better than anyone could expect. An example is 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 implementation to run.

The Template Metaprogramming technique can be used to create an specialized algorithm at the time of compiling. This technique makes the compiler interpret a subset of the C++ code in order to generate this specialized algorithm. Different methods are used to simulate different programming statements, like loops or conditional statements, at compiling time. These statements are used to tell the compiler how to generate the code for an 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.

Template specialization 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 the row or column majority of a given matrix.

6 General Structure of Kratos

Kratos is designed as a framework for building multi-disciplinary finite element programs [21]. Generality in design and implementation is the first requirement. Flexibility and extensibility are other key points in this design, enabling developers to implement very different formulations and algorithms involving in the solution of multi-disciplinary problems.

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.

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.

Finally it has to provide different levels for developers to contribute to the Kratos system, and match their requirements and difficulties in the way they extend it. Developers may not want to make a plug-in extension, create an application over it, or using IO scripts to make Kratos perform a certain algorithm. Some potential profiles for Kratos users are:

Finite Element Developers These developers are considered to be more experts in FEM, from the physical and mathematical points of view, than C++ programmers. For them, Kratos has to provide solution to their requirements without involving them in advanced programming concepts.

Application Developers 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 typical users of this kind.

Package Users Engineers and designers are other users of Kratos. They use the complete Kratos package 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, can work on it in order to create multi-disciplinary finite element applications.

6.1 Object Oriented Design

History of object-oriented design for finite element programs turns back to early 90's, and even more. Before that, time 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 [32, 33, 53, 69, 71].

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 solvers methods [15] or in the FEM itself [24, 25, 33, 90].

In Kratos we have chosen the second approach and have constructed our objects based on a finite element general

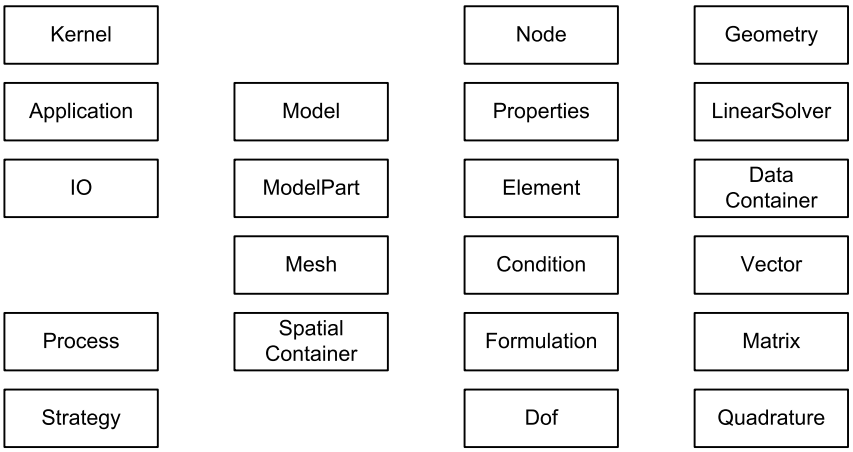


Fig. 4 Main classes defined in Kratos. The basic components in the *rightmost column*. FEM related objects in the *second column from the right*. The FEM modeling objects in the *third column*. The algorithms in *fourth column* below and library interface objects are above them

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. In addition, this approach has given us the necessary generality in the objectives of Kratos mentioned above. 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 relations are defined and their responsibilities are balanced. Figure 4 shows the main classes 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. These classes are 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. 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. 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 `Nodes` and provides basic parameters, like area or center point, to shape functions and coordinate transformation routines.

`Node` It 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 object 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 `Nodes`. Also stores the elemental data and the interface to access it.

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

`Dof` represents a degree of freedom (dof). It is a lightweight object which holds its variable, like `TEMPERATURE`, its state of freedom, and a reference to its value in the 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 `Elements` or `Conditions`. It can store any type of data and provides a variable base access to them.

`Model` stores the whole model to be analyzed. All `Nodes`, `Properties`, `Elements`, `Conditions` and solution data. It also provides an access interface to these data.

`ModelPart` holds all data related to an arbitrary part of model. It stores all existing components and data like `Nodes`, `Properties`, `Elements`, `Conditions` and solution data related to a part of model and provides interface to access them in different ways.

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

`SpatialContainer` includes 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 place 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 parts of it and provides the necessary interface to communicate with applications.

`Application` provides all the information necessary for adding an application to Kratos. A derived class from it is necessary to give a kernel its required information, like new `Variables`, `Elements`, `Conditions`, etc.

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

6.2 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 limit 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 is 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 uses advanced language features in order to provide the desirable performance.

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

Basic Tools Layer Holds all basic tools used in Kratos. In this layer the use of advanced C++ techniques 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 provide interfaces with other libraries to benefit from existing work in the field.

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 to help developer in their implementation by reusing them.

User's Algorithms Layer This is 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 the 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 5 shows the multi-layer nature of Kratos.

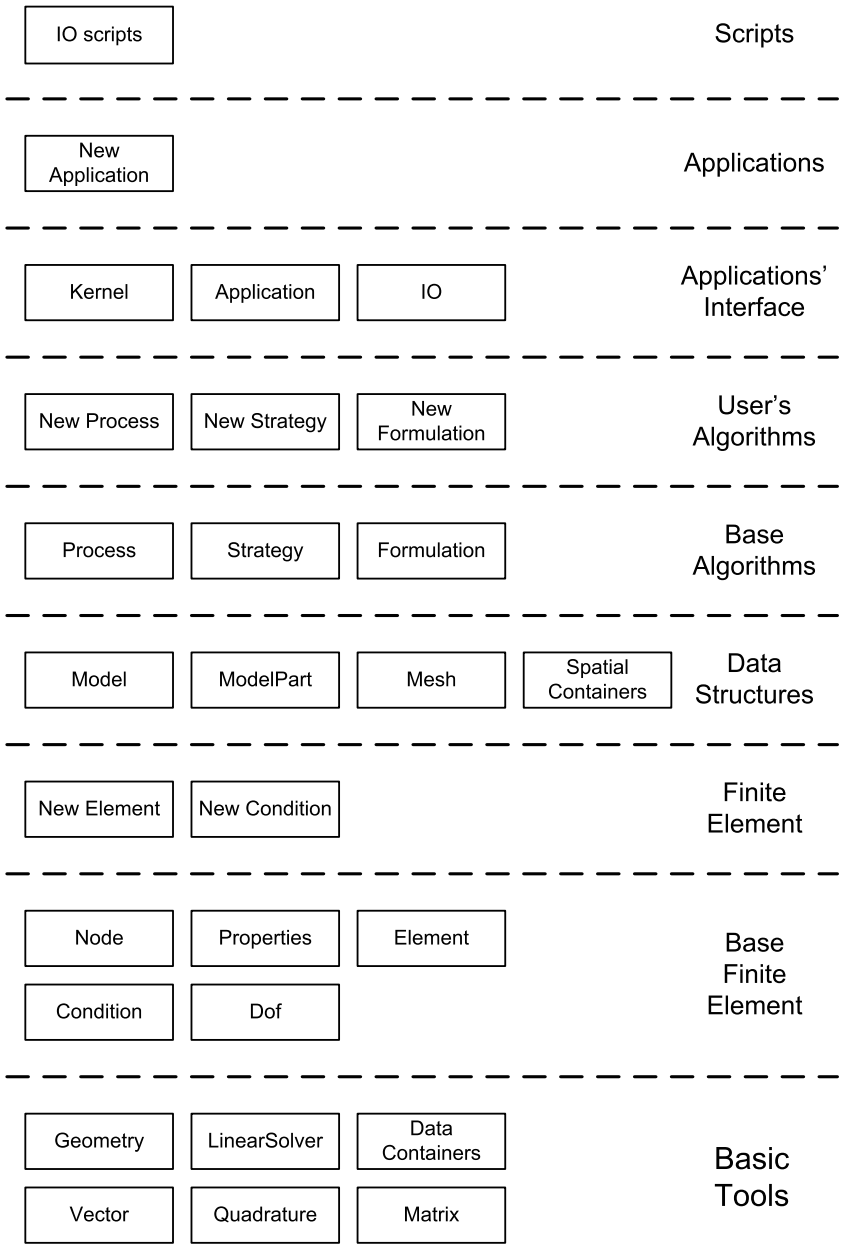


Fig. 5 Dividing the Kratos structure into layers reduces the dependency

6.3 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: `Variables`, `Elements`, `Conditions`, etc. defined in `Application`. The following code shows a typical `Application` class definition:

```
// Variables definition
KRATOS_DEFINE_VARIABLE(Matrix, MY_NEW_VARIABLE)

class KratosNewApplication
```

```
    : public KratosApplication
{
  public:
    virtual void Register();

  private:

    const NewElementType    mMyElement;
};
```

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

```
// Creating variables
KRATOS_CREATE_VARIABLE(MY_NEW_VARIABLE)

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

  // registering variables in Kratos.
  KRATOS_REGISTER_VARIABLE(MY_NEW_VARIABLE)

  KRATOS_REGISTER_ELEMENT("MyElement",mMyElement);
}
```

This interface enables Kratos to add all these `Variables`, `Elements` and `Conditions` 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.

7 Variable Base Interface

Connecting different modules and solvers in different fields of FE analysis has been always a challenge for programmers in the multi-disciplinary problem field.There are many successful examples using file interface or libraries like CORBA [87] or omniORB, though using them can cause big overheads in the code performance [47]. Also in this approach 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. This is the motivation to establish a variable base interface (see Fig. 6) which can be used at high and low levels in the same manner.

7.1 The Variable Base Interface Definition

In many connecting points between different parts of a finite element program we are asking for the value of some variable or the mapping of some variables and so on. The methodology to design this interface is sending each request with the variable or variables it involves.

In this variable base interface (VBI):

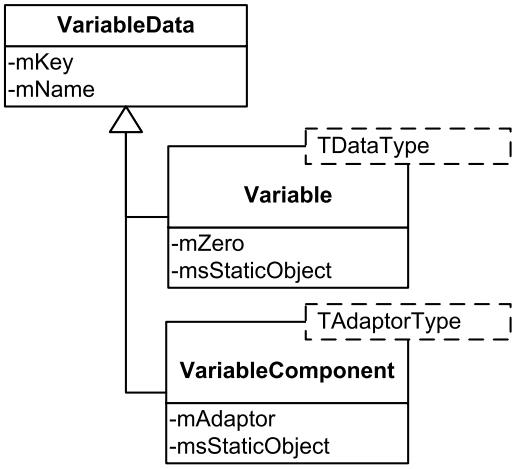


Fig. 6 Structure of the variable classes

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

This interface can be used in different situations. Passing data from one domain to other or reusing modules in a multi-disciplinary applications always requires a generic and extensible interface. This can be achieved by defining generic interfaces for modules like data structures, input-output, mapping and interpolating algorithms, etc. using the VBI.

7.2 Kratos Variable Base Interface Implementation

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

7.2.1 VariableData

`VariableData` is the base class which contains two basic informations 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 of type information in VariableData makes it unsuitable and less usable to pass it in many parts of the interface. The idea of this class, however, is to provide a lower level of information which is common to 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 in a poor performance for some cases as the function call overhead may be considerable.

7.2.2 Variable

Variable is the most important class in this structure. It has information represented by VariableData which is derived from it plus the type of the variable it represents. Variable has its data type as a template parameter. In this manner, the interface can be specialized 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 the 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 from passing another type by mistake. This feature is particularly important for cases when there are different types representing the same variable. The constitutive matrix and its corresponding vector is a good example of this situation.

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 creates a copy of the object using a copy constructor of the class. It is useful to avoid shallow copying of complex objects and also without actually having information about the variable type.
- Copy is very similar to Clone except that it also the destination pointer also passed 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 an assignment operator besides the copy constructor. Copy creates a new object while Assign does the assignment for two existing objects.
- AssignZero is a special case of Assign for 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 calls a destructor of objects to prevent memory leak and frees the memory allocated for this object assuming that the object is allocated in heap.
- Destruct eliminates an object maintaining the memory it is using. However, the unlike Delete it does nothing with the memory allocated to it. So it is very useful in case of reallocating a part of the memory.
- Print is an auxiliary method to produce output of given variable knowing its address. For example writing an heterogenous container in an output stream can be done using this method. Point assumes that the streaming operator is defined for the variable type.

All these methods are available for low level use. They are useful because they can be called by a VariableData pointer and equally for all type of data arranged in memory. However, 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 algorithm needs to initialize a variable without losing generality. For example an algorithm to calculate the norm of a variable for some Elements must return a zero value 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. Returning a zero value instead of a default constructed value keeps generality of the algorithms even for vectors and matrices, assuming that variables are defined properly.

A method is StaticObject. This method just returns None which is an static variable that 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.

7.2.3 VariableComponent

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

VariableComponent is a template taking an adaptor as its argument. An adaptor is the extending point of a component mechanism. For any new component a new adaptor (see Fig. 7) needs to be implemented. The adaptor type requirements are:

- GetSourceVariable method to retrieve parent variable.
- GetValue method to convert extract a component value from a source variable value.
- StaticObject is 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 implement it. Operations over raw pointers are not allowed on purpose. This interface manages

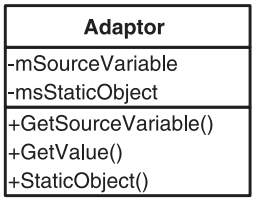


Fig. 7 Adaptor class

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 the 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.

