# Object-Oriented Finite Element Programming for Engineering Analysis in C++

Surendra Kumar

CSIR Centre for Mathematical Modelling and Computer Simulation
Council of Scientific and Industrial Research
Bangalore-560037, India.
E-mail: surendra@cmmacs.ernet.in, surend_kr@yahoo.com

*Abstract*— **Recently there has been growing interest in applying object-oriented approach to large-scale programs with a view to treating the various complexities within these. Software designed using an object-oriented approach can be significantly more robust than traditional software. More codes can be reused and it can be easier to refine, test, maintain and extend the software. Several benefits of this approach may also be observed in the area of finite element analysis. This paper describes an implementation of object-oriented programming to the finite element method for engineering analysis using C++, and illustrates the advantages of this approach.**

*Index Terms*—**engineering analysis, finite element method, data abstraction techniques, object-oriented programming, C++ programming language**

## I. INTRODUCTION

The finite element method (FEM) has been developed and applied extensively in various fields of engineering. It is a purely computer-oriented numerical tool and requires a sizable amount of programming effort. Major concerns in the development of FEM systems are placed on the computational efficiency of numerical algorithms. Therefore, procedure oriented programming techniques have been widely used and procedural programming languages such as FORTRAN, pascal etc. have been strongly supported.

The procedural approach has been proven effective in treating algorithmic complexity. However, such an approach does not address design and quality issues of the overall program. Software developed using this approach have intricate control strategies, internal data representation and algorithms. As a result, these codes face difficulties in their maintenance and extensions along the lines of the rapid advances in computational methods, computer software and hardware technology. The maintenance of finite element codes includes updating computational modules, extending the capabilities of the code and improving its performance.

Recently there has been growing interest in applying object-oriented approaches to large-scale programs with a view to treating various complexities within these. Some investigations have also been reported in the area of FEM [1-13]. It has been demonstrated that object-oriented programming can provide strong support to desirable features of FEM systems such as reusability, extensibility, easy maintenance, etc.

Mackerle [14] presents a list of published papers dealing with object-oriented programming applied to FEM and BEM. In an early paper, Zimmermann et al. [2] discussed the concept of object-oriented programming as applied to the implementation of the finite element method. Huang et al. [4] have proposed a knowledge base system in which an object-oriented analysis in the FEM domain is carried out by means of introducing entity analysis concepts. Archer et al. [5] demonstrated an object-oriented architecture for finite element analysis based on a flexible and extendible set of objects that facilitate finite element modeling and analysis. Zimmermann et al. [6] discussed the key features of an integrated environment of finite element related technique which includes an object-oriented graphic interactive environment and object-oriented operators for symbolic mathematical derivations. Yu and kumar [7] presented an object-oriented framework for implementing finite element method and explored ways to exploit the commonalities between various types of elements, loads, constraints and solvers so that duplication is reduced and software reuse is improved. Mackie [8] described a study into the object-oriented implementation of distributed finite element analysis on desktop computers using the .NET framework. Heng and Mackie [9] proposed the use of design patterns to capture best practices in object-oriented finite element programming.

While concept of basic classes like node and element are normally common in all designs, a series of papers and research work have been published on this subject depicting changes in the overall design or specific aspects in the design. Some of these papers discuss the object-oriented techniques in the context of specific problems. Tabatabai [10] suggested a object-oriented finite element environment for reinforcement dimensioning of two- and three-dimensional concrete panel structures. Pantale [11] presented benefits of using an OOP approach in comparison with traditional programming language approaches in the analysis of inelastic deformations and impact processes. Kromer et al. [12] described an approach to the design and implementation of a multibody systems analysis code using an object-oriented architecture. Franco et al. [13] discussed the aspects of the Object Oriented Programming used to develop a

Finite Element technique for limit analysis of axisymmetrical pressure vessels.

In the present investigation, object-oriented techniques have been applied in the development of a FEM software ``FACS'' for analyzing general kinds of engineering problems [15,16]. Although ``FACS'' was initially the acronym of Finite element Analysis of Composite Structures and was developed for finite element analysis of laminated composite structures [15], it presently can solve a good range of general kinds of structure analysis, heat transfer and metal working problems. C++ is used in the development of the program which has several features to support object-oriented programming and can provide high computing efficiency because of its compatibility with C.

## II. OBJECT-ORIENTED PROGRAMMING

Software applications are complex because they model the complexity of the real world. It takes a long time to implement a software and a sizable amount of programming effort is required. Therefore the primary goal when designing applications are easy maintenance, modification, reuse and extension. In this way, the behavior of the application can be kept appropriate and consistent during its life time.

