An Object-Oriented Implementation of the Finite Element Method for Coupled problems

Igor Klapka * — Alberto Cardona ** — Michel Géradin *

* *Laboratoire des Techniques Aéronautiques et Spatiales*
Dynamique des Structures, Université de Liège
rue Ernest Solvay 21, B-4000, Liège, Belgium

** *Computational Mechanics Laboratory, INTEC*
Universidad Nacional del Litoral, Conicet
Guemes 3450, 3000 Santa Fe, Argentina

ABSTRACT. *Recent advances in computational hardware allow us to consider solving complex phenomena (e.g., coupled problems, large non linearities, optimization, etc.). Complexity of problems to be analyzed is constantly increasing due to current industrial demands that pose severe constraints to software developers. Object-oriented programming has emerged as a solution to manage complexity. In this paper, we describe aspects in the development of a finite element program written in C++. Particular aspects of the formulation, as the concepts of partition and tools for the decomposition of the problem into several sub-parts, are introduced. Examples illustrating applications to the solution of piezoelectric motor modeling and of shape optimization are shown.*

RÉSUMÉ. *L'avance technologique et scientifique permet de considérer la résolution de problèmes de plus en plus complexes (p.e., problèmes couplés, problèmes non linéaires, optimisation, etc.). Poussée par la demande industrielle, la complexité des problèmes à analyser est en constante croissance, ce qui impose de sérieuses contraintes aux développeurs. La philosophie de programmation orientée objet propose de gérer cette complexité. Dans cet exposé nous décrivons des aspects du développement d'un code éléments finis écrit en C++. Certains aspects particuliers de la formulation tels que les concepts de partition et d'outils de décomposition en sous-domaines sont développés. Leur utilité est illustrée à l'aide d'exemples de solution d'un modèle de moteur piézo-électrique et d'une optimisation de forme.*

KEY WORDS: *coupled problems, piezoelectricity, object-oriented programming, C++, command interpreter.*

MOTS-CLÉS : *problèmes couplés, piézoélectricité, programmation orientée-objet, C++, interpréteur de commande*

1. Introduction

The finite element method is currently used as an extremely valuable tool of analysis in many fields of engineering and science. Since 1960, several research teams have developed analysis softwares based on this method, software which has continuously grown over the years leading to powerful packages that are in use today in industry and research. However, computers and software technology have changed a lot since the days when these programs were conceived, and propose new techniques to manage information. Also, new faster computers make way for more complex problems, requiring in turn modern information handling techniques. As a consequence, developers are evidencing an increasing difficulty in keeping their software up to date, with very high costs of maintenance and development. Yet, they are facing a growing demand for user-friendliness in the form of graphics, menus, dialog boxes, etc.

New software technology tries to remedy to deficiencies encountered with conventional programming. In particular, *object-oriented programming* (O.O.P.) is currently seen as the most promising way of devising a new application. The objective of this work is to define and evaluate an architecture for the development of finite element methods using the C++ programming language. A new package named OoFELiE(for ***Object Oriented Finite Element method Led by Interactive Executor)*** has been created. Object-oriented programming techniques have been followed, trying to give the program a maximum of flexibility for accommodating future changes.

In a first paper[CKG94] , emphasis was put on the computation engine. In this paper, we present results of our investigation into the design of finite element classes taking physical and mathematical aspects into account. A method to manage degrees of freedom, information and results accounting for partitioning in sub-domains is proposed. In the second part, the description of a very powerful and easily extensible interface to create and use objects is presented. Two examples are presented to demonstrate benefits of this integration.

2. Software Design Considerations

This research is the result of several observations. The first one concerns the growing needs of complex modelization of interaction between different physical fields. The second one concerns the evolution of new computer technology which invites developers to reconsider the design of their current tools of analysis, for example, by using new languages features or by proposing adapted algorithms to distribute tasks between computers connected in parallel. The last two decades have been marked by the growing computer power, allowing us to consider modelization of more and more complex phenomena. Both from the point of view of model size and of the interaction between different models, management of information is more complicated and has become crucial. We emphasize that achieving a correct implementation is, at least, as important as developing a sound basis for the formulation of the method.

Several people are involved in the development of large finite element software. They can be grouped as follows, from the point of view of their interests and skills:

I. final users of the code, who are interested into getting an answer to physical problem in hand,
II. numerical analysts, who propose new finite element methods and algorithms,
III. programmers, seen as users of classes and methods implemented by other people,
IV. programmers of new classes and methods, to be used by people of the previous group.

We should remark, however, that the same person may belong to more than one group. In fact, what makes the decision is the *task* being performed at a given moment. All of them should be taken into consideration, from the very beginning of the design of a new finite element program.

The main concerns about software for each group can be described by the following table:

	Tasks			
Concerned by	I	II	III	IV
Efficiency and performance	++	+++	++	+
User-friendliness	+++	+	+++	0
Extensibility	+	++	+	++
Maintainability	0	0	0	+++

Since the code will be mainly used as a research tool in an academic environment, we have decided to give priority to *extensibility* and *efficiency* features as main criteria for code design. Whenever these two features collide, the numerical efficiency aspect has been retained. In second *lieu*, the maintainability and portability issues have been considered.

3. Object-Oriented Programming

3.1. A New Philosophy

The object-oriented philosophy comes from the idea that tools (methods) must be associated with the information (data) they manage. In order to do that, some new concepts have been introduced:

Class. A class defines an abstract data type. It may represent for example a family of objects in the real world. Classes are identified during the analysis phase of application development. A class incorporates the definition of the structure as well as the operations on the abstract data type. For example, `Element` would be defined as a class in a finite element method application. Typically, data which the system analyst would define relating to the class `Element` would include such items as localization, connected nodes, number of degrees of freedom, etc. Elements belonging to the collection of objects described by the class are called *instances* of the class.

Subclasses and Inheritance. A subclass is subordinated to a class and has all of the same elements of the class from which it is *inherited.* Therefore, the term *inheritance* indicates an '*is a* ...' relationship between two classes. A subclass is usually of a special type and has additional data elements relating to it. For example, an `Isoparametric_Element` *is an* `Element` in which both positions and displacements are interpolated from nodal values.

Encapsulation. In an object-oriented application by which all data as well as functions and services offered by a class(or subclass) are *packaged* together. This (independent) self-containment of classes and their functions and services (i.e. members) is called *encapsulation.* This full modularity both greatly enhances introducing changes during the development phase as well as performing ongoing maintenance. In addition, as we will see below, encapsulation makes it possible for the unique *polymorphic* capability of object-oriented applications.

Polymorphism. As previously mentioned, functions and services required by a class are packaged (encapsulated) together. This modularity enables the same function or service to respond differently when performed on objects from different classes. For example, a single request to `fill_stiffness` will automatically be computed differently for a `bar` than for a `quad4` element. This is due to the fact that different calculation methods are encapsulated within subclasses `bar` and `quad4`. This unique capability of object-oriented applications to interpret the same request differently depending on the object being processed is known as *polymorphism.*

The two latter concepts may be used to get a complete *data abstraction,* implying for example that we can ask the elementary stiffness to a list of `Element` objects without knowing exactly what element types form each item in the list.

3.2. The Methodology

An essential factor of success in a large project is the employment of good analysis tools. As the complexity of systems increase, the analysis becomes even more important. Object-oriented philosophy has brought new techniques in modeling and analysis like OOD [CY91, COAD91] , BOOCH[Boo94] , OMT[RUM91] and OOSE (Object-Oriented Systems Engineering - Oden University). They all propose analysis methodologies to help in the design of the architecture of an object-oriented project. Today, there is a tendency to use UML (Unified Modeling Language) which fuses the concepts of previously mentioned ones into a standardized modeling language.

3.3. The Language

Discussions on the different languages which currently propose object-oriented philosophy (OOPL) can be found in books like [RUM91, CH. 15 & 16] , [CY91, CH 7] , or [BN94] for a point of view on scientific applications. Amongst the many existing object-oriented languages, we have selected C++ for portability and efficiency reasons, and also for the large number of tools available (graphic interfaces library, network tools, etc.). We remark also that C++ revealed itself to be a language which the programmer imposes to such discipline that it allows one person to maintain more than 30,000 single lines of code [STR91] .

Several researchers have worked in the application of object-oriented programming techniques to the finite element method in recent years [DF92] [ZH94] . T. Zimmermann et al. used the SMALLTALK programming language [DPBZ90] [ZDP91] . Although it is considered as a pure object-oriented language, it is not used for real numerical applications because of efficiency reasons [ZDP92] . Other researchers have presented object-oriented implementations of the finite element method using object-oriented extensions of PASCAL [FFS90] [MAC92] or ADA. However, there exists a generalized concern that the C++ language –an extension to C developed by B. Stroustrup [STR91] – is the most widespread object-oriented language for numerical computations. Applications of finite element programming using C++ have been reported in references [FD91] [MIL91] [ZH94] . We should finally mention that the new standard FORTRAN 90 [MR90] includes also features which can be considered as object-oriented extensions, making this option appealing for who are strongly involved in FORTRAN programming. However, it has many limitations since, for instance, it does not support inheritance or dynamic polymorphism.

3.4. Numerical efficiency

From a numerical efficiency point of view, it is important to understand that in C++ most of the object-oriented concepts do reach the compilation step. Only the polymorphism concept needs dynamic (during execution) binding. Historically, first versions of C++ compilers were simply translators to C, theoretically limiting efficiency to that of C. In fact, as announced by the creator of the language, C++ is designed to be a an improved C, supporting data abstraction and object-oriented philosophy. This means more concepts to be understood by the programmer and thus, more sources of mistakes. We should bear in mind that object-oriented philosophy must be considered as a tool, and not a systematic obligation which could lad to make of each *bit* an object.

We think that most of the criticisms of C++ efficiency are not valid. Practice has taught us, as also mentioned by P.R.B. Devloo [DEV94] , that inefficient implementation of C++ programs is the consequence of a bad understanding of the language. For example, when implementing a vector class you may be tempted to overload `operator[i]` so as to test if the index `i` is inside the vector range. It may be a useful feature in some cases, but you should not expect to get BLAS performance if you use this operator in a vector product. We must keep in mind that abstraction often comes with a loss of performance. Stepanov calls it *the abstraction penalty* and proposed a test to quantify it [ROB96] .

For this reason, we chose to limit the *non-anticipation* principle proposed by Dubois-Pelerin [DPBZ90] to objects accessed through the command-line interpreter and to high level methods. This principle tries to expand *data abstraction* to a *state abstrac-*

tion concept, which implies verifying the state of each data entity before using it. This idea may result in the implementation of many costly tests which can be considered as not acceptable at execution time if used on often called functions or operators (`operator[]` for example). Our choice seems to contradict the object-oriented philosophy. However, we have verified this by using techniques of analysis as *state diagrams* (OMT)[RUM91] , we can gain a good knowledge about the state of the object before using it, which makes some tests unnecessary.

