Object-oriented programming in nonlinear finite element analysis

Yves Dubois-Pélerin, Pierre Pegon

Structural Mechanics Unit, Joint Research Centre of the European Commission (TP 480), 21020 Ispra (Varese), Italy

Received 13 January 1997; accepted 1 January 1998

Abstract

Object-oriented programming has already shown to provide numerical scientists with adequate means for improving modularity in basic linear finite element codes. It is demonstrated here that this holds for nonlinear analysis, provided that the right types of objects with adequate capabilities are defined for handling nonlinear behaviours. In order to implement nonlinear models together with alternative methods for solution control, for nonlinear equation solving and for linear equation solving, a set of three classes is introduced: Problem, Nonlinear-Solver and LinearSolver. C++ examples implementing advanced algorithms indicate that these classes achieve a fair degree of programming flexibility and reusability. They thus provide an appropriate basis for modular nonlinear finite element programming. © 1998 Elsevier Science Ltd. All rights reserved.

Keywords: Finite element method, nonlinear analysis, linear equation system, nonlinear equation system, solution control, object-oriented programming, engineering software, C++

1. Introduction

1.1. Motivation

Modern finite element analysis codes comprise user-friendly tools such as pre/post-processing facilities, command languages, restart facilities. Behind these lies the analysis core, which usually consists of three main components:

● A library of finite element types. Every type implements the space discretization of a partial-differential equation. Typical activities of the finite element modules are to generate stiffness matrices, internal forces, strains. These modules manage most of the geometrically nonlinear behaviours;

● A library of material types. Every type implements a constitutive model. Typical activities of the material modules are to generate stresses and constitutive matrices. They manage materially nonlinear behaviours;

● A computational kernel, which handles the solution process. In linear analysis, this amounts to forming and solving a linear equation system. Nonlinear analysis raises two additional issues: solving nonlinear equation system and advancing the solution.

Certain algorithms are suitable for certain combinations of differential equation/solution mode/case study/platform and not for others, so it is of importance for the numericist to be able to implement alternative algorithms for each of the above issues. Accordingly the code for the computational kernel usually becomes quite large and organizing it properly becomes an increasingly formidable task as new models or algorithms are introduced. Decades of experience have shown that the conventional procedural programming approach does not provide adequate facilities toward that goal. Object-oriented programming can help a lot, mainly by ensuring a tighter control on both the data and the procedures: instead of being scattered, the data are structured into entities called objects; instead of being isolated, the procedures are assigned to objects and controlled by them. This alternative approach promotes a more modular design. Object-oriented
Object-oriented programming is thus adopted in this article as the means for organizing data and operations in nonlinear finite element programming. More precisely, this article aims at providing the reader with programming tools for combining more easily than in a classical procedural implementation the following three features:

- alternative linear equation solvers;
- alternative nonlinear equation solvers;
- alternative solution control modes.

Object-oriented programming features concepts for decentralizing operation control to low-grained objects. It will be interesting to see whether or not it also proves adequate for handling intrinsically global activities such as problem formulation, equation solving and solution control.

This article is organized as follows. This introduction is complemented by a state of the art of object-oriented programming in numerical methods, with emphasis laid on nonlinear cases. Section 2 will briefly present a finite element programming environment. Solving linear equation systems will be discussed in Section 3, nonlinear equation systems in Section 4, and solution control algorithms in Section 5.

1.2. State of the art in nonlinear object-oriented finite element programming

Although object-oriented programming traces back to the late seventies, application of its concepts to numerical methods in engineering is still a new research area. As of now (1995), the authors have not found more than 60 references on the subject, of which very few are related to nonlinear analysis.

1.2.1. Key concepts

The key concepts of object-oriented programming (encapsulation, inheritance, polymorphism) can be found in many books, like those of Meyer [1] and Cox [2]; they are briefly exposed in articles by Fenves [3] or Miller [4]. The basics of object-oriented programming applied to numerical methods can be found in contributions of Zimmermann et al. [5, 6].

1.2.2. Linear analysis

The first article ever published on object-oriented finite element programming is probably due to Rehak [7], who considered the matter from a knowledge-engineering viewpoint. Further credit goes to Peskin and Russo [8], who distinguished the problem formulation from its discretization and thus derived three separate classes Problem, Domain and Equation, and to Miller [9], who recognized the degree-of-freedom, the node and the element as the basic objects of an object-oriented finite element programming environment.

The first comprehensive implementation descriptions were released independently by Forde et al. [10] and by Baugh and Rehak [11, 12]. Forde et al. dealt with linear two-dimensional problems in solid mechanics, with emphasis on isoparametric elements. Starting from the governing equations they evidenced the need for new basic objects (such as the material and the boundary condition) acting on their own instead of being capabilities diluted within other objects; they also addressed interpolation and numerical integration. Similarly, Baugh and Rehak [11] set up a library of classes for finite element analysis, including geometrical classes for graphical preprocessing tasks. Through considerations on computer science notions such as control abstractions and dataflow principles, they also studied ways of sequencing the calculations more adequately.

Zimmermann et al. [5, 13] also started from the differential equations for choosing objects and assigning them operations. They used the purely object-oriented language Smalltalk in order to get rid of procedural programming habits and to draw maximum benefit from the concepts of data encapsulation and polymorphism. They ended up with a new concept of ‘non-anticipation’ and claimed it to be as fruitful as the former two. They separated algebraic features related to linear equation solving from the discretization features.

