Object-oriented programming in boundary element methods using C++

Wenqing Wang*, Xing Ji, Yuangong Wang

Department of Engineering Mechanics and Technology, Laboratory of Solid Mechanics, Tongji University, 1239 Siping Road, Shanghai 200092, China

Abstract

A new method of writing boundary element programmes using the programming paradigms known as object-oriented programming (OOP) is presented in this paper. Among OOP paradigms, C++ is more suited to numerical programming than a pure OOP language, and the fact that C++ is chosen in this paper intends to illustrate the efficiency of object-orient boundary element programming. The advantage of object-orient boundary element programming—i.e. being superior to other paradigms such as FORTRAN—is shown by discussion of a sample C++ code of boundary element methods. © 1998 Published by Elsevier Science Ltd. All rights reserved.

Keywords: C++; Boundary element programmes; Object-oriented programming

1. Introduction

There are several kinds of programming paradigms such as procedural oriented programming (POP) and object oriented programming (OOP). FORTRAN, a typical POP language, long considered the language of reference for scientific programming, is now giving some way to highly object oriented languages such as C++. Object-oriented languages allow one to write programmes in a much more rational manner than FORTRAN. The maintenance of programmes written in object oriented paradigms is much more easy too. Recently, the interest of using object-oriented programming (OOP) in numerical analysis increased rapidly for its efficiency and understandable data organisation. Among OOP paradigms, C++ is more suited to numerical analysis than a pure OOP language. C++ inherits the qualities of OOP, which are data encapsulation, class inheritance and polymorphy [1, 2], and the speed of traditional C. Moreover, C++ is the most commonly used and powerful OOP paradigm to date. Recently, some works on the use OOP techniques in finite element programming have been published [3–8].

In this article, an approach of object-oriented programming in boundary element methods using C++ is presented. A sample C++ code of boundary element methods is discussed in detail in order to demonstrate the efficiency of object-oriented boundary element programming.

2. Basic principles of object-oriented programming

The term object-oriented means a new name for an abstract data type, where data and operations are encapsulated into objects. Each object has its own private memory and local function, resulting in modularity and information hiding. Object and data are a concrete form of type theory, and each datum (object) is considered to be an element of type (class). Types can be related through subtype and supertype relations. It is a way of organising and sharing code in a large system. Individual programmes and the data, the manipulation are organised into a tree structure. Objects at any level of this hierarchy inherit all attributes of high level objects. The inheritance makes it possible for similar objects to share programme code and data. The true object in OOP is the class. It is a user-defined type. Through it one can manage data and their manipulations to simulate the real world. The procedure of a class definition is data abstract. Each object is an instance of a class.

Thus, programmers who develop software in an OOP way can only concentrate on data abstract and clearly object hierarchy design. Of course, the implementation of code to different platforms must be considered while C++ is involved in programming.

3. Boundary element methods with OOP

Now that the basic concept of OOP has been explained, the design of object-oriented boundary element programming can be discussed. At first, it is necessary to design a hierarchy object of boundary element methods and use the
An appropriate choice of the C++ class to implement the boundary methods is essential. Without loss of generality, the elasticity problem is considered here. The statement of the problem is given in Table 1. We can consider the problem as an object, and that means a base class, which describes the general problems, should be designed first and then classes of describing the specific problems can be derived from it. A simple hierarchy of elasticity objects and its related classes are shown in Fig. 1.

3.1. Base class
The base class must be so designed that it can describe the common attributes of the considered problems. Class BasicMessage (Fig. 2) is designed as a base class. It describes the common attributes of elasticity problems. Geometrical data, number of elements, boundary nodes and inner points are basic attributes of an element, and all of them can be considered in a base class. It performs to prepare the geometrical data and memory, which can be inherited by the future class (2D, 3D). Commonly, it can be termed as parents class.

3.2. Classes for BEM solution to the specific problems
This kind of class must be so designed that it can give a clear outline of the whole concept of BEM solutions. Class BasicMessage (Fig. 2) is designed as a base class. It describes the common attributes of elasticity problems. Geometrical data number of elements, boundary nodes and all of them can be basic attributes of an element, and all of them can be considered in a base class. It performs to prepare the geometrical data and memory, which can be inherited by the future class (2D, 3D). Commonly, it can be termed as parents class.

