Object-oriented design to automate a high order non-linear solver based on asymptotic numerical method

Arnaud Lejeune, Fabien Béchet, Hakim Boudaoud, Norman Mathieu, Michel Potier-Ferry

Object-oriented design to automate a high order non-linear solver based on asymptotic numerical method

Arnaud Lejeune [a,*], Fabien Béchet [b], Hakim Boudaoud [c], Norman Mathieu [d], Michel Potier-Ferry [d]

[a] Institut Femto-ST, UMR 6174, Département Méc'Appli, 24 rue de l'épitaphe, 25000 Besançon, France
[b] TEMPO, EA 4542 Université de Valenciennes et du Hainaut Cambrésis, 59313 Valenciennes, France
[c] Equipe de Recherche sur les Processus Innovatifs, Ecole Nationale Supérieure en Génie des Systèmes Industriels, 8, rue Bastien Lepage, BP 90647, 54010 Nancy, France [d] LEM3, Laboratoire d'Etude des Microstructures et de Mécanique des Matériaux, UMR 7239, Ile du Saulcy, 57045 Metz, France

The Manitoo library is devoted to the resolution of analytical non-linear problems using a high order method called asymptotic numerical method. We describe here the Object Oriented design of this library and especially the choices made to obtain a quite generic and flexible numerical solver.

Through classical examples, we present a comparison with some existing tools implemented in Matlab and Fortran 77.

1. Introduction

For about twenty years, many works on the resolution of non-linear problems using Asymptotic Numerical Method (ANM) have been proposed [1] and applied to a wide range of problems in fluid and solid mechanics. For instance, various applications concerning the design of marine structures [2], non-linear vibrations [3], sheet metal forming [4], biomechanics [5] or multi-scale instabilities [6] have been implemented in many research informatics tools using Fortran 77, Fortran 90 [7,8], Matlab [9,3] and also in an industrial code [10]. Within ANM, a solution branch is computed using series expansion. The step length of this branch is then defined a posteriori and it yields naturally automatic path following techniques. This automation of the non-linear computation is very important in many cases and especially for the numerical computation of physical problems involving instabilities [11–16]. The robustness of the algorithm is assessed for instance in [2], where thousand thin shell computations have been performed in order to predict thin shell buckling with random imperfections. In Computational Fluid Dynamics, the discretized problem often involves millions of degrees of freedom so parallel implementations are needed. The robustness and efficiency of ANM have been established to solve such large scale problems [17].

ANM needs high order derivatives that can be obtained by recurrence formulae. These recurrence formulae computation is easy for algebraically simple equations, as Navier–Stokes equations, but it becomes more intricate for many other physical problems [18]. A first answer to this question is the MANLAB software [9] which permits to solve generic problems with few unknowns, providing the problem written in a quadratic form. Another approach, called DIAMANT and based on Automatic Differentiation (AD) by operator overloading, has been recently proposed [19,20] and applied to small size academic problems in the Matlab and Fortran contexts [21,22]. Due to Object Oriented limitation of these languages [23], it seems then difficult to obtain a really generic, reusable and efficient ANM library. Moreover, as for most of informatics libraries, and based on a 15 years old experience, we know that the ANM library mainly requires maintainability, a wide extensibility and portability. Indeed, as mechanical models based on finite elements (or other approximation methods) often need a large number of degrees of freedom, we have to consider the use of parallel computing.

Since the 90's, Object Oriented Programming has been commonly used to design complex scientific applications. As mentioned in the context of finite element method by [24], object oriented programming permits to develop numerical tools with portability on different computer architectures such as clusters, which was not possible with the prior sequential Fortran codes. Moreover,

* Corresponding author. Tel.: +33 381666024; fax: +33 381666700.
E-mail addresses: arnaud.lejeune@univ-fcomte.fr (A. Lejeune), fabien.bechet@univ-valenciennes.fr (F. Béchet), hakim.boudaoud@ensgsi.inpl-nancy.fr (H. Boud aoud), Norman.mathieu@univ-metz.fr (N. Mathieu), michel.potierferry@univ-metz.fr (M. Potier-Ferry).

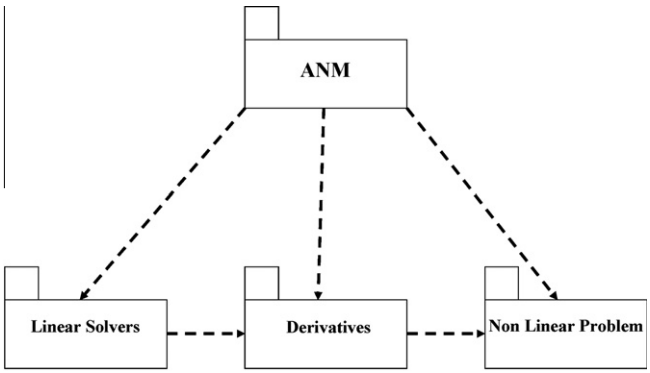

Fig. 1. Analysis and model separation.

inheritance and polymorphism are important characteristics to obtain a quite generic solver. One could argue for the loss of performance due to the overhead of oriented object implementation. Such a loss is not so obvious and Veldhuizhen demonstrates that C++ could surperform Fortran [25,26].

Despite of the propagation of object oriented and C++ programming in the field of numerical simulation of engineering problems, the ANM community has not developed an up-to-date numerical tool.

Considering all these remarks, we have been developing a new C++ library, called MANITOO which is devoted to the resolution of non-linear problems with ANM, since 2008. The main goal of this tool is to reduce development costs without losing computational performance compared to the former library developed in Fortran 77. Here we propose to deal with the object oriented design of Manitoo which has never been published and is about to be mature.

In the second part, we make a description of the ANM and present the corresponding algorithms while the third part is devoted to the Object Oriented design of the library. The fourth part shows some applications and comparisons with existing tools. We conclude with some remarks on future developments.

2. General structure

ANM consists in solving an analytical non-linear problem with a path-following (or continuation) method associated with a high order perturbation technique. First, unknowns are expanded in Taylor series with respect to a scalar path parameter. Then the non-linear problem, as a problem depending on unknowns, is also expanded in Taylor series leading to a system of linear equations at each order. Expressing high order equations with respect to Taylor coefficients requires a hard programming work. Coupling ANM with AD (namely the DIAMANT approach) allows to automate the computation of Taylor series terms in a peculiar direction using well-known recurrence formulae.

As in [27] and to obtain a generic library, the model describing the non-linear problem is distinguished from the analysis. Moreover, we split the analysis in ANM solver and linear solver.

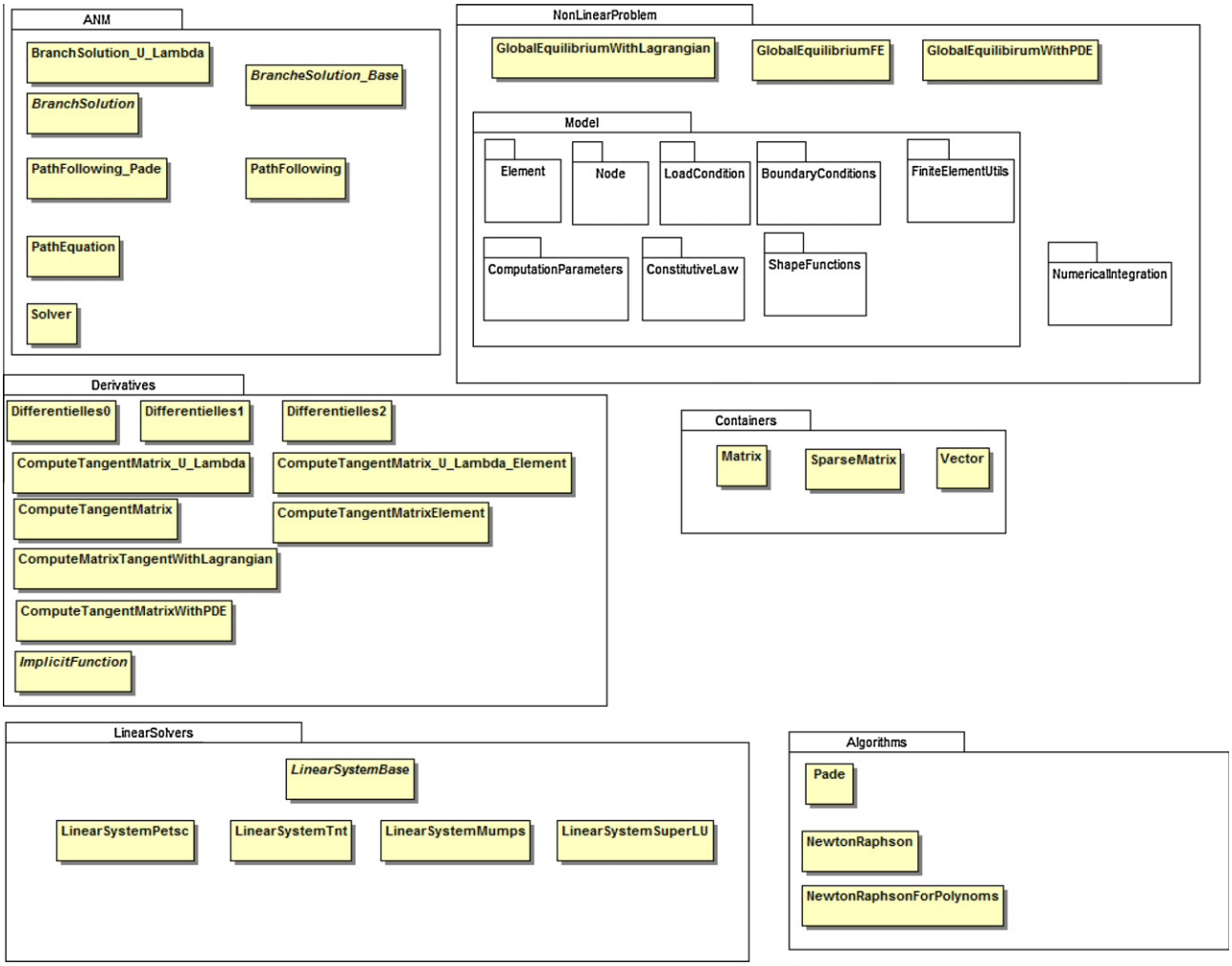

Fig. 2. Some packages and classes of the MANITOO Library.

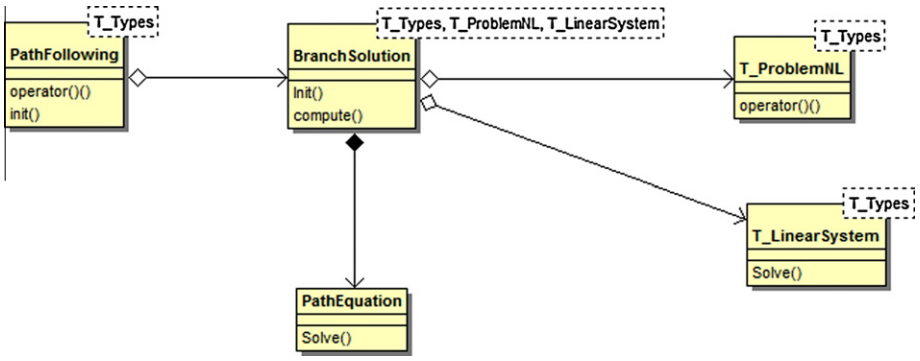

Fig. 3. ANM class diagram.