1.2.3. Nonlinear analysis

Menetrey and Zimmermann [14] extended the results of [13] to accommodate plasticity in a concrete model. They evidenced the importance of the ‘Gauss point’ object, not only for numerical integration but also as repositories of local material information. They showed how the behaviours of every constitutive model can be encapsulated in a subclass of a class material so as to interact easily with classes element and GaussPoint.

Munjiza et al. [15] implemented fractured media analysis. They derived a hierarchy of classes including a versatile representation of contact elements. Their main concern was to design a core program as small as possible, where only the object files of the modules required by the data given in the input file were linked together. The need for a permanent run program was thus avoided.

The authors of this article observed that a pitfall in the design of flexible object-oriented finite element codes was to implement adequately the boundary conditions. They showed how to model these in such a way as to cope easily with problems such as a plastic analysis with multipoint constraints [16].
1.2.4. Other topics

Contributions to other areas of object-oriented finite element analysis encompass discussions on substructuring techniques [17], numerical efficiency [18–20], portability [21], automatic code generation [22], command-interpreter environment [23], graphical tools [24–26], computer support [27], alternative programming paradigms [28–30].

2. A modular programming environment

This section presents briefly the structure of an object-oriented finite element program. Choosing an implementation language will help make the matter more tangible for the reader. On behalf of its popularity and of its numerical efficiency, C++ is chosen. Syntactical explanations will be provided for helping the reader with no previous acquaintance with that language.

For completeness a sneak overview of the file organization of a C++ program is shown first (Section 2.1). Fortran files contain procedures, whereas C++ files contain classes. These will be introduced by describing the so-called class hierarchy (Section 2.2).

2.1. File organization

A C++ program consists of a number of files which are compiled, then linked together into an executable version. The files are the following:

- the source file for the so-called main function. This file is displayed in Fig. 1; the only activity of the main function is to create problem, an object of type Problem, and then to delegate it the solving responsibility (through the message solveYourself).

```c
main ()
{
    Problem problem ;
    problem.solveYourself() ;
}
```

Fig. 1. The root function main.

- the header file and the source file for every class. Examples will be given subsequently.
- some additional header and sources files. These files implement some low-level utilities such as string manipulations, memory allocation facilities, access to I/O devices, CPU-time measurements and error messages.

In summary, an object-oriented C++ program consists primarily in its classes. These are introduced now.

2.2. The class hierarchy

The set of classes is conveniently displayed in the class hierarchy (Fig. 2). The class hierarchy lists every class and also lets the inheritance relationships show up; for example, class Quad4 is a subclass of class Element. For simplicity of discussion, the hierarchy has been splitted up in three groups.

The first group encompasses the classes which implement the components of a finite element mesh. The mesh as a whole is implemented as an object of type Domain. The domain maintains the mesh data, viz.,

- the list of the elements; so far, isoparametric bilinear elements (objects of type Quad4) or linear or geometrically nonlinear truss bar elements (objects of type Truss or TrussGNL) have been implemented;
- the list of the nodes (objects of type Node);
- the list of the materials (for example, one object of type VonMisesPlaneStrain or VonMisesUniaxial);
- the list of the loads (body loads, contour loads, nodal loads).

Class BoundaryCondition implements an essential (Dirichlet) condition. Class Dof implements a nodal degree-of-freedom; every node possesses typically two or three objects of type Dof. Class InitialCondition provides starting values for time-dependent problems. An integration point stores its position, weight and material or geometric local properties; an element of type Quad4 possesses four objects of type GaussPoint2by2, for four-point Gaussian quadrature, whereas a truss element possesses one object of type GaussPoint1. TimeStep objects are used for implementing time or pseudo-time increments. These classes have already proved to make up a convenient basis for easily maintainable finite element programs [13].

The second group of classes includes the analysis classes that have no immediate relation with space discretization. They are the focus of this article. The finite element analysis of a given problem is implemented as an object of a subclass of Problem, i.e. as an object of type NewmarkProblem or StaticProblem. Class NewmarkProblem implements transient analysis where the
Newmark family of algorithms is used for time integration, whereas class StaticProblem implements a steady-state problem. Whichever its type may be, the problem delegates to the domain the space-discretization features and takes upon itself only the time (or pseudo-time) related ones. It generates a usually nonlinear system of equations \( g(x) = 0 \), to be solved by a nonlinear equation solver (an object of type NewtonRaphson or ModifiedNewtonRaphson). The nonlinear solver in turns resorts to a linear solver whenever it comes up with a linear system \( Ax = b \). Class Crout implements a symmetric skyline solver using \( LDL^T \) decomposition, whereas class GaussSeidel implements the iterative Gauss–Seidel algorithm. The classes of this second group will be discussed in Sections 3–5.

Besides numerical analysis classes, a set of tools is needed. Class Container regroups the various types of collections: arrays of integers or double-precision floating-point numbers (classes IntArray and FloatArray), arrays indexed by characters (Dictionary), full or diagonal matrices of floating-point numbers (FloatMatrix, DiagonalMatrix), matrices containing polynomials (PolynomialMatrix), lists containing elementary matrices (UnassembledMatrix), lists containing objects such as elements, nodes, loads or materials (List). Column objects are special arrays used by class Crout for storing columns of the left-hand side. Subclasses of class Function implement various types of functions; piecewise linear and step functions are typically used for weighting loads in time, whereas polynomials are used as coefficients of elementary Jacobian matrices. Class InputStream provides customized access to the data file. These utility classes play a great part in making the higher-level ones understandable and maintainable.