Software designed using object-oriented programming technology can be significantly more robust than traditional software. More codes can be reused and it is easier to refine, test, maintain and extend the software.

The object-oriented approach attempts to manage the complexity inherent in real-world problems by abstracting out knowledge and encapsulating it within objects. *Objects* are self-contained entities (physical or conceptual) composed of both functions and data. That is, an object retains certain information and knows how to perform certain operations. Such act of grouping both information and operations into a single object is referred to as *encapsulation*. Objects which share the same behavior are said to belong to the same class. A *class* is a generic specification for any arbitrary number of similar objects. Objects that behave in a manner specified by a class are called *instances* of that class. All objects are instances of some class.

One object accesses another object by sending it a message. A *message* consists of the name of an operation and any required arguments. When an object receives a message, it performs the requested operation by executing a method. A method is the step-by-step algorithm executed in response to receiving a message whose name matches the name of the method. Limiting object access to a strictly defined interface such as the *message-send* allows another use of abstraction known as *polymorphism*. *Polymorphism* is the ability of two or more classes of object to respond to the same message, each in its own way.

Object-oriented programming languages support another abstraction mechanism: *inheritance*. *Inheritance* is the ability of one class to define the behavior and data structure of its instances as a superset of the definition of another class or classes. A program can often be organized as a set of trees or directed acyclic graphs of classes. That is, the programmer specifies a number of base class, each with its own set of derived classes. *Virtual functions* can often be used to define a set of operations for the most general version of a concept (a base class). When necessary, the interpretation of these operations can be refined for particular special cases (derived classes). Inheritance is an essential organization mechanism in an object-oriented programming and enables efficient and natural reusability of codes.

Another form of commonality can be expressed through *templates*. A *class template* is a special type of class definition that allows the programmer to generate an entire family of related classes, each of which is suited for working with a specific kind of data.

The above is a very brief introduction to the concept of object-oriented programming. A detailed account can be found in many literatures, e.g. [17-19].

## III. C++ PROGRAMMING LANGUAGE

The C++ language has evolved as the most widely used object-oriented programming language. It was developed from the C programming language and with few exceptions, retains C as a subset. The following list summarizes the merits of the C++ language [20]:

(a) C++ is designed to be a better C. It provides better support for the procedural and modular programming styles typically used in C. C++ does so without loss of generality or efficiency compared with C while remaining completely as a superset of C.

(b) C++ supports data abstraction, the ability to define and use new types. It has several features needed to make data abstraction effective. These consist of classes, the simplest access control mechanism, constructors and destructors, operator overloading, user-defined type conversions, exception handling and templates.

(c) C++ provides several facilities needed to support object-oriented programming. These consist of a class mechanism with inheritance and virtual function call mechanism, in addition to the facilities supporting data abstraction techniques.

The above list along with several other features of C++ makes it very appropriate for object-oriented design of a software application.

## IV. OBJECT-ORIENTED DESIGN OF FEM SOFTWARE

The most desirable types of general purpose finite element codes are those that are designed for comprehension, modification and updating with reduced effort. These desirable objectives are most easily met if the program is designed using object-oriented techniques. The FEM is by its nature a modular numerical tool. Object-oriented programming enables full advantage to be taken of this modularity. It reduces the scope for bugs by encouraging clearer thinking about the program design and allowing programs to be substantially altered without

the need to change the existing code. Of still greater importance is the fact that object-oriented programming allows much easier incorporation of new types of element, solution techniques and other facilities as they become available.

While performing an object-oriented design, the first task is to identify classes of objects that will model the application domain. Fortunately, it is not difficult to identify the objects in the FEM domain, because several entities such as element types, material properties, nodal points, elements, etc. can be extracted from the fundamental concepts of FEM. Several solid model entities such as points, lines, surfaces, volumes, etc. also can be directly identified as objects. However, in the FEM domain, there are a large amount of problem-solving activities which are hard to be directly identified as objects. Yet, their use and implementation may differ significantly from those in conventional packages. Some mathematical variables such as vector, matrix, etc. can also be designed as objects so as to hide their implementing details and several of these can be represented in template form so that they can take variable type (integer, real, double, etc.) as an argument. Various types of coordinate systems can also be identified as objects.

The remainder of this section briefly describes a kind of implementation of the object-oriented techniques in the development of present FEM code ``FACS''. Since the code consists of several hundreds of classes and class templates with multiple virtual inheritances, it is obviously difficult to discuss the original implementation here owing to the constraint of space. Therefore, only a few typical classes are discussed here and the description of each of these classes is kept very brief. Data and functions of each of these classes are kept to the minimum and arguments of the constructors, destructors and other member functions of the classes are also simplified for the sake of brevity.

