Object-oriented C++ boundary element solution of the vector Laplace equation

J. A. Ingber
Accurate Solutions in Applied Physics, USA

Abstract

The Boundary Element Method (BEM) lends itself well to an object-oriented implementation. Well-defined class hierarchies can reduce the size of a problem solution while improving the readability and maintainability of the solution. The BEM uses geometric elements, defined as collections of nodes, to model a surface. Boundary conditions, specified by the problem, are defined at each node. This suggests an object oriented solution that defines a base Element class that can be extended to define triangular elements and quadrilateral elements, and a base Node class that can be extended to define more specialized nodes, such as edge and corner nodes. Historically, BEM codes have been written in FORTRAN 90 and object oriented codes have been deemed too slow for such computationally intensive solutions. In this paper I will discuss the development and optimization of an object-oriented BEM code, written in C++, for solving the vector Laplace equation for the magnetic vector potential in three dimensions. The solution to the 3-D magnetic field problem was first written and tested in FORTRAN 90. Due to the complexity and size of the problem solution, the translation to C++ went through several stages. At each stage the code was tested for accuracy and speed. After optimization of the C++ code, which included optimization of memory allocation, optimization of class structures, optimization of functions required to build the discretized linear system of equations and optimization of the solver, the C++ code executed faster than the FORTRAN 90 code for all test problems.

Keywords: boundary element method, object-oriented, C++, vector Laplace equation, magnetic vector potential, class hierarchies, node, element, off functional collocation nodes.

WIT Transactions on Modelling and Simulation, Vol 50, © 2010 WIT Press
www.witpress.com, ISSN 1743-355X (on-line)
doi:10.2495/BE100261

1 Introduction

The Boundary Element Method (BEM) lends itself well to an object-oriented implementation. Well-defined class hierarchies can reduce the size of a problem solution while improving readability, extensibility and maintainability of the solution. Object-oriented programming languages have advantages over procedural programming languages, such as FORTRAN 90. Procedural programming languages rely on the use of top level subroutines and functions that are passed data, or rely on global data modules. This results in procedures that are tightly coupled, meaning that changes to the data, or changes to a procedure will effect other procedures, and global data facilitates the propagation of errors. Object-oriented programming languages support the definition of classes and class hierarchies. A well designed class defines private or protected data members (properties), and public methods (operations) that perform computations and support the protected data. As long as the public interface is supported, changes to the data within one class will not effect other classes. Thus, the resulting solution is more robust and extensible, and classes are reusable. However, the overhead associated with instantiating classes and referencing public methods can result in solutions that execute more slowly than traditional FORTRAN codes.

Historically, BEM codes have been written in FORTRAN to solve a variety of problems including potential problems [9], vorticity formulations [3], and the magnetic field integral equation [4]. More recently, object-oriented implementations of the BEM has been discussed in the literature as a means for writing solutions that are easier to read and maintain. Many of these solutions have chosen to implement a Matrix class [7, 9]. Although this is a reasonable approach, and it can be argued that this is the ”pure” object-oriented approach, this approach can reduce the execution speed of the solution by one or two orders of magnitude [5]. Performance of object-oriented C++ BEM solutions is not widely discussed in the literature, but a well designed C++ solution can actually outperform FORTRAN solutions if care is take with the design of classes, and memory allocation is managed effectively. This is illustrated with the solution of the vector Laplace equation for the magnetic vector potential in three dimensions. The solution to this problem was originally written in FORTRAN. The code was ported to C++ to improve maintenance and extensibility. The C++ code was then optimized resulting in a solution that runs faster than the original FORTRAN solution.

2 Problem formulation

The governing vector Laplace equation can be transformed into a boundary integral equation using standard techniques [6, 1]. The magnetic vector potential can be represented by the following boundary integral equation

$$\eta(\vec{x})\mathbf{A}(\vec{\mathbf{x}}) = -\int_{\Gamma}[\vec{n}(\vec{y})\cdot\mathbf{A}(\vec{\mathbf{y}})]\nabla\mathbf{G}(\vec{\mathbf{x}},\vec{\mathbf{y}})\mathbf{d\Gamma} - \int_{\mathbf{\Gamma}}[\vec{\mathbf{n}}(\vec{\mathbf{y}})\times\mathbf{A}(\vec{\mathbf{y}})]\times$$

$$\nabla G(\vec{x},\vec{y})d\Gamma - \int_{\Gamma}[\vec{n}(\vec{y}) \times \mathbf{B}(\vec{\mathbf{y}})]\mathbf{G}(\vec{\mathbf{x}},\vec{\mathbf{y}})\mathbf{d\Gamma}, \tag{1}$$

where $\mathbf{A}$ is the magnetic vector potential, $\mathbf{B}$ is the magnetic flux density, $\vec{n}$ is the normal to the boundary Γ at the source point $\vec{y}$ and G is the Green's function given by $G = 1/ \mid \vec{x} - \vec{y} \mid$. The coefficient term η is a function of the local geometry at the field point $\vec{x}$, but can be determined using standard techniques such as assuming a constant vector potential and integrating over the surface of the domain.