3. Solving linear systems

3.1. An improved modularity

A finite element analysis usually ends up with a system of linear algebraic equations \( Ax = b \) to be solved. Since this operation is both space and time-consuming, finite element programmers have always devoted special attention to it. Linear equation solvers are probably among the best modularized pieces of code in Fortran finite element software. Yet object-oriented programming can bring valuable further improvements.

First, an object-oriented version can be made much more understandable, especially for a stranger to the program. As may be gleaned from the class hierarchy displayed in Fig. 2, all features related to linear equation solving are concentrated in readily identifiable portions of the program, class LinearSolver and its subclasses Crout and GaussSeidel. These encapsulate all of the needed pieces of data (and, in particular, the complete storage mode of the left-hand side) and the requested operations (e.g. assembly, decomposition, back substitutions, updating, and even debugging facilities). Such well-closed program entities have no equivalent in Fortran.

Second, object-oriented classes may have a very limited interface. A major inconvenience when using Fortran solvers is their usually wide-open interface. For example, Fortran subroutine Colsol of Bathe’s didactic STAP software [31], designed for solving a linear system using skyline storage form, requires seven
arguments from the user. Contrariwise, and as will be shown below, in object-oriented programming the message \( x=\text{LinearSolver}{}->\text{computeSolution}() \) can trigger the same operation without any argument. In Fortran, the matter gets worse in more complex situations; for example, Bathe’s well-known Spase subroutine for solving generalized eigen-problems involves 28 arguments [31]. Object-oriented programming does not require lengthy argument list: arguments are encapsulated by objects.

Third, in object-oriented programming, straightforward use can be made of powerful lower-level programming tools. This has been illustrated by Dubois-Pélerin and Zimmermann [19] in the case of an unsymmetric skyline solver: a tool class \texttt{RowColumn} was designed in order to provide every row/column pair with easy coefficient storage, coefficient indexing, range-checking and customized dot products, thereby making the solver class itself (featuring the assembly and solving algorithms) easy to implement.

Fourth, switching from a linear solver to another one can be painful in Fortran implementation, because the differences between storage modes can require substantial upstream modifications. Object-oriented programming can help smoothing out such troubles. Class \texttt{UnassembledMatrix} provides for intermediate storage of assemblages of elementary matrices. Objects such as the problem, the domain and the nonlinear system (function setting and solving functions; the new class is called \texttt{ConjugateGradient}, and is a subclass of \texttt{LinearSolver} (Fig. 2). The third step is to supply the new class with the necessary attributes for carrying out its functions. Fig. 3 summarizes the C++ implementation of these operations. A new file has been created; it contains:

- the name of the new class (\texttt{ConjugateGradient}) and that of its superclass;
- an attribute list, below the protected keyword: attributes \texttt{leftHandSide} and \texttt{rightHandSide} store \( A \) and \( b \), respectively; attribute \texttt{preconditioner} contains the diagonal of \( A \); \texttt{maxIterations} and \texttt{tolerance} are self-explanatory;
- a function list, below the public keyword: in function \texttt{computeSolution} the solver returns \( x=A^{-1}b \); in functions \texttt{setLhs} and \texttt{setRhs} it initializes its attributes \texttt{leftHandSide} to \( A \) and \texttt{rightHandSide} to \( b \), respectively.

Finally the fourth step consists in implementing the listed functions. The key functions for assembling the system (function \texttt{setLhs}) and for solving the system (function \texttt{computeSolution}) are discussed now.

3.2. Implementing conjugate gradients

The benefits of using object-oriented programming are best illustrated by a practical example. As an alternative solver, the diagonally preconditioned conjugate gradient method is proposed. A convenient algorithm for the conjugate gradient is the following [33]:

\[
x_0 = 0
\]

For \( k = 1, \ldots, \text{maxIterations} \)

\[
r_k = r_{k-1} - z_{k-1} A p_{k-1} \quad (r_1 = b)
\]

if (\( \| r_k \| / \| r_1 \| < \text{tolerance} \))

set \( x = x_{k-1} \) and quit

Solve \( B z_k = r_k \) (\( B \) is the preconditioner)

\[
\beta_k = z_k^T r_k / z_{k-1}^T r_{k-1} \quad (\beta_1 = 0)
\]

\[
p_k = z_k + \beta_k p_{k-1} \quad (p_1 = z_1)
\]

\[
x_k = x_{k-1} + z_k p_k
\]

end loop.

It took 4 h of work for the first author to design and program the computer implementation and to run successfully a first application. It is shown now why it can go so fast.

3.2.1. The new module

The first step is one of determining the activities to be performed; these are: setting up and solving the linear system. The second step consists in assigning these activities as functions of one (or possibly more) data type; by analogy with the already implemented methods for solving linear systems, this is done by creating a new type of object and by assigning it the setting and solving functions; the new class is called \texttt{ConjugateGradient}, and is a subclass of \texttt{LinearSolver} (Fig. 2). The third step is to supply the new class with the necessary attributes for carrying out its functions. Fig. 3 summarizes the C++ implementation of these operations. A new file has been created; it contains:

- the name of the new class (\texttt{ConjugateGradient}) and that of its superclass;
- an attribute list, below the protected keyword: attributes \texttt{leftHandSide} and \texttt{rightHandSide} store \( A \) and \( b \), respectively; attribute \texttt{preconditioner} contains the diagonal of \( A \); \texttt{maxIterations} and \texttt{tolerance} are self-explanatory;
- a function list, below the public keyword: in function \texttt{computeSolution} the solver returns \( x=A^{-1}b \); in functions \texttt{setLhs} and \texttt{setRhs} it initializes its attributes \texttt{leftHandSide} to \( A \) and \texttt{rightHandSide} to \( b \), respectively.

