Stony Brook University

Elliptic Interface Problem Solved Using The Mixed Finite Element Method

A Dissertation Presented

by

Shuqiang Wang

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Applied Mathematics and Statistics

Stony Brook University

August 2007

Stony Brook University

The Graduate School

Shuqiang Wang

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

James Glimm - Dissertation Advisor
Professor
Department of Applied Mathematics and Statistics

Xiaolin Li - Chairperson of Defence
Professor
Department of Applied Mathematics and Statistics

Yan Yu - Member
Doctor
Department of Applied Mathematics and Statistics

Roman Samulyak - Outside Member
Scientist
Brookhaven National Laboratory
Computational Center

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

Elliptic Interface Problem Solved Using The Mixed Finite Element Method

by

Shuqiang Wang

Doctor of Philosophy

in

Applied Mathematics and Statistics

Stony Brook University

2007

The elliptic boundary value/interface problem is very important in many applications, for example, in incompressible flow and MHD. Many methods are used to solve these problems in a complex domain, including the finite volume method, the finite element method and the boundary element method.

For a complex computational domain, the better choice of the partition of the computational domain is to use an unstructured grid. However, it is not a straight forward task to implement a mesh generation program. Such a program requires extra computing time and resources (such as computer memory). Thus people like to use a structured mesh if possible, especially a cartesian mesh.

Popular methods using structured cartesian grids for the elliptic boundary value/interface problem include the immersed boundary method, the immersed interface method, the ghost fluid method, and the embedded boundary method.

This thesis solves the elliptic problem using several versions of the mixed finite element method on an unstructured mesh. The results are compared for speed and accuracy to the embedded boundary method.

A ghost fluid method for elliptic boundary value/interface problems is also investigated.

Finally, a simple test of the 2D Rayleigh-Taylor instability is performed using the FronTier-Lite package.

Key Words: Elliptic Boundary Value, Interface, Mesh Generation, Quadtree, Octree, Front Tracking.

To parents and my wife

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisor Professor James Glimm who has provided so much guidance and support to my research. Whenever I have questions, he will always be there to answer them.

I would also like to thank Professor Xiaolin Li, Roman Samulyak, W. Brent Lindquist, Wonho Oh and Fred Furtado. They were all very helpful when I had questions. Thanks also to Dr. Yan Yu for accepting to be in my dissertation committee in a rush. She, Xingfeng Liu and Tianshi Lu helped me to find housing during the beginning of my first semester at Stony Brook.

I would like to thank Yoon-ha Lee for the helpful discussions on the mesh generation problem. I would also thank Prof Xiaolin Li and Jinjie Liu for introducing to me the marching cubes method. It is after that discussion that I made up my mind to use this method to recover the interface for the mesh. I would also thank Jian Du and Dr. Roman Samulyak for the comparison of the mixed finite element methods and the embedded boundary method for solving the elliptic boundary problem. I would also like to thank my other friends for their helps and friendship.

At last, I would like to thank my wife, my parents, sisters and brother. With their encouragement and love, life becomes more wonderful.

Chapter 1

Introduction

This thesis discusses a method for solving elliptic interface/boundary value problem. We use an unstructured mesh based on the quadtree/octree approach to decompose the computational domain into elements and then use the mixed finite element method for solving the elliptic boundary value/interface problem.

To main contribution of this thesis is to simplify the quadtree/octree mesh generation method using the marching cubes method to recover the interface/boundary. Then we combine the quadtree/octree mesh generation with a simple implementation of the mixed-hybrid finite element method in solving the elliptic problem. By combining the two methods, we can easily treat either the elliptic boundary value or the elliptic interface problem without much change of the code. By using higher order basis functions, we can have more than 2nd order accurate method with ease. The original quadtree/octree mesh generation method by Yerry and Shephard [1, 2] assumes the interface/boundary can cross the quadrant/octant edges a finite number of times and each of the corners once. Then they use templates to triangulate/tetrahedralize the full/partial quadrants/octants. Because of their assumption, they need to generate many templates. We simplify the assumption by using the marching cubes method: there is at most one crossing on each edge and no crossing at the corners. Thus few templates are needed. Our whole mesh generation method is very similar to the marching cubes mesh generation method for imaging data or volumetric data [3–7]. The main difference is that we have geometric input instead of the imaging/volumetric data and we can move the crossing points onto the interface/boundary in the post

processing step.

The second contribution of this thesis is a timing and convergence comparison of the embedded boundary method with our mixed finite element method for solving the elliptic value problem. The result of our study will give some guidance to future research directions for the elliptic boundary value/interface problem.

The third contribution is that we extend the ghost fluids method to solve the elliptic interface problem using front tracking. Finally, we use the FronTier-Lite package to carry out a simple Rayleigh-Taylor instability simulation.

The thesis is organized as follows. In Chapter 2, we briefly review the methods for mesh generation, and then give our method which uses the marching cubes method for recovering the interface/boundary. In Chapter 3, we discuss our implementation of the mixed finite element method which assembles the matrix element by element. In Chapter 4, we perform a comparison study, comparing the foregoing with the embedded boundary method for the elliptic boundary value problem. In Chapter 5, we extend the ghost fluid method and investigate its convergence. In Chapter 6, we use the approach by Tryggvason et al. [8, 9] to carry out a simple Rayleigh-Taylor instability test using the FronTier-Lite [10] package.

In the appendices, we present several materials which are not closely related with the main content of the thesis. We show the coding issues for our computer program in Appendix A. A novel C++ interface is designed for calling functions in several different algebraic packages, including Petsc

[11], Hypre [12] and LAPACK [13]. We give a simple compact finite difference method for the calculation of the curvature of a 2D curve in Appendix B and we present a nonreflecting boundary method for an elliptic problem in Appendix C. Last, we discuss the similarity between the mixed finite element method and the cell boundary element method in Appendix D.

Chapter 2

Mesh Generation Method

2.1 Introduction

The first step for a numerical simulation is to construct a mesh for the computational domain. For simple domains, structured grids are often used, while for complex domains, unstructured or hybrid grids may be advantageous. There are mainly three methods for generating unstructured grids, namely, the Delaunay/Voronoi triangulation [14], the advancing front method [15], and the quadtree/octree-based method. Thompson [16], Owen [17] could be referenced for the most recent developments for mesh generation methods.

This thesis discusses a modified approach to the quadtree-based method discovered by Yerry and Shephard [1, 2] by using the marching cubes method to recover the interface. Notably, the FronTier code has already used the marching cubes method for grid based interface reconstruction long time ago [10].

The original quadtree/octree mesh generation method by Yerry and Shephard [1, 2] consists of the following steps:

1. Partition the computational region into a quadtree with the level difference between neighbor quadrants being no more than one. Now all those quadrants are either full quadrants or boundary quadrants.

2. Triangulate the full interior quadrants.

3. Triangulate the partial quadrants to recover the interface.

4. Post processing the mesh and move those interface crossing points onto the interface in the post processing step.

They assumes the interface/boundary can cross the quadrant/octant edges a finite number of times and the corners. Then they use templates to triangulate/tetrahedralize the full/partial cells. Because of their assumption, they need to generate many templates. We simplify the assumption using the marching cubes method: there is at most one crossing on each edge and no crossing at the corners. Thus few templates are needed. In the end, our whole mesh generation method is very similar with the method for imaging data or volumetric data [3–7]. The main difference is that we have geometric input instead of the imaging/volumetric data and we can move the crossing points onto the interface/boundary in the post processing step.

Our input, a FronTier interface [18–20], is a boundary-representation of a boundary curve which consists of vertices and segments, or a boundary surface which consists of triangles, rectangles. This kind of representation is the most frequently used model because of the popularity of the B-Rep model in CAD and its accuracy in representation. A model could alternatively be represented by a level set of implicit functions whose different values represent different components. For example, the level set method uses a distance function $\phi(x)$ to represent different materials: $\phi(x) > 0$ represents one material and $\phi(x) < 0$ represents another material. The imaging/volumetric data use the level set approach. It is worth noting that a B-Rep model could be easily augmented so that it also provides such a component function. FronTier, a code developed by Glimm et al. [18], uses the B-Rep model consisting of segments and triangles to represent the boundary. It uses integer values to represent different components. It has the ability to answer the component query: what is the

component at a particular point? Our algorithm for mesh generation will use such a component function, which is provided by the B-Rep model or the level set model.

We intend to use our mesh generation method for multiphase flow using front tracking, where the interface is explicitly represented as curves or surfaces or an implicit function. When the interface changes, we need to regenerate our mesh. Thus the mesh generation should be very fast. By using the quadtree method, we only need to regenerate the mesh for the quadrants near the interface. Thus the mesh generation method could indeed be very fast.

The quadtree/octree method and the patch based automatic mesh refinement (AMR) use a similar idea to partition the computational domain. The difference is that the AMR patches might consist of an arbitrary number of quadrants while each parent quadrant in the quadtree (octree) has exactly 4 (8) children. Thus the quadtree/octree construction is more suited to the efficient representation of complex boundary/interface.

Previous work for generating a unstructured mesh using the frontier interface includes the point shifting grid [21–23] and hybrid mesh generation [24]. The point shifted grid [21–23] has assumption that the interface/boundary only crosses the grid point of a structured cartesian grid. The hybrid method generation method [24] uses the quadtree as the underling grid and uses the Delaunay triangulation method to recover the interface.

In the following, we present the implementation of our mesh generation algorithms. These algorithms are not new. They appear either in articles/textbooks about quadtree/octree generation method or the marching

cubes method. We first present our implementation of the quadtree generation/balance algorithm. Then we show the method for recovering the interface by using marching cubes method, the generalized marching cubes method and marching cubes on the dual grid of the quadtree. Then we show the method for the octree generation/balance/interface recovering algorithm. Finally, we show the point location algorithms we used and we discuss our conclusions.

2.2 Method

2.2.1 Quadtree Based Mesh Generation Method in 2D

Quadtree Generation

The quadtree is a tree with a spatial data structure. The classic textbook for computational geometry [25] gives a good introduction to the quadtree structure. A thorough explanation of many spatial data structures can be found in [26].

A quadtree is a tree data structure in which each leaf node (called a quadrant) has exactly 4 children and each node represents a rectangle domain whose down-left and up-right corners have coordinates m_x1[2] and m_x2[2]. If a node does not contain children, it is called a leaf. If a node is not a leaf quadrant, then it is partitioned into exactly four children: the north east (NE), the north west (NW), the south west (SW) and the south east (SE). Each quadrant has a level number, m_level, to denote its depth in the tree structure. The level of a child quadrant is 1 larger than the parent quadrant. Figure 2.1

Table 2.1: A simplified quadtree structure

```
class QUADTREE {
        static int m_min_level;
        static int m_max_level;
        int m_level;
        double m_x1[2], m_x2[2];
        QUADTREE *Parent;
        QUADTREE *NE; // x > xmid & y > ymid
        QUADTREE *NW; // x ≤ xmid & y > ymid
        QUADTREE *SW; // x ≤ xmid & y ≤ ymid
        QUADTREE *SE; // x > xmid & y ≤ ymid

        QUADTREE* NorthNeighbor(void);
        QUADTREE* WestNeighbor(void);
        QUADTREE* SouthNeighbor(void);
        QUADTREE* EastNeighbor(void);

        void balance(void);
        void create(void);
};
```

gives a simple quadtree as example. The code in Table 2.1 defines a simplified quadtree class using the C++ language. Table 2.2 shows a recursively defined algorithm for creating a quadtree using pseudocode.