Also, the user of classes has an unavoidable part of responsibility. For instance, many algorithms do not need zero initialization of matrices. This is why most matrix class implementations do not initialize matrix values by default, leaving the choice to the user (e.g., LAPACK++).

In terms of pure computational efficiency, J.J.Dongarra and R.Pozo [DLN+94] obtained with the LAPACK++ library (a C++ extension of the Fortran LAPACK library for numerical linear algebra) exactly the same performance as with FORTRAN. However, the C++ implementation provided scalability, portability, flexibility and easy-to-use, features which were not so easy to obtain for large codes implemented in FORTRAN. Other research carried out by the BLITZ++ consortium [VEL95] gave in some cases, better performance than FORTRAN.

To get these results, R. Pozo and T. Veldhuizen made frequent use of the *templates* mechanism which allows to define concepts (methods, algorithms) independently of the type of object being used. The main difference with the virtual inheritance mechanism consists in the fact that the connection between data and methods is made by the compiler; then it receives enough information to produce an optimized code. This technique does not only increases legibility without losing efficiency, but also allows to methods to be reused without considering how data are implemented. The templates concept is preferred to the class inheritance mechanism in order to implement static polymorphism.

4. Numerical Solution of a Continuum Mechanics Problem

The process of solving a continuum mechanics problem customarily involves a step of discretization followed by a step of searching the solution to a discrete algebraic problem.

Discretization allows the behavior of continuum (e.g., space, time) to be presented in terms of the behavior computed at a given number of points (we then talk of values computed at the *degrees of freedom* of the model). The original problem is transformed into a discrete algebraic one which can be effectively put and solved numerically. The components of this algebraic problem express values computed at the degrees of freedom (*dof*) of our model. Several kinds of algebraic problems have to be solved, depending on the problem to analyze. For instance, the continuum mechanics problem in linear statics is expressed by a boundary value problem which is transformed after discretization into a system of linear algebraic equations. When dealing with linear vibration analysis, we have to solve a linear eigenvalue problem. In non-linear statics the problem to be solved is a system of non-linear algebraic equations.

These algebraic problems share several characteristics unique to finite element analysis. First to be mentioned, the number of equations and unknowns can be very large (e.g., it is not uncommon to talk of more than 100,000 unknowns in linear statics industrial applications). Also, the degree of coupling between unknowns is from moderate to low, so that appropriate schemes to handle sparse coefficient matrices should be employed to reach maximum computational efficiency. Several strategies, specialized to finite element applications, exist: e.g., the skyline and frontal solution schemes. Systems of non-linear equations range from mildly non-linear, for which it is fairly easy to find a solution, to very badly conditioned ones. There exists a large number of algorithms to solve non-linear systems of equations in finite element applications, each one being adapted to a particular kind of problem. In order to get a solution to a practical industrial problem, the analyst should have a variety of algorithms at his disposal, and should be able to switch from one to another even while advancing along the solution path.

What has already been said for systems of non-linear equations in the preceding paragraph applies also to the methods for extracting linear eigenvalues, integrating systems of ordinary differential equations and so forth. Many specialized algorithms exist and new ones are continuously being proposed which take into account the peculiarities of these systems and of new hardware technology. Finally we must mention that in many cases the problem to be solved consists of a sequence (or even nesting) of algebraic problems of the kind described in the previous paragraphs. Examples of these situations can be found, for instance, in optimization, stability analysis of non-linear structures and dynamic stability analysis of mechanisms and structures.

5. Basic Classes

The finite element method is an analysis tool for many different problems of continuum mechanics, most widely used for structural analysis. The method essentially consists into discretizing the continuum and transforming the system of partial differential equations into an algebraic problem.

In OOFELIE, we distinguish between two large basic class families: *mathematical* and *physical* classes. The foremost family is formed by classes specific for treating algebraic problems, e.g., vector and matrix manipulation. The latter family is a set of classes used for the description of the physical problem under study. Section 5.2 describes the physical classes family.

Besides the above-mentioned class families, there is a third group whose function is to establish the relationship between the two first class families. Within them, we can mention the classes `Dof, Dofset, Partition, Connection, Vectstr, Matrstr, Domain`. They are described in the subsequent sections.

Some utility classes have also been developed to manage variable size arrays of objects (`template Vararray<>`), stacks, buffers, pool for memory management, read and write protection on objects and execution time measurement, among others.

5.1. Mathematical Classes

At present the library consists of 30 classes organized into two groups:

1. The first group consists of different types of matrices. Most of them inherited from a virtual class `moth_mat`.
   - `Matrix`: full matrices.
   - `MatrixCx`: full complex matrices.
   - `Sparse`: sparse matrices.
   - `SparseCx`: sparse complex matrices.
   - `Matsym`: symmetric matrices.
   - `Matband`: symmetric banded matrices.
   - `Skymat`: symmetric skyline matrices
   - `SkymatCx`: symmetric skyline complex matrices.
   - `SkymatNs`: non-symmetric skyline matrices.
   - `SkymatCxNs`: non-symmetric skyline complex matrices.
   - `MatTri(U/L)`: lower/upper triangular matrices.
   - `Vect`: vectors.
   - `VectSet`: set of vectors (`Vect`).
   - `BlocSet`: set of `VectSet`.
   - `Vect3, Matr3`: three-dimensional vectors and matrices.
   - `VectN<N>, MatrN<N>`: N-dimensional vectors and matrices (template implementations for high performance in tiny dimensions).
   - `VarVect, VarVectSet, VarVarVectSet`: variable size vectors implemented by template `VarList<type>`.
2. The second group concerns calculation methods. These classes serve as *black boxes* defining robust user interfaces to the method, and taking into account particular aspects such as, e.g., memory management.
   - `StatSyst`: base class for linear systems solvers
     - Direct solvers
     - Iterative solvers (implementations of GMRES, BiCg, BiCgStab, CGS, CG from SIAM Templates book.)
   - `DynamSyst`: base class for all eigenvalue and eigenvector calculations.
     - `Jacobi`: Jacobi method for symmetric (eventually banded) matrices
     - `QR`: QR method for banded matrices
     - `SVD`: singular value decomposition implementation
     - `Lanczos`: Lanczos algorithm with single iteration vector and restart, or blocks Lanczos algorithm
   - `Opti`: optimization functions.

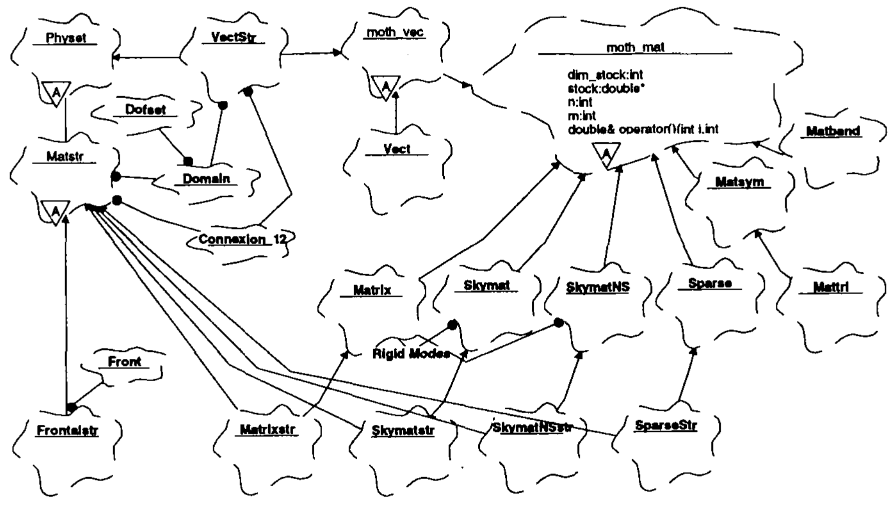


Figure 1. *Hierarchy of matrix classes*

The hierarchy of matrix classes is described in Figure 1. Classes `Physet` and `Structural` are described later in Sections 5.2 and 3. We note that this figure follows Grady Booch diagram conventions.

5.2. Physical Classes

The set of physical classes is used to describe the problem under analysis and the system state. A list of some of the currently available physical classes in OoFeLie follows:

`Nodestate:` virtual generic class providing standard interface to access state variables of a node, including:

> **`Position:`** coordinates of a point in Cartesian space,
>
> **`Displacement:`** displacements at a point,
>
> **`Force:`** data and methods to handle loads,
>
> **`Temperature:`** temperature at a point.

Also, time derivatives of the above are accessed through `Nodestate`.

`Element:` virtual master class of the finite elements library. It provides methods of access to their attributes, and of computation of the different finite element arrays and tables (e.g., `fill_stiffness()`, `fill_mass()`, etc.).

`Material:` master class for description of materials of various kinds.

`Propelem:` master class for description of element properties.

`Fixation:` master class for information concerning boundary conditions of the model.

The objects of physical classes can be grouped to form sets of objects of the same kind. For instance, the set of `Material` objects can be grouped under an instance of the class `MaterSet`. The notion of `Set` makes easier to manipulate data. For instance, it allows us to make operations on sets such as giving a common property to all objects in a set (e.g., a given material can be assigned to a whole set of elements).

The implementation of `Sets` is based on the concept of a templates to simplify programming. Different classes were implemented for the various kinds of physical classes. Some of them are: `PosiSet`, `ElemSet`, `FixaSet`, `MateSet`, `TempSet`, `DispSet`. Any values of sets of classes inherited from `NodeState` may be accessed with the structural vectors.

5.2.1. The Class `Physet`: Stack of Properties and Parental Recursion

In many situations, physical objects need to make references to properties, which usually are physical and therefore are represented as objects of the physical family (for example, an element can reference a set of nodes or a set of materials). These properties can be attributed to a single physical object or to a set of objects (e.g., a `Material` can be attributed to a single `Element` or to an `ElemSet`).

To establish such links, the virtual base class `Physet` has been created, from which every physical object inherits. This class manages, for each `Physet` object, a list of references to other `Physet` objects.

The search for a particular property is done using a mechanism called *parental recursion* implemented in the `Physet` class. In this mechanism, the object looks first for the searched property in its own list of properties (pointers to other `Physets`). If it is not found, the query is passed to a *parent object* (for example, from an `Element` to an `ElemSet`). The mechanism is applied recursively until either finding the property, or until arriving the lineage root.