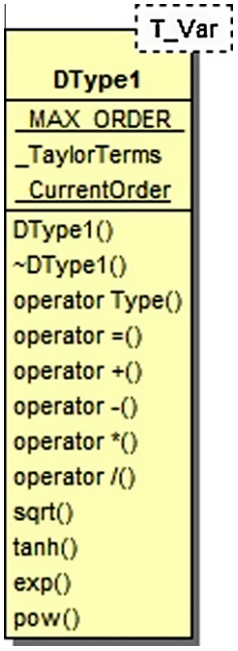

Fig. 4. A class for Taylor series variables.

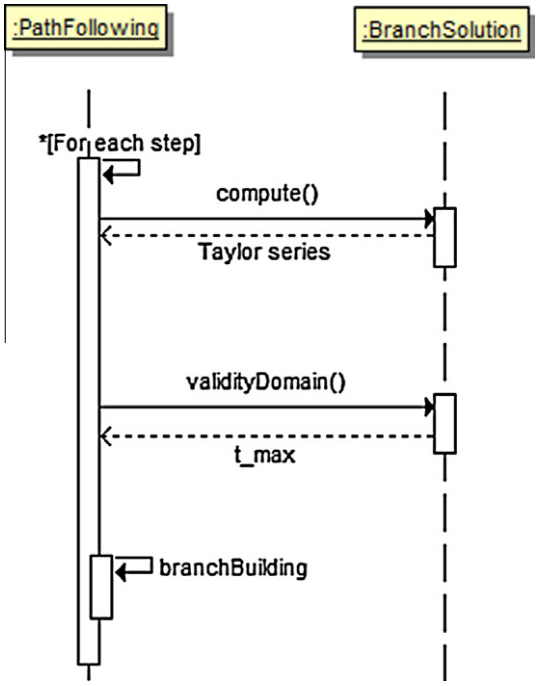

Fig. 5. Continuation sequence diagram.

We focus here on the design of the ANM solver therefore few indications about model and linear solvers integration will be given.

2.1. Asymptotic numerical method

2.1.1. General overview

First works concerning Asymptotic Numerical Method (ANM) referred it as the computation of a solution branch of a non-linear analytical problem [1]. It has been extended by [28] into a continuation method resulting in an assembly of successive solution branches. We are using the term of ANM as an alias of *path following technique via an asymptotic numerical method based on power series expansion* in the sense of [28].

Here is a brief overview of an ANM version summarized from [29]. More interested users may refer to this book.

Let us consider a quite generic quasi-static problem expressed as:

Find (u, λ) in $U \times \mathbb{R}$ such as

$$R(u, \lambda) = 0 \tag{1}$$

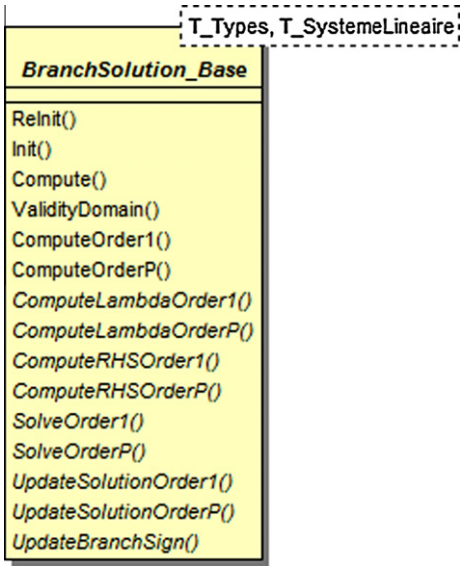

Fig. 6. Base class for branch solution computation.

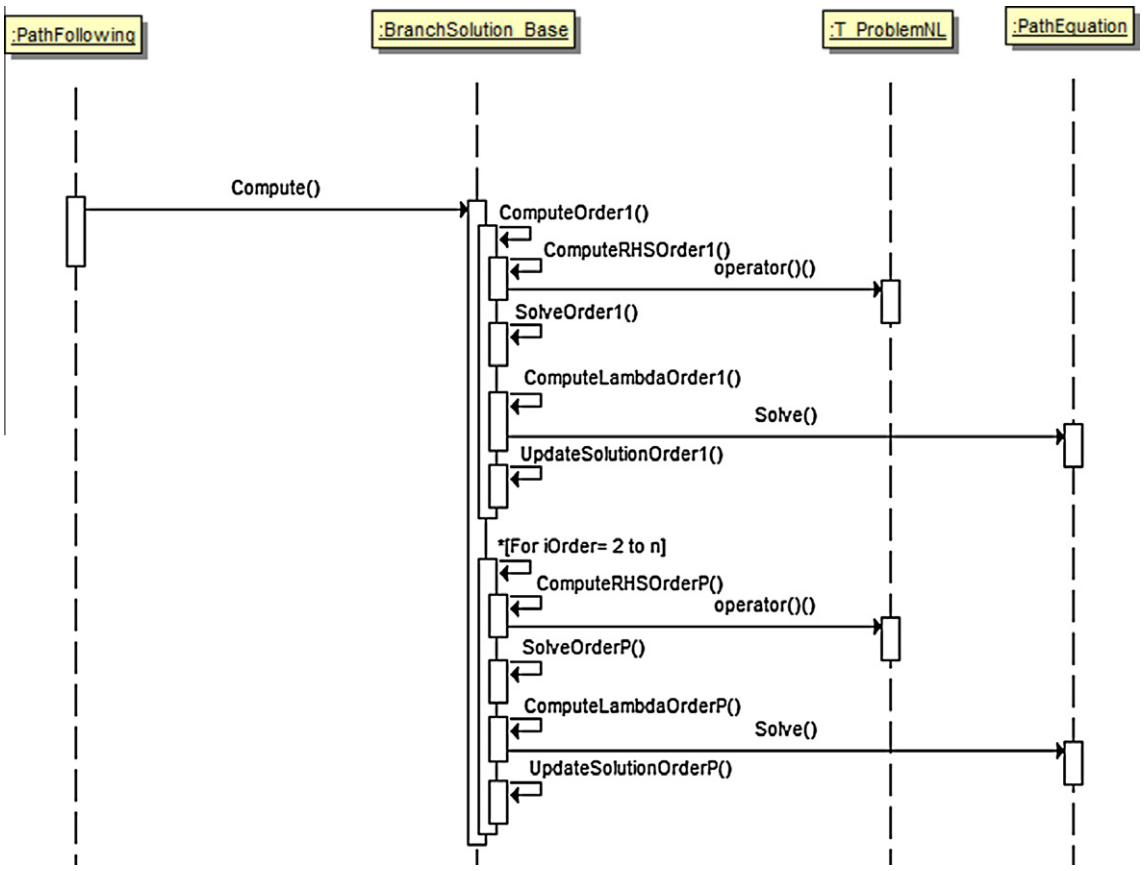

Fig. 7. Computing Taylor series terms up to an order n.

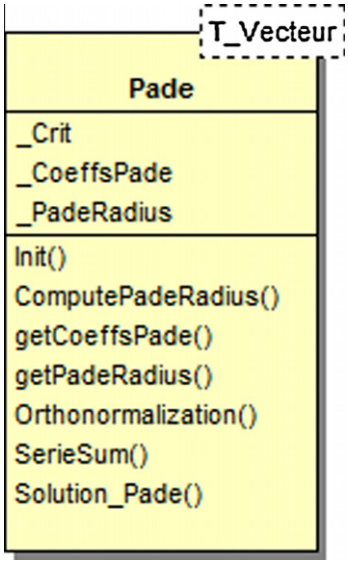

Fig. 8. Simplified Padé class.

with U a convex domain of finite dimension and R a non-linear analytical function.

Damil et al. [1] used a high order perturbation technique to solve this kind of problem by looking for solutions (u, λ) as a function of a parameter denoted by t close to a given solution (u_0, λ_0):

$$u(t) - u_0 = \sum_{i=1}^{n} t^i u_i, \;\; \lambda(t) - \lambda_0 = \sum_{i=1}^{n} t^i \lambda_i.$$

This can easily be related to the power series expansion and Taylor series terms of u and λ expressed as functions of the parameter t:

$$R(u(t), \lambda(t)) = R_0 + tR_1 + t^2 R_2 + \ldots t^n R_n$$

with

$$R_0 = R(u_0, \lambda_0) = 0$$
$$R_1 = R_{,u} u_1 + R_{,\lambda} \lambda_1$$
$$R_2 = R_{,u} u_2 + R_{,\lambda} \lambda_2 + R_{,uu} u_1 . u_1 + R_{,\lambda\lambda} \lambda_1^2 + R_{,u\lambda} \lambda_1 u_1$$
$$\vdots$$
$$R_n = R_{,u} u_n + R_{,\lambda} \lambda_n + R_n^{nl}(u_0, u_1, \ldots, u_{n-1}; \lambda_0, \lambda_1, \ldots, \lambda_{n-1})$$

where $R_{,u} = \frac{\partial R}{\partial u}(u_0, \lambda_0)$, $R_{,\lambda} = \frac{\partial R}{\partial \lambda}(u_0, \lambda_0)$, $R_{,uu} = \frac{\partial^2 R}{\partial u^2}(u_0, \lambda_0)$, $R_{,u\lambda} = \frac{\partial^2 R}{\partial u \partial \lambda}(u_0, \lambda_0)$, $R_{,\lambda\lambda} = \frac{\partial^2 R}{\partial \lambda^2}(u_0, \lambda_0)$.

Hence the computation of a solution branch is equivalent to the computation of the expansion coefficients. The latter are obviously related to the high order derivatives of the analytical function R. Every derivative R_k can be split in a linear part depending on the unknowns (u_k, λ_k) and a non-linear part depending on the previously computed terms. To compute a solution branch up to a given order n, one has to solve the following successive linear problems:

$$L_T^u(u_0, \lambda_0) u_1 + L_T^\lambda(u_0, \lambda_0) \lambda_1 = 0$$
$$L_T^u(u_0, \lambda_0) u_2 + L_T^\lambda(u_0, \lambda_0) \lambda_2 = -R_2^{nl}(u_0, u_1; \lambda_0, \lambda_1)$$
$$\vdots$$
$$L_T^u(u_0, \lambda_0) u_n + L_T^\lambda(u_0, \lambda_0) \lambda_n = -R_n^{nl}(u_0, u_1, \ldots, u_{n-1}; \lambda_0, \lambda_1, \ldots, \lambda_{n-1})$$

with $L_T^u = \frac{\partial R}{\partial u}$ the tangent operator with respect to u, $L_T^\lambda = \frac{\partial R}{\partial \lambda}$ the tangent operator with respect to λ and R_n^{nl} the non-linear part of R_n.