In the current formulation, the boundary element discretization consists of biquadratic, isoparametric quadrilateral elements and quadratic, isoparametric triangular elements. After discretization, the linear system of boundary element equations can be written as

$$[G_{ij}]\{A_j\} = [H_{ij}]\{B_j\}, \tag{2}$$

where A_j and B_j represent the components of $\mathbf{A}$ and $\mathbf{B}$, respectively in Cartesian coordinates.

It has not been widely discussed in the literature, but the solution of the BIE is almost always ill-posed in Cartesian coordinates because of the boundary conditions. In particular, after imposing the boundary conditions, the resulting coefficient matrix will be singular in all cases except for the exterior Neumann problem in which $\mathbf{B}$ is specified everywhere on the boundary. To remove the singularity, the discretized BIE (Eq. 2) must be transformed to a local tangential-normal coordinate system, and at collocation nodes along the boundary where Dirichlet conditions are specified, the normal component equation must be discarded.

3 Structure of the FORTRAN solution

The BEM FORTRAN 90 code is comprised of 2 data modules, a main program, and 18 subroutines, including the ones listed below.

```
matvec(): assembles the discretized linear system of
            equations
rqint():  performs integral evaluations over
            quadrilateral elements
sqint():  performs integral evaluations over
            quadrilateral elements with singularities
rtint():  performs integral evaluations over
            triangular elements
stint():  performs integral evaluations over
            triangular elements with singularities
decomp(): performs an LU decomposition of the linear
            system
solve():  determines the solution of the linear system
```

Table 1: Time required to execute F90 solution.

Number of Equations	Assemble Matrix	Assemble and Solve
250	0.11 s	0.15 s
2040	3.05 s	18.91 s
3040	3.75 s	55.19 s

The LU decomposition of the linear system dominates the execution speed of the solution requiring $O(n^3)$ operation count, where n is the number of linear equations in the discretized linear system. This subroutine has been optimized for the FORTRAN programming language. The assembly of the linear system, performed in the $matvec()$ subroutine, requires $O(n^2)$ operation count. Table 1 lists execution times for the FORTAN code.

The assembly of the discretized linear system of equations requires nested loops. The outer loop ranges over the boundary element collocations nodes. If the boundary condition at the node is Dirichlet (A is specified), then two equations are generated in the tangential directions. If the boundary condition at the node is Neumann (B is specified), then three equations are generated in the Cartesian directions. The inner loop ranges over the boundary elements. Each element is composed of a set of local nodes and boundary conditions. A singularity occurs when the collocation node is the same as one of the local nodes within the element. Thus, selection is required within the inner loop to determine which integral evaluation subroutine to call. Because of the ambiguity in the normal direction at an edge or corner, the boundary element representation is double-noded along edges and multi-noded at corners. Thus, the $matvec()$ subroutine calculates the distance between the collocation node and each of the local nodes within each element to determine if a singularity exists. The object-oriented C++ solution for this problem is effective in reducing the size and complexity, and improving readability and extensibility, of the $matvec()$ subroutine.

4 Structure of the object-oriented C++ solution

The boundary element discretization consists of isoparametric quadrilateral and triangular elements, each defined by a set of nodes and boundary conditions. The C++ solution defines a Node class hierarchy and an Element class hierarchy, as well as an Assembly class, a quadrature information (QuadInfo) class and a class that solves the discretized linear system of equations (LUDecomp). The class diagrams are given in Figure 1.

