Object Oriented Design of Parallel and Sequential Finite Element Codes

Boyan Lazarov[†] **& Charles Augarde**[‡]

School of Engineering
University of Durham
[†]boyan.lazarov@durham.ac.uk [‡]charles.augarde@durham.ac.uk

1. Introduction

The Finite Element (FE) Method is a widely accepted general purpose numerical modelling tool. Typical FE programs consist of several hundred thousand lines of procedural code and many complex data structures. Alternative approaches, based on object oriented programming (OOP) concepts, are becoming popular as evidenced by the exhaustive bibliography of the use of OOP techniques in FEM [3]. OOP is based on the idea of "objects" that encapsulate both the data and the operations on the data. The implementation details are hidden and every object defines itself clear interfaces for communication. This makes code very simple to maintain and modify, and hence attractive for use on research FE codes.
In this paper we describe aspects of the design and implementation of an OOP based FE code primarily for use in large geomechanics simulations. The code allows various types of FE analysis, in both sequential and parallel computing environments, as well as the modelling of multi-field physical phenomena. A novel feature of this work is that, instead of using direct generalisation of the data structures used in a procedural code (i.e. representing a domain as an object and performing the calculations on the domain) each of the finite elements in this code is defined as a separate object. In addition, material constitutive behaviour and geometric primitives are also represented as objects. This gives great flexibility for adding new elements, materials or modelling new physical phenomena. Combination of different element types, for instance mixing structural (i.e. bending) elements with continuum elements, is also made simpler. Since the mathematical operations performed by an element are very similar to the operations performed by constraints or external loads, no distinction is made between them. Extending this idea further, every operation on elements, for example removing or adding elements to the model, is also represented as an element object.
The performance of a sequential FE code depends on the performance of the individual processor, the latency for access to data in the local memory and its bandwidth. In a parallel implementation performance additionally depends on the message activity, the load balancing and the bandwidth of message passing interface between the separate processes. The modular structure in the object oriented (OO) code clarifies the communication patterns and makes data decomposition and load balancing easier. This significantly improves scalability of the code.

2. Object oriented programming - some basic principles

Detailed descriptions of the main concepts of OO programming, namely encapsulation, inheritance and polymorphism, can be found in many books [6]. Here we merely provide reminders. A traditional (non-OO) program can be viewed as a logical procedure that takes input data, processes it and returns the output. The main program is built around simpler procedures or functions. In designing a procedural code, one focuses on how to define the logic rather than defining the data and its organisation. In contrast, an OO program is built around modules (objects) which encapsulate both the data and the operations (methods) on the data. An object can be viewed as an abstraction which relates variables and methods.

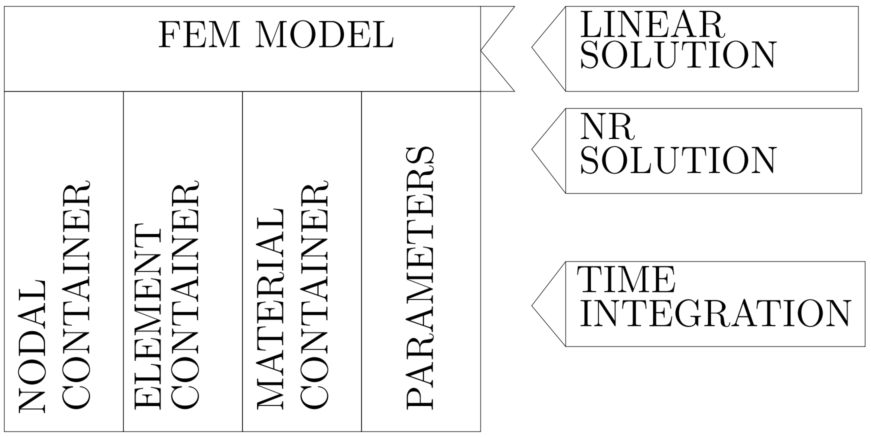

FIGURE 1. FEM data model

The first step in building an OO program therefore is to identify the objects and how they relate to each other. Once the objects are identified they can be generalised to a class of objects. The class is the actual definition of the data and the methods, and an object should be viewed as a particular instance of the class. The class definition can be reused in defining new classes. This property is known as inheritance. A new class can inherit data structures and methods from those previously defined and can add additional data structures and methods. In addition, the basic class can define access rules to its data. This characteristic (known as data hiding) provides greater system security and avoids unintended data corruption. Another very powerful property of the OO method is polymorphism, where an operation may exhibit different behaviour for different types of data. The behaviour depends not only on the operation but also on the data type or object used.