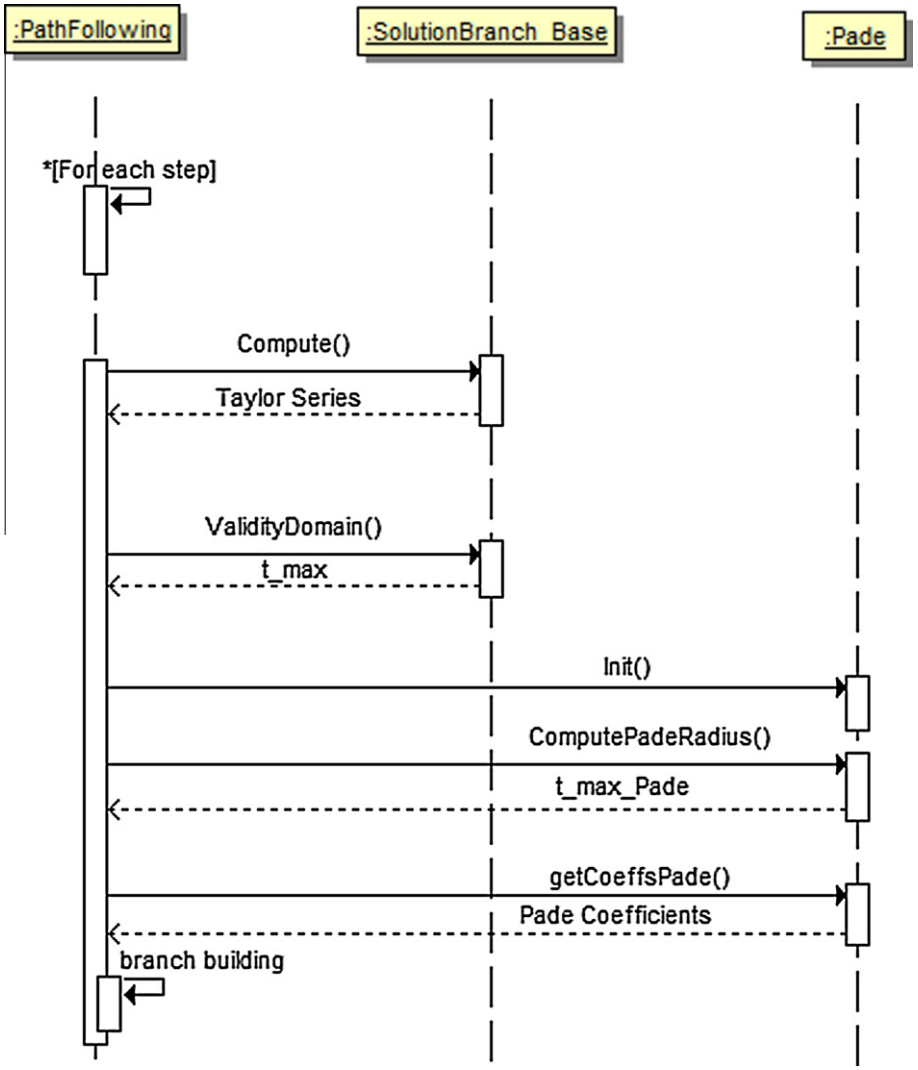

Fig. 9. Modified continuation sequence diagram with Pade add-in.

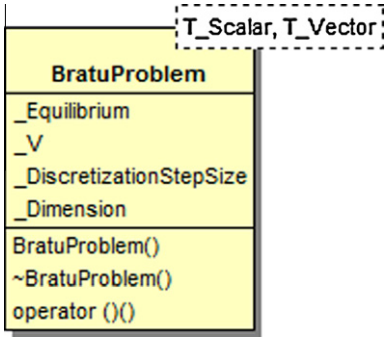

Fig. 10. BratuProblem class.

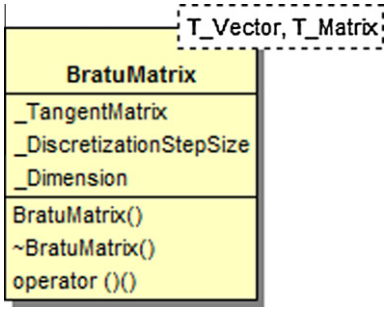

Fig. 11. BratuMatrix class.

As this set of equations is subdertermined, a path-parameter equation is added and can be expressed in a quite generic form:

$$t = < u(t) - u_0, u_1 >_u + \alpha_{arc}(\lambda(t) - \lambda_0, \lambda_1) \quad (2)$$

where $\langle \cdot, \cdot \rangle_u$ is a bilinear form defined in U and α_{arc} is a positive coefficient. Many other parameters are possible, see for instance [30]. A complete discussion on this point can be found in [31].

Usually, the classical dot product is applied and α_{arc} equals 1. A change in the bilinear form and the weight leads to a different path drive.

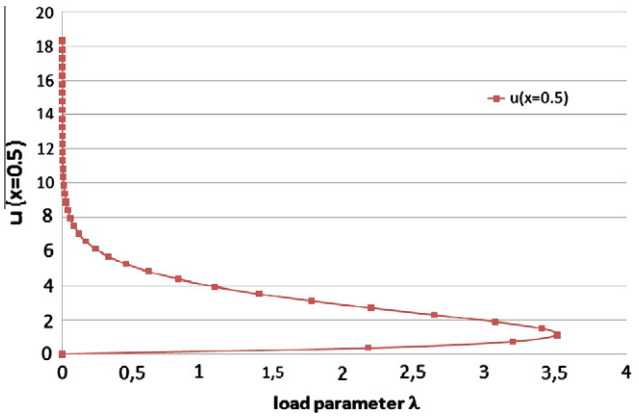

Fig. 12. Bratu solution at the point $x = 0.5$.

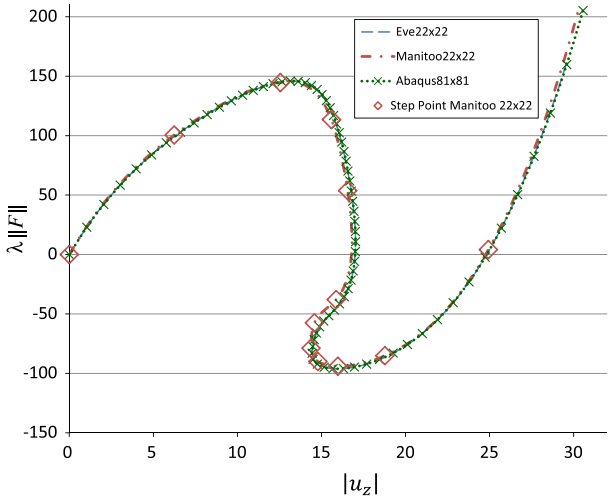

Fig. 13. Applied force (in N) versus vertical displacement (in mm) at the point of external force application.

Table 1
Total and linear solver computation time for sparse matrix.

DoF number	Matlab		Manitoo manual		Manitoo AD	
	Total	L. solver	Total	L. solver	Total	L. solver
5e3	**1.58**	0.42	**0.26**	0.05	**0.67**	0.05
7.5e3	**2.39**	0.56	**0.41**	0.16	**1.15**	0.11
1e4	**2.68**	0.73	**0.50**	0.09	**1.61**	0.12
5e4	**10.95**	2.53	**3.00**	0.748	**8.22**	0.793
1e5	**23.98**	5.09	**6.44**	1.64	**16.46**	1.73
2e5	**41.68**	9.31	**12.51**	3.36	**33.56**	3.54
2.5e5	**51.04**	11.10	**16.18**	4.52	**41.54**	4.57
5e5	**94.94**	21.72	**33.13**	9.10	**85.46**	9.91
1e6	**198.82**	45.60	**64.24**	18.41		

The derivatives of the path parameter Eq. (2) are given by:

$$\langle u_1, u_1 \rangle_u + \alpha_{arc}\lambda_1^2 = 1$$
$$\langle u_2, u_1 \rangle_u + \alpha_{arc}\lambda_2\lambda_1 = 0$$
$$\vdots$$
$$\langle u_n, u_1 \rangle_u + \alpha_{arc}\lambda_n\lambda_1 = 0$$

Once the series terms are obtained, on evaluates the interval, namely the validity domain, in which the series converge. Hence the length of the solution branch is known and the end point of this branch is the start point of the next branch.

The Algorithm 1 summarizes the resolution of the non-linear analytical problem 1 with ANM and continuation.

Algorithm 1. Global ANM algorithm

1: Initialize u_0 and λ_0
2: **for** *iStep* = 1 to *NbStep* **do**
3: Compute linear operators L_T^u and L_T^λ
4: Solve $L_T^u(u_0, \lambda_0)\hat{u}_1 = -L_T^\lambda(u_0, \lambda_0)$
5: Compute $\lambda_1 = \pm \frac{1}{\sqrt{\alpha_{arc} + <\hat{u}_1, \hat{u}_1>_u}}$
6: Compute $u_1 = \lambda_1 \hat{u}_1$
7: **for** $i = 2$ to n **do**
8: Compute high order term R_i^{nl}
9: Solve $L_T^u(u_0, \lambda_0)\hat{u}_i = -R_i^{nl}$
10: Compute $\lambda_i = -\lambda_1^2 < \hat{u}_i, \hat{u}_1 >_u$
11: $u_i = \hat{u}_i + \lambda_i \hat{u}_1$
12: **end for**
13: Using terms $(u_i)_{i=1,\ldots,n}$ and $(\lambda_i)_{i=1,\ldots,n}$ compute the validity domain upper bound t_{max}
14: $u_{end} = \sum_{i=0}^{n} u_i t_{max}^i$ and $\lambda_{end} = \sum_{i=0}^{n} \lambda_i t_{max}^i$
15: $u_0 = u_{end}$ and $\lambda_0 = \lambda_{end}$
16: **end for**

2.1.2. Advanced feature: Padé approximants

Padé approximants have been introduced in the context of ANM by [32]. Their efficiency has been shwon in terms of extending the validity domain and consequently on reducing the number of branch computation steps [33,29].

Let $(U_k)_{k=0,\ldots,n}$ and $(\lambda_k)_{k=0,\ldots,n}$ be series obtained by solving an analytical non-linear problem using ANM. From these series, a validity domain can be a posteriori computed. Using Padé approximants allows to increase the length of the solution branch by building up an orthonormal basis via the Gram–Schmidt procedure. Starting from solutions given in the form:

$$U(t) = U_0 + \sum_{i=1}^{n} t^i U_i \tag{3}$$

$$\lambda(t) = \lambda_0 + \sum_{i=1}^{n} t^i \lambda_i, \tag{4}$$

one can obtain a new representation by rational fraction of the solution path:

$$U_{pade}(t) = U_0 + \sum_{i=1}^{n} t^i \frac{\Delta_{n-i}(t)}{\Delta_n(t)} U_i \tag{6}$$

$$\lambda_{pade}(t) = \lambda_0 + \sum_{i=1}^{n} t^i \frac{\Delta_{n-i}(t)}{\Delta_n(t)} \lambda_i \tag{7}$$

where the expression of $\Delta_i(t)$ is given in [33,29].

Accounting for Padé approximants results in adding an external plug after a series computation. The increase of computation time due to this plug is, in most of cases, much smaller than the benefit due to the increase of the length of the solution branch. The global Algorithm 1 is then replaced by Algorithm 2.