Several basic classes such as ElemType, Material, Node, Element, etc. implemented in the present framework are traditional classes used for the representation of finite elements. Several specific classes are derived from these abstract classes. In most of the earlier investigations, these primitive objects directly interact with the problem domain. However, it can be revealed from the real world concept that in FEM domain, some super objects can be identified which are either aggregate of same objects or a superset of different objects. Therefore, we create an interface between the primitive objects and the problem domain by defining classes such as ElemTypeGroup, MaterialGroup, NodeGroup, ElementGroup, etc., which deal with group of same type of objects.

### A. Dynamic Data Structure

In a practical FEM application, the size of the different data items such as element types, material groups, nodal points, elements, etc. are not known before hand. It is, therefore, appropriate to organize these objects in a dynamic data structure such as a linked list so that they may be arbitrarily added, removed, found or operated on.

In the present code, a LinkedList class template has been defined and represented as a generic class which deals with lists of different objects:

```
template<class T> class LinkedList { // LinkedList of Ts
private:
    T* Head; // pointer to first element in the list
    /* . . . */
public:
    int count(); // number of items in the list
    T* find(int n); // find an item in the list
    void InsertSort(T* p, int n); // insert an item and
                         // sort the list
    void Delete(int n); // delete an item from the list
    /* . . . */
};
```

Various functions have been incorporated for the proper management of the list. For example, the function count() computes the number of objects in the list while the function find() locates a particular object in the list. The function InsertSort() is used to dynamically create an object using the constructor of its class and insert the object in the list. The list is arranged in the ascending order of the identification numbers (integers) of the objects. The function Delete() is used to dispose a particular object in the list using the destructor of its class. Several other facilities may also be incorporated in this class.

### B. Element Types

In finite element analysis, different types of one-, two- and three-dimensional elements are used for the discretization of the continuum. Each element has several characteristics such as number of nodes, degrees of freedom and shape functions. Given below is the definition of an abstract class ElemType which defines properties common to a variety of element types:

```
class ElemType {
protected:
    int num; // identification number in the list
    int NoNC; // number of nodes in the element
    int NoNodDOF; // number of nodal degrees of freedom
    ElemType* next; // pointer to next element type in the list
    /* . . . */
public:
    ElemType(); // constructor
    virtual ~ElemType(); // destructor
    virtual int NoElDOF() = 0; // pure virtual function
    /* . . . */
};
```

The data of this class include identification number in the list, number of nodes, nodal degrees of freedom, etc. The class has also a pointer to the next element in the list.

The functions such as NoElDOF() are declared to be pure virtual indicating that no definition is required for these functions in the abstract class and these functions can have different versions for different derived classes.

Now, several specific element type classes are derived from this abstract class. For example, ElemTypeB8 represents the class defined for eight-noded brick element:

```
class ElemTypeB8 : public ElemType {
private:
```

```
    int incompat; // flag for incompatible modes
    /* . . . */
public:
    ElemTypeB8(); // constructor
    ~ElemTypeB8(); // destructor
    int NoElDOF(); // number of element degrees of freedom
    /* . . . */
};
```

This class contains some additional data specific to the element type and correct implementation of functions. For example, data incompat is used to specify whether incompatible modes are to be added to shape functions.

Another class ElemTypeB8Lay can now be derived from ElemTypeB8 for layered version of eight-noded brick element.

A separate class ElemTypeGroup has been defined which deals with list of different element types:

```
class ElemTypeGroup : public LinkedList<ElemType> {
private:
    /* . . . */
public:
    ElemTypeGroup(); // constructor
    ~ElemTypeGroup(); // destructor
    ElemType* operator[] (int who); // subscript operator to
        // reference an element type
    void read(String* items); // read a particular element type
    void write(String* items); // list element type(s)
    /* . . . */
};
```

This class is derived from LinkedList class and so inherits all its operations for the proper management of the list. In addition, it has several other functions, e.g. reading and writing objects of different element type classes, etc.