7.3 Examples

A very first example is to access nodal values in a finite element program:

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

Overwriting the `[]` operators makes the syntax easier to use:

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

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

The next example shows the use of the VBI for passing additional parameters needed for calculating local contributions for each `Element`. Here a helper class named `ProcessInfo` is used to pass parameters like time step, current time, delta time and so on:

```
virtual void SomeElement::CalculateLocalSystem(
    MatrixType& rLeftHandSideMatrix,
    VectorType& rRightHandSideVector,
    ProcessInfo& rCurrentProcessInfo)
{
  // Getting process information
  double time = rCurrentProcessInfo[TIME];

  // Calculating local matrix and
  // vector ...
}
```

Writing output files can be generalized using the VBI. In this way any new extension to the library can write its results using existing output procedures. Here is an example of using the VBI to write a generic output procedure for GiD [72, 73]:

```
void GidIO::WriteNodalResults(
    Variable<double> const& rVariable,
    NodesContainerType& rNodes,
    double SolutionTag,
    std::size_t SolutionStepNumber)
{
  // Beginning results using variable's name
  GiD_BeginResult((rVariable.Name().c_str()),
    "Kratos",  SolutionTag, GiD_Scalar,
    GiD_OnNodes, NULL, NULL, 0, NULL );

  // Writing procedure ...

  GiD_EndResult();
}

void GidIO::WriteNodalResults(
    Variable<Vector> const& rVariable,
    NodesContainerType& rNodes,
    double SolutionTag,
    std::size_t SolutionStepNumber)
{
    // Writing procedure for vectors ...
}
```

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 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);
```

Finally, writing an error estimator is another example of making a generic and reusable code using the VBI. Here is an example of a simple recovery error estimator [89] implemented in a generic way:

```
void
EstimateError(const VariableType& ThisVariable,
              ModelPart& rModelPart)
{

  // initializing ...
```

```
  for(i_element = rModelPart.ElementsBegin();
      i_element != rModelPart.ElementsEnd();
      ++i_element)
    i_element->GetValue(ERROR) =
      CalculateError(ThisVariable,*i_element);
}

double
CalculateError(const VariableType& ThisVariable,
               Element& rElement)
{
  typedef typename VariableType::type data_type;

  // initializing ...

  for(i_node = element_nodes.begin() ;
      i_node != element_nodes.end() ; ++i_node)
  {
    error = i_node->GetValue(ThisVariable) -
      rElement.Calculate(ThisVariable, *i_node);