Table 2.2: Algorithm for creating a quadtree

```
Algorithm QUADTREE::create:
    if(this quadrant needs not to be divided)
        return;
    divide the current quadrant into four children
    and update member variables.
    call NE → create();
    call NW → create();
    call SW → create();
    call SE → create();
    return;
```

The criteria for deciding for whether the current quadrant should be subdivided must be supplied by the user. For our case, the quadtree is defined based on a component function which defines the boundary of the computational domain or interface. For a given point, this function gives the component number based on the interface. In fact, it is similar to a level set function which gives a contour value for a given point. If we want the boundary quadrants to be at a uniform level m_max_level and the minimum level of the quadtree be m_min_level, the checking function could be defined as in Table 2.3.

The quadtree structure could be queried to give its four neighbors. A neighbor of the quadrant v is a quadrant v' adjacent to it in the given direction. Generally a quadrant has four neighbors. In order to make a good graded mesh from the quadtree structure, it is necessary that the quadtree is balanced so

Table 2.3: Algorithm defining criteria whether to subdivide the current quadrant

```
Algorithm QUADTREE::isToDivide:
    let m_level be the level of this quadrant;
    if( m_level < m_min_level)
        return true;
    if( m_level < m_max_level )
        if( the components of the four corners of this quadrant is different )
                return true;
    return false;
```

that the level difference of any two neighboring quadrants is at most 1. In this way, the size difference of two neighboring quadrants would not be too large. For the neighbor finding and quadtree balancing algorithm in detail, refer to [25]. For completeness, we give the algorithms for finding a north neighbor in Table 2.4 and the algorithm for balancing the quadtree in Table 2.5.

Sometimes, it is necessary to update a quadtree with a changing input. Given a new interface, we could create a new quadtree. However, when the new interface is similar with the older interface with only small changes (for example, the new interface is moved for a small distance from the older interface), then the quadtree for the new interface could be created based upon the older quadtree. This approach has the potential of saving some computing time, especially when the depth of the quadtree is large. Some computational tests showed this. Table 2.7 shows the algorithm to create and balance the new quadtree for a new interface based on the old quadtree.

By using the quadtree, we partition the domain into smaller parts. A

Table 2.4: Algorithm for finding north neighbor of the current quadrant

```
Algorithm QUADTREE::NorthNeighbor:
    let this be the current quadrant;
    if(Parent==NULL)
        return NULL;
    if(this==Parent → SW)
        return Parent → NW;
    if(this==Parent → SE)
        return Parent → NE;
    μ = Parent → NorthNeighbor();
    if(μ==NULL or μ is a leaf)
        return μ;
    else
        if(this==Parent → NW)
            return μ → SW;
        else
            return μ → SE;
```

Table 2.5: Algorithm for quadtree balancing

```
Algorithm QUADTREE::balance:
    let this quadtree be the root of the quadtree;
    insert all leaf quadrant into a list L;
    while(L is not empty)
        remove an item μ from the list L;
        if(μ is a leaf and μ →NeedToBeSplit())
            subdivide μ;
            insert the four new quadrants into the list L;
            check if the four neighbors of μ need to be subdivided;
            if so, insert it/them into the list L;
    return the list L;
```

Table 2.6: Algorithm for QUADTREE::NeedToBeSplit

Algorithm QUADTREE::NeedToBeSplit:

```
let μ be the north neighbor of the current quadrant;
if(μ is not NULL and not leaf)
    if(μ → SW is not leaf or μ → SE is not leaf)
        return true;
let μ be the west neighbor of the current quadrant;
if(μ is not NULL and not leaf)
    if(μ → SE is not leaf or μ → NE is not leaf)
        return true;
let μ be the south neighbor of the current quadrant;
if(μ is not NULL and not leaf)
    if(μ → NW is not leaf or μ → NE is not leaf)
        return true;
let μ be the east neighbor of the current quadrant;
if(μ is not NULL and not leaf)
    if(μ → NW is not leaf or μ → SW is not leaf)
        return true;
```

Table 2.7: Algorithm for creating and balancing the new quadtree for a new interface based on the old quadtree; the returned list L_{new} contains the leaves of the quadtree for the new interface which is defined through the function isToDivide()

Algorithm QUADTREE::AMR_balance:
    let L_{old} be the old quadtree leaves list for the old interface;
    let L_{new} be the new quadtree leaves list which is empty at start;
    while(L is not empty)
        remove an item μ from list L_{old};
        if(μ is not a leaf)
            continue;
        else if($\mu \rightarrow Parent$ is leaf)
            delete mu and recover memory;
        else if($\mu \rightarrow Parent$ is not NeedToBeSplit() and not isToDivide())
            let $\mu = \mu \rightarrow Parent$;
            make μ be a leaf;
            insert μ into L_{old};
        else if(μ is NeedToBeSplit() or isToDivide())
            subdivide μ;
            insert the four new quadrants into the list L_{old};
            check if the four neighbors of μ need to be subdivided;
            if so, insert it/them into the list L_{old};
        else
            insert the leaf mu into the new list L_{new};
    return L_{new};

full quadrant is a quadrant that does not intersect the boundary, so that the four vertices of the quadrant have the same component, while a boundary quadrant lies on the boundary or equivalently it has different components at its four vertices.

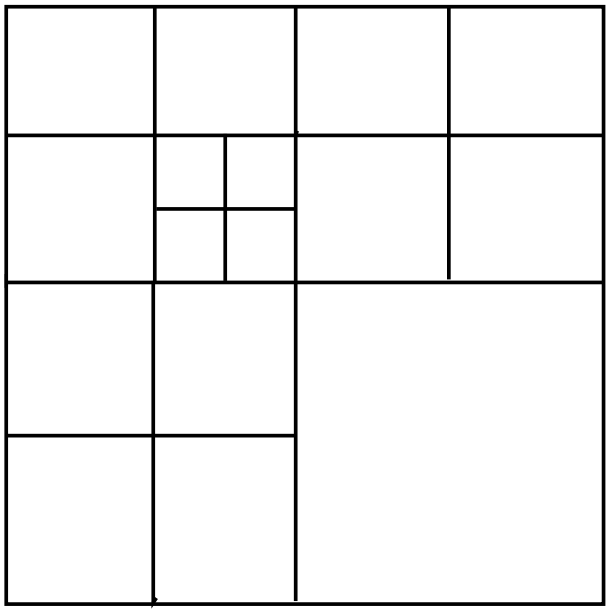

Figure 2.1: a quadtree

Quadtree Parallelization

The parallelization of the quadtree generation is easy in that the four boundaries of the quadtree are still trees. The general steps are the following:

1. Generate and balance the quadtree on each nodes;
2. Communicate the boundary information between nodes;
3. Balance the quadtree again on each nodes using the neighboring boundary information of the neighbor nodes;
4. Redo the communication step and balance step until convergence;

If we look at one of the four the boundaries of a quadtree, we would find that it is also a tree structure which is called btree here for binary tree (Note that there is a popular tree structure called B-tree [27] which means a balanced tree structure). The btree is a tree data structure with each non-leaf node having exactly two children. It is not necessarily balanced. Table 2.8 shows a simple btree data structure.

Table 2.8: A simple btree data structure

```
class BTREE {
        int m_level;          BTREE *Parent;
        BTREE *Left;
        BTREE *Right;
};
```

Before the communication of the boundary information of the quadtree across different nodes, we first encode the four boundaries of the sending quadtree, for example A, into a string of integers which represents the btree. And then we pass the string onto the neighbor nodes, for example B. Then in B, we use the string to setup a btree structure. Then when we do the quadtree balance on B, the btree will be queried for the information on the neighboring nodes.

It is well known that many tree structures can be encoded into a string of integers. The algorithm in Table 2.9 first encodes the north boundary into a list of $0's$ and $1's$. To save spaces, the list is then be packed into an integer array. After transmitting to the neighbor node, the integer array is unpacked into a list of $0, 1$ to be used to create the btree structure. When querying

Table 2.9: Algorithm for encoding the north boundary into a list of 0 and 1 using depth-first search of the north boundary of the quadtree

```
Algorithm QUADTREE::BoundaryEncoding_North:
    input: an empty list L;
    output: list L consisting of only 0, 1;
    if(Parent==NULL)
        empty L;
    if( this quadrant is a leaf quadrant)
        push back 0 into L;
    else
        push back 1 into L;
        call NW→ BoundaryEncoding_North(L);
        call NE→ BoundaryEncoding_North(L);
    return L;
```

for neighbor information beyond the boundary of the quadtree on the current node, we first go up the quadtree structure to the root, saving the path track and then use the path track to go down the btree to give the information needed. Since there are four boundaries in a quadtree, four btree structures are needed.

Quadtree-based mesh generation with the marching cubes method

A quadtree-based mesh generation method is very simple. This method was first proposed by Yerry and Shephard [1]. It consists of the following steps:

1. Partition the computational region into a quadtree with the level difference between neighbor quadrants being no more than one. Now all those quadrants are either full quadrants or boundary quadrants.

2. Triangulate the full interior quadrants.

3. Triangulate the partial quadrants to recover the interface.

4. Post processing the mesh. If we used templates to triangulate the partial quadrants and recover the interface in step 3), we need to move those crossing points onto the interface in the post processing step.

The older version of Yerry and Shephard's quadtree method assumes that the interface intersects the quadrants with finite number of crossings (namely at the corner and on the edge of the quadrants). In such a case, simple templates could be used both for the interior cells and boundary cells (later they proposed using Delaunay or advancing front method to triangulate those boundary cells). This thesis assumes that the interface crosses each edge of the quadrant at most once. Thus our method is a modification of the original version of Yerry and Shephard's quadtree method. By assuming that there is only one crossing on each edge, we exclude a number of problems that our method can deal with. But it is still robust enough to solve many problems. The reason that we assume there is only one crossing on each edge is that we

in fact are using the marching cubes method [28] to recover the interface. It should be noted that for a given boundary represented by segments for 2D and triangles for 3D surfaces, our method will generate a mesh which might not be a boundary conforming mesh. The interface recovered from our mesh is an approximation of the boundary interface. But we think this might be enough to represent the fluid interface. It is well known that the current state-of-art methods for solving multiphase fluid using interface tracking/capturing are the volume of fluid (VOF) method, level set method and the front tracking method. To represent the fluid interface, VOF uses volume fraction, the level set method uses the level set of an implicit function and the front tracking method uses segments and triangles. However they all employ some simplification of the interface.

The original quadtree method uses simple templates to triangulate the boundary quadrants. They assume that the interface could cross the edge and also the corner. By assuming that the interface could cross the corner, they avoid the cases where points lie too near the interface, thus generating triangles with a bad aspect ratio. But by doing this, they also introduce complications. They need far more templates than if they assume that the interface only crosses each edge once. By assuming that there is only one crossing along each edge, the two corners at the end of the edge have two different materials, or components. For example, see Figure 2.2.

All other cases could be reduced into these four cases by rotation. Thus if we need to triangulate the quadrant with different components, we only have four templates to generate, which are given in Figure 2.3.

