Universitat Politècnica de Catalunya

Programming for Engineers and Scientists

Master's Degree in Numerical Methods in Engineering

Design of a FE Code in C++

Authors:

Pau Márquez (pmarquez115@gmail.com)

Iván Pérez (ivan.perez.garcia@gmail.com)

Diego Roldán (diegoroldan.731@gmail.com)

Supervisor:
Prof. S. Zlotnik

Academic Year 2019-2020

Contents

1 Introduction and motivation

Up to this point in the course, the focus has been placed in the function, and we have designed a FE program created with many functions that each specify a process in the program. In procedural programming with Matlab we declared our data outside from the functions and then we passed in the data to the those functions, completing the global computation by breaking it in several sub-tasks. Clearly, one of the limitations that we saw during the completion of the first part was that all the functions were entitled to know the structure of the data, meaning the functions expected specific data type as input arguments. Therefore, whenever some bug, improvement or modification was applied to the code, all the functions had to be modified to handle the new format of the data. This clearly has a ripple effect on the time needed to modify and test all the functions on large codes, as happens when programming complex finite element programs, which involve many entities and can clearly be organized in more compact manners.

The purpose is then to extend all these features to object-oriented programming, which is supported in C++. These classes will module the entities of the code, which is a way of helping out with the growing complexity of the program. The classes will now contain data and operations dealing with that data. Summing up, the mechanisms of object-oriented programming that will help as well with complexity will be:

- Abstraction allows for generality and removes irrelevant details of the code, and defines the public interface of the classes and how their objects will interact with other objects.
- Inheritance will allow to implement class hierarchies in which the lower level classes will inherit data and methods of the higher level classes
- Polymorphism represents the fact that in the code the same message can be passed to different classes but its meaning be different depending on the class.
- Encapsulation is eventually responsible for hiding the implementation details while maintaining visible the class public interface.

These are all good properties that we will try to implement in the design of the code.

2 Designed classes and methods

The organization of the finite element classes will consist on three different levels, or at least that is the idea that we pretend to carry out. Each level will implement different data structures that will be used successively in the following levels. The justification for this is for clarity purposes and to have a clearer image of hierarchies. Another possibility would be to use more levels, for example one in which the computations of matrix and vectors operations take place, or less levels, but we think three is enough complexity for the type of problem.

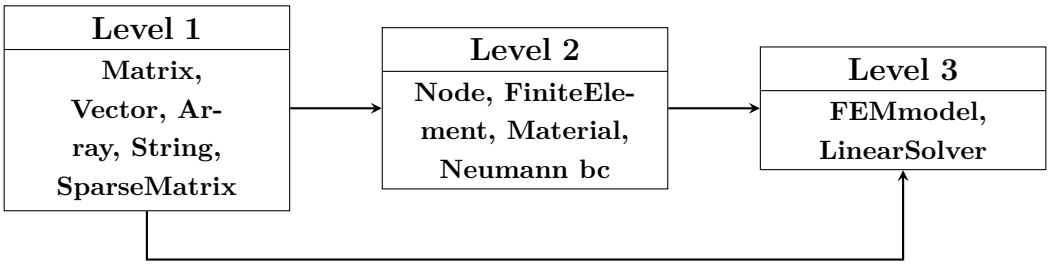

Figure 1: Organization levels of the FEM clases

2.1 First level

The first level will implement basic data structures concerning vectors, arrays and matrices. That is, it will contain classes for each kind of matrix that we may find in the problem, meaning that design should include the possibility to work with symmetric, sparse or dense matrices, which would have different methods.However, it will only be defined a generic class `Matrix` and a generic class `Vector`. There could also be a class `String` which manipulates strings.

This first level also manages the memory allocation. The reason why it is put in this level is that restricting the implementation of memory allocation procedures in the first level simplifies error corrections as one does not need to look at more deep levels of the code.

1 `class Matrix`

Brief description: Implementation of a matrix class that can be used to compute mathematical operations with matrices and how the matrix will interact with other objects, such as vectors and scalars.
Public: Operator overloading, matrix mathematical operations, matrix-scalar operations, matrix-vector operations, access to the individual elements.
Private: Number of rows, number of columns and matrix data.