Finally the fourth step consists in implementing the listed functions. The key functions for assembling the system (function \texttt{setLhs}) and for solving the system (function \texttt{computeSolution}) are discussed now.

3.2.2. Assembly

Since in the conjugate gradient method the left-hand side \( A \) is involved only multiplicatively, it need not be stored as a whole. As the code for function \texttt{setLhs} shows it (top of Fig. 4), assigning a matrix (the global stiffness matrix in static analysis or the generalized mass matrix in transient analysis) \( A \) as a left-hand side to the solver is trivial: one line of code. Class \texttt{UnassembledMatrix} provides a versatile sparse storage of the left-hand side. Observe the contrast with the more usual skyline storage, where the same function \texttt{setLhs} triggers a heavy assembly process of every elemental matrix onto the global one; the fact that a solver with a completely different data
organization can be implemented so concisely is a telling example of the object-oriented programming flexibility. Note that some procedural programming languages, such as Pascal or C, feature user-defined records, and therefore allow data to be aggregated into data types like \texttt{UnassembledMatrix}. What object-oriented programming brings on top of this is: safer encapsulation of the hidden data, functions attached to (and governed by) the user-defined record, and polymorphic use.

3.2.3. Solving

As function \texttt{computeSolution} of class \texttt{ConjugateGradient} shows it (Fig. 4), object-oriented programming allows the programmer to stick very closely to the formal algorithm displayed at the beginning of this section. Besides the declarative instructions and the \texttt{delete} instructions dealing with dynamic memory allocation, there is almost a one-to-one mapping between the steps of the algorithm and the implementing instructions. This arises from the fact that arrays and matrices manage by themselves their operations: initialization, copy, summation, products, etc. Note that these classes already include functions such as range checking when indexing and size checking when multiplying; these capabilities provide the programmer with valuable debugging information; they can be turned off when numerical efficiency is sought. The quality of these tools together with the complete decoupling of the solver from all discretization issues are the keys for a fast implementation. (It is fair to say that not all linear solvers can be implemented so quickly. For example, using a skyline storage mode in class \texttt{Crout} required designing a class \texttt{Column} and carefully implementing error-prone operations such as assembly and customized dot products. Class \texttt{ConjugateGradient} drew maximum benefit from the intermediate (unassembled) storage mode of \texttt{A}).

3.2.4. Operating mode

Once a left-hand side \texttt{A} of type \texttt{UnassembledMatrix} and a right-hand side \texttt{b} of type \texttt{FloatArray} have been computed, the solution of the system \(Ax = b\) is obtained by two declarations:

\begin{verbatim}
LinearSolver* solver;
FloatArray*x;
\end{verbatim}

and only four instructions:

\begin{verbatim}
solver = new ConjugateGradient();
solver->setLhs(A);
solver->setRhs(b);
x = solver->computeSolution().
\end{verbatim}
In the first of these four instructions, replacing `new ConjugateGradient()` with `new Crout()` or `new GaussSeidel()` would work just as well. In summary, object-oriented programming supplies the programmer with means for thoroughly modularizing linear equation solving.

Fig. 4. Two functions from the source file of class ConjugateGradient.

```java
void ConjugateGradient::setLhs(UnassembledMatrix* A)
{
    leftHandSide = A;
}

FloatArray* ConjugateGradient::computeSolution()
{
    FloatArray *x,*r,*z,*Ap,*p;
    double prod1,prod2,alpha,beta,initialNorm;
    int k;

    x = new FloatArray(rightHandSide->getSize());
    for (k=1; k<=maxIterations; k++) {
        if (k == 1) {
            r = rightHandSide->giveCopy();
            initialNorm = r->giveNorm();
        } else {
            r->subtract(alpha,Ap);
            delete Ap;
        }
        if (r->giveNorm() / initialNorm < tolerance)
            break;
        z = this->precondition(r);
        prod1 = z->dot(r);
        if (k == 1) {
            beta = 0.0;
            p = z->giveCopy();
        } else {
            beta = prod1 / prod2;
            p = p->multiply(beta)->add(z);
            Ap = leftHandSide->times(p);
            alpha = prod1 / p->dot(Ap);
            x->add(alpha,Ap);
            prod2 = prod1;
            delete z;
        }
    }
    delete p; delete r;
    return x;
}
```

(1) this function was declared in the header file (figure 3).
(2) attribute leftHandSide now points to A (no copy is performed).
(3) 3 lines of declarations, x, r, z, etc, are declared as objects of type FloatArray.
(4) creates an empty array of the same size as rightHandSide.
(5) a do-loop between 1 and maxIterations.
(6) Fortran equivalent: if (k.eq.1).
(7) the second member means: message giveCopy() is sent to object rightHandSide.-"->" is the "message" instruction.
(8) function subtract(a,y) of class FloatArray optimizes the computation of x=x-ay.
(9) recycles the memory occupied by Ap.
(10) quits the loop.
(11) returns z = diagonal(A)^{-1}r. this stands for the object (of type ConjugateGradient) which currently executes function computeSolution.
(12) function add(a,y) of class FloatArray optimizes the computation of x=x+ay.
(13) for economy, failure handling is not shown.
4. Solving nonlinear equations

The previous section showed that object-oriented programming is an adequate approach for the programmer for modularizing linear equation solving, under the proviso that a judicious choice of the type of objects is achieved. This section aims to show that the same applies when a nonlinear algebraic equation system arises on top of the linear one.