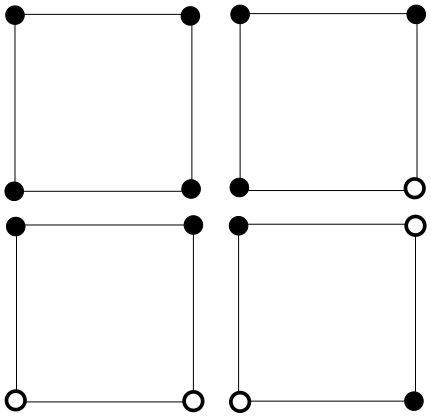

Figure 2.2: Marching Cube Cases in 2D. Black and white circle represents different components. Four different cases in total. All other cases could be transformed into this four.

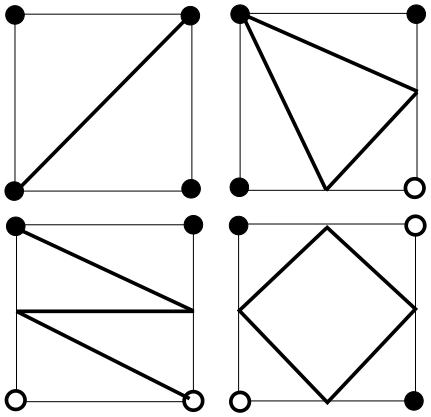

Figure 2.3: Triangulation templates for the four cases from Marching Cubes method.

It is apparent that the above representation is the same as cases from the marching cubes method [28] with two components in 2D. The marching cubes method is very popular in computer graphics and medical imaging. It is used to reconstruct the interface from volume data or the interface between two materials. The interface recovering step happens to be the same as the third step of the quadtree method for mesh generation. Thus we could use the marching cubes method to recover our interface from those partial boundary quadrants. There are many papers from this field. They are mostly used to recover 3D interfaces, or surface for mesh generation. It is indeed helpful if we could use them for volume mesh generation.

It is worth noting that there are several methods for the solid modelling which are closely connected with grid generation: the constructive solid geometry (CSG), the boundary representation (B-rep), the volume representation and the implicit surfaces. The Front tracking method [18] uses the B-rep and the level set method [29, 30] uses the implicit surfaces method. The marching cubes method has also been used for the implicit surfaces model.

By using the marching cubes method only, we could recover the interface. But we could not get a graded mesh where there are more triangles used in those important places, such as near the complex boundary or where the important physical quantities have a large gradient. We may combine the marching cubes with the quadtree construction to generate a graded mesh. Generally we constrain the maximum level difference between two neighbor quadrants to be at most 1 so that the size of the quadrants change gradually. This also facilitates the generation of the templates for the triangulations of

the quadrants since there are only a small number of cases for the quadrants with neighbors of different size.

When we only allow the max level quadrants to be partial quadrants, we have the cases in Figure 2.4 for those full quadrants. It is easy to generate the triangulations for those full quadrants using templates. And it should be noted that the templates for these cases are not unique. For example, Figure 2.5 shows a set of templates for the cases in Figure 2.4. For these templates, no new vertices are generated inside the quadrant. Another set of templates in Figure 2.6 generate more triangles for the same cases compared with the first set of templates. However these templates are easier to extend to 3D cases. The method for generating the second set of templates is as follows. If the neighbors of the quadrant are larger, then no new vertex is generated inside the quadrant. We connect the two opposite corners to generate two triangles. Otherwise, a new vertex at the center is generated and we connect the center vertex to the four corners and the centers of those edges with a smaller neighbor.

Figure 2.7 is an example of a domain separated by a circle and Figure 2.8 is the interface recovered in the mesh by the marching cubes method. Figure 2.9 is the smoothed mesh using the Laplacian smoothing procedure.

Quadtree-based mesh generation with the generalized marching cubes method

If the partial cells must lie in quadrants with the maximum level, it means that the interface must be approximated by the smallest size quadrants. If the

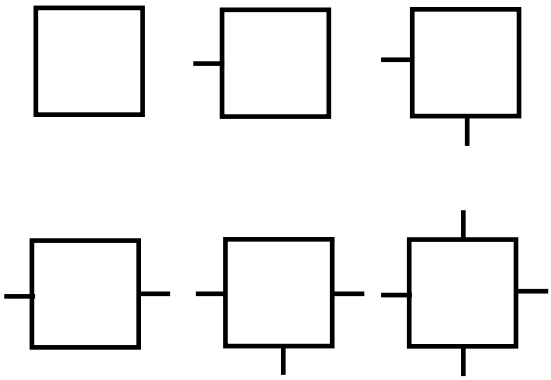

Figure 2.4: The full quadrants cases. No interface crosses these quadrants by assumption. The boxes in the figs represent the quadrants and the bar outside the edges of the boxes represents that the quadrant has a neighbor with 1 level higher (or half its size).

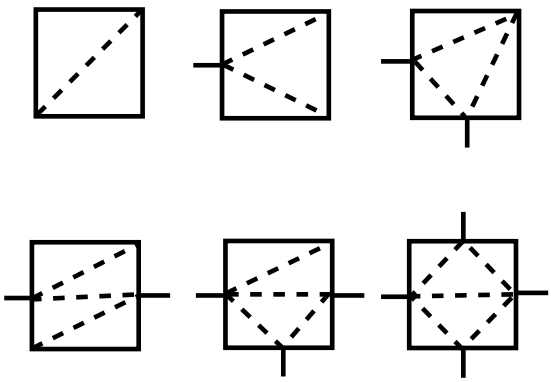

Figure 2.5: The first set of templates for the quadrants with different kind of neighbors. the bar outside the quad represents that it has a neighbor with 1 level higher.

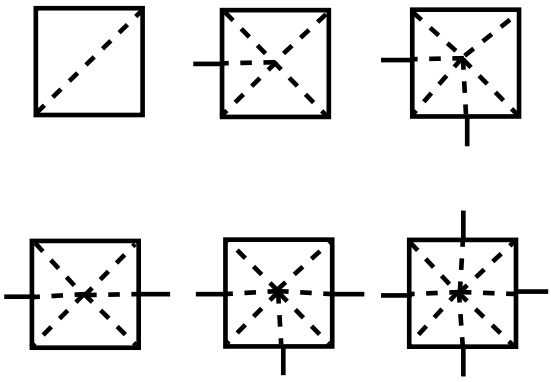

Figure 2.6: The first set of templates for the quadrants with different kind of neighbors. the bar outside the quadrands represents that it has a neighbor with 1 level higher.

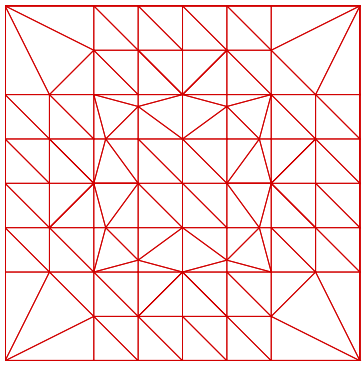

Figure 2.7: A mesh for a domain separated by a circle

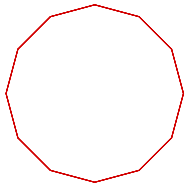

Figure 2.8: The interface recovered in the mesh of Figure 2.7.

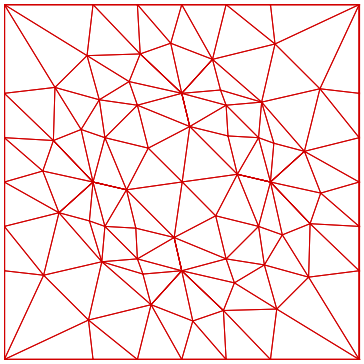

Figure 2.9: The smoothed mesh for the mesh of Figure 2.7.

interface changes dramatically everywhere, this is satisfactory. However, if the interface has a large region with small curvature, then by using the highest level quadrants (or the smallest quadrants) to approximate the interface whenever they occur would lead to unnecessary computational expenses. Thus it is preferable that the interface could be approximated automatically by using small quadrants where the interface has a large curvature and bigger quadrants where the interface changes slowly. But how do we use the marching cubes method to recover the interface in this case?

We could avoid the restriction by first triangulating all quadrants into triangles and then using the marching cube method on those triangles by assuming at most one crossing on one edge. This kind of marching cubes method is called the generalized marching cubes method [31]. In the mesh generation algorithm using marching cubes method introduced in the last section, we deal with full interior quadrants and partial quadrants differently. While for the mesh algorithm using generalized marching cubes method, we triangulate all quadrants *without* considering the whereabouts of the interface at first. After this step we *only* have triangle elements. Then we use generalized marching cubes on these triangles to recover the interface. Thus the algorithm is as follows:

1. Partition the computational region into quadtree with the level difference between neighbor quadrants at most 1. Now all those quadrants are either full quadrants or boundary quadrants.

2. Triangulate all quadrants.

3. Use generalized marching cubes method on triangle elements to recover the interface.

4. Post processing the mesh. If we used templates to triangulate the partial quadrants and recover the interface in step 3, we need to move those interface points onto the interface in the post processing step.

It is easy to triangulate the quadtree. We could use the templates in Figure 2.5 or 2.6. The method to generate the second group of templates could be extended to Octree. Previously we use the marching cubes method on rectangles. It is much easier to use it on triangles. There are only two cases to consider as in Figure 2.10. The triangulation templates are shown in Figure 2.11.

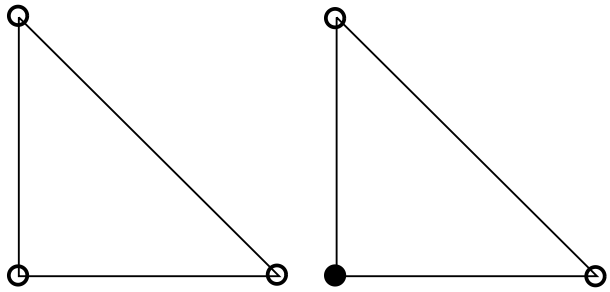

Figure 2.10: The two cases for the generalized marching cubes on a triangle element

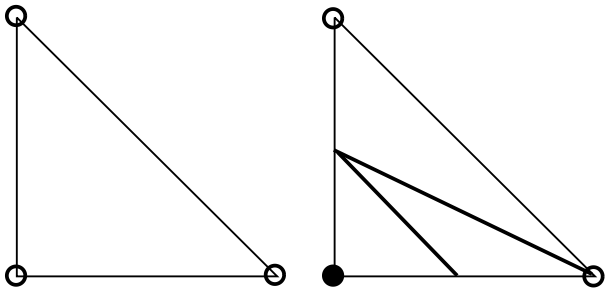

Figure 2.11: The triangulations of the two cases for the generalized marching cubes on a triangle element

Quadtree-based mesh generation with the dual marching cubes method

We could also use the marching cubes method on the dual grid of the quadtree. By using the dual marching cube method, we show that there is a great potential in combining the marching cubes method with the quadtree/octree methods for graded mesh generation. We will also use the dual marching cubes method for 3D mesh generation.

A dual grid is a grid such that we replace every cell with a vertex and every vertex with a cell. For example the Delaunay diagram of a point set is the dual graph of the Voronoi diagram of the set [25]. For the quadtree of Figure 2.1, the dual graph is given by Figure 2.12. It is apparent that the dual graph of the quadtree has the following property:

1) The cell is either triangle or rectangle.

2) The cell and its neighbor share the same edge.

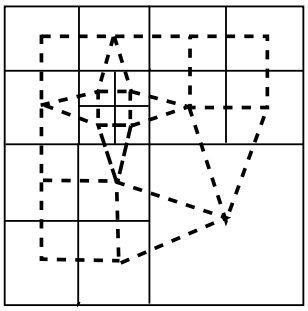

Figure 2.12: The dual graph of the quadtree in Figure 2.1

Now we may triangulate easily the full cells of the dual grid (without interface crossing), since they are either triangles which we do not need to triangulate or rectangles which we triangulate by connecting one of the two main diagonals and thus generate two triangles. For the partial cell (with interface crossing), we use the templates for rectangle in Figure 2.3 and templates for triangle in Figure 2.11.