### Table 1

<table>
<thead>
<tr>
<th>Problem</th>
<th>Governing equations</th>
<th>Interpolation functions</th>
<th>Boundary values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displacements at inner points</td>
<td>$u = \Phi^T \Psi = \Psi^T \Phi$</td>
<td>$u = \Phi^T \Psi = \Psi^T \Phi$</td>
<td>$u = \Phi^T \Psi = \Psi^T \Phi$</td>
</tr>
<tr>
<td>Stresses at inner points</td>
<td>$s_{ij} = \frac{\partial^2 \psi}{\partial x_i \partial x_j}$</td>
<td>$s_{ij} = \frac{\partial^2 \psi}{\partial x_i \partial x_j}$</td>
<td>$s_{ij} = \frac{\partial^2 \psi}{\partial x_i \partial x_j}$</td>
</tr>
</tbody>
</table>

The base class must be so designed that it can describe the common attributes of the considered problems. Class BasicMessage (Fig. 2) is designed as a base class. It describes the common attributes of elasticity problems. Geometrical data number of elements, boundary nodes and all of them can be basic attributes of an element, and all of them can be considered in a base class. It performs to prepare the geometrical data and memory, which can be inherited by the future class (2D, 3D). Commonly, it can be termed as parents class.
```cpp
class BEM_Object : public BasicMessage {
    protected:
        float* ConditionOfNode;
        Bitcard* TypeOfS_value;
        MaterialCh ThisMaterial;
        Matrix* Hmatrix;
        Matrix* Gmatrix;
        Matrix* D_and_S;
        Matrix* Element_Node;
    public:
        BEM_Object();
        ~BEM_Object();
        // configure BEM
        void InitialOut();
        virtual double Jacob(double);
        virtual double Found_Trac(int,int,Point&,Point&,Point&);
        virtual double Found_Strain(int,int,Point&,Point&,Point&);
        virtual double Found_Disp(int,int,Point&,Point&);
        virtual void ComputeH_Matrix(Point&);
        virtual void ComputeG_Matrix(Point&);
        virtual void ComputeD_and_S(Point&);
        virtual void Assembler();
        virtual void Results();
        friend double Shape_Fun(int,double);
        friend double Inteplotate(double,double,double,double);
        void EquationSolve();
};

Fig. 3. Class of boundary element.

double GP,Rx,Ry,Xj,Yj,Xk,Yk,Jac;
int i,j,l,Begin,End,m_col;
//initial
m_col = DOF*Order;
for (i=1; i<=Order; i++) {
    for (j=1; j<=m_col; j++) { (*Hmatrix)(i,j) = 0.0; }
    Xj = (*Element_Node)(1,1)-
        2.0(*Element_Node)(2,1)+(*Element_Node)(3,1);
    Xk = 0.5(*Element_Node)(3,1)-(*Element_Node)(1,1);
    Yj = (*Element_Node)(1,2)-
        2.0(*Element_Node)(2,2)+(*Element_Node)(3,2);
    Yk = 0.5(*Element_Node)(3,2)-(*Element_Node)(1,2);
// begin computing H matrix
for (l=0; l<NumInteg; l++) {
    GP = GaussPoint[l];
    Rx = Inteplotate(GP,(*Element_Node)(1,1),
                    (*Element_Node)(2,1),(*Element_Node)(3,1));
    Ry = Inteplotate(GP,(*Element_Node)(1,2),
                    (*Element_Node)(2,2),(*Element_Node)(3,2));
    Jac = this->Jacob(GP);
    Point FieldPoint(Rx,Ry);
    Rx = (GP*Yj+Yk)/Jac;
    Ry = -(GP*Xj+Xk)/Jac;
    Point Norm(Rx,Ry);
    for (int k = 0; k < Order; k++) {
        for (i=1; i <= DOF; i++) {
            Begin = k*DOF+1;
            End = k*DOF+DOF;
            for (j=Begin; j <= End; j++) {
                int jj = j-k*DOF;
                (*Hmatrix)(i,jj) = Jac*Weight[l]*Shape_Fun(k,GP)*
                    this->Found_Tract(i,jj,
                    LoadPoint,FieldPoint,Norm);
            }
        }
    }
};

Fig. 4. Member function of computing coefficient matrix H.