### C. Materials

The constitutive equation relating the stresses to the strains depends on the material properties (isotropic, orthotropic or anisotropic), material behavior (linear elastic, elastic-plastic, viscoplastic, etc.) and dimensionality (plane stress, plane strain, axisymmetric or three-dimensional). Different material classes may be defined to describe different material properties or behavior. All these classes are derived from an abstract class Material which defines their common behavior:

```
class Material {
protected:
    int num; // identification number in the list
    float density; // mass density
    Material* next; // pointer to next material in the list
    /* . . . */
public:
    Material(); // constructor
    virtual ~Material(); // destructor
    virtual void Cmat(Matrix<float>& C) = 0; // pure virtual function
    /* . . . */
};
```

For example, class MaterialOrtho can be defined for orthotropic material with nine independent properties:

```
class MaterialOrtho : public Material {
private:
    float ex,ey,ez,nuxy,nuyz,nuxz,gxy,gyz,gxz; // nine independent
        // elastic constants
```

```
    /* . . . */
public:
    MaterialOrtho (); // constructor
    ~MaterialOrtho (); // destructor
    void Cmat(Matrix<float>& C); // calculates constitutive matrix
    /* . . . */
};
```

Similar to the class ElemTypeGroup, a class MaterialGroup is defined and derived from LinkedList class to deal with list of objects of different material classes.

### D. Nodes

In FEM, the boundary and interior of the region are subdivided by lines (or surfaces) into a finite number of discrete sized subregions or finite elements. A number of nodal points are established with the mesh. The nodal points are assigned identifying integer numbers beginning with unity and ranging to some maximum value. There are several data associated with each node and each data has several components depending upon the type of the element which the node is attached to. These include spatial coordinates (location and orientation), degrees of freedom (displacements and rotations), velocity components, acceleration components, concentrated loads, etc. One abstract class Node is defined consisting of generic data and member functions. Several implementations of nodal point based on the type of problems (structural, heat transfer, metal working, etc.) are derived from this class. Class Node is represented as given below:

```
class Node {
protected:
    int num; // identification number in the list
    int NoNodDOF; // number of nodal degrees of freedom
    float x,y,z; // coordinates
    float* DofVal; // degree of freedom values
    Node* next; // pointer to next node in the list
    /* . . . */
public:
    Node(); // constructor
    virtual ~Node(); // destructor
    /* . . . */
};
```

Next, a class NodeGroup is defined which consists of several member functions to deal with nodes, such as assigning coordinates and degree of freedom values. In addition, it inherits from class LinkedList all the operations for the management of the list of nodes.

### E. Elements

Like nodes, elements are assigned identifying integer numbers beginning with unity and ranging to some maximum value. Properties required to uniquely define an element include:

*Type* : a specific element type to which the element belongs.

*Material*: properties and behavior of the material which the element is made of.

*Element connectivity*: the list of global node numbers that are attached to the element.

*Element sources*: loads applied directly to an element.

Thus, an element class must be capable of representing all of these properties. Using these information, other element characteristics such as stiffness matrix, mass matrix and load vector can be calculated.

An abstract class Element given below is defined so that several classes characterizing different categories of elements can be inherited from it.

```
class Element {
protected:
    int num; // identification number in the list
    int type; // element type number
    int mat; // material number
    Element* next; // pointer to next element in the list
    /* . . . */
public:
    Element(); // constructor
    virtual ~Element(); // destructor
    virtual void StiffMat(Matrix<float>& EK) = 0; // pure virtual
            // function
    virtual void MassMat(Matrix<float>& M) = 0; // pure virtual
            // function
    /* . . . */
};
```

Here, the data type and mat are identification numbers of element type and material in the lists defined in classes ElemTypeGroup and MaterialGroup respectively. The class also specifies the calling interface for several functions such as StiffMat() and MassMat(). Now, one derived class ElementB8 can be defined for eight-noded brick element as given below:

```
class ElementB8 : public Element {
private:
    int Connect[8]; // element connectivity
    /* . . . */
public:
    ElementB8(); // constructor
    ~ElementB8(); // destructor
    void SDMat(float r, float s, float t, Matrix<float>& B);
    // computes strain-displacement matrix
    void StiffMat(Matrix<float>& EK); // computes stiffness matrix
    void MassMat(Matrix<float>& M); // computes mass matrix
    /* . . . */
};
```

The member functions StiffMat() and MassMat() for calculating stiffness and mass matrices require certain tools such as a numerical integration scheme, in addition to the data defined in this class. These tools are embedded within these functions in the present implementation although they can also be abstracted out as objects. Some functions such as calculating shape functions, their partial derivatives, strain-displacement matrix, etc. are defined in the corresponding element type classes. These data are required by the functions StiffMat() and/or MassMat() which provide some necessary information to corresponding element type object and request it to perform these operations. Method to determine load vector for element is also implemented in a similar manner.