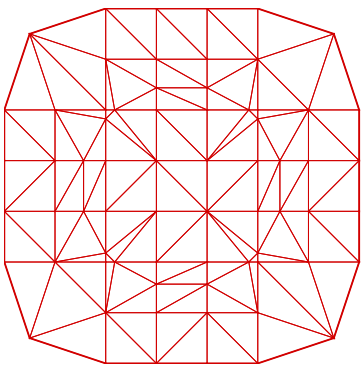

Figure 2.13: The mesh for a circle using the dual marching cubes method

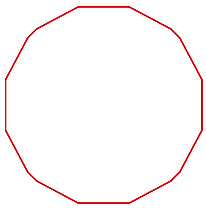

Figure 2.14: The interface for the mesh in Figure 2.13

2D Mesh Post Processing

By using all the quadtree nodes, we may generate some triangles with bad aspect ratio. We can post process the triangles so that the mesh has a better aspect ratio. We combine two processes:

1. Laplacian smoothing,

2. Edge flipping.

The two methods are the most common methods to post process the mesh. The Laplacian smoothing method moves the vertex into the center of the polygon consisting of its neighbor vertices. Edge flipping is used in the Delaunay triangulation method to make the two triangles sharing a common edge locally Delaunay [25]. By using Laplacian smoothing and edge swapping together, we can greatly improve the quality of the final mesh.

Other optimization methods can also be used. For example we could merge bad triangles together to form larger triangles.

2.2.2 Octree Based Mesh Generation Method in 3D

Octree Generation and Parallelization

The octree is used in 3 dimensional space [26]. It is also a tree data structure for which each leaf node (called octant) has exactly 8 children and each octant represents a cubic domain whose two extreme corners have the coordinates $m\_x1[3]$ and $m\_x2[3]$. Each octant has 6 neighbors instead of 4 for a quadtree. The data structure, generation and parallelization are very similar to those for quadtree. Table 2.10 gives a simple octree data structure in the C++ language. Table 2.11 shows how to generate an octree recursively.

The balance step needs special care. For a given octant A, the neighbor octants include 6 neighbor octants sharing a common face and 12 octants sharing a common edge with A. We require that the maximum level difference of A with its 18 neighbor octants is at most 1.

The parallelization of a octree is also similar to that of the quadtree. The only difference is that that the boundary of an octree is a quadtree while the boundary of quadtree is a btree. For the algorithms in detail, refer to those for the quadtree.

Octree-based mesh generation with generalized marching cubes method

Although it is possible to generate templates for the octree with the marching cubes method as with the quadtree marching cubes method in Section 2.2.1, the number of different cases for the octree is very large. Instead, we use the generalized marching cubes method on tetrahedra as we did for the

Table 2.10: A simplified octree structure

```
class OCTREE {
        static int m_min_level;
        static int m_max_level;
        int m_level;
        double m_x1[3], m_x2[3];
        bool m_bToBeSplit;
        OCTREE *Parent;
        OCTREE *NE; // NE x > xmid & y > ymid & z ≤ zmid
        OCTREE *NW; // NW x ≤ xmid & y > ymid & z ≤ zmid
        OCTREE *SW; // SW x ≤ xmid & y ≤ ymid & z ≤ zmid
        OCTREE *SE; // SE x > xmid & y ≤ ymid & z ≤ zmid
        OCTREE *NE2; // NE2 x > xmid & y > ymid & z > zmid
        OCTREE *NW2; // NW2 x ≤ xmid & y > ymid & z > zmid
        OCTREE *SW2; // SW2 x ≤ xmid & y ≤ ymid & z > zmid
        OCTREE *SE2; // SE2 x > xmid & y ≤ ymid & z > zmid

        OCTREE* NorthNeighbor(void);
        OCTREE* WestNeighbor(void);
        OCTREE* SouthNeighbor(void);
        OCTREE* EastNeighbor(void);
        OCTREE* UpNeighbor(void);
        OCTREE* DownNeighbor(void);

        void balance(void);
        void create(void);
};
```

Table 2.11: Algorithm for creating a octree

```
Algorithm OCTREE::create:
    if(this octant needs not to be divided)
        return;
    divide the current octant into eight children
    and update member variables.
    call NE → create();
    call NW → create();
    call SW → create();
    call SE → create();
    call NE2 → create();
    call NW2 → create();
    call SW2 → create();
    call SE2 → create();
    return;
```

generalized marching cubes method for the quadtree in Section 2.2.1. We first tetrahedralize the octree *without* considering the whereabouts of the interface. Thus we have only tetrahedra elements after this step. Then we use generalized marching cubes method on these tetrahedra to recover the interface. The complete algorithm is as follows:

1. Partition the computational region using an octree and balance the octree so that the level difference between any two neighbor octants are no more than 1. Now all those octants are either full octants or boundary octants.

2. tetrahedralize *all* octants.

3. Use generalized marching cubes method on the *tetrahedra* to recover the interface.

4. Post processing the mesh. If we used templates to tetrahedralize the partial quadrants and recover the interface in step 3, we need to move those interface points onto the original interface in the post processing step.

Table 2.12: Algorithm for octree balancing

Algorithm OCTREE::balance:

let this octant be the root of the octree;
insert all leaf octant into a list L;
while(L is not empty)
remove an item μ from the list L;
if(μ is not a leaf)
continue;
let $\mu_{north}, \mu_{west}, \mu_{south}, \mu_{east}, \mu_{up}, \mu_{down}$ be the octant μ's 6 neighbors;
let $lvl_{north}, lvl_{west}, lvl_{south}, lvl_{east}, lvl_{up}, lvl_{down}$ be the their levels;
if($\mu \rightarrow$ NeedToBeSplit())
subdivide μ;
insert the eight new octants into the list L;
check if the eight neighbors of μ need to be subdivided;
if so, insert it/them into the list L;
if($lvl_{north} + 1 < min(lvl_{west}, lvl_{up}, lvl_{east}, lvl_{down})$)
set $\mu_{north} \rightarrow m\_bToBeSplit = true$; push μ_{north} into L;
if($lvl_{west} + 1 < min(lvl_{north}, lvl_{down}, lvl_{south}, lvl_{up})$)
set $\mu_{west} \rightarrow m\_bToBeSplit = true$; push μ_{west} into L;
if($lvl_{south} + 1 < min(lvl_{east}, lvl_{up}, lvl_{west}, lvl_{down})$)
set $\mu_{south} \rightarrow m\_bToBeSplit = true$; push μ_{south} into L;
if($lvl_{east} + 1 < min(lvl_{north}, lvl_{up}, lvl_{south}, lvl_{down})$)
set $\mu_{east} \rightarrow m\_bToBeSplit = true$; push μ_{east} into L;
if($lvl_{up} + 1 < min(lvl_{east}, lvl_{north}, lvl_{west}, lvl_{south})$)
set $\mu_{up} \rightarrow m\_bToBeSplit = true$; push μ_{up} into L;
if($lvl_{down} + 1 < min(lvl_{east}, lvl_{south}, lvl_{west}, lvl_{north})$)
set $\mu_{down} \rightarrow m\_bToBeSplit = true$; push μ_{down} into L;
return the list L;

Table 2.13: Algorithm for OCTREE::NeedToBeSplit

Algorithm OCTREE::NeedToBeSplit:

let $\mu_{north}, \mu_{west}, \mu_{south}, \mu_{east}, \mu_{up}, \mu_{down}$ be the octant μ's 6 neighbors;

if($m\_bToBeSplit$==true)

$m\_bToBeSplit$ = false;

return true; if(μ_{north} is not NULL and not leaf)

if(one of μ_{north}'s four children: SW, SE, SW2, SE2 is not leaf)

return true;

if(μ_{west} is not NULL and not leaf)

if(one of μ_{west}'s four children: SE, NE, SE2, NE2 is not leaf)

return true;

if(μ_{south} is not NULL and not leaf)

if(one of μ_{south}'s four children: NW, NE, NW2, NE2 is not leaf)

return true;

if(μ_{east} is not NULL and not leaf)

if(one of μ_{east}'s four children: NW, SW, NW2, SW2 is not leaf)

return true;

if(μ_{up} is not NULL and not leaf)

if(one of μ_{up}'s four children: NE, NW, SW, SE is not leaf)

return true;

if(μ_{down} is not NULL and not leaf)

if(one of μ_{down}'s four children: NE2, NW2, SW2, SE2 is not leaf)

return true;

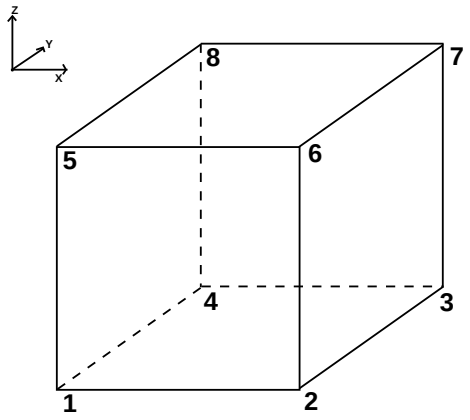

Figure 2.15: The standard cube with 8 vertex

If we have an uniform octree, all octants have the same size. We could tetrahedralize all octants in a consistent way so that we only need to tetrahedralize cube by cube with out considering how the neighbors are meshed. For example, for the cube in Figure 2.15, we connect the following diagonals: (4,5), (3,6), (1,6), (4,7), (1,3), (5,7) and (4,6) where (a, b) means the edge connecting vertex a with b. Thus we have six tetrahedra in this cube as in Figure 2.16: (1,2,3,6), (1,3,4,6), (3,7,4,6), (1,6,4,5), (6,7,4,5) and (4,7,8,5). Note that for any two parallel faces, we always connect the diagonals in the same direction. For example, for the two faces perpendicular with the X-coordinate, we connect the two parallel diagonals: (4,5) and (3,6). Since diagonals on the opposite faces of the cube are always in the same direction, we could copy the same approach for the neighbor cubes. Thus we tetrahedralize all cubes consistently octant by octant.

If the octree is not uniform, it is more complicated to generate the tetrahedra. However we could generate the tetra octant by octant following the

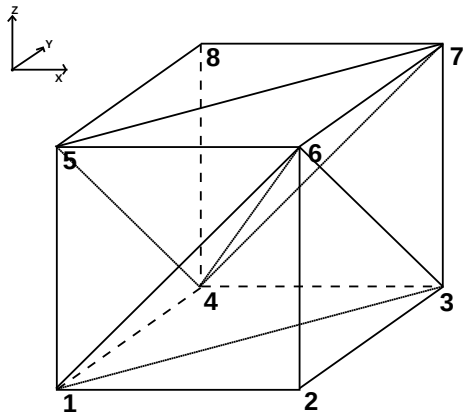

Figure 2.16: The 6 tetrahedra for the standard cube with 8 vertex

approach for the quadtree in Section 2.2.1. For a given face, we always draw the diagonal in the same direction if possible. If there exists a smaller octant neighbor, we insert a new vertex at the center of the octant, and always connect the diagonals of the opposite faces in the same direction. And then use the new vertex and the diagonals to generate the tetrahedra. As an example, we show how to generate the tetrahedra in Figure 2.18 when the octant has only one smaller neighbor as in Figure 2.17. The generated tetrahedra are (1,10,9,14), (9,10,13,14), (9,13,5,14), (5,13,12,14), (10,4,13,14), (4,11,13,14), (13,11,12,14), (12,11,8,14), (1,5,6,14), (1,6,2,14), (2,6,3,14), (3,6,7,14), (3,7,4,14), (4,7,8,14), (1,2,3,14), (1,3,4,14), (5,8,7,14) and (5,7,6,14). Similarly, we generate the templates for other cases. Note that we always draw the diagonals of the opposite faces in the same direction, so that we can tetrahedralize the whole octree consistently octant by octant.