    result += sqrt(error * error);
  }
  return result * area / element_nodes.size();
}
```

In this manner the error estimator is not depending on the domain and can work in the same way for a thermal flow or a pressure gradient.

8 Data Structure

The 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. Usually several classical containers like static and dynamic arrays, link lists and trees are used to construct a data structure [7, 35, 44]. In Kratos beside these containers some new containers suitable for multi-disciplinary finite element programming are designed and implemented. The organization of the data is also prepared for storing multi-disciplinary data.

8.1 Designing New Containers

Standard C++ containers are homogeneous due to the static typing of C++ language. In this section some heterogeneous containers capable to store different types of data are introduced. Also the VBI is used to make their interface more generic, more clear and easier to use.

8.1.1 Combining Containers

Finite element developers usually work with integers and real numbers, vectors, matrices and sometimes complex numbers as their data. 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 containers holding above data types is to take different containers and put them together as a new container. Figure 8 shows an example of this container.

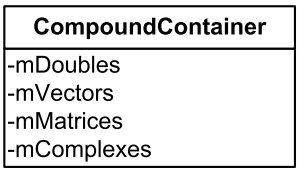


Fig. 8 Combining containers for holding doubles, vectors, matrices and complex numbers

Advantages for Combining Containers

- Fast and easy implementation. Standard containers can be reused.
- Very rigid and errorless structure. Everything relies on C++ static type checking without dangerous type casting and raw pointer manipulation.
- Keeping separated different types of data allows more specialization for each case. For example in time of copying the containers for built in types.
- Less searching time as the number of data in each container is less than the total number of data in container.

Disadvantages of Combining Containers

- Extra memory overhead is needed for supporting any new type. Supporting any new data type still increases more this overhead.
- Adding new types needs modifying the container. A good implementation can minimize this modification.

Implementation There are different ways to implement this type of container. One way is implementing the containers with sub-containers as its attributes. A better approach is to use multiple hierarchy to group different containers in a combined one. In this way 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. Figure 9 shows this approach. Finally, the VBI provides a uniform template model for accessing an element by variable.

8.1.2 Data Value Container

A data value container is a heterogeneous container with a variable base interface designed to hold the value for any type of variable.

Usually a container needs to do some basic operations over its data like: creating, copying, deleting, etc. which may vary from one type to other. A heterogenous container therefore needs a mechanism to handle each different type with its corresponding process. A common way to deal with this problem is to encapsulate all necessary operations into a handler object and associate it to its corresponding data. In

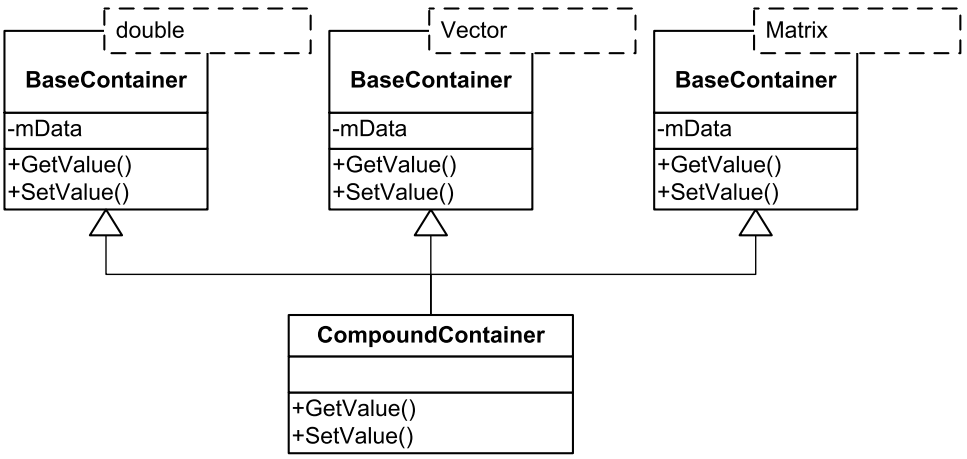


Fig. 9 Combining different containers using multiple hierarchy

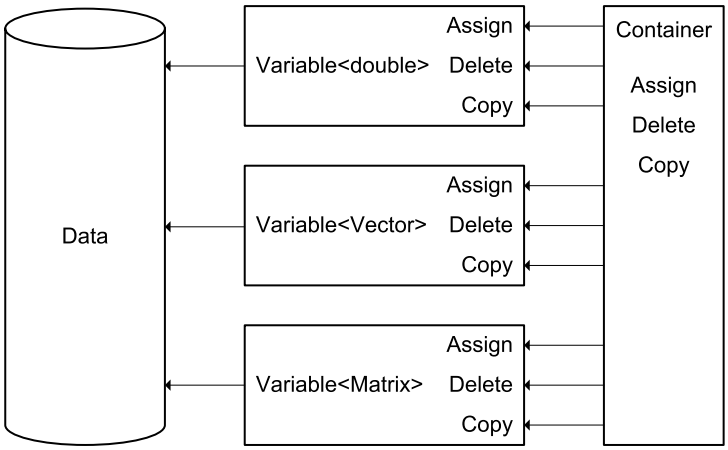


Fig. 10 Data value container uses the `Variable` class to process its data

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 10 shows the relationship between a container and the `Variable` class.

Data Value Container Advantages

- Extendibility to store any type of data without any implementation cost. 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 VBI 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.

Implementation The first step to implement a data value 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 Fig. 11.

Having a pointer to each variable and not copying it is necessary for eliminating 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 12 shows a container with this structure in memory.

The first approach is typically more efficient in use of cache but adding new data to it may invalidate all the references to its elements. The second approach lets user to get a reference of its data once and use it several times 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. Finally, a brute-force search over unsorted containers has been used for searching the keys as this has given the best performance in practice (Fig. 13).

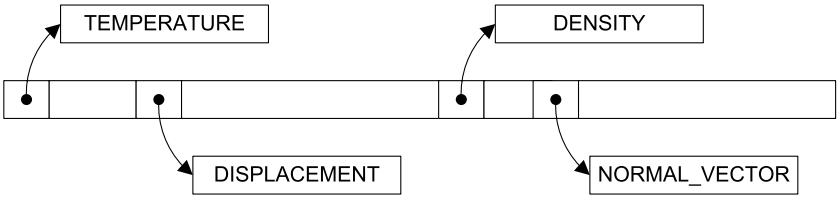


Fig. 11 A data value container with continuous memory

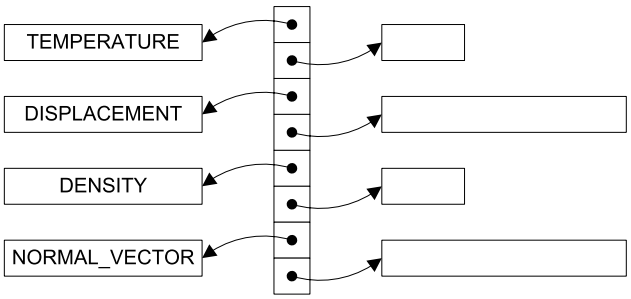


Fig. 12 A data value container with discontinuous memory

Now a VBI completes our design. Access methods use `Variable` or `VariableComponent` as data information as described for the VBI. Each access consist of a find process for given variable key and then convert the data to the given variable type. It is important to mention here that using the VBI not only increases the readability of the code but also protects users from unwanted type crashing.

8.1.3 Variables List Container

In finite element programs it is common to store the same set of data for all `Nodes` of a domain. For example in a fluid domain each `Node` has to store velocity and pressure. The previous heterogeneous container can be used to store these data but the searching procedure in order to access data makes it inefficient. So 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 assigns 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 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 14 shows this procedure.

Variables List Container Advantages

- The accessing and finding processes are very efficient.
- Extendibility to store any type of data without any implementation cost.
- Using the VBI 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 which makes them practically less independent.
- Erasing a variable from this container is a complex and difficult task.

Implementation The first approach to implement this container is to encapsulate everything in the container and taking a simple list of variables to work. This approach looks attractive but requires recalculation of the offset for each access which imposes an unacceptable 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 15 shows this structure.

An important decision here is to let the container add new variables to the shared list or not? 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 strategy 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

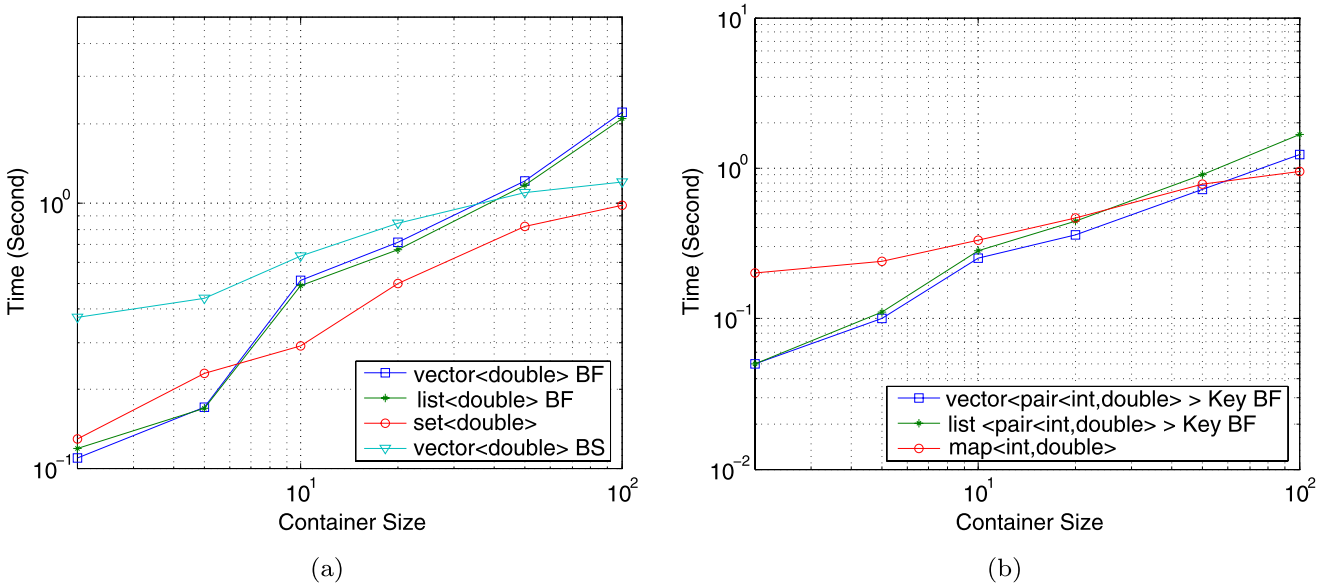


Fig. 13 Time comparison for different searching algorithms over sorted and unsorted small containers. (**a**) Comparing performance of `vector<double>` with brute-force, `list<double>` with brute-force, `set<double>` binary tree search and sorted `vector<double>` with binary search. (**b**) Comparing 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

Fig. 14 Accessing to a value in the variables list container

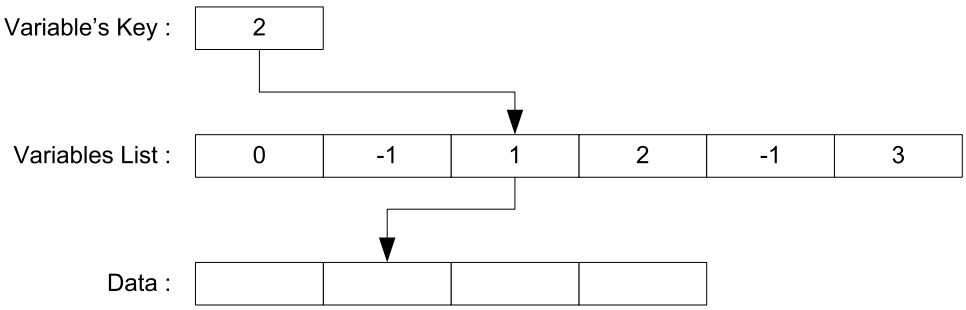
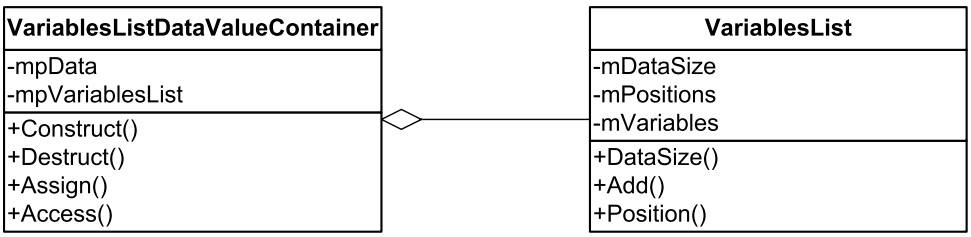


Fig. 15 The `VariablesList` class provides the list of variables and their local positions for the `VariablesListData-ValueContainer`



in another thread just after a reference is taken. So finally the inserting feature was removed from this container to reduce problems using it.

8.2 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 applications, `Nodes` and `Elements` 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.

8.2.1 Global Organization of the Data Structure

Kratos is designed to support an element-based formulation for multi-disciplinary finite element applications. Also handling mesh adaptivity is one of its goals. So the entity based data structure becomes the best choice. First because ele-

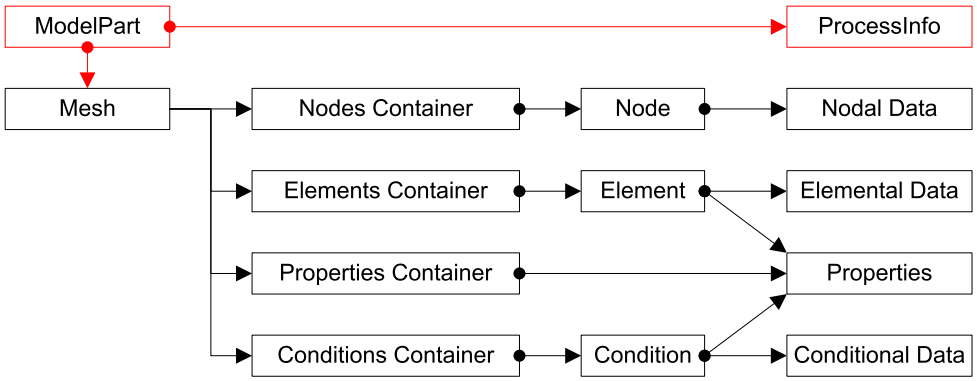


Fig. 16 `ModelPart`holds `Mesh` with some additional data referred as `ProcessInfo`

mental 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 `Nodes` and `Elements`. 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. Figure 16 shows the containers in the Kratos data structure.

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` is also a block of this structure, as a shared data between `Elements` or `Conditions`.

Separate containers for `Nodes`, `Properties`, `Elements` and `Conditions` 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 `Nodes` to a nodal data initialization procedure, sending a set of `Elements` to assembling functions, or getting a set of `Conditions` from a contact procedure.

`Mesh` is the second level of abstraction in the data structure which hold `Nodes`, `Elements` and `Conditions` and their `Properties`. In other words, `Mesh` is a complete pack of all type of entities without any additional data associated with them. So a set of `Elements` and `Conditions` with their `Nodes` 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.

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`. Each `ModelPart` can hold more than one `Mesh` which can be used to keep trace of different categories of entities like interface, local, ghost etc. `ModelPart` also manages the variables to be hold in `Nodes`. For example, all the `Nodes` belonging to one `ModelPart` sharing its nodal variables list. `ModelPart` is the nearest container to the domain concept in the multi-disciplinary FEM.

Finally `Model` is a group of `ModelPart`'s and represents the finite element model to be analyzed. It can be useful for some procedures that require 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 [62].

8.2.2 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 Fig. 17. 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`. A data value container is used for the nodal data (no historical data) and a buffered variables list container is

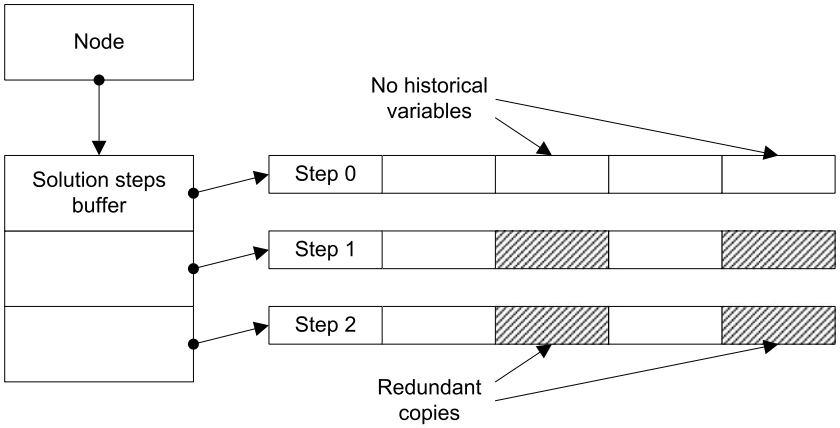


Fig. 17 Using buffer for all variables results in memory overhead due to redundant copies of no historical variables

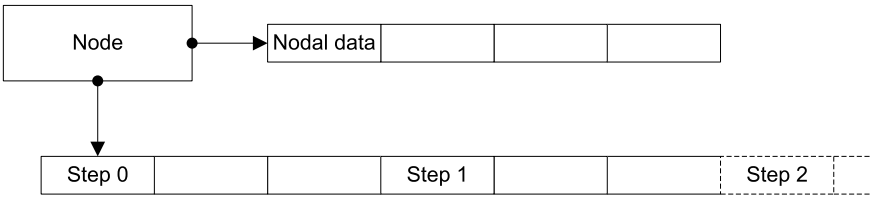


Fig. 18 The current structure allocates all buffer data in a block of memory to reduce the cache misses produced by memory jumps and also to provide a compatible data with other libraries

used for the solution step nodal data (historical data). In this way the memory overhead is eliminated because no redundant copy is produced. Also accessing to historical variables is much faster than before due to the indirecting process of accessing 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.

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 the 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.

In this implementation the buffer was introduced inside the variables list data value container instead of using a buffer of this containers. In this way cache misses are reduced and the data array can be given to other application without any conversion. Figure 18 shows this structure.

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 from afterwards. 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 no historical data only. 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, then 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`. This method guarantees the access to the proper temperature stored at each `Node`.

8.2.3 Elemental Data

Another basic unit of Kratos data structure is the elemental data. Elemental data is divided into three different categories:

properties These are all parameters that can be shared between `Element`. Usually material parameters are common for a set of element, so this category of data is referred as properties. But in general it can be any common parameter for a group of `Elements`. Sharing these data

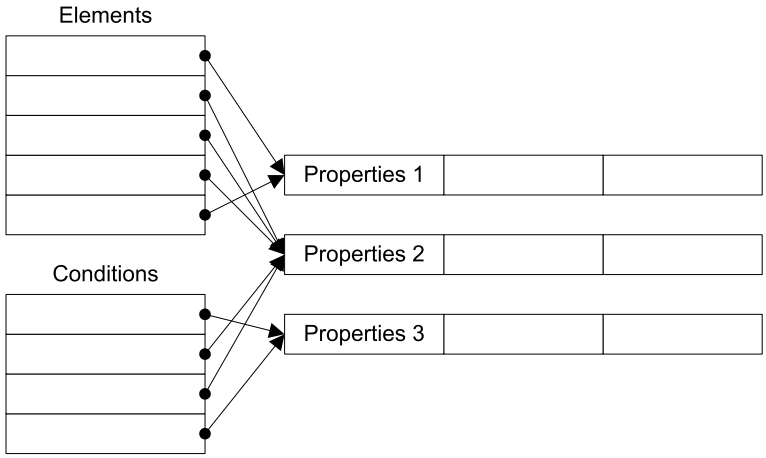


Fig. 19 Different elements or conditions use `Properties` as their share data container. This avoids redundant copies of data in memory

as properties reduces the memory used by the application and also helps updating them if necessary.

data includes 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 are 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 elements. For this reason `Element` keeps a pointer to its `Properties`. This connection lets several elements to use the same properties.

A `DataValueContainer` holds no historical data in `Element`. Using a `DataValueContainer` provides flexibility and robustness which is useful in transferring elemental data from one domain to another. As an example, one may want to set a flag over elements say active or inactive. The `DataValueContainer` can be used easily for this purpose. On the other hand a damage function may need history of some internal variable. This second result is best achieved by implementing an application specific data base inside `Element`.

8.2.4 Conditional Data