If the property is not found in any object up in the lineage, the virtual function `Physet::if_not_found()` is called (see Figures 2 and 3). This function is provided to allow the designer of a physical class implement a default action to be followed in case of failure (e.g., create a default object of the searched property and issue a warning message).

Properties can be identified by using either their type (`Ph_Ob_id` code) or their `Key`, which distinguish physical fields in a same type of properties.

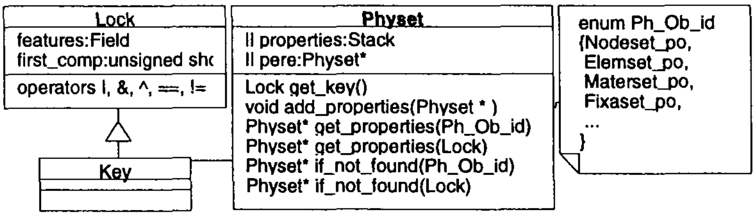

Lock
features:Field
first_comp:unsigned shc
operators |, &, ^, ==, !=
Key
Physet
|| properties:Stack
|| pere:Physet*
Lock get_key()
void add_properties(Physet *)
Physet* get_properties(Ph_Ob_id)
Physet* get_properties(Lock)
Physet* if_not_found(Ph_Ob_id)
Physet* if_not_found(Lock)
enum Ph_Ob_id
{Nodeset_po,
Elemset_po,
Materset_po,
Fixaset_po,
...
}

Figure 2. *Virtual class `Physet`*

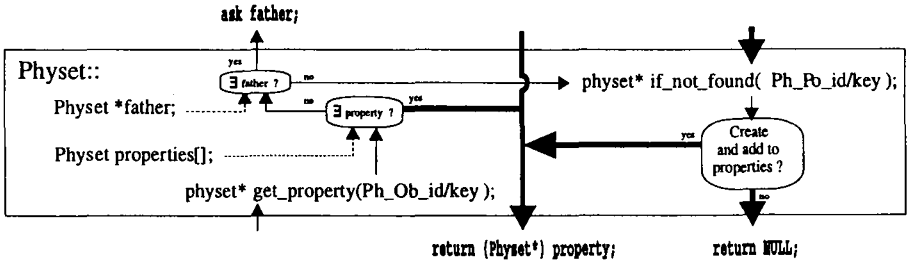

ask father;
Physet::
yes
∃ father ?
no
Physet *father;
no
∃ property ?
yes
Physet properties[];
physet* get_property(Ph_Ob_id/key);
physet* if_not_found(Ph_Po_id/key);
yes
Create and add to properties ?
no
return (Physet*) property;
return NULL;

Figure 3. *Parental recursion mechanism implemented in* `Physet`

6. Coupled Fields Management

6.1. Degrees of Freedom

The discretization process establishes a set of generalized coordinates or *degrees of freedom* for the continuum fields: i.e., displacements, temperatures, velocities. The class `Dof` implements the concept of degree of freedom. This is an essential concept in the method, so that the design of this class deserves special attention.
A *degree of freedom* is characterized by following aspects:

Physical nature: is it a displacement along the x, y or z-axis, a rotation, a temperature, an electrical potential, a Lagrange multiplier, ...?

Differentiation level: does it correspond to the primary variable or to its first or second time-derivative? In other words: is it a generalized displacement, a generalized velocity or a generalized acceleration ?

State: is it fixed or free? is it located at an interface or not?, is it a measured dof or not? (the latter concept is used in modal identification methods).

Note that the number of degrees of freedom for a given problem can be very large, so that the information of a `Dof` should be stored in a minimum of space. Also, a `Dof` object should be capable of answering as fast as possible to queries concerning its own characteristics as for instance: *does this dof correspond to a fixed displacement?* The set of characteristics of a `Dof` is stored in an object of the class `Key`, designed to satisfy the above-mentioned requirements.

Groups of `Dofs` are managed by objects of the class `Dofset`. A Dofset instance is automatically generated upon declaration by the user of a *structural* object (i.e., a structural vector or matrix, see Section 3). Any modification of the mesh implying modification of the *dof* numbering is forbidden after generation of the degrees of freedom set, unless explicitly stated, in which case the *dof* information will need to be recomputed.

Pos.	Sub-field	Abr.	Flags meaning
1:10	NATURE	TX	Displacement x
		TY	Displacement y
		TZ	Displacement z
		RX	Rotation x
		RY	Rotation y
		RZ	Rotation z
		TO	Temperature
		EP	Electric potential
		LM	Lagrange multiplier
		TM	Time (for space time-finite element)
11:12	REFERENCE	AB	Reference (absolute)
		RE	Incremental (relative)
13:14	FIXATION	FR	Free
		FR	Fixed
15:16	INTERFACE	IN	Interface
		NI	Not-interface
17:18	BOUNDARY	BO	Boundary
		IL	Internal
19:20	MEASURE	ME	Measured
		NM	Not-measured
21:24	DEGREE	GD	Generalized displacement
		GV	Generalized velocity
		GA	Generalized acceleration
		GF	Generalized force
25:28	JOKERS	J1	Joker 1 (user defined)
		J2	Joker 2 (user defined)
		J3	Joker 3 (user defined)
		J4	Joker 4 (user defined)

Table 1: Flags in field

6.1.1. Qualifying Degrees of Freedom: Classes `Key` and `Lock`

Sets of physical characteristics and their states are described by instances of the class `Key`. On the other hand, acceptance criteria are described by objects of the class `Lock`. The latter class is quite similar to the former one, from which it is derived by inheritance. Both classes work by binary encoding the information. Then, the satisfaction of a given criterion is verified by logical operations on the binary flags (which are performed very fast). Queries concerning the characteristics of a given dof can be translated into the following question: *does this key match the lock?* This section describes some details of implementation.

In an object of the class `Key`, each characteristic item (e.g., displacement x) is assigned a two-states flag. In order to keep the memory requirements to a minimum, these flags are encoded and grouped together into an instance of a class named `Field`. An instance of `Field` defines 32 bit-flags and occupies 5 bytes.

The class `Field` is actually made of several sub-fields. A `Key` allows the setting to on only one flag at each sub-field. Currently, the eight sub-fields mentioned in Table 1 have been defined. By setting to on one (and only one) bit at each sub-field, we fully specify the nature and characteristics of the corresponding degree of freedom.

The program has several predefined keys. For instance, there is a predefined key for each elementary bit information;

```
Key TX (0x10000000);    Key TY (0x20000000);
Key TZ (0x40000000);    ...
```

with the constructor of the class `Key` initializing the `Field`.

Fields describing more complex concepts can be generated as the result of logical operations between elementary `Fields`.

The result from operating two `Fields` objects is a new `Field` object, whose flags are issued from logical operations between the corresponding flags of the original `Field` objects.

Keys for typical degrees of freedom are also predefined:

```
Key DISPL_X(TX|GD);     Key DISPL_Y(TY|GD);
Key DISPL_Z(TZ|GD);     ...
```

We note that values for sub-fields not defined at construction are given by the default Key:

```
Key DEFAULT (RE|FR|NI|IL|NM|J1);
```

For instance, the full description of a free displacement degree of freedom on the y direction is made as follows:

NATURE	REF	FIX	INT	BND	MEA	DEG	JOK
0100000000	01	10	01	01	01	1000	1000

6.1.2. Class Lock

The class `Lock` defines a criterion of acceptance for objects of the class `Key`. The implementation of `Lock` is similar to that of `Key`, with the difference that it allows to set to on several flags per sub-field.

For instance, the following are valid `Locks`:

```
Lock DISPLACEMENT  (TX|TY|TZ|GD);
Lock FORCE         (TX|TY|TZ|GF);
...
Lock NATURE(TX|TY|TZ|RX|RY|RZ|TO|EP|LM);
Lock FIXATION(FR|FI);
...
Lock ALL(0xffffffff);    Lock NOTHING(0x00000000);
```

The function `int Key::match(Lock &)` verifies matching of a `Key` with a `Lock`. This function makes a logical OR between corresponding flags of both objects. If the result is different from zero for *each* sub-field, the function returns 1 (i.e. *the key opens the lock*).

The following examples illustrate a case in which a key succeeds into opening a lock and a case which it does not:

NATURE	REF	FIX	INT	BND	MEA	DEG	JOK	
1000000000	01	10	01	01	01	1000	1000	(key)
1111110000	01	10	01	01	01	1111	1111	(lock)
1000000000	01	10	01	01	01	1000	1000	⇒**True**

NATURE	REF	FIX	INT	BND	MEA	DEG	JOK	
1000000000	01	10	01	01	01	1000	1000	(key)
1111110000	01	01	01	01	01	1111	1111	(lock)
1000000000	01	00	01	01	01	1000	1000	⇒**False**

Certain acceptance criteria cannot be expressed by a single `Lock`. Think, for instance, in a case in which we ask a `Key` to be either a free displacement along x or a fixed displacement along y (a `Lock` built following these notions would also accept non-desired combinations, as fixed displacements along x). In such cases, we are obliged to test against several `Locks`. The class `UnionLock` was introduced to simplify these operations.

An instance of the class `UnionLock` expresses in a single entity a set of `Locks`. Then, the function `int Key::match(UnionLock &)` verifies matching of the Key with the sequence of `Locks`, and returns 1 (true) whenever the `Key` matches any one of them.

A `UnionLock` can be built by making the addition (operator '+') of several `Locks`.

```
Lock txyfree (TX|TY|FR);
Lock rzfix   (RZ|FI);
UnionLock rxfr_or_ryfi = txyfree + rzfix;
```

6.2. The Mathematical Point of View

Solving numerically a physical problem leads to the problem of solving a system of algebraic equations. The solution is expressed in terms of local values of the physical variables (i.e., the degrees of freedom of the model). When the behavior at all degrees of freedom is known, the model is fully determined.

In many cases, we need to split and sort the degrees of freedom into groups following different criteria and according to the solution method. This corresponds to dividing the domain into sub-domains. These sub-domains may for instance result from geometric considerations or may also correspond to grouping of dofs according to their physical nature. They can be use, for example, to solve an interface problem either in primal form (in terms of global interface unknowns) or in dual form (in term of interface Lagrange multipliers)[FR91] , or to build specials preconditioner. Criteria to make the sub-domain *splitting* are implemented using the classes `Lock` and `Partition`.

The connection operation consists in *sorting* the degrees of freedom at each subdomain and localize them inside the sub-domain(s) to which they belong. It has been implemented introducing the class `Connection`.

6.2.1. The Concept of Partition

Degrees of freedom may be classified in partitions following different criteria. This information is used to link the `Dof` to the associated mathematical unknowns.