After we generate the tetrahedra, we use the marching cubes method on the tetrahedra as on the triangles in Section 2.2.1. For more detail, refer to

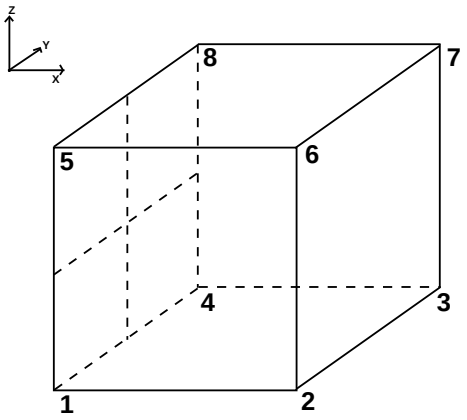

Figure 2.17: An octant with a smaller neighbor.

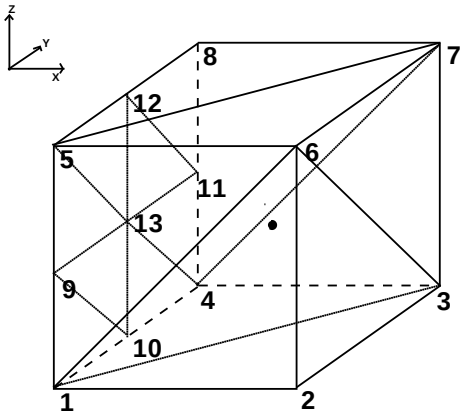

Figure 2.18: The tetrahedralization of an octant with a smaller neighbor.

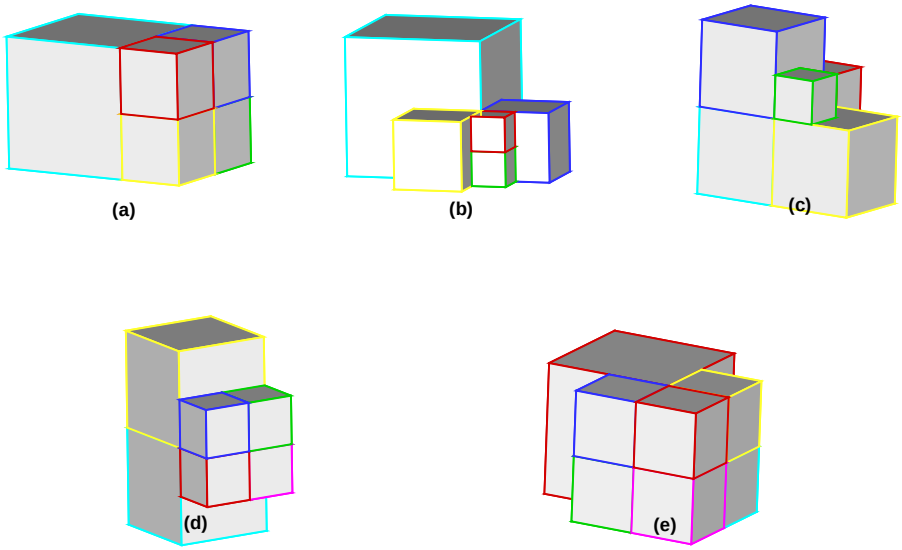

Figure 2.19: The possible cases for the cubes with 5, 6, 7 octants: (a), (b), (c) has 5 octants, (d) has 6 and (e) has 7 octants. Each octant corresponds to 1 vertex in the dual grid. Thus the 5 octants in (a), (b), (c) give a 5 vertex polygon, (d) gives a 6 vertex polygon and (e) gives a 7 vertex polygon.

[3].

Octree-based mesh generation with dual generalized marching cubes method

We could also use the dual grid of the octree similarly with the dual of the quadtree. The cells of of the dual of the quadtree have 3 or 4 vertices (triangle or rectangle), while the cells of the dual of the octree consists of polyhedral with 5, 6, 7, 8 vertices. We could regard the polyhedra with 5,6,7 vertices as degenerate cells with 8 vertices.

Thus we only need to generate the templates for the cubes. For such kind of meshing for cubes with 8 vertices, we refer the reader to Wei Guo's

thesis [23]. The meshing templates of cubes with 5,6,7 vertices could also be generated similarly. Or they could be generated using the templates for the 8 vertices cells through merging the corresponding vertices.

After we have meshed all cells into tetrahedral, we use the generalized marching cubes method on those tetrahedral as before. Figure 2.20 is an example for 3D mesh using the above approach and Figure 2.21 is the interface recovered from the mesh using marching cube method.

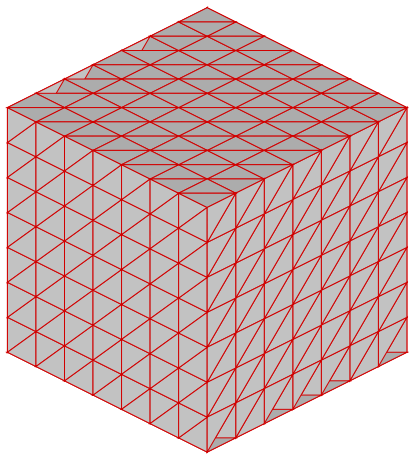

Figure 2.20: The mesh for a cube containing a sphere using the octree and the dual marching cube method.

Figure 2.23 is the interface of three spheres recovered using the octree and the marching cube method.

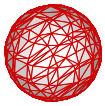

Figure 2.21: The interface for the sphere mesh in Figure 2.20.

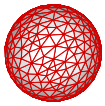

Figure 2.22: The smoothed interface for the sphere mesh in Figure 2.20.

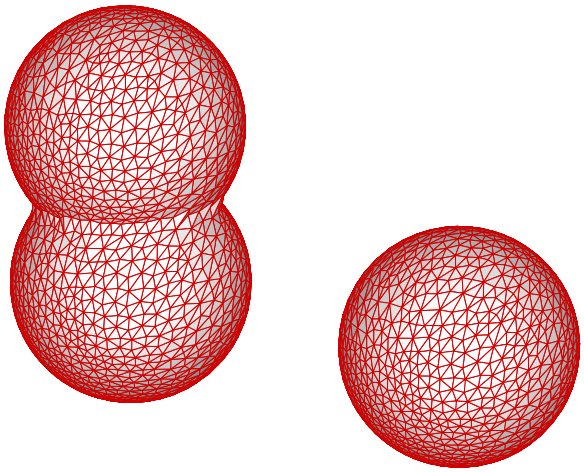

Figure 2.23: The recovered interface for three spheres.

3D Mesh Post Processing

Most two dimensional mesh optimization algorithms have 3D versions. However, they are more complicated. The most used algorithms includes:

1. Laplacian smoothing;
2. Delaunay triangulation;
3. Merge and divide.

2.2.3 Point Location

After the mesh is given, we use the finite element method to set up the matrix and solve for the unknowns. Sometimes we need to obtain the solutions for arbitrary points inside the mesh, which is called a point location problem: find the triangle/tetrahedron which contains the given point. The point location problem and the closely related range search problem are famous problems in computational geometry. See [25] and references cited therein.

If only one point is queried, we only need to loop through every triangle/tetrahedron of our mesh and test whether the triangle/tetra contains the given point. The time complexity is clearly $O(N)$ where N is the number of triangles/tetrahedra inside the mesh. If m such points are to be queried, such an approach would not be applicable when m is large such as $m = O(N)$. We would be in such a situation if we solve an elliptic interface problem using mixed finite elements on an unstructured grid and then interpolate the flux back to an cartesian grid.

To speed up the point location problem, we preprocess the mesh and set up some special data structures. For example, Edelsbrunner [32] uses a layered

directed acyclic graph (dag) structure which could be built in $O(N)$ time, uses $O(N)$ storage and achieves the point location in $O(log(N))$ time. Most mesh generation methods also use some data structures for point location during their generation processes, for example the incremental Delaunay triangulation [25] and the advancing front method. Although the quadtree/octree does not do point location during the mesh generation process, the quadtree/octree are natural data structures for point location since they are in fact tree structures. Our point location is a bucketing method and uses the quadtree/octree structure.

Our algorithm is as follows: given a point p, the quadtree/cctree and the mesh,

1. first use the quadtree/octree structure to find a leaf quadrant/octant;

2. second use the leaf quadrant/octant to find an triangle/tetrahedron which would be used as an starting point to find the target triangle/tetrahedron;

3. walk through the mesh to the given point p.

Now, we will explain our algorithm in detail. Since there is no difference in principle between the 2D and 3D algorithms for point location, we shall explain the algorithm in 2D only. When the mesh contains multiple component triangles/tetrahedra, special care will be taken to make sure that the final triangles has the same component as the component of the given point P.

For the first step of our algorithm, we use the root of the quadtree, q, to find its child quadrant q which contains the given point p. Then set

q = q and find its child q which contains p. We loop until we reach the leaf quadrant which contains the given point. The time complexity for the quadtree traversing is O(log(h)) where h is the highest level of the quadtree.

For the second step, we use the leaf quadrant to find a starting triangle T. Since the leaf quadrant is a rectangle, it must contain four vertices: the north-east corner (NE), north-west corner (NW), south-west corner (SW) and south-east (SE) corner. According to our meshing method, the leaf quadrant is either a full quadrant or a partial quadrant. The four corner points of the quadrant will become vertices of the interior mesh and they can not be vertices of the interface which separates two components. Thus the corners have the same component as those triangles incident to those corners vertices. Thus in order to find a starting triangle with the same component as the given point, we only need to find one of the four corners with the same component and use any one of the triangles incident to the corner vertices as our starting triangle. This step takes only O(1) time.

For the last step, we walk through the mesh from the starting triangle T to the given point p. Since one vertex of T and p are both contained in the leaf quadrant, they are not far from each other. In fact they should be separated by no larger than a constant number of triangles. The reason is that the triangulation of the leaf quadrant uses templates and there are finite such cases. Thus the walking algorithm only takes O(1) time. There are mainly two ways of walking [33, 34].

One method is the straight walk. This method needs an initialization step. In this method, one starts with a vertex q and circles around itself to

find a triangle T with which the ray qp crosses. After this, the walk really begins. Suppose that the ray qp goes out of the triangle T through the edge e. If T does not contain p, we need to go through T to find its neighbor T' and find the edge e' through which qp goes out of T'. And then go on until we find the final triangle T which contains p. However, during the walking, reinitialization is needed if the ray qp goes out of T through a vertex. We could say that this method walks by edges.

The other method is the visibility walk, which walks by triangles. Given a vertex q, we first find a starting triangle T (any incident triangle to q is fine). Then we get a new T' by testing which side of the triangle T's edges p lies on. Then we loop until we find the target triangle containing the point p. This method does not need to reinitialize and is much simpler. However, it is possible that the walking will fall into a cycle and do not terminate. A remedy is to walk with memory by keeping the triangles already walked through. We use the second method since it is simpler to implement.