Conditional data is very similar to elemental data and is also divided into three different categories: properties, data and historical data. It is also keeps a pointer to its shared `Properties` and uses a `DataValueContainer` to hold all data 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 helps 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 as is 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.

8.2.5 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 shown in Fig. 19.

`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`.

8.2.6 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

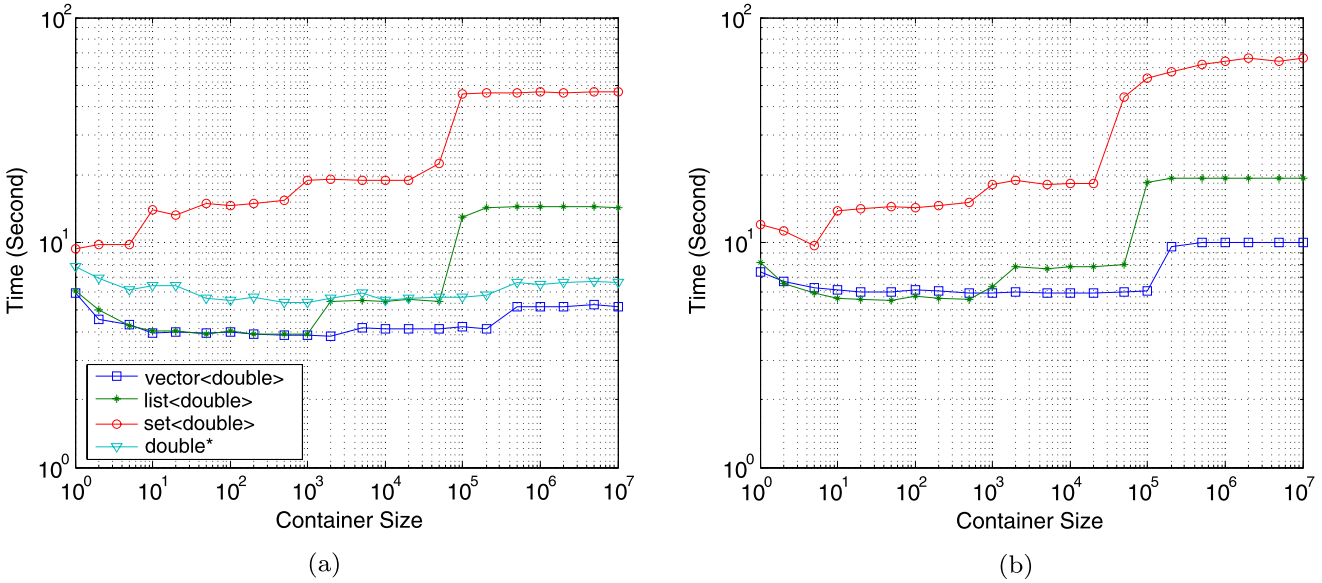


Fig. 20 Iterating time for 10^9 steps of iterations with different containers. (**a**) Comparing the performance of `double*`, `vector<double>`, `list<double>` and `set<double>`. (**b**) Comparing the performance of `vector<pair<int,double> >`, `list<pair<int, double> >` and `map<int,double>`

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 into a `Nodes` container and change some of their data in each step. As mentioned earlier, 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. 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 Several 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.

Sharing entities is the first feature to be provided by these 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* [26] instead of a normal pointer increases the robustness of the code. In this way entities that do not belong to any list anymore will be deallocated from memory automatically. So a container of smart pointers to entities is used.

Arrays are very efficient in time of iterating as shown in Fig. 20. 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 and 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 as shown in Fig. 21. Its only draw back is that its less robust and its construction it can take considerable time depending on the order of input data. For example constructing an array of `Nodes` with `Nodes` given by inverse order can take a very long time. However 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.

8.2.7 Mesh

The next level in Kratos' data structure is `Mesh`. It contains all the 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.

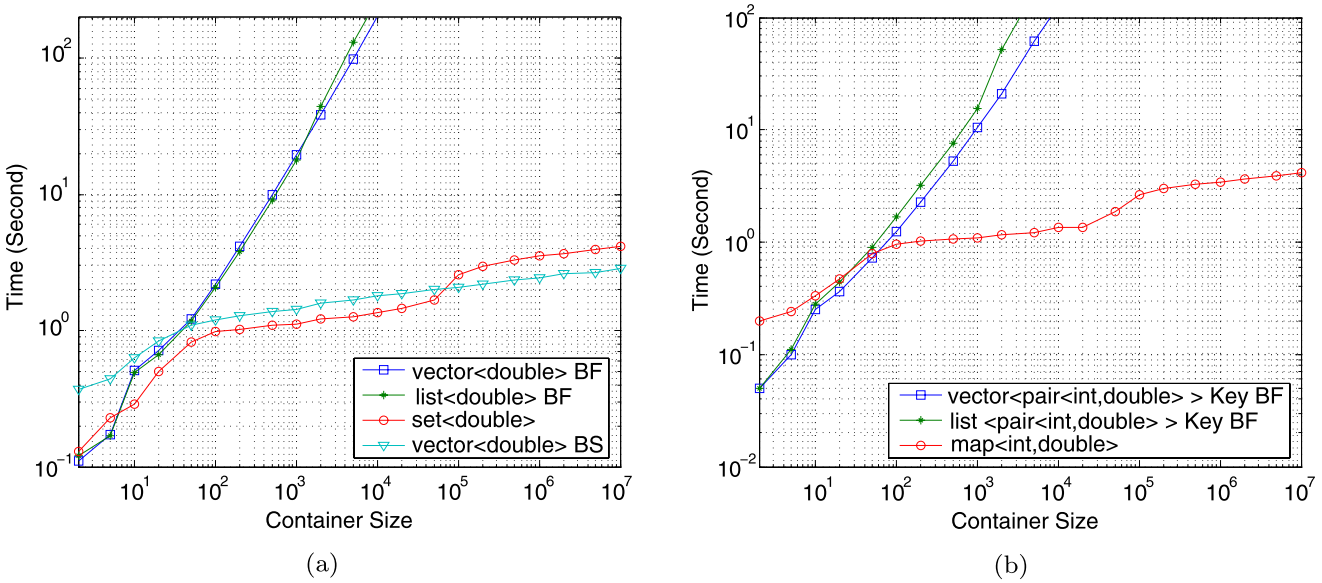


Fig. 21 Time comparison for different searching algorithms over sorted and unsorted containers. (**a**) Comparing the performance of `vector<double>` with brute-force, `list<double>` with brute-force, `set<double>` binary tree search and sorted `vector<double>` with binary search. (**b**) Comparing the 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

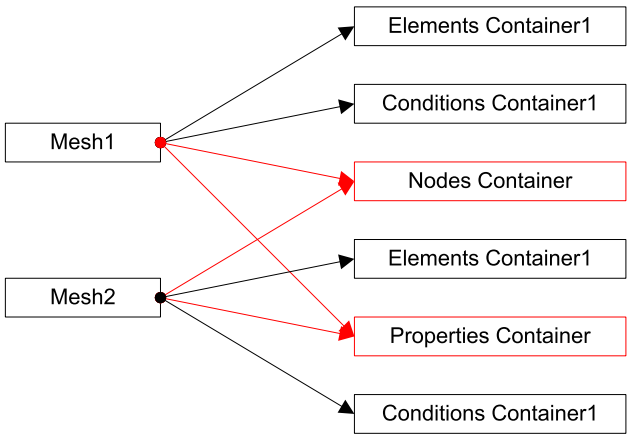


Fig. 22 Different `Meshes` can share their entities' containers

First of all `Mesh` provides a separate interface for each type of entity it stores.

`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 22 shows this ability of sharing components between `Meshes`.

8.2.8 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 a parallel process. Beside holding different `Meshes`, it also stores the solution information encapsulated in the `ProcessInfo` object.

`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 Fig. 23.

`ModelPart` uses pointers to its `Meshes`. In this way it can share them with any other model parts if necessary, so that the same node or element can be part of two different `ModelParts`. A typical use of this feature is defining two different domains over the same `Meshes`. Figure 24 shows this sharing mechanism.

`ModelPart` manages the variables lists of its components. In Sect. 8.1.3 the mechanism of the variables list container was described. These 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

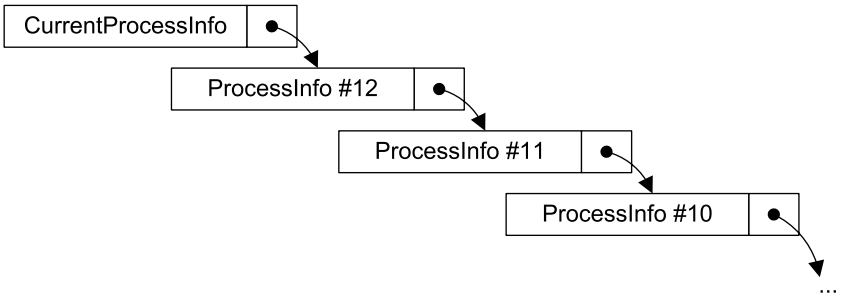


Fig. 23 `ProcessInfo`'s linked list mechanism for holding the history of solution

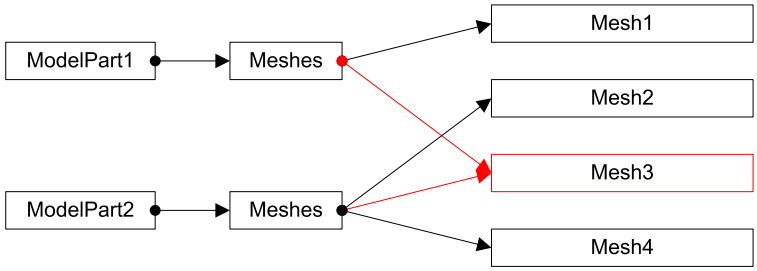


Fig. 24 `ModelPart` can share its `Meshes` with other model parts

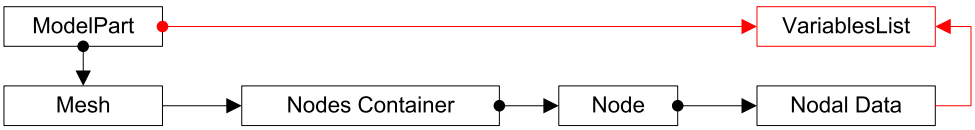


Fig. 25 `ModelPart` manages the variables list for its entities

`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 25 shows this scheme.

8.2.9 Model

`Model` is the representation of the whole physical model to be analyzed with 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.

9 Finite Element Implementation

9.1 Elements

`Elements` and `Conditions` are the main extension points of Kratos. New formulations can be introduced into Kratos by implementing a new `Element` and its corresponding `Conditions`. This makes `Element` an special object in our design.

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 a 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`.

The strategy [34] pattern is used to design `Element` structure as can be seen in Fig. 26. Using this pattern each `Element` encapsulates one algorithm separately and lets new algorithms to be added easily to Kratos without changing other parts. Also this pattern keeps them compatible with each other in order to let users interchange them, or even mix them together in a complex model.

In Kratos `Geometry` holds a set of points or `Nodes` and provides a set of common operations to ease the implementation of `Elements` and `Conditions`. The bridge pattern is used to connect `Geometry` with `Elements`. Introducing this pattern to our `Element`'s structure design results in the structure shown in Fig. 27.

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

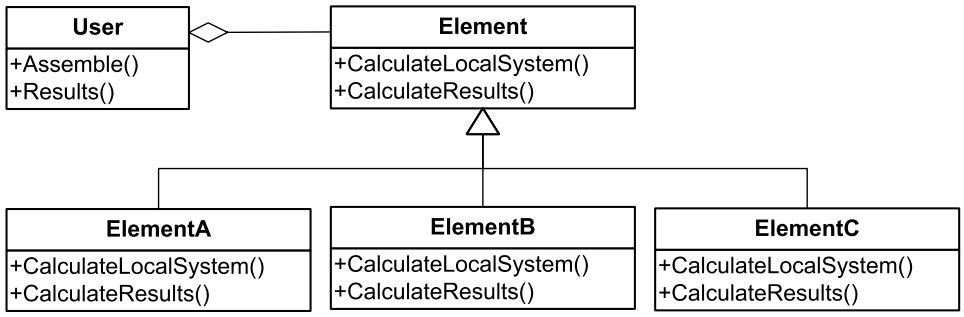


Fig. 26 `Elements`' structure using strategy pattern

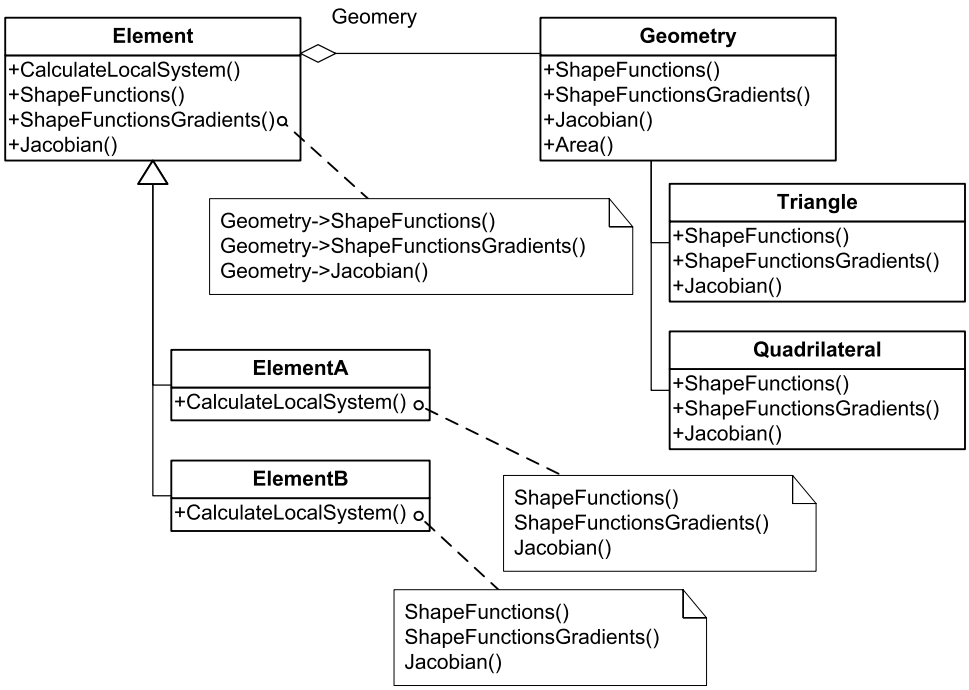


Fig. 27 `Element`'s structure using the bridge pattern

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 geometrical data respect to an `Element` derived directly. Though efficiency in `Element` is crucial, the complexity of the first approach which one's the first and which one is the second 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 features of programming language. So finally the bridge pattern structure was selected.

There are some designs in which different `Elements` can be composed to create a more complex `Element` [58]. This approach can be simulated here using a *Composite* pattern. However this structure is not implemented yet in Kratos.

The finite element methodology is used for designing a generic interface for `Element`. 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` provides three sets of methods. `CalculateLocalSystem`, `CalculateLeftHandSide` and `CalculateRightHandSide` for calculating local system matrices and vectors. `EquationIdVector` and `GetDofList` are designed to provide assembling information for `Strategy`. Finally, `Calculate` and `CalculateOnIntegrationPoints` are devoted to calculating elemental variable which are used mainly for calculating post-analysis results.

A VBI is used to provide a clear but flexible interface for these methods.To ensure the extensibility of these meth-

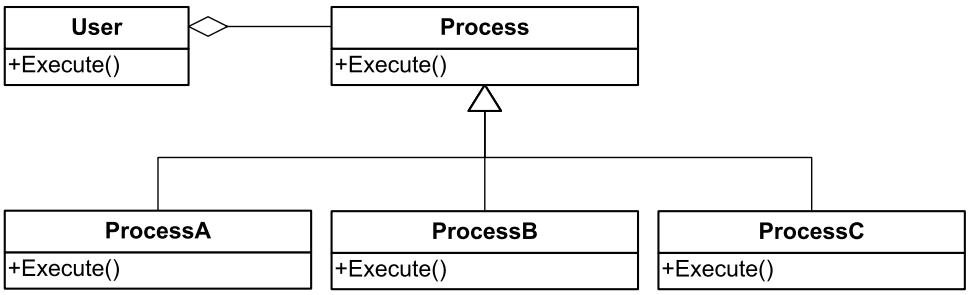


Fig. 28 `Process` structure using strategy pattern

ods a `ProcessInfo` is passed to these methods. In this way users can pass any parameter like time, time increment, time step, non-linear iteration number, some global norms which are calculated over the domain, etc. to these methods. Using `ProcessInfo` guarantees the flexibility which is necessary for `Element` to be an extension point of Kratos. In these methods passing the resulting matrices or vectors by reference as additional arguments improves the performance.

Providing an standard way to access neighbors of an element 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. In Kratos neighbor nodes and elements are stored inside the elemental data container exactly if they were standard variables. This is indeed an interesting example of the flexibility of the VBI. In this way the overhead of empty containers is 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.

9.2 Conditions

`Condition` is defined to represent the conditions applied to boundaries or to the domain itself. Neumann conditions and interfaces between domains in multi-disciplinary problems are represented by `Condition`. The only exception is the Dirichlet condition which is applied by `dof` due to the applying procedure described in Sect. 3.2.2. In Kratos `Condition` is designed very similar to `Element`. They interact with `Strategy` in the same way as `Elements`. 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`.

9.3 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. 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 [23]. 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 larger one is also helpful, specially to make a pack of small processes for handling a complex algorithm.

`Process` can be considered as a function class. `Process` is created and executed just like calling a function. The strategy pattern is used to design the family of processes. Figure 28 shows this pattern applied to the `Process` structure.

Applying this pattern allows `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.

The process interface is relatively simple. `Execute` 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 arguments without any dependency to other processes or the base `Process` class.

9.4 Solving Strategies

After designing `Process` and its derived classes, we will focus in an important family of processes which manage the solution 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 solver strategy and the Newton-Raphson iterative strategy.

`SolvingStrategy` is derived from `Process` and use the same structure. Deriving `SolvingStrategy` from `Process` lets users to combine them with some other processes using composition in order to create a more complex `Process`.

Like for `Process` users can combine different strategies in one. For example a fractional step strategy can be implemented by combining different strategies used for each step in one composite strategy. The interface of `SolvingStrategy` reflects the general steps in usual finite element algorithms like prediction, solving, convergence control and calculating results. This design yields 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` 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` is a post-solution convergence check. It can be used for example in coupled problems to verify if the solution has 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 Fig. 29.

This structure is suitable when the algorithm is not changing at all. However, 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 30 shows this structure.

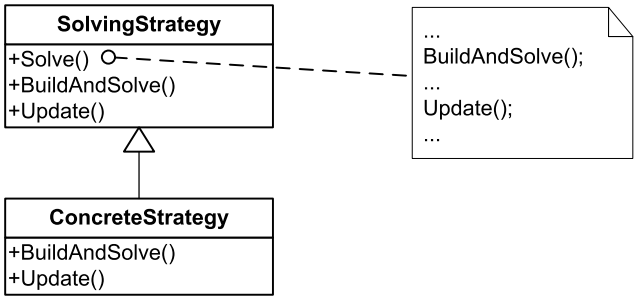


Fig. 29 Template method pattern applied to the solving strategy

The main idea of using these two additional set of objects is to increase the reusability of the code and prevent users from implementing a new `Strategy` from scratch. In practice this 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 Kratos structure both approaches can be used to implement a solution algorithm.

9.4.1 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.

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. `BuildLHS`, `BuildRHS`, `Build`, `ApplyDirichletConditions`, `SystemSolve`,

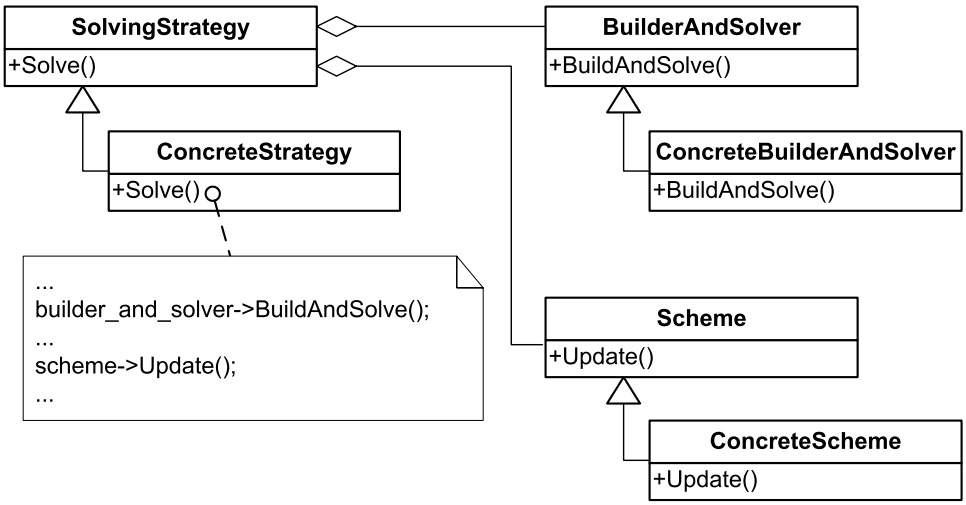


Fig. 30 Deferring different parts of the algorithm to `BuilderAndSolver` and `Scheme`

`BuildAndSolve` and `CalculateReactions` are provided for this purpose. There are also several other 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.

9.4.2 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. The definition of scheme is quote general and can be used to encapsulate other similar operation over a solution component.

According to the template method pattern the important steps of the solving procedure in standard 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. `Initialize`, `InitializeSolutionStep`, `FinalizeSolutionStep`, `InitializeNonLinIter-ation`, `FinalizeNonLinIteration` and `Update` represent these main steps.

9.5 Elemental Expressions

The finite element methodology usually consists of first converting the governing differential equation to its weak form. This is discretized over an appropriate approximation space, and finally global system of matrix equation is built from the elemental contributions. Different approaches in order to automatize this process can be found in literature [27–29, 48, 91].

Nowadays several computer algebra systems like *Matlab* [56], *Mathematica* [88], and *Maple* [55] 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 with isotropic coordinates, the elemental stiffness matrix S^e can be calculated as follows:

$$S^e{}_{ij} = k \int_\Omega (\nabla N_i)^T \mathbf{I} \nabla N_j d\Omega$$

or:

$$S^e{}_{ij} = k(\nabla_i N_l, \nabla_j N_l)$$

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 < gauss_points_number ; l++)
  {
   Matrix const& gn=shape_functions_gradients[l];

   for(int d=0 ; d < dimension ; d++)
    rLeftHandSideMatrix(i,j) += k*gn(i,d)
                                *gn(j,d)*wdj;
  }
```