Algorithm 2. Global ANM algorithm with Padé approximants

1: Initialize u_0 and λ_0
2: **for** *iStep* = 1 to *NbStep* **do**
3: Compute linear operators L_T^u and L_T^λ
4: Solve $L_T^u(u_0,\lambda_0)\hat{u}_1 = -L_T^\lambda(u_0,\lambda_0)$
5: Compute $\lambda_1 = \pm\frac{1}{\sqrt{\alpha_{arc}+<\hat{u}_1,\hat{u}_1>_u}}$
6: Compute $u_1 = \lambda_1\hat{u}_1$
7: **for** i = 2 to n **do**
8: Compute high order term R_i^{nl}
9: Solve $L_T^u(u_0,\lambda_0)\hat{u}_i = -R_i^{nl}$
10: Compute $\lambda_i = -\lambda_1^2 < \hat{u}_i,\hat{u}_1>_u$
11: $u_i = \hat{u}_i + \lambda_i\hat{u}_1$
12: **end for**
13: Using terms $(u_i)_{i=1,\ldots,n}$ and $(\lambda_i)_{i=1,\ldots,n}$ compute the validity domain upper bound t_{max}
14: Compute rational factors $(\Delta_i)_{i=1,\ldots,n}$ and associated Padé coefficients V_i^{Pade}
15: Compute the Padé validity domain t_{max}^{Pade}
16: $u_{end} = \sum_{i=0}^n u_i {t_{max}^{Pade}}^i V_i^{Pade}$ and $\lambda_{end} = \sum_{i=0}^n \lambda_i {t_{max}^{Pade}}^i V_i^{Pade}$
17: $u_0 = u_{end}$ and $\lambda_0 = \lambda_{end}$
18: **end for**

2.2. Global architecture

A previous work on Object Oriented implementation of a path-following method devoted to structural mechanics has already been proposed [34]. Our new library, called MANITOO, applies to a wider kind of analytical non-linear problems. Moreover we attempt to define a more generic implementation of ANM which is also a path-following method.

The MANITOO library needs to consider the overall ANM process: Taylor terms, validity domain computation and continuation technique.

From the previous section, it seems obvious to split the ANM in several components:

- Path-following process
- Branch solution building
- Linear system solving
- Non-linear problem definition
- Derivatives computation (optional).

Only the first two parts are really specific to the ANM. The way of computing high order derivatives, defining the model and solving linear systems should not influence the ANM process for a given kind of non-linear problems. Then a generic ANM algorithm can be defined by Algorithms 3 and 4.

The path-following flow consists in computing successive solution branches up to a given number of steps or an optional loop-breaking condition (Algorithm 3) while the solution branch is built by Algorithm 4.

Algorithm 3. ANM algorithm: basic continuation process

1: Initialize u_0 and λ_0
2: **for** *iStep* = 0 to *NbStep* **do**
3: $u_0 = u_{end}$ and $\lambda_0 = \lambda_{end}$
4: Compute high order terms $(u_k)_{k=1,\ldots,n}$ and $(\lambda_k)_{k=1,\ldots,n}$
5: Compute the validity domain upper bound t_{max}
6: $u_{end} = \sum_{i=0}^n u_i t_{max}^i$ and $\lambda_{end} = \sum_{i=0}^n \lambda_i t_{max}^i$
7: **end for**

Algorithm 4. ANM algorithm: computation of the high order terms $(u_k)_{k=1,\ldots,n}$ and $(\lambda_k)_{k=1,\ldots,n}$

1: Initialize u_0 and λ_0
2: Compute linear operators L_T^u and L_T^λ
3: Solve order 1: u_1 and λ_1
4: **for** k = 2 to *NbOrder* **do**
5: Compute right hand side R_k^{nl}
6: Solve $L_T^u\hat{u}_k = -R_k^{nl}$
7: Compute λ_k and u_k
8: **end for**

So we distinguish the analysis (ANM specific features) from the model (non-linear problem expression), the way of differentiation (derivatives terms) and the linear solver. This leads to four packages related as in Fig. 1.

2.3. Finite Element Method

In the context of solid mechanics, the Finite Element Method (FEM) is one of the most popular numerical method to approximate the solution of partial differential equations. This method is here used to model an approximation of the non-linear problem.

It is quite impossible to establish an exhaustive list of all existing Object Oriented (OO) libraries dealing with Finite Element Methods (FEMs) or other discretization methods. A bibliography has been realized until 2003 in [24] and shows the amount of works of FEM and OO.

Thus, behind the term of "Finite Element library", one could find several kind of libraries. Some of them are related to the mesh building and entity organization (AOMD [35], libmesh [36]), some of them are just related to the implementation of the computational method (OOfem [37], getfem++ [38]), others provide an upper interface and pseudo-language so that the user only has to define his problem on a strong or variational form (freefem++ [39], Fenics [40]), etc.

The library catalog of Object Oriented Finite Element is really wide so we hoped to find a convenient one for our applications. However, in the context of continuum mechanics, we want to deal with linear and quadratic 2D and 3D elements. This first requirement reduces the number of potential candidates. A second requirement is to use a really specific shell element from Büchter et al. works [41] which gives good results for very thin shells. With this second requirement, at our knowledge, the potential candidate space is a null one. Thus we decided to develop an Object Oriented Finite Element Model with basic features like iso-parametric interpolation, numerical integration, element to global assembly, etc. Our model may not offer the same level of advanced functionality as in [37,42–44] and, consequently, will not be detailed in this paper. The 2D and 3D models are quite similar (it just results in changing a template argument in most cases) while dealing with the cited shell elements needs the introduction of new internal variables, condensation technique and tensor algebra. This results to a specific class considering models based on shell elements.

2.4. Linear solver

Solving a non-linear problem results in solving successive linear systems with the same tangent matrix and changing the right hand side. Wrappers to the following solvers have been implemented:

- JAMA [45]
- SuperLU [46]
- PetsC [47–49]
- Mumps [50,51].

As the tangent matrix does not change during the successive system resolutions, we use features of these solvers to optimize the computation cost of the linear system resolution (please refer to user manuals of these libraries to get more details).

Tnt library is used for all our validation tests with small degrees of freedom and a dense linear matrix.

Three kinds of sparse matrices storage are defined to use SuperLU, PetsC, and Mumps.

2.5. Automatic Differentiation (AD)

As noted in the ANM overview, solving a problem with this method results in computing high order derivatives in a peculiar direction. A good way to automate the ANM solver is to use automatic differentiation as a pocket calculator. In this sense, an approach, named DIAMANT, coupling AD and ANM has been proposed in [19] and applied to academic problems [20]. Then right-hand side terms and tangent matrix occurring in ANM are obtained by application of AD technique. It was already an improvement avoiding the manual calculation of terms which could be a quite difficult task.

Following the definition of [52], AD can be viewed as the way to generate a code which computes the derivative of a function itself given by a code. A natural approach is to use the forward mode by applying operator overloading. This is notably well-suited with C++ language [53]. This language enables to overload classical operators in a simple way [54]. Notice that AD by operator overloading has already been implemented for a long time in other classical languages [55].

Interested readers may refer to [56] for a review of AD techniques. But in most cases, works are concerned with low order derivatives (first or second order). We are here concerned with high order derivatives in direct mode as in [57] except that we only need to compute the derivatives in a peculiar direction (i.e. the Taylor series terms of a given variable or function).

From an informatics point of view, forward differentiation by operator overloading is realized using template facilities of C++ language. First, the function to be differentiated is implemented and tested in the continuous domain, after that the template parameter is changed into the high order derivative one. Consequently, one single implementation allows the computation of residuals and derivatives. A specific procedure is applied in the case of local implicit functions because of the different nature between the continuous function (implicit) and the derivative function (explicit).

The AD feature is also applied to the computation of the linear tangent matrix, noted L_T^u in Section 2.1.1 arising from the first order differentiation of the non-linear problem.

3. Object oriented implementation

A non-exhaustive overview of the implemented classes, restricted to this paper focus and organized by packages, is given in Fig. 2.

In this section, we describe some of the techniques used in the library. The objective is to split the overall process into small size objects to obtain the higher flexibility and genericity.

As explained in Section 2.1.1, we want to apply an iterative process to the computation of algebraic functions. One important point is the way of considering mathematical/mechanical functions in our library. As mentioned in [58], function objects (also called functors) allow better optimization than function pointers in terms of computation time. However these functors have to be small enough to be inlined. They should also not be virtual. Functors are intensively used in this work to implement small functions. Moreover, to keep a reasonable computation time cost, we use meta-programming and limit the use of virtual functions.

Note that the library is implemented in french, so names of components, classes, operators have been translated for this paper.

Note also that all the UML diagrams have been built with Bouml freeware (http://bouml.free.fr).

3.1. Global class diagram

An ANM solver is a way of computing output data, usually named *results*, from input data given by a *Model* following a fixed algorithm. From the package view in Fig. 1, we focus on the ANM package. This package is made-off three main classes.

- The path-following class
- The branch solution class
- The path equation class.

An overall class diagram is illustrated in Fig. 3.

Referring to the class diagram in Fig. 3, user-defined class *T_ProblemNL* describes the non-linear problem and the computation of its high order derivatives (Taylor series coefficients). It is defined as a functor.

Using a templated class, here called *T_LinearSystem*, allows us to apply any solver of a set of linear equations.

Moreover, it seems obvious that we need to store any Taylor series terms and optionally to define devoted algebra rules to these data. Hence we defined a new data type.

3.2. Data Types

This subsection is concerned with data types of the Taylor Series variables (i.e. derivatives data type). As indicated in Section 2.5, automatic differentiation in forward mode could be done by overloading classical operators. So we would design new data type containing Taylor series coefficients and the corresponding arithmetic operators (and usual mathematics functions). An illustration of automatic differentiation by operator overloading is given in Section 3.6.

Moreover, solving non-linear problems in the field of continuum mechanics leads to using scalar variables and some containers related to linear algebra as vector or matrix. Here the term "vector" is related to the linear algebra term and not the STL term.

We make an intensive use of template ability offered by the C++ language so that a change in the scalar data type and in the container type could be done with only a few change in the library. To do so, we define a traits policy [59] containing:

- Scalar data type
- Vector data type
- Dense local matrix data type for computation with matrix of small size
- Linear tangent matrix data type which is usually a sparse one.

that could be implemented as:

```
template< class TypeVariable, class TypeVarDiff, class T_StiffnessMatrix >
struct Types
{
  // Type for non differentiate scalar variable
  typedef TypeVariable    T_Var;
  // Type for vector container of non differentiate variables
  typedef Vector< TypeVariable > T_Vector;
  // Type for small matrix container of non-differentiate variables
  typedef Matrix    T_Matrix;
  // Type for large matrix : Tangent, stiffness and mass matrix are here concerned.
      Variables are non differentiate ones.
  typedef T_StiffnessMatrix T_TangentMatrix;

  // Type for differentiate variables (i.e. Taylor Series variables)
  typedef TypeVarDiff     T_Diff;
  // Type for vector containter of differentiate variables
  typedef Vector< TypeVarDiff > T_VectorDiff;
};
```

Note that the type of the large matrices depends on the external linear solver. Some of them require Compressed Sparse Row matrix while others need Compressed Sparse Column, skyline or full storage. Moreover, we use our own implementation of vector and matrix container. Replacing by more efficient containers would be considered in future improvement of the library.

A prior feature of the differentiate data type is to contain all the terms of the series expansion. It has been chosen to define a data using static array. As a computation needs many calls to constructors/destructors, it is obvious that using dynamic allocator would lead to a loss of performance. The size of the array is defined using a macro variable. Taylor expansions are computed up to a given order less than the maximal array size.

Moreover, algebra could be added to the data class with Operator Overloading technique; this corresponds to an automatic differentiation in forward mode. Three classes of high order derivatives are available. Performances and features of these are described in [20]. As an illustration, a brief view of the class *DType1* corresponding to the first version of *DIAMANT* approach is given in Fig. 4.

Note that due to the size of the manipulated data, we avoid at most to make a copy of scalars, vectors and matrices. Only one tangent matrix is created.

3.3. Path-following

As mentioned before, continuation consists in computing the validity domain of the series expansion, building the end point of the branch, then using this point to start the computation of a new branch.

Moreover, the matrix computation object, the linear solver object and optionally the residual computation object are created at the initial step of continuation process.

The sequence Fig. 5 outlines the process of continuation.