From the above, we can easily see that the point location algorithm takes O(log(h)) time where h is the highest level of the quadtree. When the quadtree is balanced or almost balanced (the difference of the maximum level and minimum is a constant), the query time takes O(log(N)) time where N is the number of the triangles.

2.3 Conclusion

This chapter revisited the quadtree/octree method to generate the mesh by using the marching cubes method to recover the interface. By using the

marching cubes method to represent the interface, we greatly simplify the cases for generating the triangles for each quadrant in 2D and tetrahedral for octant in 3D. And therefore, this method greatly simplifies the quadtree/octree mesh generation method [1] and makes it much easier to program.

It is also worth noting that the quadtree/marching cubes method has two distinct steps:

1. use quadtree/octree to generate the underling graded grid;

2. recover the boundary using the marching cubes method.

By using marching cubes method, we create a new boundary which is an approximation to the input boundary. If the input uses B-rep and the new recovered boundary is required to be an conforming boundary of the original one, we could use the methods, for example in [15], to recover first the vertices, and then edges in 2D, and last the faces for 3D problems as the second step for recovering the boundary.

Chapter 3

Mixed Finite Element Method

3.1 Introduction

An elliptic boundary value problem seeks the solution of an elliptic differential equation with given boundary data. An elliptic interface problem is a special elliptic boundary value problem with an interior boundary; the solution has a specified jump across the interior boundary. The interior boundary is also called an interface. Many problems require the solution of an elliptic boundary value/interface problem. For example, the fractional-step projection method for the incompressible flow solves an elliptic boundary value problem [35–37]. When two phase incompressible flows are simulated using the projection method, the elliptic step solves the elliptic interface problem instead of the elliptic boundary value problem. The interior interface separates the two different materials.

When solving elliptic problems, the first step is to generate the mesh for the computational domain. An ideal mesh would require that the boundary/interfaces do not cross the cells of the mesh and lie only on the edges/faces of the cells in the mesh. Such a mesh is called a boundary conforming mesh. When such a mesh is given, no great difficulty exists in solving the boundary value/interface problem. For example, we could use finite volume/finite element method/mixed finite element method. However, to generate a boundary/interface conforming mesh is not easy; especially when the domain is complex, an unstructured mesh is needed.

Other approaches using structured nonconforming cartesian meshes have been developed to solve the elliptic boundary value/interface problem. Since cartesian meshes require little time to be generated, they are very popular.

The boundary/interface does not necessarily lie on the edges of the cartesian cells. Several methods using nonconforming cartesian mesh exist for the elliptic boundary value/interface problem. One popular method is Peskin's immersed boundary method [38]; LeVeque and Li's immersed interface face [39] is another. Peskin's method solves the problem by smoothing the discontinuous coefficient around the interface. LeVeque and Li's method uses a Taylor expansion to incorporate the influence of the discontinuous coefficient into the finite difference scheme. The embedded boundary method by Johansen and Colella [40] is also used on a cartesian grid for solving the elliptic boundary value problem.

We use the mixed-hybrid finite element to solve the elliptic interface/boundary value problem. A triangular mesh could be generated, for example, by the quadtree/octree method developed in Chapter 2. The mixed finite element method is well developed and applied using function bases from RT0 (Raviart-Thomas space of degree zero) [41–46]. However RT0 only gives first order accuracy in the flux. A few papers have used higher order schemes, such as the BDM1 basis (Brezzi-Douglas-Marini space of degree one) and the RT1 (Raviart-Thomas space of degree one) which both give a 2nd order accurate flux approximation. The reasons for preferring RT0 are that most of the practical problems have solutions with low regularity and RT0 is much easier to code. However, for the elliptic interface problems, if we use unstructured grids to make the interface lie on the edges of the triangle, the solution has sufficient regularity to benefit from higher order treatment. We will also show that the higher order mixed finite element methods are not too difficult to implement.

In this chapter, we extend the implementation by Chavent and Roberts [44] for RT0 to higher basis functions and discuss how to treat the jump condition for the elliptic interface problems. We show step by step how to setup the matrix for the elliptic problem. Our matrix setup method is a little different from the standard approach [42, 43]. It is much easier to implement.

3.2 Our Implementation of the Mixed Hybrid Finite Element Method

For the theoretical results regarding mixed finite element method for the elliptic boundary value/interface problem, see [42, 43] for more details. For implementation, see [43, 47].

In this section, we will show in detail our implementation of the mixed-hybrid finite element method for the elliptic boundary value/interface problem.

In Section 3.2.1, we show the general frame work of implementation for mixed hybrid finite elements. In Section 3.2.2, we state the implementation using the flux basis in RT0, which was also shown in detail on structured rectangle elements by Chavent and Roberts [44]. In Sections 3.2.3, 3.2.4, 3.2.5, we extend the formulation to use flux bases in BDM1, RT1, BDM2 respectively. In Section 3.2.6, we show how to use the master element. In Section 3.2.7, we show that the mixed hybrid finite element method could be easily extended to solve those problem where the potential or flux is discontinuous across the element boundary. In Section 3.2.8, we show some examples.

3.2.1 Mixed FEM for elliptic boundary value/interface problem

In this section, we show how to implement the mixed hybrid finite element method for the elliptic boundary value/interface problem. The standard elliptic interface problem is the following:

$$-\nabla \cdot a\nabla P = f, \tag{3.1}$$

where P is the potential function and a is a piecewise continuous function. The difficulty of such an problem is that the coefficient function a is not continuous and sometimes we need the flux $\vec{q} = -a\nabla P$ instead of P. In order to solve the elliptic interface problem, we first partition the domain Ω into a triangulation T_h. For simplicity, all elements are assumed to be triangles as in Figure 3.1.

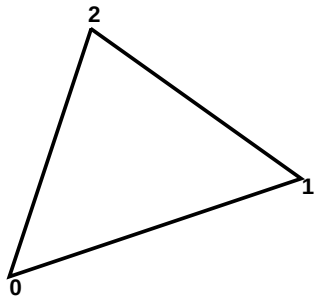

Figure 3.1: triangle element

Instead of 3.1, we use the following two first order equations:

$$\vec{q} = -a\nabla P, \tag{3.2}$$

$$\nabla \cdot \vec{q} = f. \tag{3.3}$$

Let us denote the approximation space of $(\vec{q}, P)$ by (V_h, W_h), where V_h is a space for vector functions and W_h is a space for scalar functions. Let $\vec{s} \in V_h$ be a test function. Multiply both side of 3.2 by $\vec{s}$, integrate over $K \in T_h$ and use Green's formula. Then we have:

$$\int_K \frac{\vec{q}}{a} \cdot \vec{s} = \int_K P\nabla \cdot \vec{s} - \int_{\partial K} TP\vec{s} \cdot \vec{n}, \tag{3.4}$$

where TP is the approximation of P at the boundary of K. Similarly, take $v_K \in W_h$ and multiply both side of 3.3 by v_K, we have:

$$\int_K v_K \nabla \cdot \vec{q} = \int_K v_K f. \tag{3.5}$$

We will need equation (3.4) and (3.5) in our following derivation. For convenience, we give the basis functions we are going to use. The basis for the Raviart-Thomas space of degree k are the following:

$$V_h = \{\vec{s} \in H(\nabla, \Omega) : \vec{s}|_K \in P^k(K) \times P^k(K) + \mathbf{x}P^k(K) \text{ for all } K \text{ in } T_h\},$$

$$W_h = \{v \in L^2(\Omega) : v|_K \in P^k(K) \text{ for all } K \text{ in } T_h\}.$$

The basis from Brezzi-Douglas-Marini space of degree k are:

$$V_h = \{\vec{s} \in H(\nabla, \Omega) : \vec{s}|_K \in P^k(K) \times P^k(K) \text{ for all } K \text{ in } T_h\},$$

$$W_h = \{v \in L^2(\Omega) : v|_K \in P^k(K) \text{ for all } K \text{ in } T_h\}.$$

3.2.2 RT0 basis

For RT0 (Raviart-Thomas space of degree zero), the basis function is:

$$\vec{s}|_K = a \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b \\ c \end{pmatrix},$$

$$v|_K = d,$$

where a, b, c, d are constants. Thus the flux is approximated by a linear function and the potential by a constant. The mixed hybrid finite element also must approximate the potential at the edge, TP. For RT0, TP is also approximated by a constant. For each triangle element, there is only one basis function for the potential, which we denote as P. On each edge, there exists one basis for TP, which is denoted as TP. For each element, there are three basis functions for the flux. Let us denote them as

$$\vec{s_i}|_K = \begin{pmatrix} a_i x + b_i \\ a_i y + c_i \end{pmatrix}, \quad \text{i=0,1,2}$$

We could fix the coefficient so that $\vec{s}_i|_K$ satisfies the following relation

$$\vec{s}_i \cdot n_i|_{E_j} = \delta_{i,j},$$

where E_j is the mid point of the j^th edge and $\delta_{i,j}$ is the Kronecker function. Thus any flux function $\vec{q}$ belonging to V_h could be written as

$$\vec{q} = \sum_{i=0}^{i=2} Q_i \vec{s}_i.$$

Now we are ready to get our formulation. First, substituting $\vec{q}$, P, TP into (3.4) we have:

$$\int_K \frac{\vec{q}}{a} \cdot \vec{s}_j = \int_K P\nabla \cdot \vec{s}_j - \int_{\partial K} TP\vec{s}_j \cdot \vec{n},$$

or

$$\int_K \frac{\sum_{i=0}^{i=2} Q_i \vec{s}_i}{a} \cdot \vec{s}_j = \int_K P\nabla \cdot \vec{s}_j - \sum_{i=0}^{i=2} \int_{\partial K_i} TP_i \vec{s}_j \cdot \vec{n},$$

Taking out the coefficients from the integral, we have

$$\sum_{i=0}^{i=2} Q_i \int_K \frac{\vec{s}_i}{a} \cdot \vec{s}_j = P \int_K \nabla \cdot \vec{s}_j - \sum_{i=0}^{i=2} TP_i \int_{\partial K_i} \vec{s}_j \cdot \vec{n},$$

which could be written using matrix notation:

$$AQ = DP - WTP, \tag{3.6}$$

where

$$A_{3\times 3} = (A_{i,j}) = \left(\int_K \frac{\vec{s}_i \cdot \vec{s}_j}{a}) \right),$$

$$Q_{3\times 1} = (Q_i)$$

$$D_{3\times 1} = (D_i) = \left(\int_K \nabla \cdot \vec{s}_i \right),$$

$$W_{3\times 3} = (W_{i,j}) = \left(\int_{\partial K_i} \vec{s}_j \cdot n_i \right),$$

$$TP_{3\times 1} = (TP_i).$$

Note that because of the way we fix the freedom of the basis function, the matrix W is a diagonal matrix and we could easily calculate it as

$$W_{i,i} = \text{length } (\partial K_i) \text{ and } W_{i,j} = 0 \text{ if i} \neq \text{j} .$$

Substituting the basis function into equation (3.5), we have

$$\sum_{i=0}^{i=2} \int_K v_K \nabla \cdot (Q_i \vec{s}_i) = \int_K v_K f.$$

Dividing both sides by v_K (which is an arbitrary constant) and taking the coefficients Q_i outside the integral, we have

$$\sum_{i=0}^{i=2} Q_i \int_K \nabla \cdot (\vec{s}_i) = \int_K f,$$

or using matrix notation

$$D^T Q = F, \tag{3.7}$$

where $F = \int_K f$. By using equation (3.6) and the fact that normal component of flux along an edge is continuous, we eliminate the Q and P from the final stiff matrix. Multiplying both side of equation (3.6) we have

$$Q = A^{-1}DP - A^{-1}WTP. \tag{3.8}$$

Multiplying the above equation by D^T, we get

$$D^TQ = D^TA^{-1}DP - D^TA^{-1}WTP = F. \tag{3.9}$$

Let $S = D^TA^{-1}D$. Then

$$P = S^{-1}D^TA^{-1}TP + S^{-1}F, \tag{3.10}$$

where S is a scalar for RT0. We now substitute P into equation (3.8), to obtain

$$Q = A^{-1}D(S^{-1}D^TA^{-1}TP + S^{-1}F) - A^{-1}TP. \tag{3.11}$$

Using the fact that the potential and the flux normal component are continuous across the element edges, we obtain

$$TP_{K,A} = TP_{K',A}, \tag{3.12}$$

and

$$Q_{K,A} + Q_{K',A} = 0, \tag{3.13}$$

where A is the common edge of the two neighboring triangle K and $K^{'}$. Now

we can assemble the stiff matrix equation with only TP as unknown variables.