Using elemental expressions the same formulation can be written in a simpler form as:

```
KRATOS_ELEMENTAL_GRAD_N(i,l) grad_Nil(data);
KRATOS_ELEMENTAL_GRAD_N(j,l) grad_Njl(data);

rLeftHandSideMatrix = k*(grad_Nil, grad_Njl)*wdj;
```

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. Simple overloading results in poor performance due to the redundant temporary objects that creates. Expression template technique can be used to convert above expression to previous hand written form automatically. Template metaprogramming 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.

In the current version of Kratos, element expressions are still in experimental phase. However some benchmarks have shown that their efficiency is comparable with hand coded `Elements` as supposed to be.

9.6 Formulations

Kratos has been designed to support element approaches in finite element methods. For some problems an element approach is 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 will be one of the future features of Kratos.

10 Input-Output

In general most finite element applications have to communicate with pre and post processors, except in some special cases for which the application generates its own input. This makes the input-output (IO) operation an essential part of the application. For a multi-disciplinary code IO must deal with new variables and data types. Also providing some additional features like multi format and media support, *serialization* [1] and script language support are very useful.

10.1 IO Structure Design

The Strategy pattern is used to encapsulate each format separately and make it extensible to any new format easily. The same structure can be used for supporting different media separately. Then the `Bridge pattern` can be used to connect them to the format structure. However a simpler approach is to treat the media like formats and use the same structure for both and enrich it via templates or several levels of hierarchy as shown in Fig. 31.

There are different concepts (like temperature, displacement, viscosity, etc.) and data types in a FEM data. Supporting different concepts and data type is very important for a multi-disciplinary code. Here the VBI comes handy and it is used to generalize the IO.

The first step is providing IO extendibility to new concepts. This can be done by introducing a lookup table which relates the concept names and their internal handlers. A simple list of `Variables` 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 `"NODES[29](TEMPERATURE,0)=418.651"` statement from input, it 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.

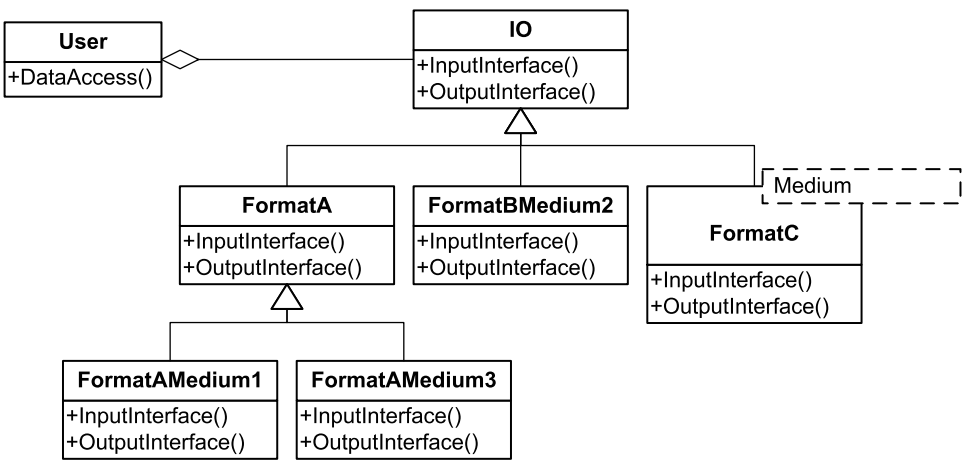


Fig. 31 Extended Multi format and Medium IO structure

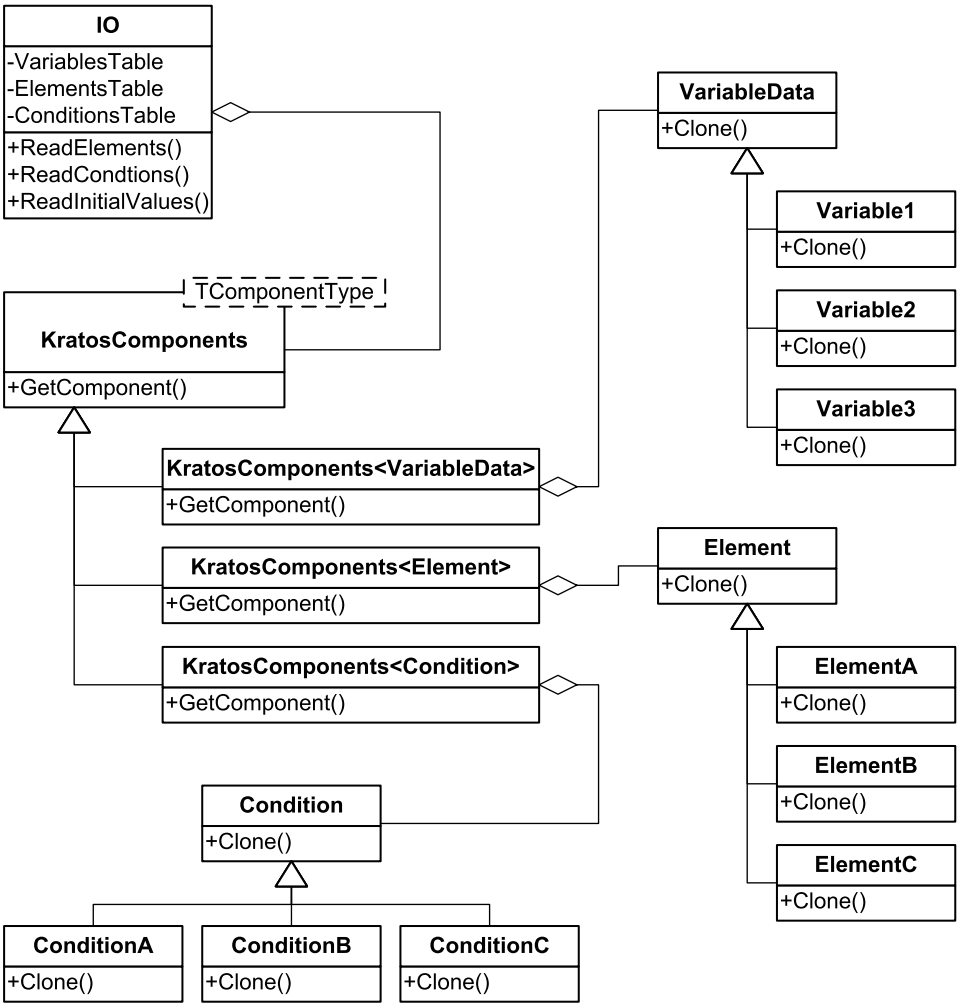


Fig. 32 Using `KratosComponents` in IO

```
template<class TDataType>
void WriteNodalResults(
    Variable<TDataType> const& rVariable,
    NodesContainerType& rNodes,
    std::size_t SolutionStepNumber)
{
  NodesContainerType::iterator i;
  for(i=rNodes.begin() ; i!=rNodes.end() ; ++i)
    Output << rVariable.Name() << "="
           << i->GetSolutionStepValue(rVariable,
                           SolutionStepNumber))
           << std::endl;
 }
```

Each new `Element` or `Condition` also is a new type which implies that IO does not know how to create it. 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.

IO uses a 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 `Elements` there is no need to search in `Variables` and `Conditions` and to put all of them together can slow down the parsing process unnecessarily. Dividing the lookup table into three family of classes: `Variables`, `Elements` and `Conditions` helps

to distinguish them in the time of search. Doing this also eliminates unnecessary type casting and makes the implementation easier and clearer. Figure 32 shows the resulting structure.

Finally an interpreter is necessary to provide the script language input feature for this module. This feature provides a high level of extensibility in algorithms which is very useful in order to deal with different algorithms as is typical in multi-disciplinary applications.

10.1.1 IO Interface Design

The input interface is quite straightforward and consists of methods to read `Nodes`, `Properties`, `Elements`, `Conditions`, initial values, `Meshs` and `ModelParts`. The two latter ones are useful for reading multi-disciplinary data for various domains.

The output interface consists of methods for writing `Nodes`, `Properties`, `Elements`, `Conditions`, `Meshs` and `ModelParts` and also results for postprocessing. The VBI is used to define the result methods interface in a generic way.

10.2 Writing an Interpreter

In modern applications an interpreter is used to read the input file and understand the input grammar [43, 66]. An interpreter usually is divided into two parts as shown in Fig. 33.

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 eliminate all white spaces and also the 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 1 shows some examples of tokens.

Table 1 Some typical tokens with examples of matching sequences

Token's Type	Examples
INTEGER	1 34 610
REAL	4.57 2e-4
IF	if
FOR	for
LPARENTHESES	(
RPARENTHESES	)

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.

The second part is the *parser*. A parser does the syntax analysis. It takes the tokens from the lexical analyzer and tries to find their relation and meaning due to its grammar. A 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 the above sequence of tokens to a C parser would result in a parse tree as presented in Fig. 34.

There are several reasons to separate the lexical analyzer from the parser. The first one is simplifying the design and reducing the 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 in the lexical analysis, separating it and using techniques like buffering can decrease 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 divided into two phases. A scanner which does simple tasks like removing comments and whitespaces, and an analyzer which does the real complex job.

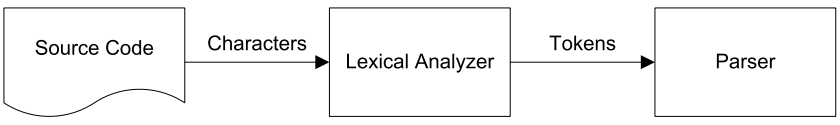


Fig. 33 Global structure of an interpreter

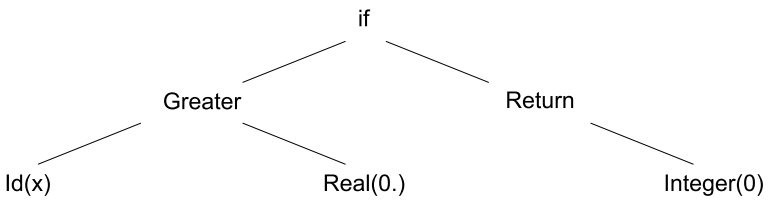


Fig. 34 Parse tree for an `if` statement

There are various ways to implement a lexical analyzer and a parser in the literature [5, 6, 9]. Also there are some tools to generate them. Some classical ones are *Lex* and *Yacc*, and their derivatives like *Fast lexical analyzer generator* (*Flex*) and *Bison* and their C++ variations *Flex++* and *Bison++*.

The first interpreter for Kratos was implemented using Flex++ and Bison++. They 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 creating a sophisticated interpreter to using an existing one. 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 maintainability issues, specially in Windows, due to the incompatibilities between the Linux and Windows compilations of these tools. The current interpreter is implemented using *Spirit*.

Spirit is an object-oriented parser generator framework implemented in C++ using template meta-programming techniques which 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. This interpreter can be used to read a data file containing nodes, elements, properties, conditions, and initial values and automatically reads any new variable in the variables list.

10.3 Using Python as Interpreter

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 code cost. Also implementing an interpreter requires knowledge of 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 development 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.

For selecting a script language, the following requirements are used as selection criteria: interface to C++ or at least to C. Portability. To be object-oriented. Language syntaxes and its readability. Flexibility and extendibility and popularity

From the long list of languages first some of more popular ones were selected: *Lisp*, *Perl*, *Ruby*, *Tcl* and *Python*. Python was finally selected due to some details and also our background. Tcl was a good choice as it is already used at CIMNE for example in GiD [72, 73] 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. An existing well designed interface to Python and some practical use of Python in finite element applications [3, 4] were another reasons for choosing Python.

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 performed easily at this level without changing Kratos or its applications. Also Python 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.

10.3.1 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.

11 Validation Examples

In this chapter some applications developed using Kratos are presented to test the desired flexibility and extensibility of the framework.

11.1 Incompressible Fluid Solver

The Kratos features an incompressible CFD solver, implemented in an "incompressible fluid application". Both monolithic and fractional step solvers (see [22]) are implemented in the same application. Most of the examples described in the following section were run using a Fractional-Step type approach, in order to achieve maximal efficiency.

11.1.1 Implementation in Kratos

The main CFD solvers of the Kratos are based on an Arbitrary Lagrangian Eulerian (ALE) formulation. The solver used in the examples is based on a fractional step method

[18] using equal order pressure-velocity elements stabilized by Orthogonal SubScales (OSS) [19], or Finite Increment Calculus (FIC) [63].