```
**void BEM_Object::ComputeG_Matrix(Point& LoadPoint)**

```cpp
complex Fd;
double GP,coef,Rx,Ry,tmp;
int i,j,i1,j1,k,Begin,End;
int _f_index;
jj = DOM*Order;
for (i=1; i<=Order; i++) {
  for (j=(i-1); j<=Order; j++) {
    (*Gmatrix)(i,j) = 0.0;
  }
  _f_index = -1;
}
// coefficient matrix
for (l = 1; l <= Order; l++) {
  Rx = LoadPoint.getxy() - (*Element_Node)(1,1);
  Ry = LoadPoint.getxy() - (*Element_Node)(1,2);
  GP = sqrt(Rx*Rx+Ry*Ry);
  if (GP < 1.0e-6) _f_index = l-1;
}
for (l = 0; l < Order; l++) {
  for (i=1; i<=DOM; i++) {
    Begin = 1*l*DOM1;
    End = 1*l*DOM1;
    for (j=Begin; j <= End; j++) {
      for (int k=0;k < NumInteg; k++) {
        GP = GaussPoint[k];
        Rx = Inteplotate(GP,(*Element_Node)(1,1),
                        (*Element_Node)(2,1),(*Element_Node)(3,1));
        Ry = Inteplotate(GP,(*Element_Node)(1,2),
                        (*Element_Node)(2,2),(*Element_Node)(3,2));
        Point FieldPoint(Rx,Ry);
        Fd = this->found_Disp(i,jj,LoadPoint,FieldPoint);
        (*Gmatrix)(i,j) += this->Jacob(GP)*Weight[k]*
                        Shape_Fun(1,GP)*imag(Fd);
        if ( (_f_index==0)&&(l==1) ) {
          GP = 2*Q_Gauss[k]-1;
          coef = log(1./Q_Gauss[k]);
          Rx = Inteplotate(GP,(*Element_Node)(1,1),
                          (*Element_Node)(2,1),(*Element_Node)(3,1));
          Ry = Inteplotate(GP,(*Element_Node)(1,2),
                          (*Element_Node)(2,2),(*Element_Node)(3,2));
          Point FieldPoint(Rx,Ry);
          (*Gmatrix)(i,j) += 2.*this->Jacob(GP)*Q_Weight[k]*
                          Shape_Fun(1,GP)*real(this->found_Disp
                          (i,jj,LoadPoint,FieldPoint))/coef;
        } else if ( (_f_index==2)&&(l==1) ) {
          GP = 1.0-2.0*Q_Gauss[k];
          coef = log(1./Q_Gauss[k]);
          Rx = Inteplotate(GP,(*Element_Node)(1,1),
                          (*Element_Node)(2,1),(*Element_Node)(3,1));
          Ry = Inteplotate(GP,(*Element_Node)(1,2),
                          (*Element_Node)(2,2),(*Element_Node)(3,2));
          Point FieldPoint(Rx,Ry);
          (*Gmatrix)(i,j) += 2.*this->Jacob(GP)*Q_Weight[k]*
                          Shape_Fun(1,GP)*real(this->found_Disp
                          (i,jj,LoadPoint,FieldPoint))/coef;
        } else if ( (_f_index==1)&&(l==1) ) {
          GP = -Q_Gauss[k];
          coef = log(1./Q_Gauss[k]);
          Rx = Inteplotate(GP,(*Element_Node)(1,1),
                          (*Element_Node)(2,1),(*Element_Node)(3,1));
          Ry = Inteplotate(GP,(*Element_Node)(1,2),
                          (*Element_Node)(2,2),(*Element_Node)(3,2));
          Point FieldPoint(Rx,Ry);
          tmp = this->Jacob(GP)*Shape_Fun(1,GP)*
               real(this->found_Disp(1,jj,LoadPoint,FieldPoint))/coef;
          GP = Q_Gauss[k];
          Rx = Inteplotate(GP,(*Element_Node)(1,1),
                          (*Element_Node)(2,1),(*Element_Node)(3,1));
          Ry = Inteplotate(GP,(*Element_Node)(1,2),
                          (*Element_Node)(2,2),(*Element_Node)(3,2));
          Point FieldPoint(Rx,Ry);
          (*Gmatrix)(i,j) += this->Jacob(GP)*Shape_Fun(1,GP)*
                           real(this->found_Disp(1,jj,LoadPoint,FPoint))/coef;
          (*Gmatrix)(i,j) += tmp*Q_Weight[k];
        } else {
          (*Gmatrix)(i,j) += this->Jacob(GP)*Weight[k]*
                          Shape_Fun(1,GP)*real(Fd));
        }
      }
    }
  }
}
```