This starts from the same evidence as for linear equation solving: a nonlinear algorithm (e.g. Newton–Raphson) is a purely algebraic device whose unique function is to solve an equation \( g(x) = 0 \) (right of Fig. 5). In particular, it has nothing to do with analysis data such as displacements, internal forces, out-of-balance forces, time steps, increment steps or load multipliers (left of Fig. 5). This will be used as the guideline for modularizing nonlinear equation solving.

In Section 4.1 an implementation of the Newton–Raphson method will be presented. Section 4.2 will show that the new module is truly easy to use. Not only easy to use, but also easy to enrich: alternative solvers are readily accommodated (Section 4.3).

4.1. Class Newton–Raphson

Object-oriented programming focuses primarily on data, not on functions. Modularizing nonlinear equation solving will thus consist in making a data type of it (i.e. a class) rather than just a functionality of another class. This is why a specific class NonlinearSolver is created (see Fig. 2). Two algorithms are proposed: the Newton–Raphson [34] and modified Newton–Raphson methods. Whenever a problem class (StaticProblem or Newmark–Problem) generates a nonlinear equation system, it will use an object of type NewtonRaphson or ModifiedNewtonRaphson to implement and solve that system.

Class NewtonRaphson is shown in Fig. 6. The unique functionality of an object of this type is to solve its associated problem, through function Solve. In order to do this, it is supplied with four attributes, mainly problem and LinearSolver. Function Solve implements the following algorithm:

\[
\begin{align*}
\text{given } x_0 \\
\text{for } i = 1, \ldots, \text{max Iterations} \\
y_i &= g(x_{i-1}) \\
\text{if } (\|y_i\|/\|y_1\| < \text{tolerance}) \\
\text{set } x = x_{i-1} \text{ and quit} \\
solve Vg \delta x_i = -y_i \\
x_i = x_{i-1} + \delta x_i
\end{align*}
\]

As for the case of a linear solver, the C++ translation is very close to the mathematical formulation (Fig. 7). As a matter of fact, class NewtonRaphson achieves a fair level of modularity:

- class NewtonRaphson is completely problem independent (so is class ModifiedNewtonRaphson).
- Any problem class that wishes to resort to class NewtonRaphson must only provide the following three functions:
  - GiveInitialGuess(),
  - ComputeFunctionalAt(x),
  - ComputeJacobianAt(x),

which are used in function solve() of class NewtonRaphson, as was shown in Fig. 7. For example, a problem of type StaticProblem returns

\[ g(x) = f_{\text{internal}}(x) - f_{\text{external}}(t) \]

for the functional and for the Jacobian, respectively (\( t \) denotes the current pseudo-time). A problem of type

![Fig. 5. Solving nonlinear equations.](image-url)
class NewtonRaphson : public NonlinearSolver
{
    protected:
        Problem* problem;
        LinearSolver* linearSolver;
        int maxIterations;
        double tolerance;
    public:
        NewtonRaphson (Problem* p);
        ~NewtonRaphson () ;
        FloatArray* solve () ;
};

(1) see also the legends of figure 3.
(2) means: attribute problem is an object of class Problem or of any subclass of Problem (StaticProblem or NewmarkProblem).
(3) this special function, called constructor, initializes the 4 attributes.
(4) this special function, called destructor, deinitializes the attributes.

Fig. 6. Header file of class NewtonRaphson.

FloatArray* NewtonRaphson :: solve ()
{
    UnassembledMatrix *jacobian;
    FloatArray x,y,*dx;
    double initialNorm,norm;
    int i;

    x = problem->giveInitialGuess () ;
    for (i=1 ; i<=maxIterations ; i++) {
        y = problem->computeFunctionalAt(x) ;
        norm = y->giveNorm();
        if (i == 1)
            initialNorm = norm ;
        if (norm/initialNorm < tolerance) {
            delete y ;
            break ;
        }
        jacobian = problem->computeJacobianAt(x) ;
        linearSolver->setLhs(jacobian) ;
        linearSolver->setRhs(y->minus());
        dx = linearSolver->computeSolution() ;
        x->add(dx) ;
        linearSolver->updateYourself();
        delete jacobian ; delete y ; delete dx ;
    }
    return x ;
}

(1) see also the legend of figure 4.
(2) function minus of class FloatArray switches the sign of the coefficients.
(3) clears out the left- and right-hand sides of linearSolver.

Fig. 7. Function solve from the source file of class NewtonRaphson.
NewmarkProblem with, say, accelerations as primary unknowns $x$ returns
\[ g(x) = M \dot{x} + f_{\text{external}}(d) - f_{\text{external}}(t) \]
and
\[ \nabla g = M + K_{\text{tangent}}/\beta \Delta t^2, \]
respectively, where the problem obtains the displacements $d$ from the accelerations $x$ by means of its finite difference formulae. Note that class Newton-Raphson does not interfere in the choice of the initial guess $x_0$: it is up to the problem to return a default value such as 0, or the last converged solution, or any kind of predictor of its own.