Now, one ElementGroup class is defined which deals with the lists of elements and performs several tasks including assembly of element stiffness matrices and load vectors, and solution of the system equations:

```
class ElementGroup : public LinkedList<Element> {
private:
    /* . . . */
public:
    ElementGroup(); // constructor
    ~ElementGroup(); // destructor
    Element* operator[] (int who); // subscript operator to
            // reference an element
    void assembly(int loadcase); // assembly of stiffness
            // matrices and load vectors
    void SkySolve(int loadcase); // skyline reduction solution
    void FrontSolve(int loadcase); // frontal solution
    /* . . . */
};
```

Each of these tasks is decomposed into smaller tasks performed by different procedures implemented in this class.

*F. Other Classes*

Several other classes, in addition to those presented above, need to be defined in a complete finite element library. For example, a group of object classes are defined that perform modal generation and results processing.

Engineering variables such as strains and stresses can also be abstracted out. For example, a stress class is defined to provide several facilities including computation of principal stresses and their orientations, von Mises stresses and stress transformations.

Several utility classes are also defined to manage the finite element objects. The object oriented approach has facilitated the natural extension of the code in implementation of several other features. For example, the code consists of some classes which perform interpretation of FEM commands written in C language syntax. The code also includes several inherent classes for text editing facility which can be used for preparing command file and also for making reasonably well documentation related to the software [16].

As stated earlier, the above discussion on classes are only representative and actual implementation differs significantly. For example, in the original implementation a class Elem3DStBrick8Lay defining three dimensional eight noded structural layered brick element is virtually inherited from multiple base classes each having own set of base classes, as depicted in Figure 1.
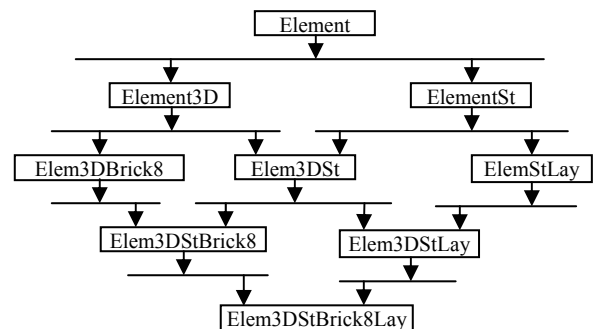


Figure 1. An example of inheritance of class Elem3DStBrick8Lay.

## V. FEM SOLUTION OF AN ENGINEERING PROBLEM USING 'FACS'

The present object-oriented FEM code ``FACS'' can be used to solve a considerable range of general kinds of engineering problems in the fields of structural analysis, heat transfer and metal working. It is worthwhile here to discuss the different steps and roles of different classes in solving an engineering problem. In order to do so, an example problem of foreign object impact on laminated composite plate is considered which was solved in the initial implementation of the code [15,21]. Problem description is sketched in Figure 2. The steps required to solve the problem using present object-oriented implementation are briefly described below.
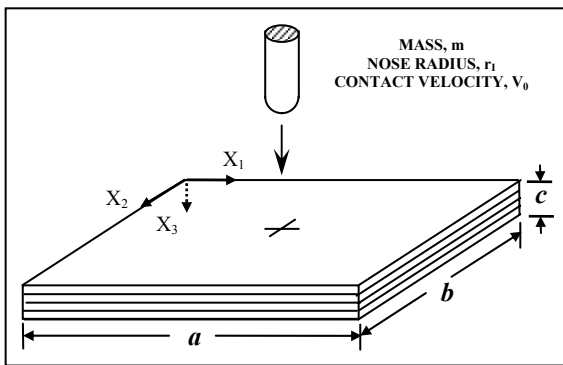


Figure 2. Problem description of transverse impact by a foreign object on a rectangular laminated plate.

### A. Discretization and Pre-Processing of Finite Element Model

The element type is defined to be layered version of eight-noded isoparametric brick element. This is done by requesting object of ElemTypeGroup class which dynamically creates an object of class ElemType3DStB8Lay and inserts in the list of element types. While the object is created, several characteristics of the element such as number of nodes, number of nodal degrees of freedom, etc. are also defined. Orthotropic material properties are defined by requesting MaterialGroup class which creates an object of class MaterialOrtho and assign values of the material density, elastic constants and strength values either using the constructor or member functions of the material class. The number of plies and orientation, thickness and material property reference number of each ply in the laminate are described using separate classes LayerData and LayerDataGroup.

Solid modelling, mesh control and mesh generation are performed using appropriate classes such as Keypoint and KeypointGroup, Line and LineGroup, Area and AreaGroup and Volume and VolumeGroup. While meshing, a number of nodal points are established which are created once the solid model objects request object of class NodeGroup to do so. The coordinates, etc. are assigned to each nodal point during the process. Similarly elements are defined by object of class ElementGroup which creates objects of class Elem3DStBrick8Lay based on the element type currently set and arranges them in a

list. The data input to each element include the element type reference number, material reference number and element connectivity. Plies within an element are automatically defined using the coordinates of its nodal points.