3.2.3 BDM1 basis

For the BDM1 (Brezzi-Douglas-Marini space of degree one), the 6 basis functions for the flux are:

$$\vec{s}|_K = \begin{pmatrix} a_1x + a_2y + a_3 \\ b_1x + b_2y + b_3 \end{pmatrix},$$

and the basis for the potential is the same as RT0:

$$v|_K = d,$$

where $a_1, a_2, a_3, b_1, b_2, b_3, d$ are constants. The potential at the edge, TP, is approximated by a linear function. We use the two Gauss-Legendre points (a_1, a_2) on the edge A to fix the two degrees of freedom for TP as

$$TP|_A = TP_1w_1 + TP_2w_2,$$

where $w_i(a_j) = \delta_{i,j}$.

Then the derivation of the method to assemble the stiff matrix element by element is the same as for the the RT0 bases, except that we now have 6 basis elements for the flux, 6 basis elements for TP and

$$A_{6\times 6} = (A_{i,j}) = \left(\int_K \frac{\vec{s}_i \cdot \vec{s}_j}{a}) \right),$$

$$Q_{6\times 1} = (Q_i),$$

$$D_{6\times 1} = (D_i) = \left(\int_K \nabla \cdot \vec{s}_i \right),$$

$$W_{6\times 6} = (W_{i,j}) = \left(\int_{\partial K_i} \vec{s}_j \cdot n_i \right) = (\text{length}(\partial K_i)),$$

$$TP_{6\times 1} = (TP_i).$$

As in RT0, we could easily calculate the matrix W as

$$W_{2i,2i} = W_{2i+1,2i+1} = \frac{length(\partial K_i)}{2} \text{ and } W_{i,j} = 0 \text{ if i} \neq \text{j} .$$

3.2.4 RT1 basis

For the RT1 (Raviart-Thomas space of degree one), the 8 basis functions for the flux are:

$$\vec{s}|_K = \begin{pmatrix} a_1 x + a_2 y + a_3 \\ b_1 x + b_2 y + b_3 \end{pmatrix} + \begin{pmatrix} x \\ y \end{pmatrix} \times (c_1 x + c_2 y) \Bigg),$$

and the potential $v|_K$ expanded in linear basis elements gives

$$v|_K = P_1 p_1 + P_2 p_2 + P_3 p_3,$$

where $a_1, a_2, a_3, b_1, b_2, b_3$ are constants and p_1, p_2, p_3 are basis functions. p_i satisfies $p_i(E_j) = \delta_{i,j}$ where E_j are the mid points of the edges. The potential

at the edge, TP, is approximated by a linear function as BDM1.

Then the derivation is the same as RT0, except that we have 8 bases for the flux and 6 bases for TP and

$$A_{8\times 8} = (A_{i,j}) = \left(\int_K \frac{\vec{s}_i \cdot \vec{s}_j}{a}) \right),$$

$$Q_{8\times 1} = (Q_i),$$

$$D_{8\times 3} = (D_{i,j}) = \left(\int_K p_i \nabla \cdot \vec{s}_i \right),$$

$$W_{8\times 6} = (W_{i,j}) = \left(\int_{\partial K_i} \vec{s}_j \cdot n_i \right) = (length(\partial K_i)),$$

$$TP_{6\times 1} = (TP_i).$$

As in RT0 and BDM1, we calculate W as

$$W_{2i,2i} = W_{2i+1,2i+1} = \frac{length(\partial K_i)}{2} \text{ and } W_{i,j} = 0 \text{ if i} \neq \text{j} .$$

3.2.5 BDM2 basis

For the BDM2 basis (Brezzi-Douglas-Marini space of degree two), there are 12 basis functions for the flux and 3 basis functions for the potential in a triangle element. The basis functions for the Lagrangian multipliers are quadratical functions on the inter element edges. The matrix setup is similar to that for the RT1. Refer to [42] for detail.

3.2.6 Reference Element

As we have noted, the implementation in the previous sections requires the computation of the following integration:

$$A_{3\times 3} = (A_{i,j}) = \left(\int_K \frac{\vec{s_i} \cdot \vec{s_j}}{a}) \right),$$

$$D_{3\times 1} = (D_i) = \left(\int_K \nabla \cdot \vec{s_i} \right),$$

$$W_{3\times 3} = (W_{i,j}) = \left(\int_{\partial K_i} \vec{s_j} \cdot n_i \right).$$

It is possible to evaluate the above numerical integrations using arbitrary elements. This approach requires that we invert a matrix to solve for the coefficients of the flux bases for each element. Although the basis functions from RT0 could be written down explicitly for arbitrary elements, it is not easy to write down the flux bases for other higher order spaces. Thus we need to use the conditions that the bases have to satisfy to set up a system of equations and then invert the matrix. Although these are possible, it is not encouraged since the inverse of a matrix for every element takes unnecessary time and also the inverse of the basis coefficient matrix of a distorted element is less accurate than those taken in a well shaped element.

The general approach for the integration of the basis functions on arbitrary elements is to perform integration on a reference element. The use of reference elements is an important part of the finite element method both for convergence analysis and the implementation. First the basis function on the reference element is defined and then the basis functions on arbitrary element

are defined by a change of variables like the following:

$$v_{h|K} = \hat{v} \circ F^{-1}.$$

However, the above transformation would change the value normal components of the basis function from zero to non zero. To overcome this difficulty, the Piola's transformation was introduced [42]:

$$\vec{q}(x) = \frac{1}{J(\hat{x})} DF(\hat{x}) \hat{\vec{q}}(\hat{x}).$$

3.2.7 Mixed Hybrid Finite Elements for Discontinuous Potential/Flux Problems

The previous derivations for elliptic problems all assume that the potential/flux normal component is continuous across the element edges. However, it is easy to solve those problems with jumps in the potential/flux across edges. We need only to modify equations (3.12) and (3.13) to incorporate such jumps

$$TP_{K,A} = TP_{K',A} + \text{jump}TP, \tag{3.14}$$

and

$$Q_{K,A} + Q_{K',A} = \text{jump}Q. \tag{3.15}$$

3.2.8 Examples

In this section, we use the mixed hybrid finite element method to solve a simple elliptic interface problem 3.1 where the interface is a cicle centered around the origin with radius $r_0 = 0.5$. The computational domain is inside $[-1, 1] \times [-1, 1]$. The coefficient a is defined as

$$a = \begin{cases} 1, & r \leq 0.5, \\ 10, & r > 0.5. \end{cases}$$

and $f = -9r$. The exact solution is

$$P = \begin{cases} r^3, & r \leq 0.5, \\ \frac{r^3}{10} + (1 - \frac{1}{10})0.5^3, & r > 0.5. \end{cases}$$

In order to show the accuracy of the mixed finite element method, we generate our triangulation in the following way:

1. generate a uniform grid consisting of rectangles;

2. use marching cubes method to recover the interface.

Then the interface lies on the edges of the triangles. Figure 3.2 gives a mesh with the uniform grid being 32×32.

We only give the flux error in L^2 norm for BDM1 method in Table 3.1. From the table, we find that the method gives a 2nd order accurate flux in the L^2 norm. Figure 3.3 shows the graph of the potential with a 32×32 uniform grid.

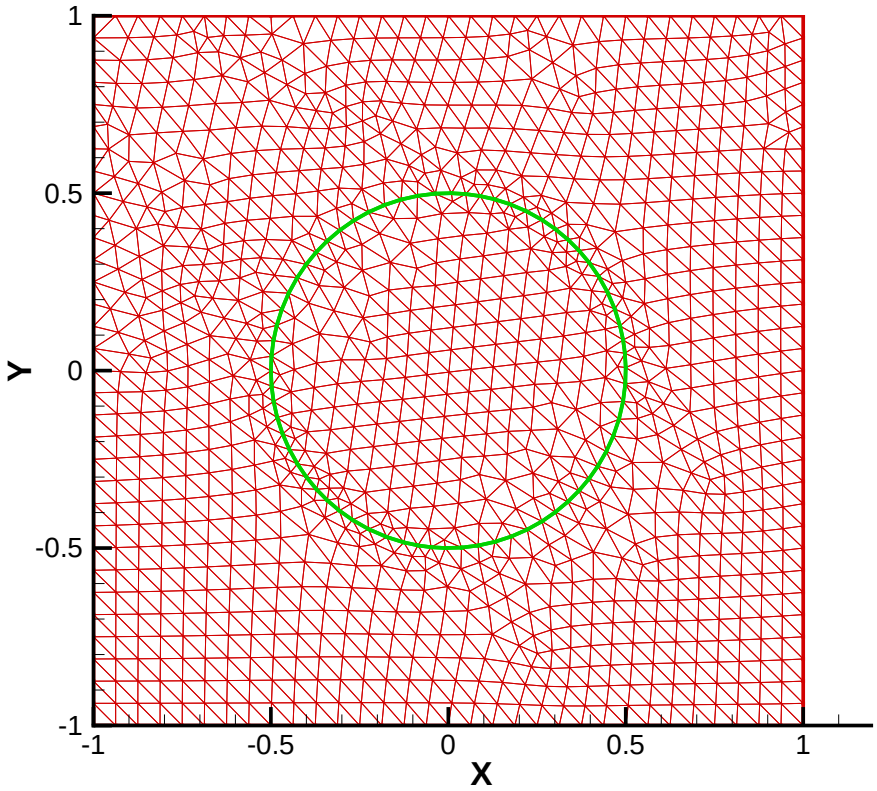

Figure 3.2: mesh

Table 3.1: The flux errors for an elliptic interface problem with jump in the flux

Mesh Size	flux error in L^2 norm
8×8	0.046384
16×16	0.011289
32×32	0.002862
64×64	0.000725
128×128	0.000189