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 was implemented by creating a new `SolvingStrategy` combining existing ones for different steps. The strategy was hard coded in C++, however the implementation was such that all the different steps could be solved separately. This allowed the definition of a flexible Python interface which in turn permits the end-user to control the flow of the program. This flexibility in particular provides major advantages in defining new fluid structure interaction coupling strategies based on the existing ALE solver.

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(m_step, mtime_order);
  double Dp_norm;

  // predicting the velocity
  PredictVelocity(m_step,mprediction_order);

  // initialize projections at the first steps
  InitializeProjections(m_step);

  // Assign Velocity To Fract Step Velocity
  // and Node Area to Zero
  AssignInitialStepValues();

  if(m_step <= mtime_order)
      Dp_norm = IterativeSolve();
  else
  {
      if(mpredictor_corrector == false)
          Dp_norm = FracStepSolution();
      else  //iterative solution
          Dp_norm = IterativeSolve();
  }

  if(mReformDofAtEachIteration == true )
      Clear();

  m_step += 1;


  mOldDt =
    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
  SolveStep1(mvelocity_toll, mMaxVelIterations);

  // solve for pressures
  // (and recalculate the nodal area)
  double Dp_norm = SolveStep2();

  ActOnLonelyNodes();

  //calculate projection terms
  SolveStep3();

  //correct velocities
  SolveStep4();

  return Dp_norm;
}
```

This strategy can be exported to Python without loss of performance, by providing access to the specific methods implementing each step. We can write in Python an equivalent solution step as shown in Table 2.

It can be seen that 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. This `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 `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 `Element`:

```
void Fluid3D::CalculateLocalSystem(
        MatrixType& rLeftHandSideMatrix,
        VectorType& rRightHandSideVector,
        ProcessInfo& rCurrentProcessInfo)
{
  KRATOS_TRY

  int FractionalStepNumber =
        rCurrentProcessInfo[FRACTIONAL_STEP];

  if(FractionalStepNumber <= 3)
  {
      Stage1(rLeftHandSideMatrix,
             rRightHandSideVector,
             rCurrentProcessInfo,
             FractionalStepNumber - 1);
  }
  else if (FractionalStepNumber == 4)
  {
      Stage2(rLeftHandSideMatrix,
             rRightHandSideVector,
             rCurrentProcessInfo);
  }

  KRATOS_CATCH("")
}
```

Where `Stage1` and `Stage2` are private methods.

Table 2 The fractional step strategy written in Python

```
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()
```

11.1.2 Benchmark

Kratos is a general purpose code. Therefore, it is expected to show a slightly 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 a comparison was possible here with the code Zephyr, an in house program developed at 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 [45]. 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 35 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 1 m/s. The mesh generated by FEFLO has a resolved boundary layer and contains 30000 nodes and 108000k tetrahedral elements. The values computed were the lift and drag history for the cylinder. Figure 36 shows a view of velocity field obtained.

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 Fig. 37. Note that the initiation of the shedding is not deterministic and may happen at any time which justifies the phase difference between the two codes.

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.

On the other hand, Zephyr features an element based formulation and implements the same fractional method 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 twice as faster than Zephyr.

11.2 Fluid-structure Interaction

Coupled problems can be naturally implemented inside Kratos via the Python interface. The fluid solver and the structural solver can be implemented separately and coupled using this interface. The first action required to solve a fluid structure interaction (FSI) problem is to load the different applications involved. The code in Table 3 shows this step in Python. Then a very simple explicit coupling procedure can be expressed by a Python script as shown in Table 4.

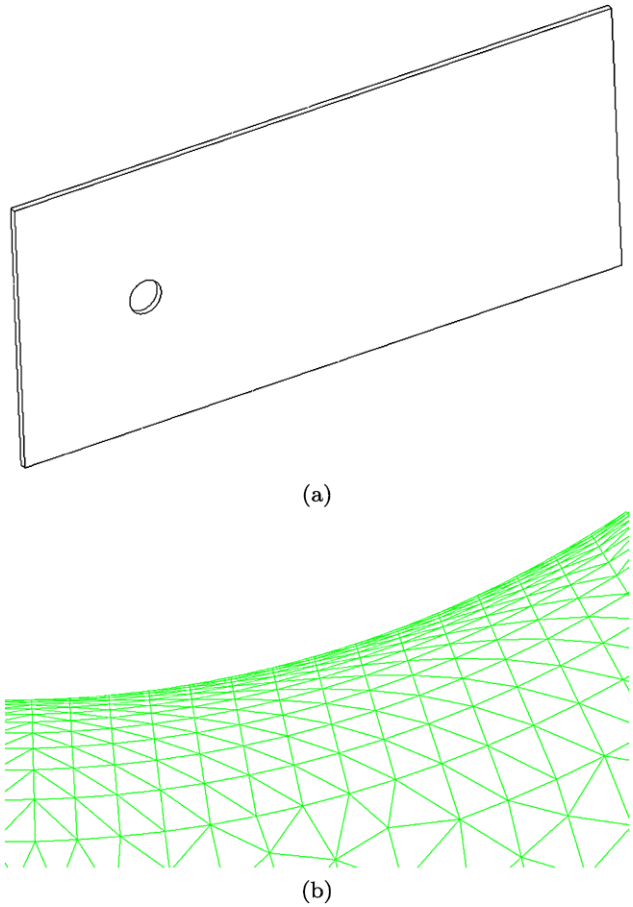


Fig. 35 (**a**) Geometry of cylinder example, domain dimension $19.0 \times 8.0 \times 0.2$ and $R_c = 0.5$. (**b**) Detail of the mesh used for the cylinder example

Of course many different alternative coupling schemes exist, see for example [40, 77]. Many of them can be implemented without making changes to the single field solvers. An example of fluid-structure interaction is given in Fig. 38.

11.3 Particle Finite Element Method

The Particle Finite Element Method (PFEM) [37–39, 64, 65] is a method for the solution of fluid problems on varying domains in time. 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 are the efficient regeneration of the mesh and the optimized recalculation of all element 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. A part of the Python script is given in Table 5.

This example shows how a previously implemented fluid solver can be reused when implementing a new algorithm.

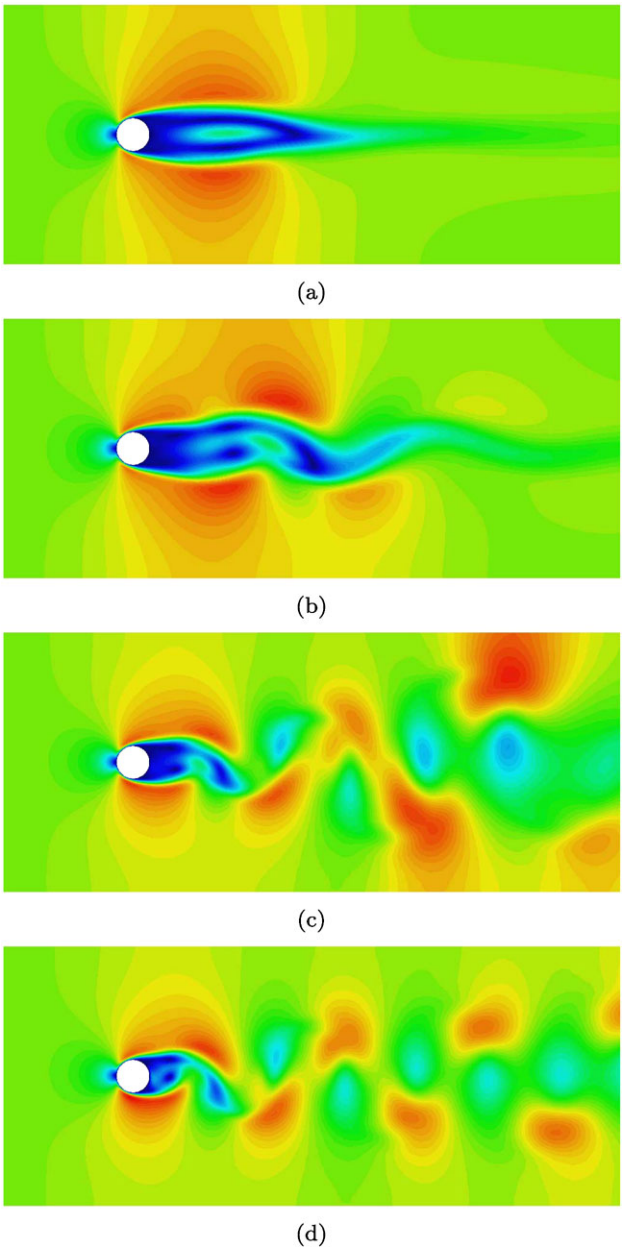


Fig. 36 Velocity at different time steps

This reusability allows the fast development of new formulations which can be tested by solving large scale real-life problems. Examples of simulations done with the PFEM implemented in Kratos can be found in [46].

Figure 39 shows an object (an unmanned vehicle) which floats on the surface of the water after landing. This is a good example of application of the PFEM for dealing with a Fluid-Structure Interaction problem.

Two validation examples showing the impact of wedges with different shapes on the surface of the water are showed in Fig. 40 on the top of the experimental results. These last two examples were run in Kratos using a different CFD solver as described in [78].

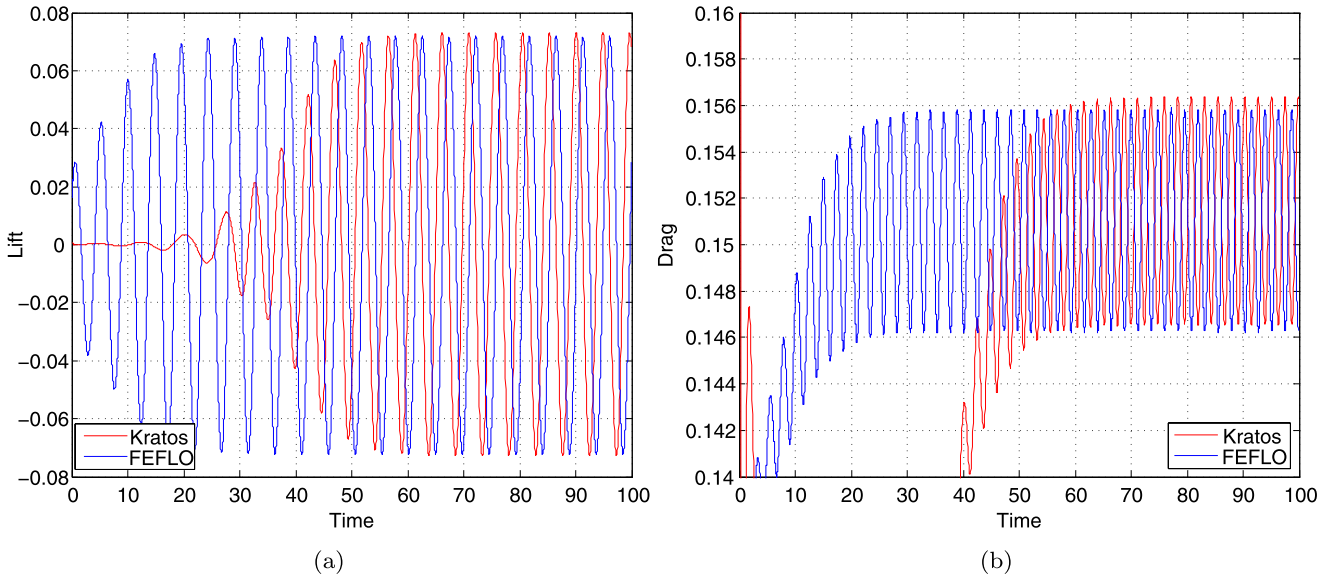


Fig. 37 (Color online) The comparison of results obtained by Kratos (*red line*) using a predictor corrector scheme and FEFLO (*blue line*). (**a**) Lift calculated for the cylinder. (**b**) Drag calculated for the cylinder

Table 3 Importing different applications involved in solving the FSI coupled problem

```
#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)
```

11.4 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 [41]. 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 system input and output are known and the properties are to be estimated [41].

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 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 [36]. There are many different types of neural net-

Table 4 A very simple explicit FSI coupling procedure implemented in Python

```
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()
```

works, of which the *multilayer perceptron* is an important one [81]. Neural networks provide a direct method for the solution of general variational problems and, consequently, inverse problems [20].

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 [51] developed at CIMNE is an open source neural networks library written in C++. Flood was used to create the neural network necessary for solving this problem. While the *Flood* 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 into another project. It also demonstrates its robustness due to the fact that the Neural Network 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.

11.4.1 Methodology

The general solution of variational problems using Neural networks consists of three steps [20]:

- Definition of the functional space. The solution here is represented by a multilayer perceptron.
- Formulation of the variational problem. For this performance functional $F(y(x,a))$ must be defined. In order to evaluate the functional we solve a partial differential equation 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.

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 this methodology is applied to solve two different thermal inverse problems.

11.4.2 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 access the internal data of Kratos and change the boundary conditions assigned to the different `Nodes`. This is done without any file interface which would dramatically reduce the performance. The first task is implement the interface for the direct solution using Kratos. The following code shows the main part of this interface:

```
// Initializing Kratos kernel
Kernel kernel;
kernel.Initialize();

// Initializing Kratos thermal application
KratosThermalApplication thermal_application;
kernel.AddApplication(thermal_application);

// Read mesh
GidIO gidIO("thermal_problem");
gidIO >> mesh;