Class NewtonRaphson is completely independent from the linear equation solver it uses. According to the input supplied by the user, the linear equation system may be of type Crout, GaussSeidel or ConjugateGradient, indifferently. As was shown in Section 3, these implement very diverse storage modes for their left-hand side. The nonlinear solver does not care: it requests a Jacobian matrix from its problem, receives an object of type UnassembledMatrix, and transmits this object to its linear solver, without interfering on the way the linear solver handles it. Note also that if the linear solver is trivial, as in explicit formulations of transient problems, class NewtonRaphson still applies: it just turns out that the jacobian contains only diagonal elemental matrices, and the linear solver will take little time to invert it.

### 4.2. Operating mode of the nonlinear solvers

Class NonlinearSolver is easy to use. Class NewmarkProblem illustrates this: the attributes of an object of this type are listed in Fig. 8: the $\beta$ and $\gamma$ parameters, the vectors of kinematic unknowns $d$, $v$, $a$, the number of time steps, the domain for all space-discretization features, and the nonlinear solver, which is an object of any subclass of NonlinearSolver, i.e. Newton-Raphson or ModifiedNewtonRaphson.

The root function of the program (Fig. 1) creates a problem of type, say, NewmarkProblem, and sends it the message `problem->solveYourself()`. Accordingly the problem executes function `solveYourself` implemented by class NewmarkProblem (Fig. 8). This function is displayed in Fig. 9: the problem loops over the time steps, and obtains successively $a$, $d$ and $v$. The vector of the primary unknowns $a$ is obtained by message `nonlinearSolver->Solve()`.

```cpp
class NewmarkProblem : public Problem
{
    protected:
        double beta, gamma;
        Domain* domain;
        int nsteps;
        NonlinearSolver* nonlinearSolver;
        FloatArray* displacements;
        FloatArray* velocities;
        FloatArray* accelerations;
        // ...

    public:
        void solveYourself();
        FloatArray* giveDisplacements();
        FloatArray* giveVelocities();
        TimeStep* giveNextStep();
        void terminateCurrentStep();
        FloatArray* giveInitialGuess();
        FloatArray* computeFunctionalAt(FloatArray* x);
        UnassembledMatrix* computeJacobianAt(FloatArray* x);
        // ...
};
```

(1) returns the next time step $\Delta t_{n+1}$.
(2) performs end-of-step printing and updating.
(3) see section 4.1.

Fig. 8. Header file of class NewmarkProblem.
Function Solve of class NewtonRaphson is the function that was described in the previous section.

Remark. Using a nonlinear solver in class Problem and subclasses is by no means a design requirement. If class NewmarkProblem implemented linear transient analysis, it would just declare and use an attribute of type LinearSolver rather than NonlinearSolver.

4.3. Implementing an alternative nonlinear solver

In certain nonlinear problems, the Newton–Raphson method and its modified version may be ill-advised, because not applicable, inefficient or non-converging. The scientist may thus wish to use alternative schemes, such as initial-tangent, quasi-Newton or secant-Newton methods (see [35]). The BFGS method [36, 37], a popular member of the quasi-Newton family, has been implemented. Its formulation is displayed in appendix to give the reader a hint on how many programming details are involved. However, it took no more than 6 h of work to program and debug a new class BFGS (see Fig. 2), including the line-search function, until the first example ran successfully. Again, this was made possible by the modular design of this object-oriented implementation, which restricted to the minimum all ‘upwards’ communications with the problem and ‘downwards’ communications with the linear solver of the nonlinear solver.

5. Solution control

The previous section dealt with solving the nonlinear algebraic equation system resulting at every time increment or load increment from the discretization process. The last step is to advance the solution process from increment to increment. Schemes to do so are coined ‘continuation’, ‘path-following’ or ‘solution control’ methods. In nonlinear steady-state analysis, the simplest control mode is by load control: at every step, the external load is incremented by a given factor; it is well known that this method fails around limit points. More refined schemes have been devised: displacement control [38], fictitious-spring control [39, 40], stiffness-parameter method [41], arc-length control [42–45], control by constant increments of external work [46], etc. As suggested by this enumeration, there is no general agreement on a presumably ‘best’ control method. Therefore the programmer should rather have means for implementing alternative methods easily. Object-oriented programming is assessed here with respect to this goal. To start, the simple case of control by the time variable is considered. Next the arc-length method is discussed as a typical example of a more advanced solution.

5.1. Time control

In many situations, the time \( t \) turns out to be the adequate parameter for monitoring the solution. This is usually the case in transient analysis. It is also the case in certain nonlinear steady-state analyses; for example, it includes load control and a simple form of displacement control where loading consists only in prescribed displacements; in both cases, loading evolution is appropriately implemented by weighting external loads or prescribed unknowns by load–time functions ([47], p. 677).

Controlling the solution by the time is trivial, as illustrated by class NewmarkProblem (Fig. 9). The only noteworthy point is that control issues are separated from solving nonlinear equation systems: shifting from one nonlinear solver to another one is a matter of changing one word in the data file, like BFGS instead of NewtonRaphson.

```c
void NewmarkProblem :: solveYourself ()
{
    domain->setAllGlobalNumbers();

    while (this->giveNextStep()) {
        accelerations = nonlinearSolver->solve();
        displacements = this->giveDisplacements();
        velocities = this->giveVelocities();
        this->terminateCurrentStep();
    }
}
```

(1) the domain assigns an equation number to every nodal degree of freedom.