Different criteria of partitioning can be defined, namely:

- according to the nature of dof (e.g., translation / rotation / internal dof),
- according to its fixity state (e.g., fixed / free),
- according to geometric or physical partitioning (e.g., substructuration for parallel computations, substructuration for constructing super-elements),
- or any combination of the above-mentioned criteria.

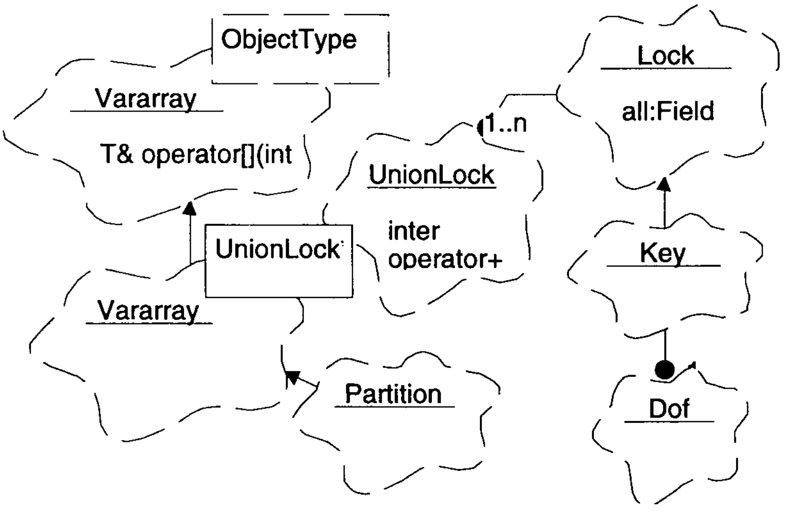


Figure 4. *Implementation of class* `Partition`

The class `Partition` has been created to implement this concept. This class allows the defining of a distribution of the degrees of freedom set in separate subsets following one or more criteria. An object of the class `Partition` gives an exclusive mapping between dofs characteristics and part numbers. It can be built simply by giving the sequence of `Locks` (or `UnionLocks`) that define each part. A dof belongs to the part whose `Lock` is first matched, when verifying the parts assignment in ascending order. The last part is conventionally defined to be the `Lock all`, so that this part comprises all those degrees of freedom not included in the preceding ones.

The following example illustrates a case in which we partition the degrees of freedom set into four parts:

Group 1:	free displacement dofs along x, y and z axes
Group 2:	free rotation dofs along x, y and z axes
Group 3:	free temperature dofs
Group 4:	remaining dofs (fixed displacements, rotations and temperatures).

Each of the first three groups is characterized by a particular `Lock`, while the fourth one is characterized through the `Lock all` which has all bits set to `on`. An example of the set of instructions defining an object P of the class `Partition` follows:

```
>> Partition P;
>> P.add_new_part( TX|TY|TZ | FR );
>> P.add_new_part( RX|RY|RZ | FR );
>> P.add_new_part( TO | FR );
>> P.print();
Partition =
Field no 1
1 1 1 0 0 0 0 0 0 0 0  0 1  1 0  0 1  0 1  0 1  1 0 0 0  1 0 0 0
Field no 2
0 0 0 1 1 1 0 0 0 0 0  0 1  1 0  0 1  0 1  0 1  1 0 0 0  1 0 0 0
Field no 3
0 0 0 0 0 0 1 0 0 0 0  0 1  1 0  0 1  0 1  0 1  1 0 0 0  1 0 0 0
Field no 4
1 1 1 1 1 1 1 1 1 1 1  1 1  1 1  1 1  1 1  1 1  1 1 1 1  1 1 1 1
```

6.2.2. The Connection

Each dof is individualized by a unique global label. Structural matrices and vectors are assembled separately for each part. The connection establishes to which part and in which position inside its part, should be assigned each dof. This concept is implemented by an object of the class `Connection`.

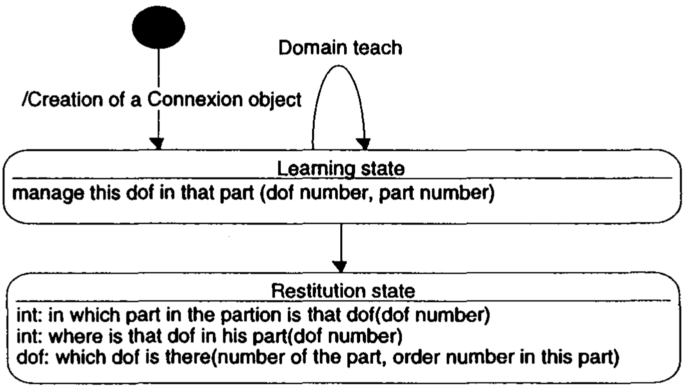

Domain teach
/Creation of a Connexion object
Learning state
manage this dof in that part (dof number, part number)
Restitution state
int: in which part in the partion is that dof(dof number)
int: where is that dof in his part(dof number)
dof: which dof is there(number of the part, order number in this part)

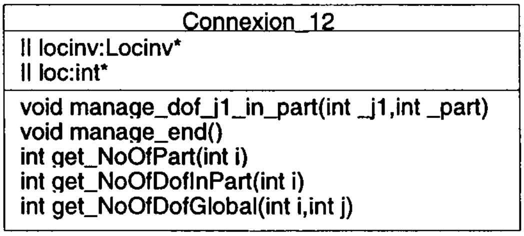
Figure 5. *State diagram for class Connection*

We may distinguish two phases when using an instance of the class. During the first phase, the *learning* phase, we teach the Connection to which part belongs each dof. In this phase, we are able to modify the dofs ordering at each part (e.g., using different bandwidth optimization algorithms) or even the dof/part assignment.

In the second phase, the *restitution* phase, we inquire the object to get information about the particular connection. For instance, we may ask to which part does a given dof belong, what is the order number inside its part, etc.

Both phases are separated by a call to the member function `manage_end()`, which enables the user to make inquiries and disables any further intent to modify the connection.

The domain has a member function `void build_connect(Connection&, Partition&, Opt_ReOrder_Type=Sloan)` which is responsible for the first learning phase of `Connection`.



```
Connexion_12
-----------------------------------------------
|| locinv:Locinv*
|| loc:int*
-----------------------------------------------
void manage_dof_j1_in_part(int _j1,int _part)
void manage_end()
int get_NoOfPart(int i)
int get_NoOfDofInPart(int i)
int get_NoOfDofGlobal(int i,int j)
```

Figure 6. *Interface for class* `Connection`

An example of construction of an object of the class `Connection` follows.

```
>> Domain BarTher{...};
>> Partition P;
>> P.add_new_part(TX | FR);
>> P.add_new_part(TO | FR);
>> Connection C;
>> BarTher.build_connex(C,P); C.print();
Connection = Number of parts: 3
             Number of dofs : 30
 - part 1 : 8 dofs
 - part 2 : 2 dofs
 - part 3 : 20 dofs
```

First, we defined a partition with three parts: free x-displacements, free temperatures and the rest of dofs. Then, based on this partition, we built a connection for the domain `barther`.

6.2.3. *Structural Vectors and Matrices*

Knowing a `DofSet` and a corresponding `Connection` we can construct *structural vectors* and *matrices* which will be used in the numerical solution phase. If no sub-domain has been defined using partitions and connections, matrices and vectors for the whole domain are constructed.

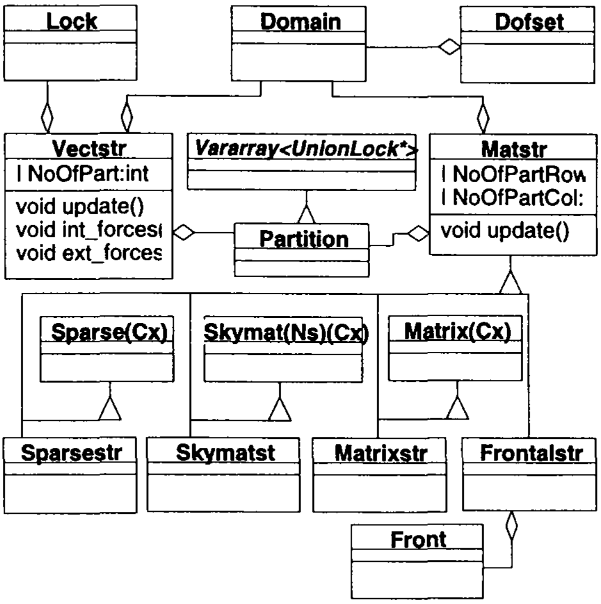


Figure 7. *Structural vectors and matrices*

6.3. The Physical Point of View

6.3.1. Storage Sets

To solve a problem we need to manage information on the model. We may consider we have a database with already known information which represents the model, and from which we evaluate new pieces of information during computations, that are to be stored later in the database. A mechanism has been designed to manage, store and recall *nodal* results, and interface them to the mathematical classes. This mechanism is implemented using the structural vector, which is in fact a vector of pointers on values stored in the database.

The structural vector is built from three kinds of data: the domain, the connection, and the part number in the partition. For instance, we may define a structural vector that comprises all degrees of freedom in the second part of domain A, for the connection Co. In this way, we may have different vectors for the different parts in which a given domain has been partitioned. We can also define several connections on a same domain and different structural vectors for each one. We use these notions, for instance, in optimization problems (see Example 8.1).

In order to completely define the structural vector we should give also its type; that is, we have to indicate further qualifiers that apply to all dofs in the structural vector. These qualifiers are: the differentiation degree (i.e. generalized displacements, generalized velocities, accelerations), absolute or relative quantities, measured quantities or not? Normally, all this information is contained in the Key associated to each dof. We have limited the possibilities of constructing structural vectors to cases in which all dofs are of the same type, for instance, they have all the same differentiation degree (i.e. they are all displacements). This is the most usual situation and does not introduce any practical limitation. The rest of the sub-fields in the Key (NATURE, FIX, INT, BND and JOK) are used to define the index number inside the structural vector by an appropriate algorithm. The situation is illustrated next.

NATURE	0 0	FIX	INT	BND	0 0	0 0 0 0	JOK	⇐ dof
0 0 0 0 0 0 0 0 0 0	REF	0 0	0 0	0 0	MEA	DEG	0 0 0 0	⇐ vectstr

In the following example, we define a structural vector v for the domain BarTher, covering all the generalized displacements of the model.

```
>> Vectstr v (BarTher, GEN_DISPL);
```

where GEN_DISPL is a key that indicates degree of differentiation 0.

Figure 8 gives a brief description of the data structure implementation through flow diagram. For instance, the identifier and coordinates of each node are stored into an object of the class Position. The set of nodes of the discretization is then grouped into an object of the class PositSet. And the set of node coordinates is put together with the other objects that form the discrete model (sets of elements, fixations, loads, materials, etc.) to form an object of the class Domain. The class Domain contains also pointers to objects of classes that describe results of computations; e.g., displacements, stresses, velocities, accelerations, etc.