// Set properties to optimization values d,c,k
```

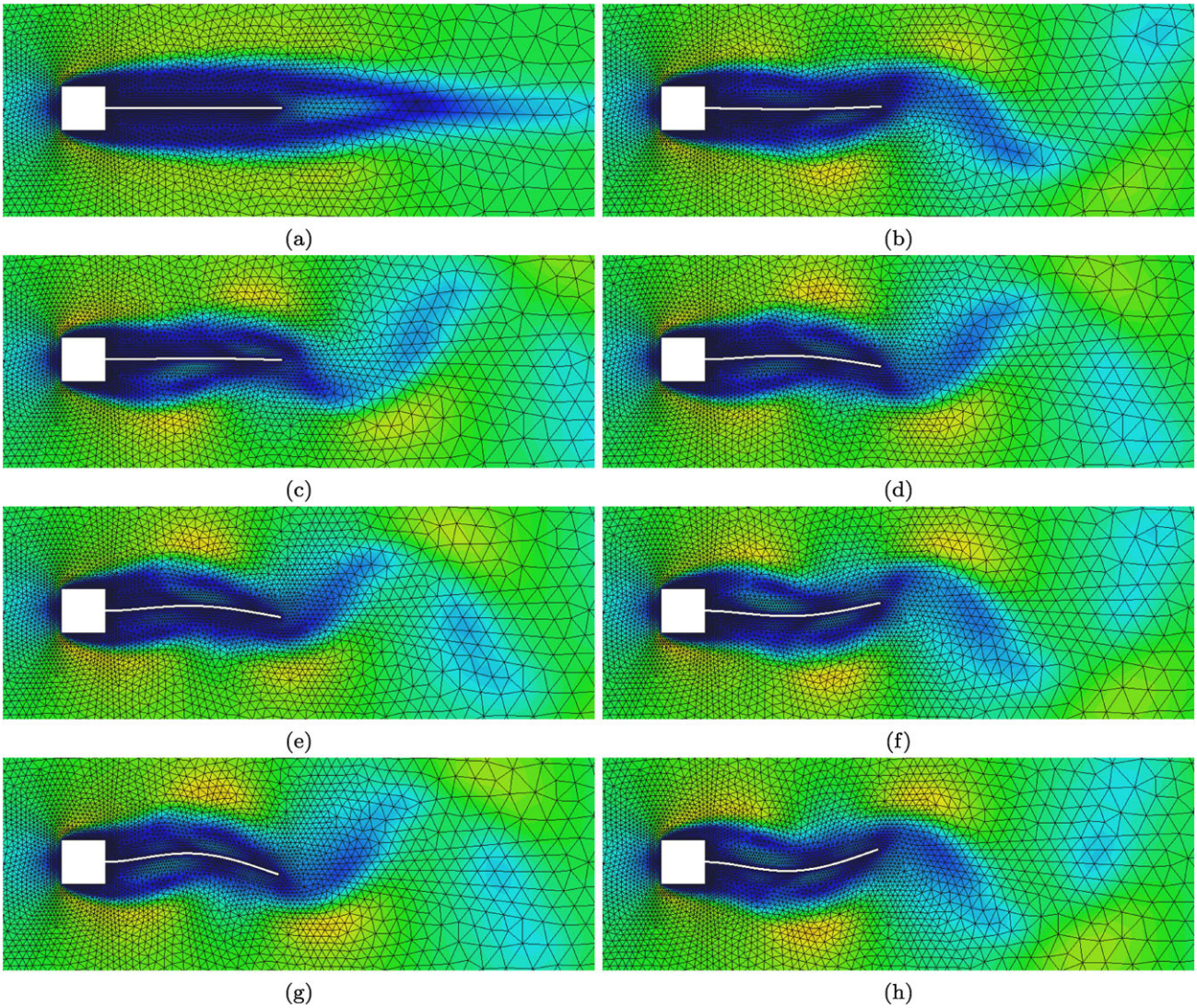


Fig. 38 Flag flatter simulation using fluid structure interaction with mesh movement

```
Properties& r_properties = mesh.GetProperties(1);
r_properties[DENSITY] = d;
r_properties[SPECIFIC_HEAT_RATIO] = c;
r_properties[THERMAL_CONDUCTIVITY] = k;

// Assign initial temperature to nodes
// with fixed temperature
MeshType::NodeIterator i_node;
for(i_node =  mesh.NodesBegin() ;
    i_node != mesh.NodesEnd() ; i_node++)
  if(!(i_node->IsFixed(TEMPERATURE)))
    i_node->SetSolutionStepValue(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)
  numerator /= (2.0*pow(sigma,2)));
  double denominator = 8*sigma*sqrt(2.0*pi);

  boundaryTemperature[i] = numerator/denominator;

  // Assign boundary temperature
  for(i_node = mesh.NodesBegin() ;
      i_node != mesh.NodesEnd() ; i_node++)
    if(i_node->IsFixed(TEMPERATURE))
      i_node->SetSolutionStepValue(TEMPERATURE,
                      boundaryTemperature[i]);

  // Solving using thermal solver
  Solve();

  // Now updating the nodal temperature values
  // by result of solved equation system.
  Update();
```

Table 5 Implementation in Kratos of particle finite element method (PFEM)

```
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()
```

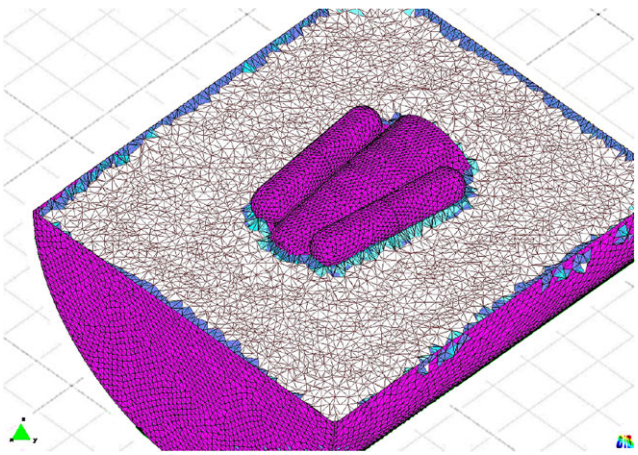


Fig. 39 An unmanned vehicle floats on the surface of the water after landing

```
  // Obtain center node temperature
  nodeTemperature[i] =
   center_node.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 the FEM problem using different boundary conditions assigning fixed values of temperature to `Nodes`. Finally it retrieves 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 integration algorithms.

The inverse problem also involves 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 any problem.

The tool obtained by combining these two application has been successfully used for solving *boundary temperature estimation problems* and *diffusion coefficient estimation problems* [20, 52].

Some results can be seen in Fig. 41 which shows the agreement between the estimated diffusion coefficient and the diffusion coefficient estimated by the neural network for a square domain problem.

The results of a different inverse problem are shown in Fig. 42. The graphs shown distribution a comparison of the actual boundary temperature and the one estimated by the neural network for a square domain.

Full details of the examples, as well as a complete description of the theory used can be found in [52].

Further applications and other class of problems implemented in Kratos can be found in [2, 61, 74–76, 79].

12 Conclusions and Future Work

12.1 Conclusions

Kratos, a framework for developing multi-disciplinary FEM programs has been designed and implemented. This framework provides a high level of flexibility and generality which is required for dealing with multi-disciplinary problems. Developers in different areas can adapt Kratos for their needs without altering the standard interface used to communicate with other fields in coupled analysis. The applications already implemented in the Kratos framework can be used for solving multi-disciplinary problems using any master and slave strategies or even by solving simultaneously. Finally a 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

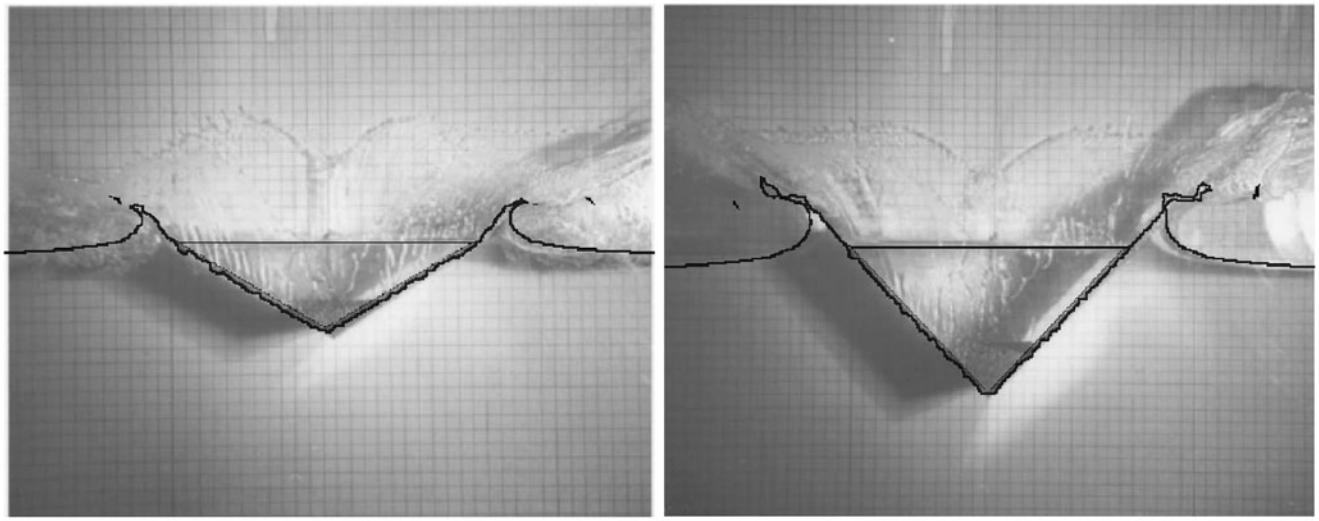


Fig. 40 Impact of wedges with different shapes on the surface of the water

the application development not only faster, but also ensures compatibility with other tools for solving multi-disciplinary problems.

Kratos is also extensible at different levels of implementation. Each application can add its variables, degrees of freedom, `Properties`, `Elements`, `Conditions`, 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.

Last but not least, the performance of Kratos is comparable even to single purpose programs and different benchmarks has shown this in practice. This makes Kratos a practical tool for solving industrial multi-disciplinary problems.

All these objectives were achieved not only by a number of innovative development, but also by collecting and reusing several existing works. Some of these aspects are as follow:

Kratos has an object-oriented and multi-layer structure 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. The implementation difficulties needed for each layer are also tuned for the knowledge of users.

A new variable base interface has been designed and implemented. This interface is used at different levels of abstraction and has proven to be clear, flexible, and extensible.

New heterogeneous containers have been implemented in order to hold different types of data without any modifications. The `DataValueContainer` is designed to be very flexible while the `VariablesListContainer` is designed to be fast but less robust. 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 `Nodes` or `Elements` 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 `Nodes` and `Elements`. Several levels of abstraction like `Mesh`, `MoldePart` and `Model` are provided to help users group model and data information in different ways. These objects are effectively used for separating domains information in multi-disciplinary problems or sending a single part to a process.

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.

A flexible and extensible IO module for finite element programs has been developed. Any application built with

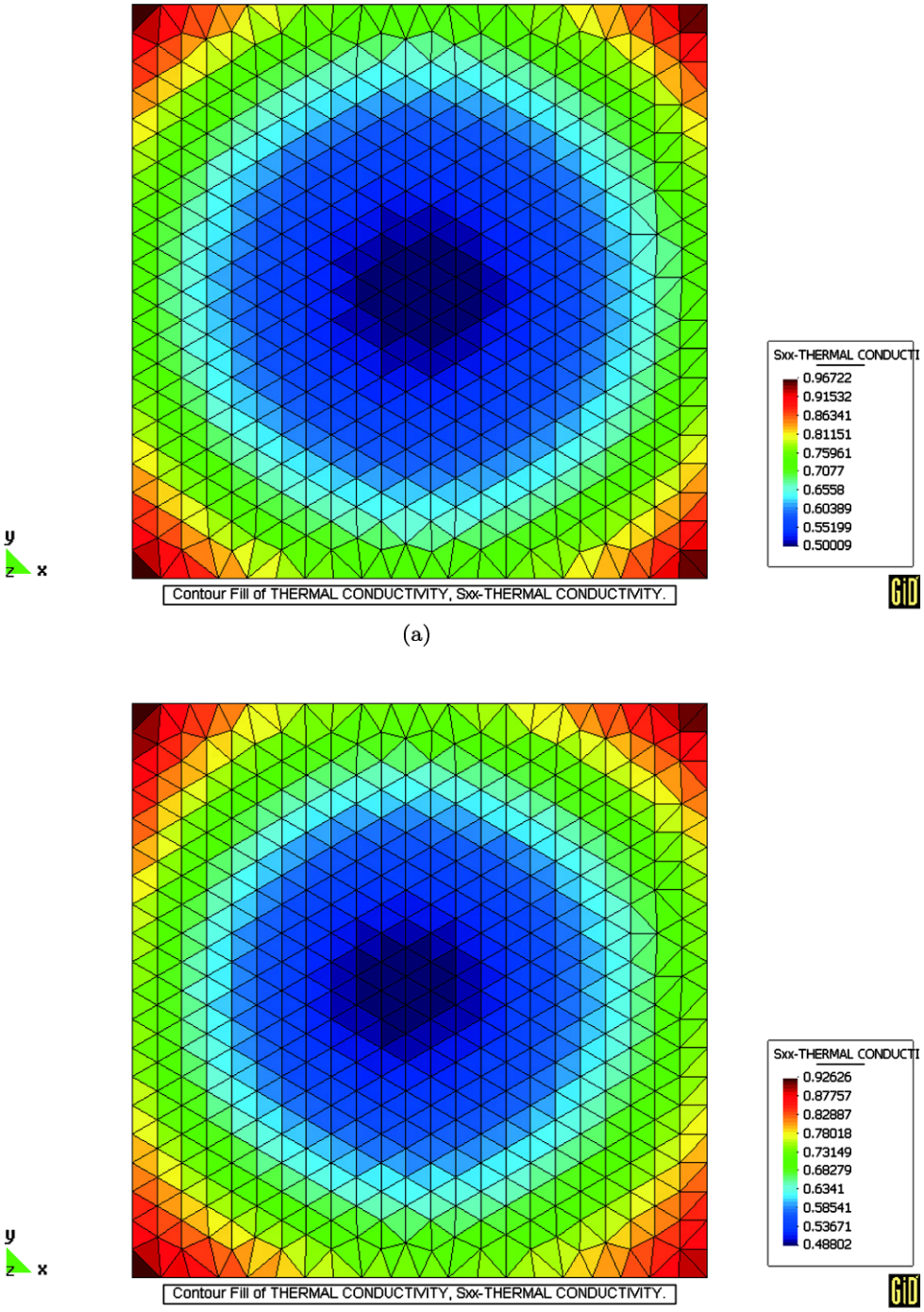


Fig. 41 Actual diffusion coefficient (**a**) and estimated diffusion coefficient (**b**) for the diffusion coefficient estimation problem

Kratos can use IO for reading and writing its own concepts without making any change to it. An interpreter is also implemented to handle Kratos data files but 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.