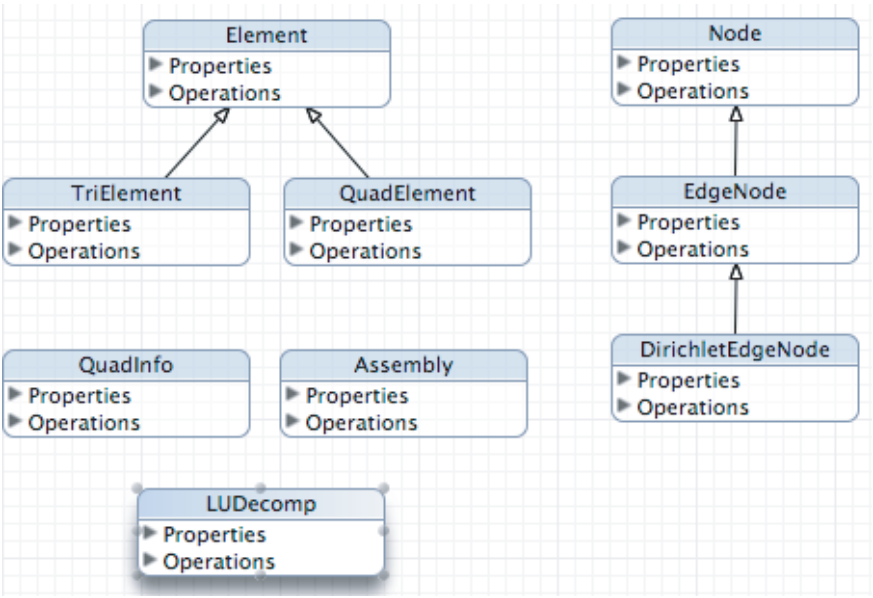

Figure 1: Class hierarchies.

An Element object has a set of nodes, and implements two integration functions. Thus, every Element object has immediate access to the data and functions required for correct mapping to the master element, and correct calculations of the integral evaluations. The Element class hierarchy simplifies the logic of the assembly routine through the use of virtual methods. The operations of the base Element class include two pure virtual methods that must be implemented by all subclasses as shown below:

```
virtual void regularIntegration(Node* collocationNode,
                                const QuadInfo& qi) = 0;
virtual void singularIntegration(Node* collocationNode,
                                 const QuadInfo& qi,
                                 int singularNode) = 0;
```

During assembly of the discretized linear system, an element calls the appropriate integration routine and dynamic binding determines the implementation (triangular or quadrilateral). Integration routines require access to Gauss points and Gauss weights for accurate computation. This information is constant, and specific to the boundary integral equation. Gauss points and Gauss weights are calculated in the QuadInfo class. A single instance of the QuadInfo class is instantiated in the Assembly class and passed by reference to the integration routines. The integration routines reference only the public interface of the QuadInfo class, thus the QuadInfo class is not tightly coupled with the Element classes. A Node object has a type, a set of boundary conditions, and data members

for the accumulation of integral evaluations. The properties of the base Node class are as shown below:

```
char nodeType;
double x,y,z; //geometric/functional node values
double cX, cY,cZ;  //collocation node values
double integrationResultA[3][3],
  integrationResultB[3][3];
double uVect[3], vVect[3], nVect[3];
double boundaryConditions[3];
int rcIndex;  //position in DLS
int globalNodeNumber;
```

The operations of the base Node class include necessary accessor and mutator methods to support the class properties as well as the following method:

```
int isSingular(Element e) const;
```

The $isSingular()$ routine returns the node number of the singularity, if a singularity exists for the given element, and returns zero if no singularity exits in the given element.

The EdgeNode class has one additional property as shown below:

```
std::map<int> multiNodeNumbers;
```

The EdgeNode class overrides the $isSingular()$ method, and references $multiNodeNumbers$ to determine if a singularity exits in the given element. This greatly simplifies the logic and time required to determine if a singularity exists, as distances do not need to be calculated during execution of the solution.

When edge and corner nodes have Dirichlet boundary conditions for at least 2 duplicate nodes, off functional node collocation is required to avoid singularities in the discretized linear system of equations [4]. The DirichletEdgeNode class has one additional property:

```
char side;  //The edge side of element
```

The singular integration routines can reference the $side$ property of a DirichletEdgeNode to modify the collocation nodes within the element to maintain a well-posed problem. Note, Dirichlet edge node detection is not implemented in the F90 solution. However, with the use of objects, the extension of the Node class to handle this special case is rather straight forward.