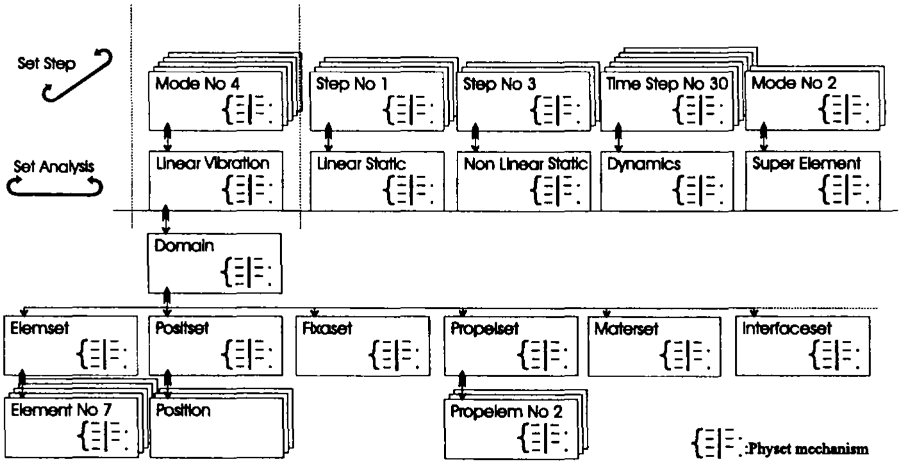


Figure 8. *Domain and physical data*

6.3.2. The Physical Space: the Domain

The domain contains all information about the representation and modelisation of physical space. It acts as a container for references to all data sets of a given computation (see Figure 8), and provides methods of access to any physical data. Physical data is accessed by using the stack of properties mechanism (see Section 1). An object for each one of classes `Posiset`, of `Elemset` and of `Fixaset` are minimally required to build a `Domain`. But several other physical properties may be added to extend the capabilities of analysis, for example, `Partition`, `Connection`, `Interfaceset`, etc. The `Domain` is also the class that constructs the `Dofset`.

Once a `Domain` is defined, structural objects like instances of `Vectstr` of `Skymatstr` may be built with reference to it, giving reference to the corresponding `Dofset` or subset of the `Dofset` if a partition is defined.

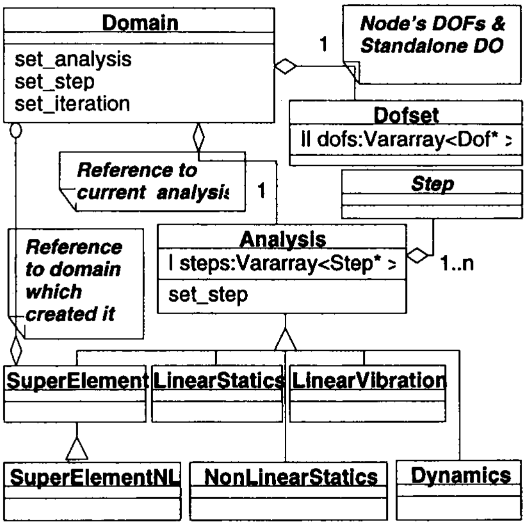


Figure 9. *Classes* `Domain` *and* `Analysis`

7. Dialog with Objects

The software we have built tries to answer the following requirements of design:

- users have to dispose of a fairly large variety of algorithms to be applied in many changing situations;
- it should be easy to incorporate new algorithms to the program;
- linking and chaining the execution of different solution algorithms ought to be easy and transparent to the user.

The answers we gave to these demands is by means of a language interpreter which accesses data usually restricted to internal use in a standard program. The language can be used to describe both data and algorithms. In other words, it is able to define in detail the methods by which data should be transformed to reach the final results.

7.1. The Command Line Interpreter

The main program works in dialog with two modules: an *interpreter* –which employs a specialized high-level language– and an *execution manager.* The interpreter translates user instructions into a low-level language internal to the program. A first detection of errors is made at the interpreter level, followed by appropriate error recovering. If the command is correct, the equivalent command in low-level language is transmitted to the execution manager, which gives course to the required action.

We distinguish between three different kinds of commands: *flow-control, declaration* and *expression* statements. Flow control actions are directly taken by the execution manager. Declarations require the construction of an object, followed by the storage of its name and address into a symbol table. These actions are also taken by the execution manager, after calling the appropriate constructor of the class.

For the other statements, which involve algebraic operations and function calls, the execution manager simply invokes the appropriate method of the concerned class. This is programmed using the dynamic binding capability of an object-oriented language [AMM91] . For instance, when the user asks for the addition of two scalars, the execution manager will call the addition method of the scalar class. But when the addition of two vectors is required, the execution manager will make use instead of the addition method for the vector class, by simply recognizing the type of variables involved in the algebraic operation.

7.2. Choice of Grammar

The command interpreter was defined in terms of a language whose grammar is as close as possible to C++. Therefore, once an algorithm is implemented and tested (interactively or through a commands file), the resulting procedure can be easily added to the compiled part of the software (for example, as a member function).

Here is a list of functionalities which have been given to the interpreter-executor[1]:

- creation of objects from classes and use of their members functions,
- use of inheritance and polymorphism,
- execution of algebraic operations through operators (+ $-$ $*$ / [ ] & | ^),
- use of Stream capabilities ($\ll$ $\gg$)[STR91] ,
- possibility to add and/or to overload member functions,
- material or element properties may depend on other properties, through functions declared with the interpreter.

[1] a description of the parser/interpreter implementation itself is made in reference[CKG94]

Two major problems have been encountered in the specification of the interpreter capabilities. They have requested special adaptations of the grammar and are briefly discussed below.

7.2.1. Function Definition

We can see there exists an analogy between the prompt command line in the interpreter and the `main(){...}` function in a C++ program. The only exception is that in C++, functions cannot be declared and/or defined inside the main program.
A declaration such as

```
Complex csqrt(float x, float)
```

leads to an ambiguity, since the parser cannot see the difference between the declaration of a `Complex` variable whose name is `csqrt` with constructor `(float, float)` and the declaration of a function `csqrt` which returns a `Complex` from two arguments. For this reason, a new reserved key word `Function` has been introduced in the language.

7.2.2. Inheritance

In a compiled program, the name of an object is specified at its creation (e.g., `float a;`). That name will never appear in the object file resulting from the compilation, since the executable file no longer needs that information at run time.

In an interpreted language, however, we need that information. Therefore, according to encapsulation philosophy, each object must have a member function which returns its name given at declaration time.

The same problem happens when finding the name of public data and member functions, in which cases the problem of inheritance is encountered. Supposing we have a class `sA:` inheriting from a class `A:`. `A:` has a member function `funcA` which does not need to be redefined in `sA:` (Figure 10). How could the interpreter know that `sA:` has a member function `funcA` due to its inheritance from `A:`?

In a compiled use of class, static information on polymorphism, encapsulation and inheritance does not survive to compilation; thus, it no longer exists in the executable file. In an interpreter, compilation and execution co-exist, in which case, this information has to be stored and controlled to maintain the object-oriented concept in the interpreted language. This fact naturally leads to the concept of *I_* classes.

7.3. I_ Classes

Each class to be accessed through the interpreter, has a corresponding `I_class` which inherits from the virtual class `I_` (see Figure 10).

Objects on the interpreter stack are of type `I_`. Each one has a name and a pointer to the basic object to whom it corresponds. In order to execute the member function whose name and arguments are on the stack (if it exists), the `I_` object uses the proce-

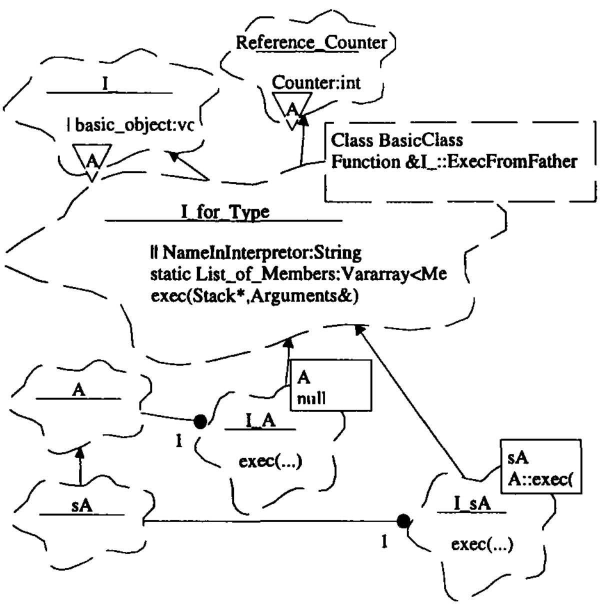


Figure 10. *I_ mechanism to interface basic objects with the command line interpreter.*

dure `exec (Stack*, Arguments&)`. Also, it manages a reference counter on the basic object to allow deletion when appropriate.

The function `I_::exec(...)` is defined for each class `I_`. It manages the transfer of control and data to functions of the basic object which we want to make accessible through the interpreter. If the basic object class inherits from another class, and if `I_::exec(...)` fails to find the member function, control is transferred to the `I_::exec(...)` corresponding to the base class. In this way, the inheritance mechanism is implemented in the interpreter.

Let us again consider the example of the class `sA:` which inherits from a class `A:`. `I_sA::exec(funcA,...)` makes a call to `I_A::exec(...)` since it does not find the member function `funcA`defined for `A:`. The call is finally made in `I_A::exec(...)`. The mechanism has easily been extended to multiple inheritance.

7.4. Example: Newton Algorithm

We next show an example of a function written in the high-level language of OOFELIE, which describes an elementary implementation of the Newton algorithm to get the solution of a non-linear algebraic problem.

```
while   $\| (r_i = f - f_{int}(q_i)) \| < \epsilon$
do      $q_i = K_{tan}(q)^{-1} r_i(q_{i-1})$
        $q = q + q_i$
```

```
Function Vectstr Newton (Domain dom)
{
   dom.set_analysis (NONLINEARSTATICS_PO);
   Vectstr u(dom, GEN_DISPL); Vectstr f(dom, GEN_FORCE);
   Vectstr du(dom).set_to_zero(); Vectstr residue (dom);

   int     itmax = 10      ;      int iter   = 0   ;
   double  tol   = 0.00001;    double prec   = 1.0;
   double  nfext = tol;

   residue = f - du.int_forces();
   nfext  += f.norm() + du.norm();

   Skymatstr k ( dom, GEN_TANG_STIFF );
   while( (iter<itmax) & (prec>tol) )
   {  iter++;
      k.update();
      k.factor();
      k.back( residue , du ) ;
      u += du ;
      residue = f - du.int_forces() ;
      prec = residue.norm() / nfext;
   };
   iter.print();
   return u;
};
```

Once a `Domain` has been defined –i.e., a notion embodying all data for a given problem– a simple interactive call to this method returns the *structural vector* `u` for which equilibrium is re-established. This short example shows that the interactive executor is able to manage function calls, iteration and conditional branching statements.