### B. Boundary Conditions and Loading

Degree of freedom constraints are applied either on nodes, keypoints, lines or areas using the member functions defined in these classes. Concentrated forces are specified at either keypoints or nodes. Surface loads are specified on lines and areas or on nodes and element faces. If loads are specified on the solid model, the node and element classes request the solid model objects for load data and transform these data to the equivalent nodal and element loads using appropriate numerical algorithms. For transient dynamic analysis of structures, several data such as time-step, number of load steps, etc. are defined within the class ElementGroup.

### C. Computation of Element Properties

Finite element transient dynamic equilibrium equation for structural analysis can be derived using Hamilton's variational principle. The equation for the case of no damping can be written as

$$[M]\{\ddot{U}\}+[K]\{U\}=\{F\}, \qquad (1)$$

where $[M]$ and $[K]$ are structural mass and stiffness matrices, $\{U\}$ and $\{\ddot{U}\}$ are the nodal displacement and acceleration vectors and $\{F\}$ is the applied load vector. These can be calculated as $[M]=\sum_e [m]_e$, $[K]=\sum_e [k]_e$ and $\{F\}=\sum_e \{f_b\}_e + \sum_e \{f_s\}_e + \{F_c\}$, (2)

where $[k]_e$ is the element stiffness matrix, $[m]_e$ is the element stiffness matrix, $\{f_b\}_e$ is the element body force vector, $\{f_s\}_e$ is the element traction force vector and $\{F_c\}$ is the global concentrated force vector.

The above finite element equation is integrated step-by-step with respect to time using Newmark direct integration method with constant average acceleration ($\alpha = 0.5$ and $\beta = 0.25$). After applying this method, Eqn (1) can be evaluated at time $t_{n+1}$ to form [15]:

$$[\hat{K}]\{U_{n+1}\}=\{\hat{F}_{n+1}\} \qquad (3)$$

Where $[\hat{K}]$ is effective stiffness matrix and $\{\hat{F}_{n+1}\}$ is the effective load vector defined as,

$$[\hat{K}]=[K]+\frac{1}{\beta(\Delta t)^2}[M] \qquad \text{and}$$

$$\{\hat{F}_{n+1}\}=\{F_{n+1}\}+[M]\left(\frac{1}{\beta(\Delta t)^2}\{U_n\}+\frac{1}{\beta(\Delta t)}\{\dot{U}_n\}+\frac{1-2\beta}{2\beta}\{\ddot{U}_n\}\right). \quad (4)$$

For each element $e$, the element stiffness matrix $[k]_e$ and mass matrix $[m]_e$ are calculated by member functions StiffMat() and MassMat() defined in class

Elem3DStBrick8Lay. This class also consists of functions to calculate the effective element stiffness matrix $\left[\hat{k}\right]_e$ and the effective element load vector $\left[\hat{f}_{n+1}\right]_e$. Here $\left[\hat{f}_{n+1}\right]_e$ contains element surface force vector and element body force vector which are computed using separate functions implemented in this class or its base classes.

### D. Assemblage of Elements and Solution of Equilibrium Equations

The effective element stiffness matrices and load vectors are assembled to constitute the effective global stiffness matrix $\left[\hat{K}\right]$ and effective global load vector $\left\{\hat{F}_{n+1}\right\}$. The function assembly() defined in the class ElementGroup gets the stiffness matrix and load vector from each element and assembles them in skyline vector storage mode. This compacted storage is used by the skyline reduction solution method SkySolve() to solve the system of equations. Another solution scheme called frontal solution method is implemented by function FrontSolve(). However, the complete assembly of all element contributions is not required in case of frontal solution method (FrontSolve()) in which assembly and reduction of equations are performed at the same time. Solution of the system of equations using any of two methods determines the global displacement vector $\left\{U_{n+1}\right\}$ at (n+1)th time-step.

Since the contact force at the impact point is not known at the beginning of each time-step, a Newton-Raphson iterative method is used to implement a non-linear contact law in the analysis. The contact force is calculated using a function defined within the class NodeGroup and its value is assigned to the nodal point at the impact point. Once the global displacement vector $\left\{U_{n+1}\right\}$ is known, the velocity vector $\left\{\dot{U}_{n+1}\right\}$ and acceleration vector $\left\{\ddot{U}_{n+1}\right\}$ are computed within the class NodeGroup. This procedure is repeated for each time-step.