3.4. Branch computation

Branch computation consists in finding terms of the Taylor series expansion (i.e. high order derivatives terms). Knowing the initial solution (U_0, λ_0), Taylor series terms up to an order n are obtained.

As for continuation, this class is perhaps the most generic. However it needs some specifications depending on the kind of non-linear problem. Then a virtual base class has been designed (Fig. 6). In term of computation time, this should not be dramatic. Indeed, on non-academic simulations with a large number of degrees of freedom, access to the virtual table, also called vtable, is really less time-consuming than solving each right hand side term at each order and linear tangent assembly.

The sequence diagram related to the computation of the series terms is given in Fig. 7. Applied class would be expressed in the next section dealing with examples.

3.5. Padé class

As highlighted in Section 2.1.2, Padé approximants could be designed as a computational class. A simplified version of this class is described in Fig. 8. Members named *Orthonormalization*, *SerieSum* and *Solution_Pade* are private ones used to compute the validity domain and coefficients related to Padé method.

The sequence Fig. 9 illustrates the easiness of including a posteriori computation of Padé approximants and the extended validity domain.

3.6. Formulations

Formulations describe the non-linear problem. It computes function results with data and parameters given in the model.

A formulation has to be defined as a functor class even if some of them are not so small. Then formulations classes need at least parenthesis *()* operator. However, we want to compute the function and its derivatives up to a given order. There are at least two ways of implementing the function operator. The first corresponds to the classical hand-written developments. Each algebraic operation is developed by hand and, sometimes, smart and really specific optimizations could be investigated before the computer implementation.

The second way is to apply operator overloading based on the fact that a function is a compound of elementary and well-known operators. This leads to compute high order terms of a given function using AD technique in implements only once adequate formulae. This "pocket calculator" ability allows us to obtain more genericity in our library. The user should only implement computation function and then, with just referring to the appropriate data-type, one could compute residual value or differentiate terms.

The designed ANM library is free from the way of computing high order terms so that each user could choose his own preference. One has just to define a formulation as a functor with a good data type. However, independently of the computational way of high order derivatives, it is strongly recommended to enable different kinds of data treated by the formulation. Hence, with changing the data type, one could compute high order derivatives or evaluate the residual of a given solution. As an illustration, the function $f(x, y) = x * y$ could be naively implemented with the following templated *ScalarProduct* class:

```
template < class T_Scalar, int T_Automatic = 1>
class ScalarProduct
{
  public :
    ScalarProduct() {};
    ~ScalarProduct() {};
    T_Scalar operator() (T_Scalar & x, T_Scalar & y){return (x*y); };
}

template < class T_Scalar>
class ScalarProduct<T_Scalar, 0>
{
  protected:
    const int Order;
  public :
    ScalarProduct(int order):Order(order) {};
    ~ScalarProduct() {};
    T_Scalar operator() (T_Scalar & x, T_Scalar & y){
    T_Scalar z;
    z[Order] = 0;
    for (int i=0, j= Order ; j>=0; ++i, --j)
        z[Order] += x[i]*y[j];
    return z; };
}

template < >
class ScalarProduct < double, 0>
{
  public :
    ScalarProduct() {};
    ~ScalarProduct() {};
    double operator() (double & x, double & y){
    return (x*y); };
}
```

It is quite easy to compute the function value in the continuous domain with automatic differentiation or with the hand-coded expansion:

```
  double z_AD, z_Hand;
  ScalarProduct<double> f_scalprod_AD;
  ScalarProduct<double,0> f_scalprod_Hand;
  z_AD = f_scalprod_AD(x,y);
  z_Hand = f_scalprod_Hand(x,y);
```

and its derivatives:

```
// We here suppose that T_Diff is the class describing derivative types with operator overloading
  T_Diff z_AD;
// We here suppose that T_Vector is a class containing data in a vector style with a "good" default size
  T_Vector z_Hand;
// We here suppose the computational order in knwon and given by the variable named "order"
  ScalarProduct< T_Diff > f_scalprod_AD;
  ScalarProduct< T_Vector,0> f_scalprod_Hand(order);

  z_AD = f_scalprod_AD(x,y);
  z_Hand = f_scalprod_Hand(x,y);
```

Switching between scalar and derivative computation amounts to changing the variable type. With AD, recurrence formulae are implemented in the *T_Diff* class with operator overloading ability. It seems obvious that a hand-coded version leads to separate operations on derivatives from data so that any formulations need to rewrite associated recurrence formulae.

Coupling the automatic computation of formulation with our quite generic version of ANM allows us to claim that we obtain a quite generic and automatic version of a non-linear solver based on an asymptotic numerical method. We briefly discuss the performances and illustrate the implementation of a new formulation in the incoming application part.

4. Applications

In this section, we propose to solve two kinds of problems with our library. Each problem has different mathematical features and illustrates flexibility, reusability and genericity of MANITOO.

Computations have been performed under a 64 bit Windows 7 operating system on a Lenovo x200 Tablet. The processor is an *Intel Core2 Duo L9400 @ 1.86 GHz* and 2.9 Go of RAM are available.

4.1. Academic example: Bratu problem

4.1.1. Problem formulation and discretization

This example was solved in the ANM context by [29]. It was the first problem solved with the here-concerned MANITOO object-oriented library. The initial formulation of the problem is given by:

Find (u, λ) defined on $\Gamma = [0, 1]$ such as

$$R(u,\lambda) = \begin{cases} u_{,xx} + \lambda e^u = 0 \\ u(0) = u(1) = 0 \end{cases}$$

or in its equivalent form Find (u, v, λ) defined on $\Gamma = [0,1]$ such as

$$R(u,v,\lambda) = \begin{cases} u_{,xx} + \lambda v = 0 \\ dv - v du = 0 \\ u(0) = u(1) = 0 \end{cases}$$

The second formulation of the Bratu problem has been introduced to simplify the expansion made by hand. Hence in [29], a new kind of generic problems has been designed with a main equation and a local one expressed in a differential form. Using AD enables to work with the first formulation without any introduction of new variable or specific smart expansion. In the following part, we will deal with the second formulation, as the first is less complicated.

Concerning the discretization and numerical approximation of second order derivative in the spatial domain, the finite difference method is used to compute approximate the solution (u_h, v_h, λ_h). The domain Γ is discretized with a constant step size h.

ANM is applied on this discretized problem given by:

$$R(u_h, v_h, \lambda_h) = \begin{pmatrix} v_0 - e^{u_0} \\ \frac{-2u_0+u_1}{h^2} + \lambda_h v_0 \\ v_1 - e^{u_1} \\ \frac{u_0-2u_1+u_2}{h^2} + \lambda_h v_1 \\ v_2 - e^{u_2} \\ \frac{u_1-2u_2+u_3}{h^2} + \lambda_h v_2 \\ \vdots \\ v_{n-2} - e^{u_{n-2}} \\ \frac{u_{n-3}-2u_{n-2}+u_{n-1}}{h^2} + \lambda_h v_{n-2} \\ v_{n-1} - e^{u_{n-1}} \\ \frac{u_{n-2}-2u_{n-1}+u_n}{h^2} + \lambda_h v_{n-1} \\ v_n - e^{u_n} \\ \frac{u_{n-1}-2u_n}{h^2} + \lambda_h v_n \end{pmatrix} = \vec{0} \tag{9}$$

Computing the tangent matrix respect to u_h is obvious:

$$L_T^u(u_h, v_h, \lambda_h) = \begin{pmatrix} -\frac{2}{h^2} + \lambda_h v_0 & \frac{1}{h^2} & 0 & & \dots & & 0 \\ \frac{1}{h^2} & \ddots & \ddots & & & & \\ 0 & \ddots & \ddots & \frac{1}{h^2} & & \ddots & \vdots \\ & & \frac{1}{h^2} & -\frac{2}{h^2} + \lambda_h v_i & \frac{1}{h^2} & & \\ \vdots & & \ddots & \frac{1}{h^2} & \ddots & \ddots & 0 \\ & & & & \ddots & \ddots & \frac{1}{h^2} \\ 0 & & \dots & & 0 & \frac{1}{h^2} & -\frac{2}{h^2} + \lambda_h v_n \end{pmatrix} \tag{10}$$

4.1.2. ANM

The unknowns are sought in the form:

$$u(t) = u_0 + \sum_{i=1}^{n} t^i u_i$$

$$v(t) = v_0 + \sum_{i=1}^{n} t^i v_i$$

$$\lambda(t) = \lambda_0 + \sum_{i=1}^{n} t^i \lambda_i$$

and the path-parameter is defined as:

$$t = < u(t) - u_0, u_1 > + (\lambda(t) - \lambda_0)\lambda_1$$

This problem could then be solved using Algorithm 5. At each order, the right hand side comes from the contribution of non-linearities, and then, in this problem, to the computation of the product λv. Note that the variable v is not really an unknown but an internal variable (also called condensation variable). This variable is used for continuation at each branch computation. The algorithm could be improved for a hand-written differentiation (refer to Section 2.2 of [29]) but here is not the purpose.

Algorithm 5. ANM algorithm with condensation variable

1: Initialize u_0, $v_0 = e^{u_0}$ and λ_0
2: **for** *iStep* = 1 to *NbStep* **do**
3: Compute linear operators $L_T^u(v_0, \lambda_0)$ and $L_T^\lambda(v_0)$
4: Solve $L_T^u(v_0, \lambda_0)\hat{u}_1 = -L_T^\lambda(v_0)$
5: Compute $\lambda_1 = \pm \frac{1}{\sqrt{\alpha_{arc} + \langle \hat{u}_1, \hat{u}_1 \rangle_u}}$
6: Compute $u_1 = \lambda_1 \hat{u}_1$
7: Compute $v_1 = v_0 u_1$
8: **for** $i = 2$ to n **do**
9: Compute high order term R_i^{nl}
10: Solve $L_T^u(v_0, \lambda_0)\hat{u}_i = -R_i^{nl}$
11: Compute $\lambda_i = -\lambda_1^2 \langle \hat{u}_i, \hat{u}_1 \rangle_u$
12: $u_i = \hat{u}_i + \lambda_i \hat{u}_1$
13: Compute $v_i = v_0 u_i + \sum_{l=1}^{i-1} \frac{l+1}{i} u_{l+1} v_{i-1-l}$
14: **end for**
15: Using terms $(u_i)_{i=1,\ldots,n}$ and $(\lambda_i)_{i=1,\ldots,n}$ compute the validity domain upper bound t_{max}
16: $u_{end} = \sum_{i=0}^{n} u_i t_{max}^i$, $v_{end} = \sum_{i=0}^{n} v_i t_{max}^i$ and $\lambda_{end} = \sum_{i=0}^{n} \lambda_i t_{max}^i$
17: $u_0 = u_{end}, v_0 = v_{end}$ and $\lambda_0 = \lambda_{end}$
18: **end for**

4.1.3. Implementation

To implement the Bratu problem, we do not have to worry about condensation technique.

ANM is compound by a continuation class, a branch solution class, a linear solver class, the specific *BratuProblem* and *BratuMatrix* classes devoted to the computation of the function $R(u, \lambda)$, the high order derivatives and the tangent matrix with respect to the u variable. In this case AD is not used to compute the linear matrix.

The continuation, branch solution and linear solver classes are generic ones available in the library. As noticed from subSection 4.1.1, the Bratu problem is of the form $R(U, \lambda)$ with non-linearities in (U, λ). Hence we use the *BranchSolution_U_Lambda* class. For this example we decide to use the SuperLU linear solver. The corresponding sparse matrix storage is defined to build the tangent matrix.