3. The non-linear FE method

The code described here is primarily a material non-linear FE [1, 2] program for solid mechanics, and uses the Newton-Raphson method to solve the equations arising from a discretisation of a weak form of the equations of equilibrium, compatibility and material constitutive behaviour. The field variable is displacement at nodal degrees-of-freedom (DOFs). The incremental displacement vector $\mathbf{u}$ at the end of a load step, is obtained by summing displacements at successive sub-steps $\Delta\mathbf{u}_i$ from

$$\mathbf{K}_{i-1}\Delta\mathbf{u}_i = \mathbf{r}_{i-1} \tag{1}$$

where $\mathbf{K}_{i-1}$ and $\mathbf{r}_{i-1}$ are the tangential stiffness and the residual (out-of-balance) force vector at step $i-1$ respectively. The residual at each substep is found from an equilibrium check against current stresses arising from the current state of deformation. The basic operations repeated at element level are therefore determination of the elemental stiffness and residual contributions.

4. Data model and object identification

The main steps performed in FE analysis are geometric modelling, domain discretisation, solution and post-processing. Many FE codes try to accomplish all these tasks in one program. Such an approach increases significantly the complexity of the code and reduces ease of maintenance and extension. The code developed here is for research and so we have concentrated on the solution phase only. The program kernel performs the following tasks: reading the discretised model and the solution parameters, solution and saving of results. The data model used in the program is shown in Fig(1). The FE data model can be represented as a set of nested object databases (DBs) for (1) the objects connected with the geometry description, (2) the elements, (3) the material models (or parameters for the PDE) and (4) the parameters needed for solution. Each DB is implemented as an object with common operations, namely *get object, add object* and *delete object.*

4.1. **Nodes.** The node object can take two forms in this code. The first is an object containing information on the degrees-of-freedom (DOFs) at that node. In this case, each node object carries a table which relates the DOFs to their global numbers. A node object of this nature might not include any information about its position. We do not restrict the numbers of DOFs at nodes to be the same throughout the domain, a restriction which prevents mixing of element types in other codes. The DOFs are assigned to the node by the elements which are connected to this node. The DOFs table consists of unique DOFs, i.e. the container cannot contain two displacements in X direction or two rotations around Y axis. Once the DOFs are set by the elements, the FE data model will assign a global number to each DOF by cycling through all nodes. Other node objects are possible (called fake nodes) which are used to implement the constraints, such as prescribed boundary conditions. The Nodal DB (see Fig(1)) contains all nodes of all types.

4.2. **Elements and operations on elements.** We have indicated, above, that the element object contains information about the topology of the element. In addition material information (including history variables, such as hardening parameters) is kept internally. The common operations performed by every element are the formation of the element stiffness matrix and contribution to the residual vector and the storage of these contributions in their global counterparts. The notion of an element object is further generalized in the code to include every operation on an element or set of elements. For example, consider the modelling of tunnelling in geomechanics. To simulate the tunnelling process, excavation must be modelled. Implementation by removing elements on a global level will limit the program extensibility. The easiest way, therefore, to incorporate such a feature is to implement the excavation process as an “element” which will remove elements from the element DB.

4.3. **System input or external loads.** Considering small displacement analysis for simplicity, external loads will contribute to the residual vector only. However, to allow extension to finite strain, loads are considered here as another type of element object. This avoids adding an additional DB, specifically for loads. The drawback is that a small computational overhead is added in the case when the load does not contribute to the tangent matrix (i.e. the small displacement case).