Results of contact force and centre displacement are presented in Figure 2 for a test case of $[90_4/0_8/90_4]$ graphite-epoxy square plate of size 100 mm having clamped edges and impacted by a steel mass of 200 gm traveling at 5 ms$^{-1}$.

### E. Computation of Stresses and Post-Processing of Results

Once the displacements are known, element strains and stresses within each ply in the element are calculated using the member functions defined in the Elem3DStBrick8Lay class or its base classes. Failure criteria for prediction of matrix cracking and delamination are implemented in separate functions within the class. Result on prediction of impact-induced delamination for the above test case, as obtained directly by post-processor classes of the code, is plotted in Figure

3. The result depicts strength ratio, $e_d$ (calculated based on impact-induced delamination criterion [22]) in the bottom 0/90 plies interface of $[90_4/0_8/90_4]$ graphite-epoxy laminated square plate of size 100 mm which is having clamped edges and impacted by 200 gm mass at a velocity of 5 ms$^{-1}$. The region, where $e_d$ is greater than or equal to unity at the end of the impact, gives the estimation of the delamination size [22].
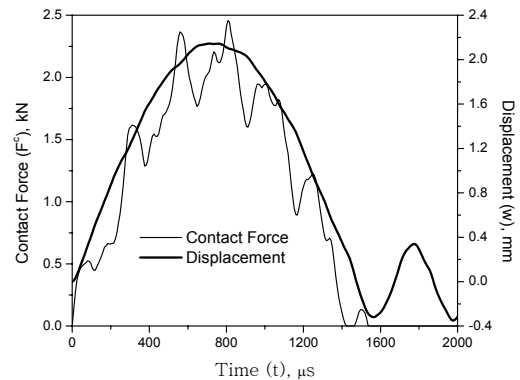


Figure 2. Contact force and centre displacement in graphite/epoxy plate ($[90_4/0_8/90_4]$) (dimension: a = b = 100 mm), having clamped edges and impacted by blunt-ended steel cylinder of nose radius 5 mm and mass 200 gm having initial velocity of 5 ms$^{-1}$.
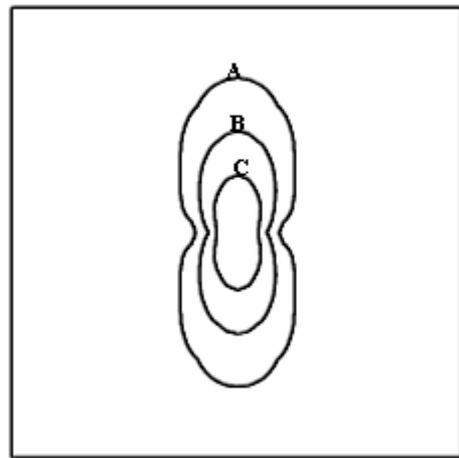


Figure 3. Maximum strength ratio, $e_d$ and predicted delamination size at 0/90 interface of $[90_4/0_8/90_4]$ graphite-epoxy square plate of size 100 mm having clamped edges and impacted by 200 gm mass at a velocity of 5 ms$^{-1}$ ($e_d$ values: A = 0.2, B = 0.5, C = 1.0).

## VI. DISCUSSION AND CONCLUSION

Finite element application programs are significantly large and complex, and therefore, key issues in developing these codes are easy testing, maintenance, extension and reusability. Object-oriented programming can provide stronger support to these desirable features than traditional programming. It concentrates on modelling the real world aspects of the system.

In object-oriented design, the approach used is to identify and implement a library of finite element data

types or classes that corresponds to high-level concepts in engineering and mathematics. Each class has well-defined roles and interfaces and therefore can be developed, validated and maintained independently. The use of data abstraction promotes the modularity of the finite element program and permits efficiency concerns to be more easily addressed at the implementation level of each class. It also makes complex data structures more convenient to use because the complexity is hidden by the abstract operators of the class. The concept of inheritance enables efficient and natural usability of finite element codes. Several new facilities such as new element types, materials and solution techniques may be incorporated with much reduced effort. The programmer may also be able to make use of existing code which may continue to run in the new system.

The present one is a kind of implementation of object-oriented approach to the design of FEM system. Concepts of object-oriented programming and some of the sample finite element classes implemented in the present software "FACS" have been briefly discussed. C++ was used in the development of the program because it has several features to support object-oriented programming. This language also provides hybrid object-oriented environment which allows the programmer to define objects, but also contains intrinsic data types that are not objects. The paper also discussed the role of different classes and their interfaces in solving a practical engineering problem using FEM.