**Fig. 5.** Member function of computing coefficient matrix G.
C++ object way. From Table 1 we know that $H$, $G$, $D_{ik}$ and $S_{jk}$ should be calculated in each element, so we thought of these as the behaviour of a BEM object. The class is so designed that it can compute coefficient, assemble system equations and solve the system equations. The attributes of this class are

- `ConditionOfNode` stores the boundary values;
- `TypeOfB_value` stores the type of given boundary values;
- `Thismaterial` a class, stores the characteristics of material;
- `Hmatrix`, a matrix class, records coefficient $\int_{\Gamma} \frac{u}{p} \psi^T d\Gamma$;
- `Gmatrix`, a matrix class, $\int_{\Gamma} \psi^T d\Gamma$;
- `D_and_S`, a matrix class, records coefficient $\int_{\Gamma} S_{ij} \Phi^T d\Gamma$ and $\int_{\Gamma} D_{ik} \Phi^T d\Gamma$ for stress and strain analysis;
- `Element_Node`, a matrix class, stores coordinates of nodes on each element temporarily.

The methods of `BEM_Object` or the task of this class should be performed by member functions as following

- Computing coefficient $G_{ij}$ on each element by function `ComputeG_Matrix(Point&)`;
- Computing coefficient $H_{ij}$ on each element by function `ComputeH_Matrix(Point&)`;
- Assembling coefficient matrices $G$ and $H$, forming system equations with boundary applied. This is implemented by function `Assembler()`;
- Solving system equations by function `EquationSolve()`;
- Output the results by function `Results()`.

In `BEM_Object`, the keywords “protected” and “public” are the access specifiers. Only this class itself or its child class can access attributes and functions that are listed “protect”, and that assure the mechanism of encapsulation. Functions that are listed “public” can be referred by other classes. Keyword “virtual” states that member function declared by that keyword allows the appropriate version of the member functions to be used regardless of the circumstances in which it is being called. With this definition, polymorphism is achieved, and that means the multiplication routines can be implemented.

When 3D problems are involved, a class of 3D can be easily created by derived from class `BEM_Object` and just performing multiplication of the virtual functions of `BEM_Object`. The new class can inherit whole attributes of this class and have access to the parent class. This is also a kind of polymorphism.

The other classes used in this class are `Point`, `MaterialCh`, `Matrix`, `BitCard`. `Point` is designed to describe the real point in a Cartesian system. `MaterialCh` is defined for the material type and `Matrix` is the object statement of mathematical matrices. `BitCard` is defined to only have a byte length in computer memory and is used to record the type of boundary.

The source code of two key member functions is given in Figs. 4 and 5.

### 3.3. Problem solving

The procedure of BEM is very concise with above class definitions. At first, define an object with designed BEM class data type, then execute the specific action according to the alternative messages. A simple main programme in C++ code is shown in Fig. 6. The source code of classes is included in headfiles.

```cpp
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <math.h>
#include <complex.h>
#include "pointm.h"
#include "const.h"
#include "basic.h"
#include "bem.h"

//main
void main()
{
    BEM_Object thisProblem;
    thisProblem.InitialOut();
    thisProblem.Assembler();
    thisProblem.EquationSolve();
    thisProblem.Results();
}
```

Fig. 6. Source code of main programme.

### 4. Discussions and conclusions

Some elementary concepts of BEM approaches with OOP have been presented. Compared to the procedural oriented programming concerning modular procedures (subroutines) to data, BEM with OOP’s main job is how to design an object to simulate a real problem. The characteristics of OOP such as inheritance, encapsulation and polymorphism are very effective in writing general BEM software that deals with different mechanics problems. Firstly, we begin with data abstract of general BEM attributes and behaviour (functions) and design a basic class. Secondly, the classes of different problems can be derived from the basic class, which inherit parts or whole attributes and behaviour of the basic class. The additional characteristics of the derived class will be beneficial to the polymorphism of OOP. Lastly, an object of a specific problem is defined with messages of the problem, then let the object behave according to procedure given above, and the problem is solved. The code of BEM with OOP is very concise, easy to read, rewrite and maintain. Besides, OOP language itself can be used to write system software other than application software. So, to write BEM software that has an elegant computer interface and is user-friendly should not be too difficult.
References