(2) means: as long as message this->giveNextStep() returns a non-zero value.
5.2. Arc-length control

Crisfield’s cylindrical arc-length method [48] is one of the most popular general continuation methods. As far as programming is concerned, this method possesses two features of interest: it enforces the constraint equation to be fulfilled at every iteration; it is tightly related to one nonlinear solving algorithm—the modified Newton–Raphson method. For these two reasons, the module where control is implemented will have to master the whole nonlinear process, a situation clearly not akin to that of time control. The nice thing is that the object-oriented data organization described in this article lends itself to cope easily with this new situation.

A new class called StaticProblemArcLength is created as a subclass of StaticProblem (see Fig. 2); as such it inherits all attributes and functions already defined in class StaticProblem. Class StaticProblem is shown in Fig. 10. It is very similar to class NewmarkProblem (Fig. 8), although the contents of the declared functions is different. Fig. 11 displays class StaticProblemArcLength. The attributes of an object of this type are those listed below keywords protected in both Figs. 10 and 11. The new class defines the specific attributes for the arc-length method: lambda stands for the current load-level parameter \( l \), initialLambda for its starting value \( l_0 \), referenceLoadVector for the nominal value of the external load vector \( q \) to be multiplied by \( l \), desiredIterations for the desired number of modified Newton–Raphson iterations per increment. The inherited attribute nonlinearSolver will not be used; since it is a pointer, the space overhead is insignificant. Instead, the new class must define an attribute linearSolver, because the nonlinear algorithm is embedded within the arc-length procedure, rather than within the nonlinear solver classes. Class StaticProblemArcLength inherits from its superclass not only attributes, but also functions. This is the case of function computeJacobianAt(\( x \)), which, therefore, needs not be reimplemented. Contrariwise, function solveYourself() is reimplemented; it translates every detail of the arc-length-controlled solution process given in [48]; for economy, this lengthy and, to be honest, procedure-like function is not displayed here.

Use of class StaticProblemArcLength is exemplified by the two-dimensional shallow arch for complex snap-through analysis shown in Fig. 12. A problem of type StaticProblemArcLength is used with a reference load vector with \( P=10^4, l_0=0.5 \), three desired iterations, 250 arc-length steps and a Crout linear solver. The vertical displacement of two nodes are in agreement with the values recorded by Powell and Simons [49]. Ideally, one would implement continuation methods as separate class, say, ContinuationMethod. The general spherical arc-length method would be implemented as a subclass ArcLength of ContinuationMethod. A problem class would be client of ArcLength, in turn client of NonlinearSolver, in turn client of LinearSolver. In contrast, this example showed

```c++
class StaticProblem : public Problem
{
    protected:
        int numberOfSteps;
        Domain* domain;
        FloatArray* displacements;
        NonlinearSolver* nonlinearSolver;
        // ...

    public:
        virtual void solveYourself () ;
        virtual void giveNextStep () ;
        virtual void terminateCurrentStep () ;
        void giveInitialGuess () ;
        FloatArray* computeFunctionalAt (FloatArray* x);
        UnassembledMatrix* computeJacobianAt (FloatArray* x);
        // ...
};
```

(1) virtual means that the function is likely to be reimplemented by the subclasses.
(2) this function resembles that of class NewmarkProblem (figure 9).

Fig. 10. The header file of class StaticProblem.
that a practical-use algorithm, namely, Crisfield’s cylindrical method, may require resorting to a more convenient alternative, which seems to be best handled directly within a problem class.

In summary, the programmer can easily implement alternative solution control methods. This is obtained by enriching the library of problem classes (often by inheritance) or a library of continuation-method classes, and by using the capabilities of the existing tools. These tools are classes LinearSolver and NonlinearSolver (and subclasses), plus classes like Domain or UnassembledMatrix. Such object-oriented program organization really tends to behave as a programming tool-box, not as a rigid framework where slots are to be filled: for example, this subsection showed that a problem class can bypass the nonlinear solvers and rather handle directly a linear solver.

6. Conclusions

The potential of object-oriented programming for implementing the finite element method in nonlinear analysis has been investigated; the goal was to enhance programming modularity. Since geometrically nonlinear behaviour can be rather easily handled within the element and node modules and material nonlinearities within material modules, the challenge was to design a data structure for the nonlinear process itself, at the global level. This covers three aspects, which have been reviewed successively:

- linear equation solving. It was shown that this task can be conveniently enclosed within a LinearSolver module. Programming flexibility was gained by defining a powerful intermediate data storage (class UnassembledMatrix) for handling all global matrices;
- nonlinear equation solving. Considering nonlinear equation solving as a strictly formal, mechanics-independent, operation ended up with the definition of a new NonlinearSolver module. An algorithm for solving nonlinear equation systems is simply mapped into a subclass of NonlinearSolver, like class NewtonRaphson;
- solution control. Nonlinear equation systems are a by-product of a model equation (for example, that of steady-state structural equilibrium equation) together with a control method for advancing the solution.

Much more than a procedural program, an object-oriented program lends itself to be used as a tool box. In Fortran a tool is by nature a function. It is reasonably easy to reuse Fortran tools that perform basic operations, like array and matrix manipulations. However, the Fortran programmer is essentially deterred from using elaborated tools, which are usually too complicated, too specific or too context-dependent.
The strong point of object-oriented programming is to allow the programmer to define modules of any granularity, and in particular sophisticated ones. This article introduced modules NonlinearSolver and LinearSolver, which interact loosely with others: Problem, Domain, Element, Node, Matrix, UnassembledMatrix, etc. These classes achieve a high degree of modularity, by taking benefit of the object-oriented programming concepts:

- they manage information autonomously. For example, the attributes defined in class Crout contain all required pieces of information for storing the system’s left-hand side in a symmetric skyline form. Using class Crout does not require knowing anything of its internal representation;
- they provide clear capabilities. The functions dealing with a given kind of operations are regrouped in one or few classes. Also they provide limited interface: their name, their (short) argument list, their return value, and possibly their commented description.