8. Applications

8.1. The MBB Beam: Example of Optimization Problem

We consider a simplified step of the MBB beam topological optimization problem [OBR90] . It consists into finding the best position of two nodes that minimize the work of external forces in an eight bars 2D truss (Figure 11).

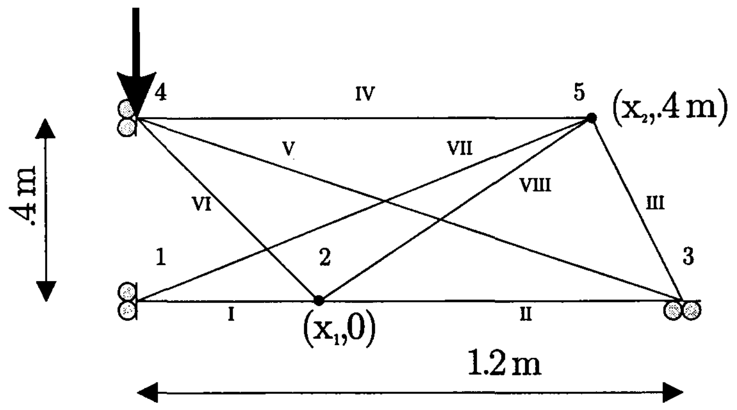


Figure 11. *MBB domain*

We next describe data for this problem. First, we define nodes, positions, bars, loads, etc.:

```
float height=40;          float width =120;
float aX2 = width/2.;     float aX5 = width;

Positset  a { 1 0 0 0 ;
              2 aX2 0 0 ;
              3 width 0 0 ;
              4 0 height 0 ;
              5 aX5 height 0 };
Elemset b { 1 BARELST  1 2 ; 2 BARELST  2 3 ;
            3 BARELST  3 5 ; 4 BARELST  4 5 ;
            5 BARELST  3 4 ; 6 BARELST  2 4 ;
            7 BARELST  1 5 ; 8 BARELST  2 5 };
Set cha(FORCE).define(4, 2, -100);
Fixaset  fix;
fix.define (1, TX); fix.define (1, TZ);
                    fix.define (2, TZ);
fix.define (3, TY); fix.define (3, TZ);
fix.define (4, TX); fix.define (4, TZ);
                    fix.define (5, TZ);
```

In order to have direct access to structural vectors defined over the set of positions we want to optimize, we define an `Interfaceset` that comprises the selected dofs:

```
Interfaceset Moving_pos;
 Moving_pos.define(2,TX);
 Moving_pos.define(5,TX);
```

Then, a set of materials and elements properties is given:
```
Materset maters;
Material steel (ISOTROPIC);
 steel.put (ELASTIC_MODULUS, 21e4);
 steel.put (POISSON_RATIO,   0.3);
 steel.put (MASS_DENSITY,    7.8e-6);
maters.put (1,steel);
Propelem prp(BARELST);
 prp.put (CROSS_SECTION_1,1);
 prp.put (MATERIAL,1);
 b.add_properties(prp);
```

We next instruct the `Domain` about the particular problem to analyze:
```
Domain MBB { a b maters cha fix Depl_on_interf };
MBB.set_analysis (NONLINEARSTATICS);
```

A linear static analysis of the `Domain` MBB can be done by defining the function `LinStat`:
```
Function Vectstr LinStat (Domain dom)
{dom.set_analysis(LINEARSTATICS);
 Skymatstr k (dom, GEN_STIFF );
 Vectstr u   (dom, GEN_DISPL );
 Vectstr f   (dom, GEN_FORCE );
 k.factor();
 k.back(f, u);
 return u;
};
```

In order to perform a non-linear static analysis, we may use function `Newton` defined in Example 7.4

We remark that the interpreter allows quite involved computations to be very easily expressed. For instance, the difference between the linear and non-linear solutions for a given problem is written:
```
>> (LinStat(MBB)-Newton(MBB)).print();
```
The result is:
```
Vectstr = Vect. Dim: 7
 Node               Component               Value
 ==========================================
    1                 Y Trans.              0.0072476
    2                 X Trans.             -0.00019790
    2                 Y Trans.              0.00050828
    3                 X Trans.              0.00042267
    4                 Y Trans.              0.0026230
    5                 X Trans.              0.0013617
    5                 Y Trans.              0.00034099
```

We will now compute the horizontal position of nodes 2 and 5 that give minimal deformation energy in the system. To this end, we use Newton's method for unconstrained minimization[DS83] :

Given $f : R^n \longrightarrow R$ twice continuously differentiable, $x_0 \in R^n$; for each iteration k,

$$\text{solve}\Delta^2 f(x_k)\, s_k^N = -\Delta f(x_k),$$
$$x_{k+1} = x_k + s_k^N$$

Δf and $\Delta^2 f$ will be evaluated by finite differences.
We first define function `ff` which returns the work of external forces, for a given configuration:

```
>> Function scalar ff (Domain dom )
   { scalar p;
     Vectstr f(dom, GEN_FORCE);
     p = f * newton(dom);
     return p;
   };
```

Displacement and force vectors are created for the `Domain` MBB:

```
>> Vectstr U ( MBB , GEN_DISPL );
>> Vectstr F ( MBB , GEN_FORCE );
```

We next define a second independent partition, that will allow us to have direct access to positions x_2 and x_5:

```
>> Partition Pa ;
>> Pa.add_new_part( TX | IN ).print();
Partition Pa =
Field no 1
1 0 0 0 0 0 0 0 0  0 1  1 0  0 1  1 0  1 0  1 0 0 0  1 0 0 0
Field no 2
1 1 1 1 1 1 1 1 1  1 1  1 1  1 1  1 1  1 1  1 1 1 1  1 1 1 1
```

The connection for partition `Pa` is built and used to define the structural vector `p`, which will reference positions x_2 and x_5 at the physical database:

```
>> Connection Co;
>> MBB.build_connect ( Co , Pa );
>> Vectstr p ( MBB , AB , Co );
```

`p` gives direct access to the independent parameters of the optimization problem, with objective function `ff`.

In the following, we build the matrix of second-order derivatives of the objective function by numerical differentiation, and iterate up to obtaining convergence for a minimum work configuration, by updating the horizontal positions of nodes 2 and 5.

```
>> double e=0.0001, Fp;
>> Vect x0(p.dim), sN(p.dim), Df(p.dim);
>> Matsym DDf(p.dim,p.dim);
>> x0=p;
>> int i, j, NbrIter;
>>
>> while (sN.norm>e)
   {Nbre_d_iteration++;
    p=x0;     Fp=ff(MBB);
    i=0;
    while (i<p.dim)
    { i++;       p=x0;       p[i]+=e;
      Df[i]=ff(MBB);
    };
    i=j=0;
    while(i<p.dim)
    { i++; j=i-1;
      while(j<p.dim)
      { j++;      DDf[i,j]=Fp;
        DDf[i,j]=DDf[i,j]-Df[i];
        DDf[i,j]=DDf[i,j]-Df[j];
        p=x0;       p[i]+=e;         p[j]+=e;
        DDf[i,j]=(DDf[i,j]+ff(MBB))/e/e;
      };
    };

    i=0;
    while(i<p.dim)
    {  i++;
       Df[i]=(Df[i]-Fp)/e;
    };

    DDf.solve(Df,sN);

    double exp=1.;
    p=x0-sN;
    Fpp=ff(MBB);
    while( (Fpp > (Fp + 1e-4 * exp * (Df * sN)))
          &(sN.norm>e)   )
    {  exp/=2.;         sN*=exp;
       p=x0-sN;         Fpp=ff(MBB);
    };
    x0=p;
   }
```

```
Vect x0 = 60.0000 ! 120.000 !  scalar Fpp =  66.6880
Vect x0 = 61.1483 ! 98.6275 !  scalar Fpp =  66.3469
Vect x0 = 62.1338 ! 103.560 !  scalar Fpp =  66.3148
Vect x0 = 63.3607 ! 105.256 !  scalar Fpp =  66.3147
Vect x0 = 63.4108 ! 105.171 !  scalar Fpp =  66.3147
Vect x0 = 63.4214 ! 105.185 !
```

Results of computations are the positions of both nodes 2 and 5:

```
>> Nbre_d_iteration.print;
scalar Nbre_d_iteration = 5.00000
>> p.print;
Vectstr p = Vect. Dim: 2
 Node            Component               Value
 ===========================================
    2              X Trans.              63.421
    5              X Trans.             105.18
```

8.2. *The Ultrasonic Motor: Example of a Coupled Physical Problem*

8.2.1. Description

The ultrasonic motor is based on an annular circular plate in free-vibrations state.

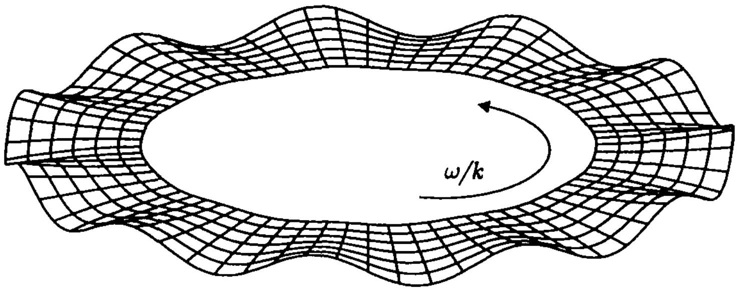


Figure 12. *Transverse vibration wave in an annular plate*

Vibration modes are given by

$$w_{kn}(r,\theta) = A_{kn}R_{kn}(r)\begin{cases}\cos(k\theta) & k=0,1,\ldots\\ \sin(k\theta) & n=1,2,\ldots\end{cases}$$

where $R_{kn}(r)$ is the Bessel function with k the number of nodal diameters.