The Assembly class is responsible for the input of the problem data and construction of the global node and element arrays, calculation of the normal and tangential vectors to the boundary at each boundary element node, and assembly of the discretized linear system of equations. The main properties and operations of the Assembly class are shown below:

```
//Dynamic Global Node Array
//Each element in the Node* array holds the address
```

```
of a node
Node** globalNodes;

//Dynamic array
//Each element in Element* array holds address
of an element
Element** elements;

//Define the Quadrature data
QuadInfo qi;

//Define the geometry
void inputGeom();

//Calculate vectors
void calcVec();

//Build discretized linear system
void assembleMatVec(double[] bVector,
                    double[] aMatrix);
```

Arrays of pointers are used for memory efficiency and to support dynamic binding through virtual methods. Dynamically allocated arrays are used for speed and efficient use of memory. The coefficient matrix for the discretized linear system is defined as follows:

```
//allocate and initialize aMatrix
double *aMatrix =
          (double*)calloc(numEquations*numEquations,
           sizeof(double));
```

The C routine, $calloc()$, has an advantage over the C++ operator new in that $calloc()$ initializes the allocated memory space to zero at a time that is most efficient. When referencing the $aMatrix$, simple arithmetic is used for correct mapping into a two-dimensional array, or matrix, as illustrated in the following assignment statement:

```
aMat[row*neq + col] += tempVecs[0][0];
```

The LUDecomp class implements Crout's algorithm, optimized for C++. This is an outstanding algorithm and the LU decomposition routine requires about $1/3N^3$ executions [8]. Table 2 lists execution times for the optimized C++ code.

Table 2: Time required to execute C++ solution.

Number of Equations	Assemble Matrix	Assemble and Solve
250	0.09 s	0.14 s
2040	2.95 s	18.07 s
3040	5.02 s	54.26 s

5 Discussion

The initial motivation for porting the FORTRAN 90 solution to C++ was to improve the readability, extensibility and reusability of the solution for commercial purposes. Improving the execution speed was not anticipated in the beginning, although every effort was made to make the code as efficient as possible. The C++ solution went through several iterations. The first iteration of the C++ code was nearly two orders of magnitude slower than the FORTRAN 90 code. The poor performance, was the motivation to begin optimization of the C++ code. Many modifications were made, but a few of the changes were quite significant.

Initially, the data type $long\ double$ was used instead of $double$, and the Gauss points and Gauss weights were properties of the Element class since this data is required to perform the integral evaluations. Defining the Gauss points and Gauss weights in the base Element class slowed performance due to the fact that these values were generated every time an element was instantiated. (Unlike Java, C++ does not support initialization of static data members within a constructor.) Creating a quadrature class and instantiating a single object that was passed, by reference, to the integration routine resulted in a small yet significant increase in performance. Changing the data type from $long\ double$ to $double$ reduced the execution time by close to eighty percent, and the accuracy of the results did not change within the seven significant digits that were printed. Implementing Crout's algorithm for solving the linear system of equations reduced the execution speed by nearly fifty percent. Finally, eliminating the need to calculate distances between collocation nodes and local nodes to check for singularities, as discussed in section 4, improved performance of the solution.

Acknowledgements

This work was funded by the Air Force Research Laboratory under Small Business Innovation Research (SBIR) contract number FA945-08-M-0084. The starting point for this work was initiated by a graduate student, Paula Higgins, at the University of New Mexico for a Master's Thesis [2].

References

[1] Z. Fang and M.S. Ingber. The Solution of Magnetostatic BEM System of Equations Using Iterative Methods. *Engineering Analysis with Baoundary Elements*, 26:789–794, 2002.
[2] P. Higgins. Boundary Element Method Solution of Laplace's Equation for the Magnetic Vector Potenial. Master's thesis, University of New Mexico, 2003.
[3] M. S. Ingber and S. N. Kempka. A Galerkin implementation of the generalized Helmholtz decomposition for vorticity formulations. *J. Comp. Phys.*, 169:215–237, 2001.
[4] M.S. Ingber and R.H. Ott. An Application of the Boundary Element Method to the Magnetic Field Integral Equation. *IEEE Transactions on Antennas and Propagation*, 39:606–611, 1991.
[5] I. A. Jones, P. Wang, A.A. Becker, D. Chen, and T.H. Hyde. Efficient object-oriented implementation of boundary element software. In *Proceeding of the Eighth International Conference on the Application of Artificial Intelligence to Civil and Structural Engineering Computing*, Stirling, Scotland, 2001.
[6] L. Li. Boundary Element Method for Three-Dimensional Magnetostatic Fields in Terms of Vector Variables. *Acta Polytech. Scan. Elec. Engr. Ser.*, 61:1–58, 1998.
[7] R.J. Marczak. An object-oriented framework for boundary integral equation methods. *Computers and Structures*, 82:1237–1257, 2004.
[8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes The Art of Scientific Computing*. Cambridge University Press, 32 Avenue of the Americas, NY,NY 10013-273, USA, 2007.
[9] H. Qiao. Object-oriented programming for the boaundary element method in two-dimensional heat transfer analysis. *Advances in Engineering Software*, 37:789–259, 2005.