In the following, we give an implementation example of *BratuProblem* and *BratuMatrix* classes described in the diagram Figs. 10 and 11. *BratuProblem* class is an implementation of the discretized problem (9) while *BratuMatrix* is an implementation of the tangent matrix computation (10) (without condensation technique). Results are obtained with a call to the operators parenthesis returning a reference on a *T_Vector* in case of equilibrium computation (*BratuProblem*):

```
template < class T_Scalar, class T_Vector >
T_Vector & BratuProblem< T_Scalar, T_Vector >::operator () (const T_Vector & U, const
    T_Scalar & lambda)
{
  _Equilibrium.init(0.);
  for (int i=0; i < _Dimension; ++i)
    _V[i] = exp(U[i]);

  _Equilibrium[0] = -2. /(_DiscretizationStepSize*_DiscretizationStepSize) * U[0] + U
      [1]/(_DiscretizationStepSize*_DiscretizationStepSize) + lambda * _V[0];
  for (int idim = 1; idim < _Dimension-1 ; ++idim)
    _Equilibrium[idim] = (U[idim+1]-2*U[idim] + U[idim-1])/(_DiscretizationStepSize*
        _DiscretizationStepSize) + lambda * _V[idim];
  _Equilibrium[_Dimension-1] = -2. /(_DiscretizationStepSize*_DiscretizationStepSize) *
      U[_Dimension-1] + U[_Dimension-2]/(_DiscretizationStepSize*
      _DiscretizationStepSize) + lambda * _V[_Dimension-1];

  return _Equilibrium;
}
```

As *T_Scalar* and *T_Vector* are template parameters of the class BratuProblem, on can choose to.

- Compute the constant value by using for example the type *double* for *T_Scalar* and *Vecteur⟨double⟩* for *T_Vector*
- Compute the differentiate terms (or Taylor series terms) by using for example the type *DType1* for *T_Scalar* and *Vecteur⟨DType1⟩* for *T_Vector*.

In case of matrix computation (*BratuMatrix*), the result is a reference to a *T_Matrix*:

```
template < class T_Vector, class T_Matrix >
T_Matrix & BratuMatrix< T_Vector, T_Matrix >::operator()(const T_Vector & U0, const
    T_Scalar & lambda0)
{
  T_Vector v0(U0.getDimension(),1.);
  for (int i=0; i < _Dimension; ++i)
       v0[i] = exp(U0[i]);

  _TangentMatrix(0,0) = -2./(_DiscretizationStepSize*_DiscretizationStepSize) + lambda0
      *v0[0];
  _TangentMatrix(0,1) = 1./(_DiscretizationStepSize*_DiscretizationStepSize);

  for (int idiag=1; idiag < _Dimension-1; ++idiag)
  {
    _TangentMatrix(idiag,idiag-1) = 1./(_DiscretizationStepSize*_DiscretizationStepSize)
        ;
    _TangentMatrix(idiag,idiag) = -2./(_DiscretizationStepSize*_DiscretizationStepSize)
       + lambda0*v0[idiag];
    _TangentMatrix(idiag,idiag+1) = 1./(_DiscretizationStepSize*_DiscretizationStepSize)
        ;
  }

  _TangentMatrix(_Dimension-1,_Dimension-2) = 1./(_DiscretizationStepSize*
      _DiscretizationStepSize);
  _TangentMatrix(_Dimension-1,_Dimension-1) = -2./(_DiscretizationStepSize*
      _DiscretizationStepSize) + lambda0*v0[0];

  return _TangentMatrix;
};
```

where the template parameters *T_Vector*, *T_Matrix* can be defined to choose the vector container type (for example *Vector⟨double⟩*) and the type of linear matrix. Latter has to be coherent with the linear system solver called from ANM.

The main file could be described in the following listing:

```
1
2  // Taylor series variable class
3  #include "DType1.h"
4
5  // Solution branch class
6  #include "SolutionBranch_U_Lambda.h"
7  // Class for the tangent matrix computation for Bratu problem
8  #include "BratuMatrix.h"
9  // Class for the computation of Bratu equilibrium
10 #include "BratuProblem.h"
11 // Class for the resolution of non linear problem
12 #include "ANM_Solver.h"
13 // Wrapper to the SuperLU linear solver
14 #include "RSL_LinearSystemSuperLU.h"
15
16 // Define the data structure
17 #include "Types.h"
18 typedef DType1 T_Diff;
19 typedef MAT::SparseMatrix T_Matrix;
20 typedef Types< double, T_Diff, T_Matrix > T_Types;
21
22 // Some aliasing
23 typedef BratuProblem< typename T_Types::T_Diff, typename T_Types::T_VectorDiff >
      T_BratuProblem;
24 typedef BratuMatrix< typename T_Types::T_Vector, typename T_Types::T_TangentMatrix >
      T_BratuMatrix;
25
26 typedef RSL::LinearSystemSuperLU<typename T_Types::T_Vector, typename T_Types::
      T_TangentMatrix >  T_LinearSystem;
27 #define storageType SPARSE_COL
28
29 typedef SolutionBranch_U_Lambda <T_BratuProblem, T_LinearSystem, typename T_Types::
      T_Vector, typename T_Types::T_Diff, typename T_Types::T_VectorDiff > T_Branch;
30 typedef ANM::PathFollowing<T_Types, T_BratuProblem, T_Branch, T_BratuMatrix,
      T_LinearSystem > T_ANMSolver;
31
32
33 int main(int argc, char * argv[])
34 {
35   int NbDiscretizationStep = 9;
36   if (argc > 1)
37     NbDiscretizationStep = atoi(argv[1]);
38
39   // Parameters for the ANM algorithm
40   const int order = 20;
41   const double accuracy = 1e-6;
42   const int nbANMStep = 4;
43   const int nbSavePoints = 10; // we save the value of nbSavePoints on a piece of
        solution branch
44
45   //Define the Bratu problem
46   const double DiscretizationStepSize = 1./(NbDiscretizationStep+1);
47   T_Types::T_Vector U0(NbDiscretizationStep,0);
48   double lambda0 = 0;
49   T_Types::T_Vector v0(NbDiscretizationStep,1);
50
51   T_BratuProblem * pbBratu = new T_BratuProblem(NbDiscretizationStep);
52   T_BratuMatrix * computeKt = new T_BratuMatrix(NbDiscretizationStep, storageType);
53   T_LinearSystem * LinearSystem = new T_LinearSystem();
54   T_ANMSolver ANMsolver(order, accuracy, nbANMstep, nbSavePoints);
55
56 // Container of the result : for each branch, at each savePoint the value of lambda is
      mapped to u values
57   T_ANMSolver::T_ConteneurResultat SolutionPath;
58
59   ANMsolver.attachNonLinearProblem( pbBratu);
60   ANMsolver.attachTangentMatrixComputation(computeKt);
61   ANMsolver.attachLinearSystem(LinearSystem);
62   ANMsolver(U0, lambda0, SolutionPath );
63
64   ANMsolver.releaseLinearSystem();
65   ANMsolver.releaseTangentMatrixComputation();
66   ANMsolver.releaseNonLinearProblem();
67
68   delete LinearSystem;
69   delete computeKt;
70   delete pbBratu;
71
72   return 1.;
73 }
```

The automatic differentiation of the Bratu problem is done by declaring the *BratuProblem* class with the appropriate data type **BratuProblem**⟨**typename T_Types::T_Diff, typename T_ Types::T_VectorDiff**⟩ with *T_Types::T_Diff* a derivative (i.e. Taylor series) data type and *T_Types::T_VectorDiff* a vector of this last.

Note that an object of the class **BratuProblem⟨double, Vector⟨double⟩⟩** allows to compute the residual for a given solution through its parenthesis operator.

4.1.4. Results

To test the efficiency of MANITOO, we compare our implementation with a Matlab code. This code corresponds to a hand-written asymptotic numerical method. As the operations involved in the Bratu problem are few and quite simple, it is expected that the Matlab code is optimized in term of runtime performance. Moreover we use this example to evaluate the performance of automatic differentiation compared to hand-written ANM.

The classical solution curve at the middle of the domain is given in Fig. 12. Each point on the curve illustrates the end of a solution branch and the beginning of the next one.

Obviously, the finite difference method applied to Bratu problem leads to a really sparse matrix. In this case, we use the superLU solver with sparse matrix to solve linear systems in the Manitoo library while the *sparse* keyword is used in Matlab.

Table 1 describes computation time (in s). The total and linear–solver cpu-time of Matlab, hand-written ANM and ANM with automatic differentiation are reported. Total time is the time spent to solve the problem while linear–solver time is the time spent to solve for each branch at each solver the linear systems $K\ U_k = F_k$.

We observe that the Matlab computation time seems to be greater than the computation time needed by AD or classical ANM. As the matrix is really sparse, we observe that AD time is more than twice of the hand-written code. Due to the use of condensation technique in the Matlab code, it could be viewed similar to the second version of DIAMANT [20] (i.e. there is no recomputation of the previous order) while we are here using the first version of automatic differentiation: for a given order we recompute all terms of the series up to this order. This kind of AD avoids us to store all intermediate variables. Moreover we do not need specific AD techniques such as check-pointing or sequence recording. This leads to a better flexibility and genericity of the algorithm. Note also that as observed on this case, memory management is a critical point for engineering problems.

4.2. Geometrically non-linear mechanical problem

4.2.1. Formulation

Let us consider an elastic deformable body. We look for the displacement field, denoted by u, of this body subjected to an external force.

Usually, the Green–Lagrange strain tensor γ is defined by

$$\gamma(u) = \frac{1}{2}({}^T\theta(u) + \theta(u) + {}^T\theta(u)\theta(u)) \tag{11}$$

where $\theta(u) = \nabla_x u$ is the displacement gradient tensor. In elasticity framework, the second Piola–Kirchhoff stress tensor, denoted S, is related to the Green–Lagrange strain tensor by the fourth order elastic tensor L:

$$S(\Gamma(u)) = L : \Gamma(u) \tag{12}$$

The problem to solve is expressed as:

Find (u, λ) such as

$$\int_\Omega {}^t S(\Gamma(u)) : \Delta\Gamma(u) d\Omega - \lambda P_e(\Delta u) = 0 \tag{13}$$

In this example, non-linearity only lies in the expression of the strain field.

The problem (13) is discretized using the finite element method with 2D, 3D or shell elements. A Gauss quadrature approximation is used to integrate the non-linear form over the discretized domain. Hence the stress tensor is a local variable evaluated at each integration point (or Gauss point).