For a given $k \neq 0$, it can be seen that the sinus mode has the same frequency ω_{kn} as the cosine mode, where they are linearly independent. The combination of these two modes gives:

$$\begin{aligned} w_k(r,\theta,t) &= w_{kn_{\sin}} + w_{kn_{\cos}} \\ &= A_{\sin} R(r)\cos(k\theta)\cos(\omega t) + A_{\cos} R(r)\sin(k\theta)\sin(\omega t+\varphi) \\ &= \tfrac{1}{2}R(r) \quad [(A_{\sin} + A_{\cos}\cos\varphi)\cos(k\theta - \omega t) \\ &\qquad +(A_{\sin} - A_{\cos}\cos\varphi)\cos(k\theta + \omega t) \\ &\qquad +2A_{\cos}\sin\varphi\sin k\theta\cos\omega t] \end{aligned}$$

with constants $A_{\sin}, A_{\cos}$ and φ determined from the initial conditions. The two first terms represent waves rotating with θ while the third term is a stationary wave.

If we choose initial conditions such as $A_{\sin} = A_{\cos} = A$ and $\varphi = 0$, we find

$$w(r,\theta,t) = AR(r)\cos(k\theta - \omega t)$$

which is a moving wave around the axis plate at the velocity of ω/k (fig. 12).

Due to the Kirchhoff hypothesis for plates, the motion of a peripheral point **Q** is given by

$$[u_r, u_\theta, w]_{\mathrm{Q}} = \left[-\frac{h}{2}\frac{\partial w}{\partial r}, -\frac{h}{2r}\frac{\partial w}{\partial \theta}, w\right]$$

where h is the plate thickness. Knowing $w(r,\theta,t)$, velocity at **Q** is given by

$$[\dot{u}_r, \dot{u}_\theta, \dot{w}]_{\mathrm{Q}} = -A\omega\left[-\frac{h}{2}\frac{\partial R}{\partial r}\sin(\omega t - k\theta), \frac{hk}{2r}R\cos(\omega t - k\theta), R\sin(\omega t - k\theta)\right]$$

which describe an ellipse in plane (θ, w).

By putting a rigid body (rotor) in contact with the plate, the contact point **Q** at the top of the ellipse when $\cos(\omega t - k\theta) = 1$, has a speed motion given by

$$[\dot{u}_r, \dot{u}_\theta, \dot{w}]_{\mathrm{Q}} = \left[0, -A\omega\frac{hk}{2r}R, 0\right].$$

Without slipping, the rotation speed of the rotor is given by $\dot{\theta} = -A\omega hkR(r)/2r^2$.

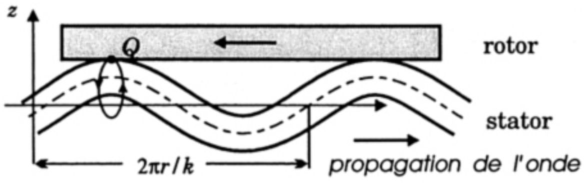


Figure 13. *Motion of the rotor*

In practice, the vibration of the plate is obtained by distributing piezoelectric actuators on opposite sides, to put in resonance the two linearly independent modes (opposed by $\pi/2$).

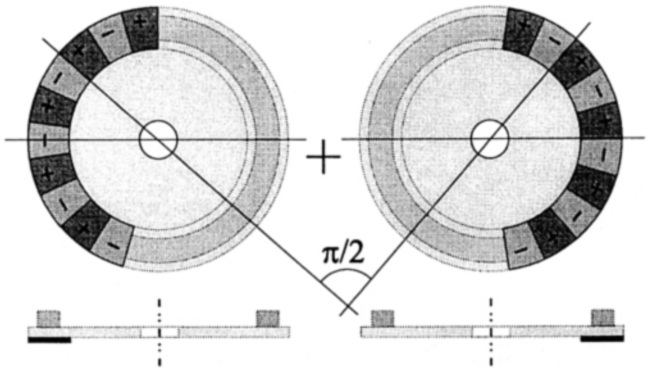


Figure 14. *Set of piezo-actuators for each mode*

8.2.2. Numerical Implementation

When simulating piezoelectric devices, we should be aware that there exists large differences in magnitude between elastic, thermal and electric terms. In order to avoid numerical troubles, we divide the equations according to the particular nature of the involved degrees of freedom.

The finite element discretization of the stator of the ultrasonic motor, leads to the system:

$$\begin{pmatrix} \mathbf{M}_{uu} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \ddot{\mathbf{u}} \\ \ddot{\boldsymbol{\phi}} \\ \ddot{\boldsymbol{\theta}} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \mathbf{C}_{u\theta} & \mathbf{C}_{\phi\theta} & \mathbf{C}_{\theta\theta} \end{pmatrix} \begin{pmatrix} \dot{\mathbf{u}} \\ \dot{\boldsymbol{\phi}} \\ \dot{\boldsymbol{\theta}} \end{pmatrix} + \begin{pmatrix} \mathbf{K}_{uu} & \mathbf{K}_{u\phi} & \mathbf{K}_{u\theta} \\ \mathbf{K}_{u\phi}^{\mathbf{T}} & \mathbf{K}_{\phi\phi} & \mathbf{K}_{\phi\theta} \\ 0 & 0 & \mathbf{K}_{\theta\theta} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \boldsymbol{\phi} \\ \boldsymbol{\theta} \end{pmatrix} = \begin{pmatrix} \mathbf{F} \\ \mathbf{Q}_{\phi} \\ 0 \end{pmatrix} \tag{1}$$

which connects the elastic field **u**, the electric field $\boldsymbol{\phi}$ and the temperature field $\boldsymbol{\theta}$. The temperature field $\boldsymbol{\theta}$ may be uncoupled from the two first equations, leading to

$$\begin{pmatrix} \mathbf{M}_{uu} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \ddot{\mathbf{u}} \\ \ddot{\boldsymbol{\phi}} \end{pmatrix} + \begin{pmatrix} \mathbf{K}_{uu} & \mathbf{K}_{u\phi} \\ \mathbf{K}_{u\phi}^{\mathbf{T}} & \mathbf{K}_{\phi\phi} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \boldsymbol{\phi} \end{pmatrix} = \begin{pmatrix} \mathbf{F} - \mathbf{K}_{u\theta}\boldsymbol{\theta} \\ -\mathbf{K}_{\phi\theta}\boldsymbol{\theta} \end{pmatrix} \tag{2}$$

and

$$\begin{pmatrix} \mathbf{C}_{u\theta} & \mathbf{C}_{\phi\theta} & \mathbf{C}_{\theta\theta} \end{pmatrix} \begin{pmatrix} \dot{\mathbf{u}} \\ \dot{\boldsymbol{\phi}} \\ \dot{\boldsymbol{\theta}} \end{pmatrix} + \mathbf{K}_{\theta\theta}\boldsymbol{\theta} = 0. \tag{3}$$

By assuming that the temperature field response is much slower than that of fields **u** and $\boldsymbol{\phi}$, we may consider in a first approximation that $\boldsymbol{\theta}$ is constant inside each time integration step of equation [2].

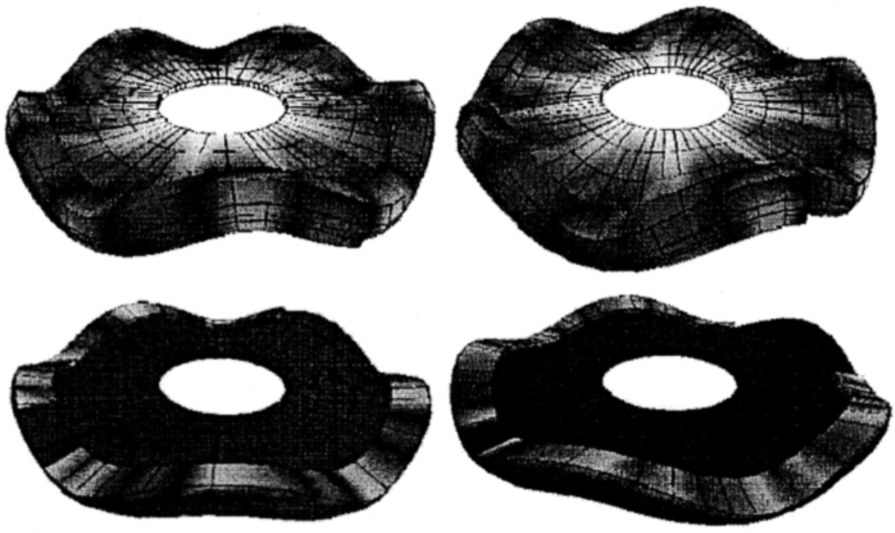


Figure 15. *Selected pair of orthogonal modes (top and bottom view)*

This allows the integration and updating of the temperature $(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}})$ separately by

$$\mathbf{C}_{\theta\theta}\dot{\boldsymbol{\theta}} + \mathbf{K}_{\theta\theta}\boldsymbol{\theta} = -\mathbf{C}_{u\theta}\dot{\mathbf{u}} - \mathbf{C}_{\phi\theta}\dot{\boldsymbol{\phi}},$$

once $\mathbf{u}, \boldsymbol{\phi}$ are known. Note that in this way both systems of equations become symmetric.

Usually, potentials at some points are known *a priori*. They include both reference neutral potentials $\phi_0 = 0$ as well as particular potential values at some electrodes ϕ_1. We may thus write

$$\begin{pmatrix} \mathbf{M}_{uu} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \ddot{\mathbf{u}} \\ \ddot{\boldsymbol{\phi}}_i \\ \ddot{\boldsymbol{\phi}}_0 \\ \ddot{\boldsymbol{\phi}}_1 \end{pmatrix} +$$
$$\begin{pmatrix} \mathbf{K}_{uu} & \mathbf{K}_{u\phi_i} & \mathbf{K}_{u\phi_0} & \mathbf{K}_{u\phi_1} \\ \mathbf{K}^{\mathbf{T}}_{u\phi_i} & \mathbf{K}_{\phi_i\phi_i} & \mathbf{K}_{\phi_i\phi_0} & \mathbf{K}_{\phi_i\phi_1} \\ \mathbf{K}^{\mathbf{T}}_{u\phi_0} & \mathbf{K}^{\mathbf{T}}_{\phi_i\phi_0} & \mathbf{K}_{\phi_0\phi_0} & \mathbf{K}_{\phi_0\phi_1} \\ \mathbf{K}^{\mathbf{T}}_{u\phi_1} & \mathbf{K}^{\mathbf{T}}_{\phi_i\phi_1} & \mathbf{K}_{\phi_0\phi_1} & \mathbf{K}_{\phi_1\phi_1} \end{pmatrix} \begin{pmatrix} \mathbf{u} \\ \boldsymbol{\phi}_i \\ \boldsymbol{\phi}_0 \\ \boldsymbol{\phi}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{F} - \mathbf{K}_{u\theta}\boldsymbol{\theta} \\ -\mathbf{K}_{\phi\theta}\boldsymbol{\theta} \\ \mathbf{Q}_0 \\ \mathbf{Q}_1 \end{pmatrix}$$

with $\mathbf{u}$, $\boldsymbol{\phi}_i$, $\boldsymbol{\phi}_0$ respectively the elastic displacements vector, internal electric potentials vector and the neutral reference potentials. Since $\boldsymbol{\phi}_0 = 0, \boldsymbol{\phi}_1$ are known, the system above is reduced to

$$\begin{pmatrix} \mathbf{M}_{uu} & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} \ddot{\mathbf{u}} \\ \ddot{\phi}_i \end{pmatrix} + \begin{pmatrix} \mathbf{K}_{uu} & \mathbf{K}_{u\phi_i} \\ \mathbf{K}^{\mathrm{T}}_{u\phi_i} & \mathbf{K}_{\phi_i\phi_i} \end{pmatrix}\begin{pmatrix} \mathbf{u} \\ \phi_i \end{pmatrix} = \begin{pmatrix} \mathbf{F} - \mathbf{K}_{u\theta}\theta - \mathbf{K}_{u\phi_1}\phi_1 \\ -\mathbf{K}_{\phi\theta}\theta - \mathbf{K}_{\phi_i\phi_1}\phi_1 \end{pmatrix} = \begin{pmatrix} \mathbf{F}_u \\ \mathbf{F}_\phi \end{pmatrix}.$$