Improved modularity implies improved maintainability. As a matter of fact, the new classes are:

- reusable. For example, when it came to implementing arc-length controlled nonlinear problems, most attributes and functions of the existing class StaticProblem were reused. This was done in the simplest way: by inheriting existing features. A new class StaticProblemArcLength was created as a subclass of StaticProblem, and automatically received all features of StaticProblem. Other example: class UnassembledMatrix stores elementary matrices (objects of type Matrix); summing two unassembled matrices means summing repeatedly small matrices, which is obtained by reusing the capabilities of class Matrix. A further example is if eigensolvers are to be implemented later on, class UnassembledMatrix will be reused without modifications;
- flexible. For example, a nonlinear-problem class can rely on a nonlinear solver of any type (e.g. NewtonRaphson, BFGS); similarly, this nonlinear solver can in turn use a linear solver of any type (e.g. Crout, GaussSeidel).

Furthermore, these well-closed modules are not black boxes. The programmer can easily open them up and understand their internal details, as substantiated by the numerous pieces of code displayed in this article, although use was made of a language famous for its abstruse syntax. Note also that making oneself acquainted with the program can be done progressively: the class hierarchy provides a dense overview; more insight is gained by scrutinizing the class specifications; eventually one can step into the implementation details of the functions [19]. This opens the gate for software design rules applicable to an isolated programmer or to a large team, which can speed up code design and implementation [50].

Appendix A

A.1. The BFGS algorithm (from [51])

given $x_0$
for $i = 1, \ldots, \text{maxIterations}$
(1) compute $G_i = g(u_{i-1})$.
(2) if ($\|G_i\|/\|G_1\| < \text{tolerance}$) set $x = x_{i-1}$ and quit.
(3) if $i = 1$
compute the initial jacobian $A_1$.
(4) if $i \neq 1$, compute and store $v_j$ and $w_j$:
\[
\begin{align*}
\gamma_i &= G_i G_{i-1}^- \\
v_j &= -s d_{i-1}^- G_{i-1}^- s - s d_{i-1}^- (G_{i-1}^-)^{1/2} G_{i-1}^- - \gamma_i \\
w_j &= d_{i-1}^- d_{i-1}^- G_{i-1}^- v_j
\end{align*}
\]
(5) compute the search direction $d_i$
\[
\begin{align*}
b^{(i)} &= G_i \\
for j = i-1, \ldots, 1 \\
b^{(j+1)} &= b^{(j+1)} + [w_j + 1 b^{(j+1)}] v_j + 1 \\
solve A_i d^{(j)} &= b^{(j)} \\
for j = 1, \ldots, i-1 \\
d^{(j+1)} &= d^{(j)} + [v_j + 1 d^{(j)}] w_j + 1 \\
d_i &= d^{(i)}.
\end{align*}
\]
(6) compute the new solution $x_j$ solve for $s$: $d_i^T g(x_{i-1} - s d_i) = 0$ (line search)
\[
x_i = x_{i-1} - s d_i
\]
A.2. C++ implementation: function solve() of class BFGS [51] (for economy, failure handling is not shown)

```c
FloatArray* BFGS :: solve ()
{
    UnassembledMatrix *A;
    FloatArray **v,**w;
    FloatArray *x, *gamma, *b, *d,*G_old,*G_new=NULL;
    double s, initialNorm, norm, prod;
    int i, j;

    // allocate space for vectors v[i] and w[i]
    v = allocFloatArray(maxIterations+1);
    w = allocFloatArray(maxIterations+1);

    // starting point
    x = problem->giveInitialGuess();

    // corrections
    for (i=1; i<=maxIterations ; i++) {

        // compute functional
        if (!G_new) 
            G_new = problem->computeFunctionalAt(x);

        // check convergence
        norm = G_new->giveNorm();
        if (i==1)
            initialNorm = norm;
        if (norm/initialNorm < tolerance)
            break;

        // compute jacobian
        if (i==1) {
            A = problem->computeJacobianAt(x);
            this->giveLinearSolver()->setLhsTo(A);
        }

        // compute and store v[i] and w[i]
        if (i>1) {
            gamma = G_new->minus(G_old);
            prod = d->dot(gamma);
            V[i] = G_old->multiply(-s*sqrt(prod/-s*d->dot(G_old)))
                ->subtract(gamma);
            w[i] = d->divide(prod);
            delete gamma ;
        }

        // compute the search direction d
        b = G_new->giveCopy();
        for (j=i ; j>=2 ; j--)
            b->add(w[j-1]->dot(b),v[j]);
        linearSolver->setRhsTo(b);
        d = linearSolver->computeSolution();
        for (j=2 ; j<i ; j++)
            d->add(v[j]->dot(d),w[j]);
        delete b;

        // compute the new solution x[i]
        G_old = G_new;
        G_new = this->performLineSearch(d,x,G_old,s);
        x->minus(s,d);
    }

    // release memory
    linearSolver->updateYourself();
    delete d; delete G_old; delete G_new; delete A;
    freeFloatArray(v,maxIterations+1);
    freeFloatArray(w,maxIterations+1);

    return x;
}
```
References


[29] Yu G, Adeli H. Object-oriented finite element analysis