4.2.2. ANM

The unknown are sought in the form:

$$u(t) = u_0 + \sum_{i=1}^{n} t^i u_i$$

$$\lambda(t) = \lambda_0 + \sum_{i=1}^{n} t^i \lambda_i$$

and the path-parameter is defined as:

$$t = \langle u(t) - u_0, u_1 \rangle + (\lambda(t) - \lambda_0)\lambda_1.$$

One has also to differentiate the intermediate variables:

$$\theta_k = \nabla_x u_k \tag{14}$$

$$\Gamma_k = \frac{1}{2}\left({}^T\theta_k + \theta_k) + \sum_{i=0}^{k} {}^T\theta_i \theta_{k-i}\right) \tag{15}$$

$$S_k = L : \Gamma_k \tag{16}$$

and the global formulation:

$$\int_\Omega {}^T S_k : (\Delta\Gamma)_0 + {}^T S_0 : (\Delta\Gamma)_k d\Omega + \int_\Omega \sum_{i=1}^{k-1} {}^T S_i : (\Delta\Gamma)_{k-i} d\Omega = \lambda_k P_e(\Delta u) \tag{17}$$

using

$$\Delta\Gamma(u) = \frac{1}{2}({}^T\theta(\Delta u) + \theta(\Delta u) + {}^T\theta(\Delta u)\theta(u) + {}^T\theta(u)\theta(\Delta u))$$

implying

$$(\Delta\Gamma)_k = \frac{1}{2}({}^T\theta(\Delta u)\theta_k + {}^T\theta_k\theta(\Delta u)).$$

This problem could then be solved using Algorithm 1. At each order, the right hand side comes from the contribution of geometric non-linearities. According to the Bratu example, one could condense intermediate variables, such as the stress tensor. This leads to the convenient quadratic formulation [29]. In Manitoo, we recall that we work with any kind of formulation.

4.2.3. Design of the formulation class

As for the Bratu example, ANM is compound by a continuation class, a branch solution class, a linear solver class and classes defining non-linear problem and matrix computation.

In this example, the problem (13) could be written as $R(U) - \lambda F = 0$. Hence we do not need to compute the tangent term L_λ^T related to the λ variable. This term is here always equals to $-F$. Then we are here using the *BranchSolution* class accounting for this specificity. The continuation process does not change from the Bratu problem.

The non-linear formulation consists in computing an approximation of $R(U)$ over a discretized domain. As the process of finite element computation is always the same, we design a generic function that iterates on all the elements of a mesh, apply on each one the computation of a given function, and assembles the element results to the global result. This function is called by a class named *GlobalEquilibriumFE*. It computes a function overall a finite element domain through the parenthesis operator.

Then we have to implement a class, called for example *FML::InternalWorkElt*, computing the numerical approximation of the internal work on an element. Concerning the computation of tangent matrix, even if it could be easily obtained and computed by hand, as explained in [29], we use the automated computation of this matrix. This allows us to change for example the constitutive law in the formulation without any change in the implementation of tangent matrix computation. As for the computation of non-linear formulation, computation of tangent matrix is quite generic. On each element a local tangent matrix is computed and

```
1
2  // Derivatives data type
3  #include "DType1.h"
4
5  // Solution branch class
6  #include "SolutionBranch.h"
7  // Class for the tangent matrix computation using automatic differentiation
8  #include "DMT_ComputeTangentMatrix.h"
9
10 // Class for the computation of global equilibrium
11 #include "GlobalEquilibriumFE.h"
12 // Class for the computation of element equilibrium
13 #include "FML_InternalWorkElt.h"
14 // Class for the resolution of non linear problem
15 #include "ANM_Solver.h"
16 // Wrapper to the SuperLU linear solver
17 #include "RSL_LinearSystemSuperLU.h"
18
19
20 // Define the data structure
21 #include "Types.h"
22 typedef DType1 T_Diff;
23 typedef MAT::SparseMatrix T_Matrix;
24 typedef Types< double, T_Diff, T_Matrix > T_Types;
25
26
27 // Some aliasing
28 typedef RSL::LinearSystemSuperLU<typename T_Types::T_Vector, typename T_Types::
       T_TangentMatrix >  T_LinearSystem;
29
30 typedef FML::InternalWorkElt<typename T_Types::T_Diff, typename T_Types::T_VectorDiff >
        T_InternalWorkElt;
31 typedef FML::GlobalEquilibriumFE<typename T_Types::T_Diff, typename T_Types::
       T_VectorDiff, T_InternalWorkElt > T_GlobalEquilibrium;
32
33 typedef SolutionBranch< T_GlobalEquilibrium , T_LinearSystem, typename T_Types::
       T_Vector, typename T_Types::T_Diff, typename T_Types::T_VectorDiff > T_Branch;
34
35
36
37 typedef ANM::Solver<T_Types, T_GlobalEquilibrium, T_Branch, T_BratuMatrix,
       T_LinearSystem > T_ANMSolver;
38
39
40 int main(int argc, char * argv[])
41 {
42
43   std::string inputFilename;
44
45   if (argc > 1)
46     inputFilename = argv[argc-1];
47
48   // A class object containing finite element mechanical model features (discretization
         , constitutive law, load and prescribed boundary conditions, ANM parameters...)
49   Model model;
50   // All parameters are read in an input file
51   IMP::ImportCLE import;
52   import(inputFilename, model);
53
54   // Creating the global non linear problem computation
55   T_GlobalEquilibrium * globalNonLinearEquilibrium = new T_GlobalEquilibrium(model);
56   // Creating the element non linear computation
57   T_InternalWorkElt * eltNonLinearEquilibrium = new T_InternalWorkElt(model);
58   // Creating the tangent matrix computation
59   DMT::ComputeTangentMatrix< T_InternalWorkElt, T_Types > * computeKt = new DMT::
         ComputeTangentMatrix< T_InternalWorkElt, T_Types >(eltNonLinearEquilibrium);
60   // Creating the linear system solver
61   T_LinearSystem * LinearSystem= new T_LinearSystem();
62
63   // Inital starting point
64   typename T_Types::T_Vector U0(modele.nbDdls(),0.);
65   double lambda0 = 0.;
66
67   T_ANMSolver ANMsolver(model.getAlgoParameters());
68
69 // Container of the result : for each branch, at each savePoint the value of lambda is
       mapped to u values
```

```
  T_ANMSolver::T_ConteneurResultat SolutionPath;

  ANMsolver.attachNonLinearProblem( globalNonLinearEquilibrium );
  ANMsolver.attachTangentMatrixComputation(computeKt);
  ANMsolver.attachLinearSystem(LinearSystem);
  ANMsolver.apply(U0, lambda0, SolutionPath );

  ANMsolver.releaseLinearSystem();
  ANMsolver.releaseTangentMatrixComputation();
  ANMsolver.releaseNonLinearProblem();

  delete LinearSystem;
  delete eltNonLinearEquilibrium;
  delete computeKt;
  delete globalNonLinearEquilibrium;

  return 1.;
}
```

assembled into the global matrix. The local matrix is computed with successive evaluations of the first derivative of element equilibrium equation on the canonical basis directions.

We give hereafter a simplified example of the element internal work computation class in a 2D case. This is a non-generic implementation of this kind of class and it is not the one used for the

```
// Compute the element equilibrium through the parenthesis operator ()
template < class T_Scalar, class T_Vector , class T_Force , class T_LocalMatrix >
T_Vector & FML::TravailInterneElt< T_Scalar, T_Vector, T_Force, T_LocalMatrix >::
    operator () (ELT::ElementBase * elt, const T_Vector & U)
{
  _EquilibreElt.fill(0.);
  double detJ = 0.;
  T_LocalMatrix D = elt->getConstitutiveMatrix();
  T_Vector ThetaGauss(4); // a T_Vector containing 4 data
  T_Vector GammaGauss(3); // a T_Vector containing 3 data
  T_Vector SigmaGauss(3); // a T_Vector containing 3 data
  T_Vector B_S(4); //a T_Vector containing 4 data

  for (int iGauss = 0; iGauss < elt->nbIntegrationPt; ++iGauss)
  {
    // Compute the gradient of displacement stored in a vector-style container and the
        determinant of Jacobian matrix
    T_LocalMatrix GradMatrix = ComputeGradOfShapheFunction(elt, iGauss, detJ);
    ThetaGauss = GradMatrix*U;

    // Compute the strain stored in a vector-style : here lies the non-linearity
    GammaGauss[0] = Theta[0]+ 0.5*(Theta[0]*Theta[0])+ 0.5*(Theta[2] * Theta[2]);
    GammaGauss[1] = Theta[3]+ 0.5*(Theta[3]*Theta[3])+ 0.5*(Theta[1] * Theta[1]);
    GammaGauss[2] = Theta[1]+ Theta[0]*Theta[1] + Theta[2] + Theta[3]* Theta[2];

    // Compute the stress in case of elastic behavior
    SigmaGauss = D*GammaGauss;
    // Compute the virtual internal work contribution at the Gauss point
    B_S[0] = SigmaGauss[0] + ThetaGauss[0] * SigmaGauss[0] + ThetaGauss[1]*SigmaGauss
        [2];
    B_S[1] = ThetaGauss[1] * SigmaGauss[1] + SigmaGauss[2] + ThetaGauss[0]*SigmaGauss
        [2];
    B_S[2] = ThetaGauss[2] * SigmaGauss[0] + SigmaGauss[2] + ThetaGauss[3]*SigmaGauss
        [2];
    B_S[3] = SigmaGauss[1] + ThetaGauss[3] * SigmaGauss[1] + ThetaGauss[2]*SigmaGauss
        [2];

    _EquilibreElt += (GradMatrix.transpose() * B_S) * (detJ *GaussWeight[iGauss]);
  }
  return _EquilibreElt;
}
```

following results subsection. However the purpose is here to illustrate the relative programming effort to implement a new formulation at an element level.

4.2.4. Results

MANITOO is used to solve an academic buckling example dealing with the snap-through of a thin curved roof as in [12]. We consider that the behavior of the deformable body is linear and elastic. Shell elements are used to discretize the roof as in [12].

The mesh used, called *mesh22* $\times$ *22*, is made up of 484 elements, 1541 nodes and 9246 degrees of freedom.

On Fig. 13 we compared the results obtained with MANITOO, Abaqus and Eve (a Fortran77 implementation of the ANM). On the Manitoo and Eve curves, points indicate the end of a branch and the beginning of the next one. Consequently, each point corresponds to the computation of a tangent matrix and its factorization.

On the Abaqus curves, each point corresponds to an increment computation and consequently to a linear system solution. In this case, each point is linked to the next one by a straight line. Accuracy of Abaqus results depends on the size of the increments and then to the number of matrix inversions. From Fig. 13, we see that when we need 13 matrix computation and factorization with the ANM, Abaqus needs about 75 matrix inversions to obtain a quite accurate curve.

However, the comparison of these two softwares is not really appropriate because MANITOO has a specific goal: ANM was designed to build analytical solutions that would be post-processed.

As MANITOO has been designed and developed to replace the Eve code, we compare the computation time. Simulation with Eve requires 2254 s while Manitoo only needs 1957 s.

We observe a saving of about 13%. Comparing results on this academic mechanical case does not lead to a global conclusion. However, this comparison ensures us that at least on this basic case, object oriented C++ implementation coupled with automatic differentiation overperforms the available Fortran77 implementation of ANM. We have obtained a faster solver with more genericity and flexibility than the Eve version.

5. Concluding remarks

This paper deals with the object oriented design of *MANITOO* which is an automatic non-linear solver based on asymptotic numerical method. We try to build a quite generic and flexible library by using some concepts such as functors, traits and classical Object Oriented paradigms.

Due to the nature of the mechanical problems, design of MANITOO integrates a wrapper to numerous linear solvers. Moreover we have quickly designed our own finite element library due to the lack in the opensource community of a library accouting for our specific shell elements.

As illustrations we have applied our library to solve two different kinds of academic problems. On the one hand, we note the quality of the obtained solution from the solver which is a necessary condition to go on with this library. On the other hand, by comparing our library with Eve, a Fortran77 implementation of ANM, on a simple mechanical problem, we note that MANITOO allows to decrease the computation time. For our application, the object oriented design and C++ implementation is not slower than the Fortran77 one.