4.4. **Constraints.** In most FE codes, constraints (e.g. prescribed displacements) are either applied by removing the constrained DOF from the node or by modifying the global tangent matrix. Both approaches are unsuitable for this code as they are global operations requiring communication between all processes. The only way to add constraints without changing the data organisation, and the relations between objects, is by applying the constraints either using the Penalty method or using Lagrange multipliers [1, 4]. The advantage of using Lagrange multipliers is that the applied boundary conditions or constraints are exactly satisfied. The drawback is that each constraint increases the length of the vector of unknowns by one. Therefore, for each constraint equation an additional “fake” node is added. If the Lagrange multiplier method is applied on a subdomain then the DOF representing the multiplier field will be associated with a geometric node.

4.5. **Materials.** Objects representing materials are the most difficult to unify, because they are directly dependent on the underlying physical problem. In [5] element-to-material communication is achieved by introducing another abstraction, the *material point*. Such an approach adds, in some cases, additional data which it is not necessary to store in the element. For example, elasticity parameters where the domain is uniformly linear elastic. The material point concept solves the problem with the internal history variables, for example the plastic strain in plasticity, but does not solve the problem of unifying the output from the material model. Let us consider a solid mechanics problem including heat transfer. The mechanical element will integrate the residual over the stress, however the heat transfer element will integrate the residual over the temperature. The output from the material object given material point as an input in the first case will be stress in the second case will be temperature. Therefore it is necessary to use these element objects with some care.

5. Solution algorithms. Vectors and matrices

The solution algorithms are designed as procedures (objects) which plug into the FE model. The solution procedure works with global vectors and matrices rather than elements and nodes. The connection between the algorithms and the FE model are the global vectors and matrices. They are implemented as global objects with various methods for manipulation and updating. Using this abstraction makes the underlying storage independent from the application. For large matrices, sparse storage is used and objects are the perfect tool for hiding the internal organization.

6. Parallel implementation

The main challenge in the parallelisation of a numerical algorithm is load balancing and syncronization between the different processes. The clear modular structure of the proposed data model makes data decomposition and the implementation of communications very easy. The data organization Fig(1) is the same on each process. The communications are organized on the element, the node and the material DB levels. Nodes and element objects are not allowed to communicate directly between processes. If an object requires an object from another process, the request and the data synchronization is done by the DB responsible for managing this object. For example, if an element requires a node, the request has to be send to the nodal DB. The majority of communications are for synchronizing the nodes between the different processes. Every operation over a set of elements (e.g. removal to model excavation) is considered to be performed by an element object. Such operations are global objects and they require communication, but for the rest of the element objects "inside" processes, no communication or synchronization is necessary. The current solution vector is kept in a distributed form between the processes, however communication between these objects is organized in a separate mathematical library for vectors and matrices. Load balancing will be added in the future as an element object.

7. Remarks

The described data structures and algorithms have been successfully implemented. The C++ language has been chosen due its implementation efficiency and portability. The described data organization leads to modular implementation and easy extendibility of the code. The program provides robust computational tool for modelling complex processes such a tunnelling in geomechanics and has good possibilities for many other interesting physical problems.

8. Acknowledgements

This research is funded by the UK EPSRC under grant no GR/S42712/01.

References

[1] Bathe, K.J.,*Finite Element Procedures*, Prentice Hall, **1996**

[2] Belytschko,T., Liu,W.K. and Moran, B.,*Nonlinear Finite Elements for Continua and Structures*, Wiley, **2000**

[3] Mackerle,J., Object-oriented programming in FEM and BEM a bibliography (1990-2003), *Advances in Engineering Software*, 35, **2004**, 325-336

[4] Dubois-Pelerin, Y. and Pegon, P., Linear constraints in object-oriented finite element programming, *Comput. Methods Appl. Mech. Eng*, 154 , **1998**, 31-39

[5] Patzak, B., Bittnar, Z., Design of object oriented finite element code, *Advances in Engineering Software*, 32 , **2001**, 759-767

[6] Stroustrup, B., The C++ Programming Language,*Addison-Wesley*, **1997**