We may eliminate the second equation using $\mathbf{K}_{\phi_i\phi_i}\phi_i = \mathbf{F}_\phi - \mathbf{K}^{\mathrm{T}}_{u\phi_i}\mathbf{u}$ leading to the final system

$$\mathbf{M}_{uu}\ddot{\mathbf{u}} + (\mathbf{K}_{uu} - \mathbf{K}_{u\phi_i}\mathbf{K}^{-1}_{\phi_i\phi_i}\mathbf{K}^{\mathrm{T}}_{u\phi_i})\mathbf{u} = \mathbf{F}_u - \mathbf{K}_{u\phi_i}\mathbf{K}^{-1}_{\phi_i\phi_i}$$

We remark that because of the difference of magnitude between elastic $\mathbf{u}$ and electric ϕ fields, condensation is preceded by an inverse diagonal scaling.

Plots of vertical displacement of all Q points at the periphery of the motor during the starting phase are given in Figure 16. We remark that the system requires a time delay of nearly 3×10^{-4} sec to reach stationary vibration under the piezoelectric excitation. Finally, in Figure 17 we plot the vertical displacement evolution in time for one Q point.

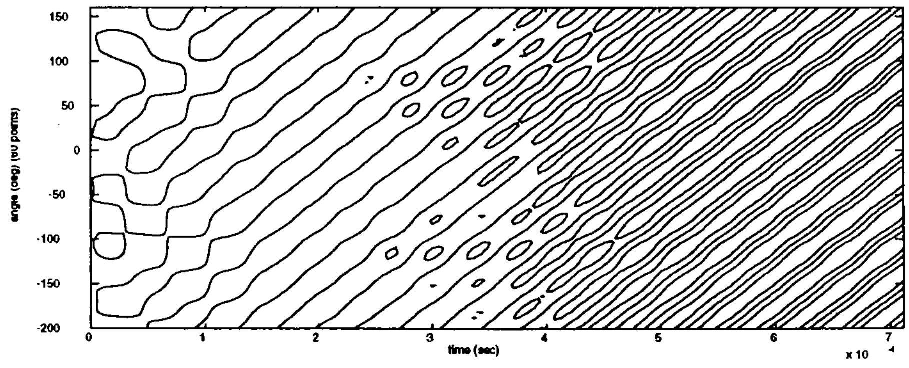


Figure 16. *Vertical displacement variations (time) isovalues of all Q points at the periphery during the starting phase*

9. Concluding Remarks

The architecture of a new finite element software built around the idea of a powerful commands interpreter has been presented. A specialized high-level language for describing data and computations has been developed. The program has been written following object-oriented programming techniques and using the C++ programming language.

The program has demonstrated to be able to adapt quite easily to a large variety of applications. People involved in the project is working in such different topics as thermoviscoplastic analysis, piezoelectricity, cable dynamics, structural optimization, elasto accoustic coupling and modal updating methods. The program conception has shown to be flexible enough to handle all these modeling problems without troubles.

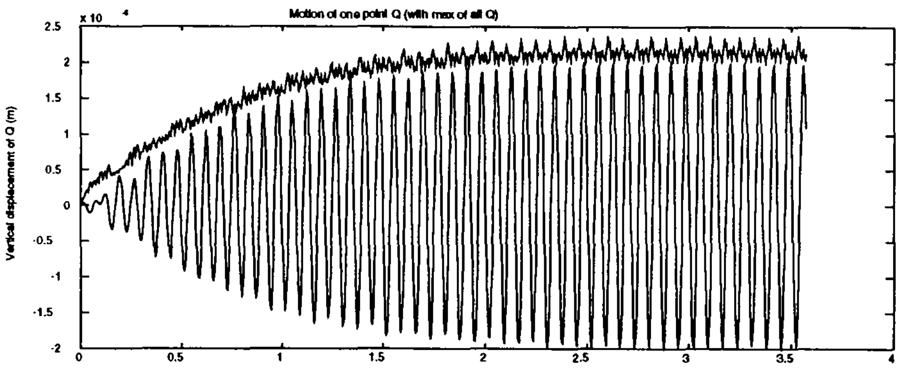


Figure 17. *Time evolution of vertical displacement of one Q point together with plot of the maximum value of vertical displacement at the periphery*

This flexibility in application was made possible thanks to the clear modularization that can be reached in object-oriented programming, with a marked separation of functionalities. Extensibility and reusability features of object oriented programming are clearly shown off.

The interpreter has evidenced to be of extreme utility to users in by introducing new functionalities and algorithms to the environment. Complex computations which may require entirely reviewing a standard program have been introduced without pain in OoFeLie. Users have been able to naturally incorporate new features, enlarging the program's capabilities.

Examples illustrating the functionalities of the program have been shown. In particular, an example of structural optimization and a second example concerning the modelization of the rotor of a piezoelectric motor have been developed.

Acknowledgements

The first and third authors acknoledge the partial supports of the Federal Services for Technical and Cultural Affairs (SSTC) under contract PAI/P3-050 and of the Ministry of Research and Technology of the "Région Wallonne" to perform this research.

The second author acknowledges also the support of the Belgian FNRS for providing the financial support of his visits to the University of Liège.

References

[Amm91] L. AMMERAAL, *C++ for Programmers.* John Wiley & Sons, 1991.

[BN94] J. BARTON, L. NACKMAN, *Scientific and Engineering C++.* Addison Wesley, 1994.

[Boo94] G. BOOCH, *Object Oriented Analysis and Design.* 1994.

[CKG94] A. CARDONA, I. KLAPKA, M. GERADIN, Design of a new finite element programming environment. *Engineering Computations*, 11:365-381, 1994.

[CY91] P. COAD, E. YOURDON, *Object-Oriented Design*. Yourdon Press Computing Series, 1991.

[DEVLOO94] P.R.B. DEVLOO, Efficiency issues in an object-oriented programming environment. *Artificial Intelligence and Object-Oriented Approaches for Structural Engineering, Civil-Comp Press*, pages 147–151, 1994.

[DF92] P.R.B. DEVLOO, J.S. RODRIGUES ALVES FILHO, An object-oriented approach to finite element programming: a system independent windowing environment for developing interactive scientific programs. *Advances in Engineering Software*, 14:41–46, 1992.

[DLN$^+$94] J. DONGARRA, A. LUMDSDAINE, X. NIU, R. POZO, K. REMINGTON, Sparse matrix libraries in C++ for high performance architectures. In *Proceedings of the Second Annual Object-Oriented Numerics Conference (OON-SKI'94)*, pages 122–138, April 24–27, 1994.

[DPB90] Y. DUBOIS-PELERIN, P. BOMME, T. ZIMMERMAN, Application de la programmation orientée objet à la méthode des éléments finis - développement d'un logiciel pilote en Smalltalk. *IREM* - rapport interne 90/5, École Polytechnique Fédérale de Lausanne, 1990.

[FD91] J.S. RODRIGUES ALVES FILHO, P.R.B. DEVLOO, Object-oriented programming in scientific computations: the beginning of a new era. *Engineering Computations*, 8:81–87, 1991.

[FFS90] B.W.R. FORDE, R.O. FOSCHI, S.F. STIEMER, Object-oriented finite element analysis. *Computers and Structures*, 34:355–374, 1990.

[FR91] C. FARHAT, F.X.ROUX, A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 1205-1227, 1991.

[DS83] J.E.DENNIS, R.B.SCHNABEL, *Numerical Methods for Unconstrained Optimization and Non-linear Equations*. Prentice Hall Series, 1983.

[MAC92] R.I MACJKIE, Object oriented programming of the finite element method. *International Journal for Numerical Methods in Engineering*, 35:425–436, 1992.

[MIL91] G.R. MILLER, An object-oriented approach to structural analysis and design. *Computers and Structures*, 40:75–82, 1991.

[MR90] M. METCALF, J. REID, *Fortran 90 Explained*. Oxford Science Publications, 1990.

[OBR90] N. OLHOFF, M.P. BENDSOE, J.RASMUSSEN, On Cad-integrated structural topology and design optimization. *Report No 27, Inst. of Mech Eng, Aalborg University, Denmark*, 1990.

[ROBISON96A] Arch D. ROBISON, The abstraction penalty for small objects in C++. In *POOMA'96: The Parallel Object-Oriented Methods and Applications Conference*, February 28 - March 1 1996. Santa Fe, New Mexico.

[RUM91] James RUMBAUGH, *Object Oriented Modeling and Design*. 1991.

[STR91] Bjarne STROUSTRUP, *The C++ programming language*. Addison Wesley, 2nd Edition, 1991.

[VEL95] Todd VELDHUIZEN, Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May, 1995.

[ZDP91] T. ZIMMERMAN, Y. DUBOIS-PELERIN, Object-oriented finite element programming, i. governing principles; ii. a prototype program in Smalltalk. *LSC - internal report 91/2, École Polytechnique Fédérale de Lausanne*, 1991.

[ZDP92] T. ZIMMERMAN, Y. DUBOIS-PELERIN, The object-oriented approach to finite elements: concepts and implementations. *Proceedings of the First European Conference on Numerical Methods in Engineering, Brussels*, pages 865–870, 1992.

[ZH94] G.W. ZEGLINSKI, R.P.S. HAN, Object-oriented matrix classes for use in a finite element code using C++. *International Journal for Numerical Methods in Engineering*, 37:3921–3937, 1994.