As a final result, we achieve to design, at our knowledge, the first generic non-linear solver based on Asymptotic Numerical Method with performance at least equivalent to the Fortran implementation existing in the ANM community.

Of course more works have to be realized to obtain a more complete library. Firstly, coupling MANITOO with an automated Finite Element tool such as the Fenics project [60] could result in a really generic finite element solver and could be of great interest for the ANM community. Secondly, a deeper analysis of the used design patterns as in [27] could result in improving the Oriented Object structure and adding more flexibility related to the Model design. Thirdly, improving internal algorithms and data structures, without changing the interface for the final user, would lead to decreasing the computation time and improving the memory management. And finally, allowing parallel or multi-threads feature would enable us to solve cases close to industrial problems.

Acknowledgement

This work has been supported by the French "Agence Nationale de la Recherche" (ANR) in the framework of the Project INSTABANDE.

References

[1] Damil N, Potier-Ferry M. A new method to compute perturbed bifurcations: application to the buckling of imperfect elastic structures. Int J Eng Sci 1990;28(9):943–57.
[2] Noirfalise C, Bourinet J, Fogli M, Cochelin B. Approche fiabiliste de la stabilité des coques minces avec imperfections géométriques aléatoires. In: 18ème Congrès Français de Mécanique; 2007.
[3] Cochelin B, Vergez C. A high order purely frequency-based harmonic balance formulation for continuation of periodic solutions. J Sound Vibr 2009;324(1–2):243–62.
[4] Abdelkhalek S, Montmitonnet P, Potier-Ferry M, Zahrouni H, Legrand N, Buessler P. Strip flatness modelling including buckling phenomena during thin strip cold rolling. Ironmak Steelmak 2010;37(4):290–7.
[5] Niroomandi S, Alfaro I, Cueto E, Chinesta F. Accounting for large deformations in real-time simulations of soft tissues based on reduced-order models. Comput Meth Program Biomed 2012;105(1):1–12.
[6] Nezamabadi S, Yvonnet J, Zahrouni H, Potier-Ferry M. A multilevel computational strategy for handling microscopic and macroscopic instabilities. Comput Meth Appl Mech Eng 2009;198(27–29):2099–110.
[7] S. Baguet, Suivi de branches de points limites par des méthodes asymptotiques numériques., Ph.D. thesis, Université de la Méditerranée, Aix-Marseille II, 2001.
[8] Galliet I, Cochelin B. Une version parallèle des méthodes asymptotiques numériques. Revue Européenne des Eléments Finis 2004;13(1–2):177–95.
[9] R. Arquier, B. Cochelin, C. Vergez, User guide MANLAB 1.0., 2009.
[10] Daridon L, Charras T. Implémentation dans Cast3m de la méthode asymptotique numérique. Revue Européenne des Eléments Finis 2004;13(1–2):165–76.
[11] Vannucci P, Cochelin B, Damil N, Potier-Ferry M. An asymptotic-numerical method to compute bifurcating branches. Int J Numer Meth Eng 1998;41(8):1365–89.
[12] Boutyour EH, Zahrouni H, Potier-Ferry M, Boudi M. Bifurcation points and bifurcated branches by an asymptotic numerical method and padé approximants. Int J Numer Meth Eng 2004;60(12):1987–2012.
[13] Baguet S, Cochelin B. On the behaviour of the ANM continuation in the presence of bifurcations. Commun Numer Meth Eng 2003;19:459–71.
[14] Allery C, Cadou J, Hamdouni A, Razafindralandy D. Application of the asymptotic numerical method to the Coanda effect. Revue Européenne des Eléments finis 2004;13(1-2):57–77.
[15] Cadou J, Potier-Ferry M, Cochelin B. A numerical method for the computation of bifurcation points in fluid mechanics. Euro J Mech – B/Fluids 2006;25(2):234–54.
[16] Brezillon A, Girault G, Cadou J. A numerical algorithm coupling a bifurcating indicator and a direct method for the computation of hopf bifurcation points in fluid mechanics. Comput Fluids 2010;39(7):1226–40.
[17] Medale M, Cochelin B. A parallel computer implementation of the asymptotic numerical method to study thermal convection instabilities. J Comput Phys 2009;228(22):8249–62.
[18] Potier-Ferry M, Damil N, Braikat B, Descamps J, Cadou J-M, Cao HL, et al. Traitement des fortes non-linéarités par la méthode asymptotique-numérique. Comptes Rendus de l'Académie des Sciences – Series IIB 1997;324(3):171–7.
[19] Charpentier I, Potier-Ferry M. Différentiation automatique de la méthode asymptotique numérique typée: l'approche diamant. Comptes Rendus Mécanique 2008;336(3):336–40.
[20] Charpentier I, Lejeune A, Potier-Ferry M. The diamant approach for an efficient automatic differentiation of the asymptotic numerical method. In: Bischof CH, Bücker HM, Hovland PD, Naumann U, Utke J, editors. Advances in automatic differentiation. Springer; 2008. p. 139–49.
[21] Koutsawa Y, Charpentier I, Daya EM, Cherkaoui M. A generic approach for the solution of nonlinear residual equations. Part i: the diamant toolbox. Comput Meth Appl Mech Eng 2008;198(3–4):572–7.
[22] Bilasse M, Charpentier I, Daya EM, Koutsawa Y. A generic approach for the solution of nonlinear residual equations. Part ii: Homotopy and complex

nonlinear eigenvalue method. Comput Meth Appl Mech Eng 2009;198(49-52):3999–4004.
[23] Cary JR, Shasharina SG, Cummings JC, Reynders JVW, Hinker PJ. Comparison of C++ and Fortran 90 for object-oriented scientific programming. Comput Phys Commun 1997;105(1):20–36.
[24] Mackerle J. Object-oriented programming in fem and bem: a bibliography (1990–2003). Adv Eng Softw 2004;35(6):325–36.
[25] Veldhuizen TL, Jernigan ME. Will C++ be faster than Fortran? In: Proceedings of the 1st international scientific computing in object-oriented parallel environments (ISCOPE'97). Lecture notes in computer science. Springer-Verlag; 1997.
[26] Veldhuizen TL. Scientific computing: C++ versus Fortran: C++ has more than caught up. Dr. Dobbs J Softw Tools 1997;22(11):91.
[27] Heng B, Mackie R. Using design patterns in object-oriented finite element programming. Comput Struct 2009;87(15–16):952–61.
[28] Cochelin B. A path-following technique via an asymptotic-numerical method. Comput Struct 1994;53(5):1181–92.
[29] Cochelin B, Damil N, Potier Ferry M. Méthode asymptotique numérique. Hermes Lavoisier, 2007.
[30] Lopez S. An effective parametrization for asymptotic extrapolations. Comput Meth Appl Mech Eng 2000;189(1):297–311.
[31] Mottaqui H, Braikat B, Damil N. Discussion about parameterization in the asymptotic numerical method: application to nonlinear elastic shells. Comput Meth Appl Mech Eng 2010;199(25-28):1701–9.
[32] Cochelin B, Damil N, Potier-Ferry M. Asymptotic-numerical methods and padé approximants for non-linear elastic structures. Int J Numer Meth Eng 1994;37(7):1187–213.
[33] Najah A, Cochelin B, Damil N, Potier-Ferry M. A critical review of asymptotic numerical methods. Archiv Comput Meth Eng 1998;5:31–50.
[34] Olsson A. An object-oriented implementation of structural path-following. Comput Meth Appl Mech Eng 1998;161(1-2):19–47.
[35] Remacle J-F, Shephard MS. An algorithm oriented mesh database. Int J Numer Meth Eng 2003;58(2):349–74.
[36] Kirk BS, Peterson JW, Stogner RH, Carey GF. `libMesh`: a C++ library for parallel adaptive mesh refinement/coarsening simulations. Eng Comput 2006;22(3–4):237–54.
[37] Patzák B, Bittnar Z. Design of object oriented finite element code. Adv Eng Softw 2001;32:759–67.
[38] Renard Y, Pommier J. Getfem ++, an open source generic C++ library for finite element methods <http://home.gna.org/getfem>.
[39] Pironneau O, Hecht F, Morice J. Freefem++ <www.freefem.org>.
[40] Fenics <http://www.fenicsproject.org/index.html>.
[41] Büchter N, Ramm E, Roehl D. Three-dimensional extension of nonlinear shell formulation based on the enhanced assumed strain concept. Int J Numer Meth Eng 1994;37(15):2551–68.
[42] Zimmermann T, Dubois-Pèlerin Y, Bomme P. Object-oriented finite element programming: I: governing principles. Comput Meth Appl Mech Eng 1992;98:291–303.
[43] Archer GC, Fenves G, Thewalt C. A new object-oriented finite element analysis program architecture. Comput Struct 1999;70(1):63–75.
[44] Lin B-Z, Chuang M-C, Tsai K-C. Object-oriented development and application of a nonlinear structural analysis framework. Adv Eng Softw 2009;40(1):66–82.
[45] Pozo R. Template numerical toolkit and jama/C++ linear algebra package <http://math.nist.gov/tnt/>; 2004.
[46] Demmel JW, Eisenstat SC, Gilbert JR, Li XS, Liu JWH. A supernodal approach to sparse partial pivoting. SIAM J Matrix Anal Appl 1999;20(3):720–55.
[47] Balay S, Gropp WD, McInnes LC, Smith BF. Petsc home page <http://www.mcs.anl.gov/petsc>; 2008.
[48] Balay S, Gropp WD, McInnes LC, Smith BF. Petsc users manual. Tech. Rep. ANL-95/11 – Revision 3.0. Argonne National Laboratory; 2008.
[49] Balay S, Gropp WD, McInnes LC, Smith BF. Efficient management of parallelism in object oriented numerical software libraries. In: Arge E, Bruaset AM, Langtangen HP, editors. Modern software tools in scientific computing. Birkhauser Press; 1997. p. 163202.
[50] Amestoy PR, Duff IS, Koster J, L'Excellent J-Y. A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J Matrix Anal Appl 2001;23(1):15–41.
[51] Amestoy PR, Guermouche A, L'Excellent J-Y, Pralet S. Hybrid scheduling for the parallel solution of linear systems. Parall Comput 2006;32(2):136–56.
[52] Bischof CH, Bücker HM. Computing derivatives of computer programs. In: Grotendorst J. editors. Modern methods and algorithms of quantum chemistry: proceedings, NIC Series, NIC-Directors, 2nd ed., vol. 3, July 2000, p. 315–27.
[53] Jerrell ME. Function minimization and automatic differentiation using C++. SIGPLAN Not 1989;24(10):169–73.
[54] Stroustrup B. The C++ programming language. Addison-Wesley Longman Publishing Co., Inc.; 2000.
[55] Jerrell ME. Automatic differentiation using almost any language. SIGNUM Newslett 1989;24(1):2–10.
[56] Bischof CH, Hovland PD, Norris B. On the implementation of automatic differentiation tools. High Order Symbol Comput 2008;21(3):311–31.
[57] Rall LB. Automatic differentiation: techniques and applications. Lecture notes in computer science, vol. 120. Berlin: Springer; 1981.
[58] J.T.C.S. Committee. Technical report iso/iec 18015:2006 on C++ performance. Tech. rep., Information Technology Programming languages, their environments and system software interfaces, 2006.
[59] Myers N. A new and useful technique: "traits". j-C-PLUS-PLUS-REPORT 1995;7(5):32–5.
[60] Logg A. Automating the finite element method. Archiv Comput Meth Eng 2007;14(2):93–138.