The vector class would also have the same characteristics as that of the Matrix, although simpler. There is also necessary the overloading of the operators of multiplication and access to the matrices and its product with vectors and scalars. Another possibility noted in the diagram is to include another class, very similar to the Matrix class, with sparse matrices, which would turn out indispensable for large problems involving many unknowns.

2.2 Second level

The second level of the hierarchy has the classes describing the characteristics of the problem. It would have a `Nodes` class with the variables defining the problem dimension, the total number of nodes and the nodal coordinates in the private definition. It would also have a class denoted as `FiniteElement` , from which all element-dependent variables are derived, and it will depend on the choice of the user. There will also be a class `Material` to represent the material

properties and eventually two classes with the prescribed degrees of freedom `Prescribed_dof` and a `Neumann_bc` class to account for non-essential boundary conditions.

2 **class Node**

Brief description: Node class is the minimum entity that defines the domain and can contain information of its id, position, and solution.
Public: get node information (id, coordinates), get solution, store solution, construct node information (id, coordinates).
Private: Node id, vector of coordinates, data solution.

3 **class FiniteElement**

Brief description: Contains all data related to the element entity.
Public: Set element identifier, set element conductivity, set local connectivity, set local coordinates, get local connectivity, get local stiffness matrix, get local nodal force vector, get nodes per element.
Private: Element identifier, local coordinates and local connectivity of the element, number of nodes per element, number of Gauss points, dimension of the problem, conductivity of the material, number of Gauss points, vector with Gauss weights, matrix with shape functions evaluated at each Gauss point, matrix with derivatives of elemental shape functions at each Gauss point, local stiffness matrix, local nodal vector.

4 **class Material**

Brief description: Contains a description of the material behavior.
Public: Set material properties and return material properties, defined inside void functions.
Private: Material conductivity.

5 **class Neumann_bc**

Brief description: Computes the contribution of the nodes on the boundary to the right-hand side vector.
Public: Void compute contribution.
Private: None.

2.3 Third level

The third level contains the solver class, which in this case will be a Linear Solver which employs Gaussian elimination.

6 `class LinearSolver`

Brief description: The finite element solution is calculated by a direct method.
Public: Apply Neumann and Dirichlet boundary conditions inside void functions. Solve the system by inversion in solver void function.
Private: Global matrix of the system, right-hand side vector and solution vector.

The class `LinearSolver` is developed on the already developed matrix and vector classes. The Dirichlet boundary conditions will be enforced through the standard row substitution technique, and the Lagrangian multiplier method will not be coded, although it is mentioned here as a possible extension.

If the problem type involved non-linear terms, then obviously this would imply the user to select parameters such as the maximum number of iterations, norm type, convergence criterion and precision for the iterative solver, and new functions should be created inside the class to allow for this computation. Then, as an extension, classes should be defined that store non-symmetric sparse matrices. Gauss elimination should then be available along with other iterative numerical methods for all matrix classes, such as Jacobi, Gauss-Seidel, multi-grid or the pre-conditioned conjugate gradient method. However, this will be out of the scope of the code to be developed.

3 Strong points and limitations

Now the data and the operations are together in the class where they belong, and that is better to having a lot of functions, each passing and receiving data. As explained in the introduction, there are a few properties of object-oriented programming which makes the code easier to maintain and modify, as well as making it reusable in other programs since classes are encapsulated units of data and operations. Other advantages of the design developed here is that the code is of higher quality and more easily to maintain as the classes have already been tested and do not change from application to application.

As for the bad properties of the mentioned design, it is clear that there may be entities which are not best decomposed into a class, and a simple function would serve the purpose. The designed program is probably also slower and more complex to design and understand than a common Matlab program, although it is clearly of higher quality if analyzed in terms of its re-usability properties, among others.

4 Concluding remarks

This work has established the procedure and methodology for constructing and solving the Poisson equation in a C++ environment. It has been clear that designing a program in C++ takes more up-front design time to create good models and hierarchies that when programming it in Matlab.