12.2 Future Work

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. Fortunately, several aspects of Kratos become useful in this process.

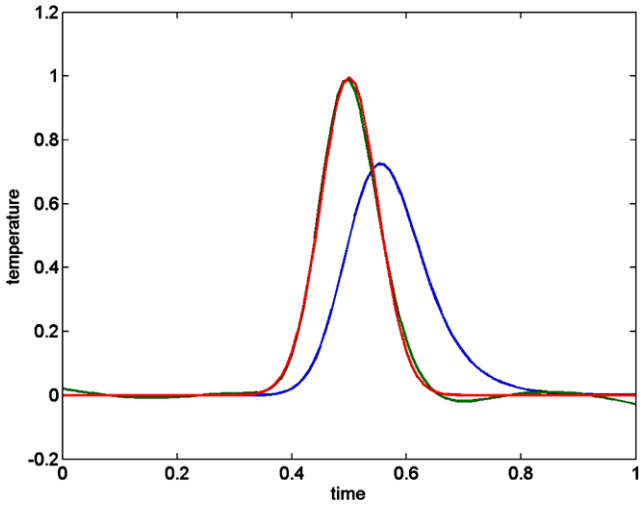


Fig. 42 (Color online) Actual boundary temperature (*red*), estimated boundary temperature (*green*) and measured temperature at the center of the square (*blue*) for the boundary temperature estimation problem

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. Also new solvers and preconditioners should be added to extend the solving abilities of Kratos.

Implementing missing components for elemental expression 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.

Serialization for automatization of problem loading and saving and passing data over network is another line of extensions. Supporting binary format for input can reduce significantly the data reading time.

Acknowledgements We would like to thank the whole Kratos team for their contributions and valuable feedback that was decisive for the evolution of Kratos. In particular thanks to Pavel Ryzhakov and Dr. Roberto Lopez for providing some of the examples. This work was partially supported by XPRES project of the Spanish government.

References

1. Boost serialization library. http://www.boost.org/libs/serialization/doc/index.html
2. Papadrakakis M, Oñate E, Schrefler B (eds) (2007) Updated Lagrangian formulation of a quasi-incompressible fluid element, Ibiza, Spain, Proceedings. CIMNE, Barcelona
3. ABAQUS Inc. ABAQUS Scripting Reference Manual
4. ABAQUS Inc. ABAQUS Scripting User's Manual
5. Aho AV, Sethi R, Ullman JD (1986) Compilers principles, techniques and tools. Addison-Wesley, Reading
6. Aho AV, Ullman JD (1978) Principles of compiler design. Addison-Wesley, Reading
7. Aho AV, Ullman JD, Hopcroft JE (1983) Data structures and algorithms. Addison-Wesley, Reading
8. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Croz JD, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1995) LAPACK users' guide, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia
9. Appel AW, Ginsburg M (1999) Modern compiler implementation in C. Cambridge University Press, Cambridge
10. Archer GC (1996) Object-oriented finite element analysis. PhD thesis, University of California at Berkeley
11. Archer GC, Fenves G, Thewalt C (1999) A new object-oriented finite element analysis program architecture. Comput Struct 70(1):63–75
12. Bangerth W (2000) Using modern features of C++ for adaptive finite element methods: Dimension-independent programming in deal.II. In: Deville M, Owens R (eds) Proceedings of the 16th IMACS World congress 2000, Lausanne, Switzerland, Document Sessions/118-1
13. Bangerth W, Hartmann R, Kanschat G. `deal.II` Differential equations analysis library, Technical reference. http://www.dealii.org
14. Bangerth W, Hartmann R, Kanschat G (2006) deal.II—a general purpose object oriented finite element library. Technical Report ISC-06-02-MATH, Institute for Scientific Computation, Texas A&M University
15. Bangerth W, Kanschat G (1999) Concepts for object-oriented finite element software—the `deal.II` library. Preprint 99-43 (SFB 359), IWR Heidelberg, Oct. 1999
16. Bathe K-J (1996) Finite element procedures. Prentice-Hall, New York
17. Cardona A, Klapka I, Geradin M (1994) Design of a new finite element programming environment. Eng Comput 11(4):365–381
18. Codina R (2001) Pressure stability in fractional step finite element methods for incompressible flows. J Comput Phys 170:112140
19. Codina R (2002) Stabilized finite element approximation of transient incompressible flows using orthogonal subscales. Comput Methods Appl Mech Eng 191(39):4295–4321
20. Dadvand P, Lopez R, Oñate E (2006) Artificial neural networks for the solution of inverse problems. In: ERCOFTAC
21. Dadvand P, Mora J, González C, Arraez A, Ubach P, Oñate E (2002) Kratos: An object-oriented environment for development of multi-physics analysis software. In: Proceedings of the WCCM V fifth world congress on computational mechanics, July 2002
22. Donéa J, Huerta A (2003) Finite element methods for flow problems. Wiley, New York
23. Dubois-Pélerin Y, Pegon P (1997) Improving modularity in object-oriented finite element programming. Commun Numer Methods Eng 13:193–198
24. Dubois-Pèlerin Y, Zimmermann T (1993) Object-oriented finite element programming: Iii. an efficient implementation in C++. Comput Methods Appl Mech Eng 108(1–2):165–183
25. Dubois-Pèlerin Y, Zimmermann T, Bomme P (1992) Object-oriented finite element in programming: Ii. a prototype program in smalltalk. Comput Methods Appl Mech Eng 98(3):361–397
26. Edelson DR (1992) Smart pointers: They're smart, but they're not pointers. Technical report, Santa Cruz, CA, USA
27. Eyheramendy D, Zimmermann T (1996) 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

28. Eyheramendy D, Zimmermann T (1996) Object-oriented finite elements ii. a symbolic environment for automatic programming. Comput Methods Appl Mech Eng 132(3):277–304(28)
29. Eyheramendy D, Zimmermann T (1998) Object-oriented finite elements iii. theory and application of automatic programming. Comput Methods Appl Mech Eng 154(1):41–68(28)
30. Felippa CA, Geers TL (1988) Partitioned analysis of coupled mechanical systems. Eng Comput 5:123–133
31. Felippa CA, Park KC, Farhat C (2001) Partitioned analysis of coupled mechanical systems. Comput Methods Appl Mech Eng 190:3247–3270
32. Filho JRA, Devloo PRB (1991) Object oriented programming in scientific computations: the beginning of a new era. Eng Comput 8(1):81–87
33. Forde BWR, Foschi RO, Stiemer SF (1990) Object-oriented finite element analysis. Comput Struct 34(3):355–374
34. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
35. Gonnet GH, Baeza-Yates R (1991) Handbook of algorithms and data structures: in Pascal and C, 2nd edn. Addison-Wesley, Reading
36. Haykin S (1994) Neural networks: A comprehensive fundation. Prentice Hall, New York
37. Idelshon SR, Oñate E (2006) To mesh or not to mesh. That is the question. Comput Methods Appl Mech Eng 195:4681–4696
38. Idelsohn SR, Oñate E, Calvo N, Pin FD (2003) The meshless finite element method. Int J Numer Methods Eng 58:893–912
39. Idelsohn SR, Oñate E, Pin FD (2004) The particle finite element method: a powerful tool to solve incompressible flows with free-surfaces and breaking waves. Int J Numer Methods Eng 61:964–989
40. Idelsohn SR, Pin FD, Rossi R, Oñate E (2009) Fluid-structure interaction problems with strong added-mass effect. Int J Numer Methods Eng 80:1261–1294. doi:10.1002/nme.2659
41. Kirsch A (1996) An introduction to the mathematical theory of inverse problems. Springer, Berlin
42. Klapka I, Cardona A, Geradin M (1998) An object oriented implementation of the finite element method for coupled problems. Rev Eur Elements Finis 7(5):469–504
43. Klapka I, Cardona A, Geradin M (2000) Interpreter oofelie for pdes. In: European congress on computational methods in applied sciences and engineering (ECCOMAS 2000), Barcelona
44. Knuth DE (1998) The art of computer programming: sorting and searching, vol 3, 2nd edn. Addison-Wesley, Reading
45. Laboratory for Computational Physics and Fluid Dynamics (LCP&FD). FEFLO project
46. Larese A, Rossi R, Oñate E, Idelsohn SR (2008) Validation of the particle finite element method (pfem) for simulation of free surface flows. Eng Comput 25:385–425
47. Lindemann J, Dahlblom O, Sandberg G (2006) Using Corba middleware in finite element software. Future Gener Comput Syst 22(1–2):158–193
48. Logg A (2007) Automating the finite element method. Arch Comput Methods Eng 14(2):93–138
49. Lu J, White D, Chen W-F (1993) Applying object-oriented design to finite element programming. In: SAC '93: proceedings of the 1993 ACM/SIGAPP symposium on applied computing. ACM, New York, pp 424–429
50. Lu J, White DW, Chen W-F, Dunsmore HE (1995) A matrix class library in C++ for structural engineering computing. Comput Struct 55(1):95–111
51. López R. Flood: an open source neural networks C++ library. CIMNE
52. López R, Balsa-Canto E, Oñate E (2008) Neural networks for variational problems in engineering. Int J Numer Methods Eng 75(11):1341–1360
53. Mackie RI (1992) Object oriented programming of the finite element method. Int J Numer Methods Eng 35(2):425–436
54. Mackie RI (1997) Using objects to handle complexity in finite element software. Eng Comput 13(2):99–111
55. Maplesoft. Maple's documentation
56. MathWorks. Matlab's documentation
57. Menetrey P, Zimmermann T (1993) Object-oriented non-linear finite element analysis: application to j2 plasticity. Comput Struct 49(5):767–773
58. Miller G (1991) An object-oriented approach to structural analysis and design. Comput Struct 40(1):75–82
59. Miller G (1994) Coordinate-free isoparametric elements. Comput Struct 49(6):1027–1035
60. Miller GR, Banerjee S, Sribalaskandarajah K (1995) A framework for interactive computational analysis in geomechanics. Comput Geotech 17(1):17–37
61. Mora J, Otín R, Dadvand P, Escolano E, Pasenau MA, Oñate E (2006) Open tools for electromagnetic simulation programs. COMPEL 25(3):551–564
62. Mount D, Arya S (1997) Ann: A library for approximate nearest neighbor searching. citeseer.ist.psu.edu/mount97ann.html
63. Oñate E (2004) Possibilities of finite calculus in computational mechanics. Int J Numer Methods Eng 60(1):255–281
64. Oñate E, Idelsohn S, Celigueta M, Rossi R (2008) Advances in the particle finite element method for the analysis of fluid-multibody interaction and bed erosion in free surface flows. Comput Methods Appl Mech Eng 197:1777–1800
65. Oñate E, Idelsohn S, Pin FD, Aubry R (2004) The particle finite element method. An overview. Int J Comput Methods 1(2):267–307
66. Open Engineering. OOFELIE
67. Patzák B. OOFEM documentation. Czech Technical University, Faculty of Civil Engineering, Department of Structural Mechanics
68. Patzák B, Bittnar Z (1999) Object oriented finite element modeling. Acta Polytech 39(2):99–113
69. Pidaparti RMV, Hudli AV (1993) Dynamic analysis of structures using object-oriented techniques. Comput Struct 49(1):149–156
70. Press WH, Vetterling WT, Teukolsky SA, Flannery BP (2002) Numerical recipes in C++: the art of scientific computing. Cambridge University Press, Cambridge
71. Raphael B, Krishnamoorthy CS (1993) Automating finite element development using object oriented techniques. Eng Comput 10(3):267–278
72. Ribó R, Pasenau M, Escolano E, Ronda JSP. GiD user manual. CIMNE, Barcelona
73. Ribó R, Pasenau M, Escolano E, Ronda JSP, González LF. GiD reference manual. CIMNE, Barcelona
74. Rossi R (2005) Light weight structures: structural analysis and coupling issues. PhD thesis, University of Bologna
75. Rossi R, Idelsohn SR, Oñate E (2006) On the possibilities and validation of the particle finite element method (pfem) for complex engineering fluid flow problems. In: Proceedings of ECCOMAS CFD 2006, Egmond aan Zee, The Netherlands
76. Rossi R, Oñate E (2010) Validation of a fsi simulation procedure—bridge aerodynamics modelproblem. Eng Comput (to appear)
77. Rossi R, Oñate E (2010) Analysis of some partitioned algorithms for fluid-structure interaction. Eng Comput 27:20–56. doi:10.1108/02644401011008513
78. Rossi R, Ryzhakov P, Oñate E (2009) A monolithic fe formulation for the analysis of membranes in fluids. Int J Space Struct 24(4):205–210
79. Rossi R, Vitaliani R (2004) Numerical coupled analysis of flexible structures subjected to the fluid action. In: 5th PhD symposium in civil engineering, Delft

80. Saad Y (2003) Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics, Philadelphia
81. Šíma J, Orponen P (2003) General-purpose computation with neural networks: A survey of complexity theoretic results. Neural Comput 15(12):2727–2778
82. Touzani R. OFELI documentation
83. Touzani R (2002) An object oriented finite element toolkit. In: Proceedings of the fifth world congress on computational mechanics (WCCM V)
84. Veldhuizen TL (1995) Expression templates. C++ Rep 7(5):26–31
85. Veldhuizen TL (1998) Arrays in blitz++. In: Proceedings of the 2nd international scientific computing in object-oriented parallel environments (ISCOPE'98). Lecture notes in computer science. Springer, Berlin
86. Veldhuizen TL, Jernigan ME (1997) Will C++ be faster than Fortran? In: Proceedings of the 1st international scientific computing in object-oriented parallel environments (ISCOPE'97). Springer, Berlin, Heidelberg, New York, Tokyo
87. Vinoski S (1997) CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Commun Mag, 14(2)
88. Wolfram Research. Mathematica's documentation
89. Zienkiewicz OC, Taylor RL, Zhu JZ (2007) Finite element method: its basis and fundamentals. Butterworth-Heinemann, Stoneham
90. Zimmermann T, Dubois-Pèlerin Y, Bomme P (1992) Object-oriented finite element programming: I: Governing principles. Comput Methods Appl Mech Eng 98(2):291–303
91. Zimmermann T, Eyheramendy D (1996) Object-oriented finite elements i. principles of symbolic derivations and automatic programming. Comput Methods Appl Mech Eng 132(3):259–276 (18)