Inspite of a number of advantages of object-oriented programming, special care must be taken to preserve the computational efficiency of the numerical algorithms. In order to do so, a hybrid implementation can be used at those places in the code which are computationally intensive and are required during the solution phase of the analysis. The static member function concept and other facilities in C++ such as use of this pointer allow the developer to implement a hybrid approach wherever required.

The general conclusion is that use of object-oriented programming with C++ is attractive for the development of finite element application programs.

REFERENCES

[1]  G. R. Miller, "An Object-Oriented Approach to Structural Analysis and Design", *Comput. Struct.*, vol. 40, pp. 75-82, 1991.
[2]  T. Zimmermann T, Y. Dubois-Pelerin and P. Bomme, "Object-oriented finite element programming. I. Governing principles", *Comput. Meth. Appl. Mech. Eng.*, vol. 98, pp. 291–303, 1992.
[3]  X.A. Kong, "Data design approach for object-oriented FEM programs", *Comput. Struct.,* vol. 61, pp. 503-513, 1996.
[4]  S. Y. Huang, S. Nakai, H. Katukura, and M. C. Natori, "An Object-Oriented Architecture for a Finite Element Method Knowledge-Based System", *Int. J. Numer. Meth. Engng.*, vol. 39, pp. 3497-3517, 1996.
[5]  T. Zimmermann, P. Bomme, D. Eyheramendy, L. Vernier and S. Commend, "Aspects of an object-oriented finite element environment", *Comput. Struct.*, vol. 68, pp. 1-16, 1998.
[6]  G.C. Archer, G. Fenves and C. Thewalt, "A new object-oriented finite element analysis program architecture", *Comput. Struct.*, vol. 70, pp. 63-75, 1999.
[7]  L. Yu and A.V. Kumar, "An object-oriented modular framework for implementing the finite element method", *Comput. Struct.*, vol. 79, pp. 919-928, 2001.
[8]  R.I. Mackie, "Object oriented implementation of distributed finite element analysis in .NET", *Adv. Eng. Software*, vol. 38, pp. 726–737, 2007.
[9]  B.C.P. Heng and R.I. Mackie, "Using design patterns in object-oriented finite element programming", *Comput. Struct.*, vol. 87, pp. 952–961, 2009.
[10] S.M.R. Tabatabai, "Object-oriented finite element-based design and progressive steel weight minimization", *Finite Elem. Anal. Des.*, vol. 39, pp. 55–76, 2002.
[11] O. Pantale, "An object-oriented programming of an explicit dynamics code: application to impact simulation", *Adv. Eng. Software,* vol. 33, pp. 297–306, 2002.
[12] V. Kromer, F. Dufossé, M. Gueurya, "On the implementation of object-oriented philosophy for the design of a finite element code dedicated to multibody systems", *Finite Elem. Anal. Des*., vol. 41, pp. 493–520, 2005.
[13] J.R.Q. Franco, F.B. Barros, F.P. Malard and A. Balabram, "Object oriented programming applied to a finite element technique for the limit analysis of axisymmetrical pressure vessels", *Adv. Eng. Software,* vol. 37, pp. 195–204, 2006.
[14] J. Mackerle, "Object-oriented programming in FEM and BEM: a bibliography (1990–2003)", *Adv. Eng. Software*, vol. 35, pp. 325–336, 2004.
[15] S. Kumar, Finite element analysis of impact-induced deformations, stresses and damages in composite laminates. Ph.D. thesis. Indian Institute of Technology, Kharagpur, 1998.
[16] FACS software, URL: http://www.facssoft.com.
[17] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, N.J., 1990.
[18] I. Graham, *Object Oriented Methods*, Addison-Wesley, Reading, MA, 1991.
[19] G. Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings, Menk Park, CA, 1991.
[20] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 2nd ed., 1991.
[21] B. Pradhan, and S. Kumar, "Finite Element Analysis of Low-Velocity Impact Damage in Composite Laminates", *J. Reinf. Plast. Comp.,* Vol. 19, pp. 322–339, 2000.
[22] H.Y. Choi and F.K. Chang. "A model for predicting damage in graphite/epoxy laminated composites from low-velocity point impact", *J. Compos. Mater.*, vol. 26, pp. 2134-2169, 1992.

**Surendra Kumar** is currently a senior scientist at CSIR Centre for Mathematical Modelling and Computer Simulation, Bangalore; which is a constituent laboratory of Council of Scientific and Industrial Research, India. He is a Ph.D. in mechanical engineering from Indian Institute of Technology, Kharagpur. He has about thirteen years of job experience in research and teaching. His major fields of interest include Computational Mechanics of Materials, Finite Element Method, Metal Working Processes and Software Development.