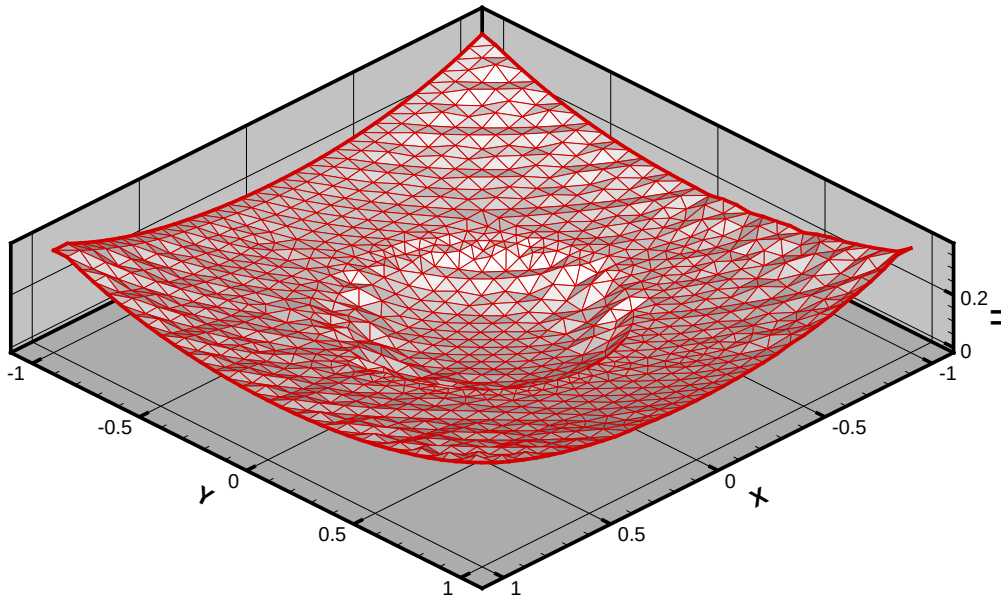

Figure 3.3: potential

3.3 Conclusion

In this chapter, we have given the implementation of the mixed hybrid finite element method for elliptic problem by using four different basis functions: RT0, BDM1, RT1 and BDM2. Note that we used the same approach for the all basis functions to assemble the stiff matrix. The implementations for higher order basis functions are the same. For elliptic interface problems, we first triangulate the domain so that the interface lies on the triangle edges only. If the potential/flux is continuous across the interface, then we use the mixed hybrid methods directly to solve them.

Note, however, that there are also elliptic interface problems where the potential/flux have jumps across the interface. After only slight modification, the mixed hybrid finite element method can again be used to solve these problem. Compared with the popular immersed boundary method and the immersed interface method, the mixed finite element method can give higher order accurate solutions by using higher order basis functions. For example, BDM1 and RT1 both give a 2rd order accurate flux in the L^2 norm. BDM2 gives a 3rd order accurate flux in the same norm.

Chapter 4

A Comparison Study

4.1 Introduction

The purpose of this chapter is to present a comparative study of two popular methods for the solution of elliptic boundary value problem: the embedded boundary method (EBM) and the mixed finite element methods (MFEM). The methods are quite different in their performance characteristics and the mixed finite element methods come in several versions, using different basis functions.

To present our main results in an easily accessible manner, we arrange the results in a table of solution time for comparable accuracy. We find that the EBM is better than lower or the same order accurate MFEM, but not as good as the higher order accurate MFEM we test here.

We observe that no single study of comparison can be definitive, as comparison results may be dependent on the problem chosen, the accuracy desired and comparison method selected. To begin, we distinguish between two not so different kinds of elliptic problems: the elliptic boundary value problems and the elliptic interface problem. For the elliptic boundary value problem, the computational domain exists only on one side of the boundary, for example, interior/exterior boundary value problem. For the elliptic interface problem, there is some internal boundary called an interface across which the solutions on the two sides satisfy some jump conditions.

There are many methods for solving elliptic boundary value/interface problems. Several popular methods have been developed on cartesian meshes for the boundary value/interface problems: the immersed boundary method (IBM) by Peskin [38], the immersed interface method by LeVeque and Li [39],

the ghost fluid methods (GFM) by Liu, etc. [48], the embedded boundary method by Johansen and Colella [40], integral equation method by Mayo [49], Mckenney, Greengard and Mayo [50]. The advantage of these methods is that they are defined on a cartesian mesh. Therefore, there is no need to generate a mesh. For the cells away from the boundary/interface, they use a central finite difference method which is simple and second order accurate. For the cells near or crossing the boundary/interface, a special different algorithm treatment is needed. When a (structured/unstructured) mesh is generated before hand, we could use a finite element/finite volume method. It is not easy to get high accuracy by using a finite volume method. The finite element method could have very high accuracy if high order basis functions are used. For elliptic boundary/interface problems, we could use Galerkin finite elements, the discontinuous Galerkin method, and the mixed finite element method. When the boundary/interface is complex, the apparent choice is to use a finite element method (FEM) with an unstructured mesh. However, it is not easy to generate an unstructured mesh especially when the boundary is very complex and the boundary changes with time. Another disadvantage of using FEM with an unstructured mesh is that it does not have the super convergence property which follows when using a uniform structured mesh.

Most of the comparison studies for elliptic boundary value/interface problems are conducted either through mesh refinement or by comparing methods using cartesian mesh [39, 40, 48–50]. In this chapter we are to perform a comparison study of two methods for solving the elliptic boundary value problem: the embedded boundary method using a cartesian mesh and the mixed fi-

nite element method using an unstructured mesh. The EBM uses ghost cells along the boundary and the finite volume method to achieve 2nd order accuracy in the potential and flux. The MFEM uses an unstructured triangular mesh. Instead of solving the second order elliptic equation, it solves two first order equations and gives the potential and flux at the same time. Higher order basis functions give higher order of accuracy. Refer to [42] for a thorough discussion of mixed and hybrid finite element methods. For a more implementation oriented view, see [44]. The advantages and disadvantages of the MFEM are discussed briefly in [46]. For the comparison between FEM and MFEM, see the references cited in [45]. In this chapter we use the RT0 (Raviart-Thomas space of degree zero), the RT1 (Raviart-Thomas space of degree one), BDM1 (Brezzi-Douglas-Marini space of degree one) and BDM2 (Brezzi-Douglas-Marini space of degree two) as basis functions of the flux. We use the mixed-hybrid FEM. The final algebraic equations have only the potentials on the mesh edges as unknowns. To use the MFEM, we need to generate the mesh for the computational domain. There are mainly three methods for meshing: the Delaunay triangulation [14], the advancing front method [15] and the quadtree/octree method [1]. In this chapter we use a method based on the quadtree/octree method. This method simplified the original construction by using the marching cubes method to recover the interface.

For the implementation of the embedded boundary method, see [40, 51]. The numerical results by the embedded boundary method is run by Dr. Jian Du using his code.

4.2 A Comparison Study

We are to solve the elliptic problem:

$$\begin{cases} \phi_{xx} + \phi_{yy} = f \\ \frac{\partial \phi}{\partial n} = g \end{cases} \tag{4.1}$$

in a complex domain, where $\phi(x, y)$ is called the potential. Since the gradient of the solution $\nabla\phi$ is often needed and more difficult to solve for, we will use the gradient errors as the comparison criterion. The gradients at both the regular grid centers and the boundary points are calculated and compared.

For our test problem, we use $\phi = e^{\frac{x^2+y^2}{2}}$ as the exact solution of the elliptic equation (4.1); f and g are obtained by differentiating ϕ. We will show two different test problems using the same equation and analytic solution. The difference between the two problems lies only in the boundary: the second boundary is more complex than the first one. The EBM uses a structured cartesian grid. The mixed finite element methods use an unstructured grid based on the quadtree/octree construction. The quadtree/octree have minimum and maximum levels. In order to compare the results, we need to have comparable grids by letting the minimum/maximum level of the quadtree to be equal. Fig. 4.1 shows the grid used by the mixed finite element methods when EBM uses the 128×128 grid. Thus the mesh is uniform. We compare

the results using the L_2 norm of the flux $\|\nabla\phi\|_2$. The norm is defined as:

$$\begin{cases} \|\nabla\phi\|_2 = \sqrt{\sum_{\text{face}} \|\nabla\phi\|^2_{2,\text{face}}} \\ \|\nabla\phi\|_{2,\text{face}} = \text{Area(face)} \times \sqrt{\phi_x(x_0,y_0)^2 + \phi_y(x_0,y_0)^2} \end{cases} \tag{4.2}$$

where (x_0, y_0) is the center of the rectangle for the cartesian grid used by the EBM. For the MFEM, we first interpolate the fluxes at the center of the cartesian grid, and then compute the norm.

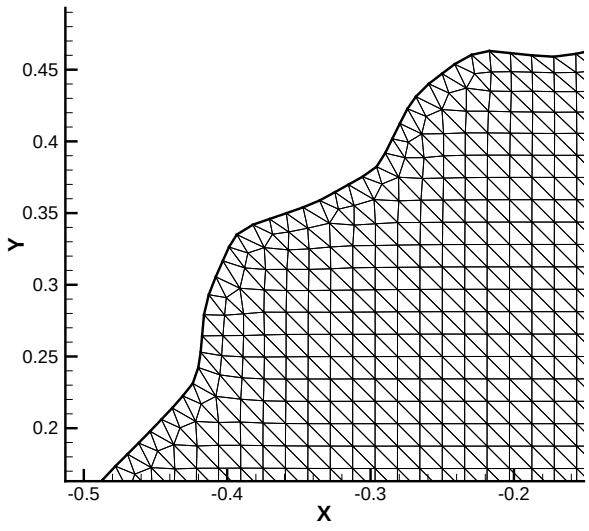

Figure 4.1: Detail of the unstructured computational mesh for a 128×128 mesh

The matrices for both methods are solved using methods in the PETSc [11] package. Here we use the BiCGSTAB method with the ILU method as preconditioner. We have tried different methods (such as LU, Cholesky, CG, GMRES, BiCGSTAB etc) with different preconditioners in the PETSc packages and find that the BiCGSTAD method with ILU as preconditioner is

the fastest for solving our matrices.

The first problem uses a simple boundary. The computational domain lies inside a perturbed circle as in Fig. 4.2.

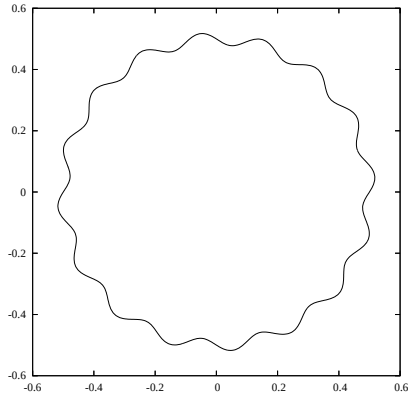

Figure 4.2: Boundary for the first test

Table 4.1 displays the errors and timing results for different mesh sizes. The convergence ratios with mesh refinement, the number of unknowns for the linear system, and the number of iterations for the linear solver are also listed. The maximum relative tolerance is $1e^{-9}$. The errors are measured by the L_2 norm of $\nabla\varphi$ defined by (4.2). From the table, we see that RT0 has only first order accuracy, EBM/BDM1/RT1 have 2nd order accuracy and BDM2 has 3rd order accuracy. The EBM is much faster than the other four methods when the same mesh size is used. The most apparent reason is that it has fewer unknowns than the other four methods. As expected, the RT0 is faster than BDM1/RT1/BDM2 since it has at most one half the number of the unknown variables. However, RT0 only has 1st order accuracy. Although BDM1/RT1 are both 2nd accurate method with the same number of unknowns, they have different characteristics in their timing and accuracy. The BDM1 is less accu-

Table 4.1: Convergence and Timing Study for the Boundary in Fig. 4.2

Mesh Size	EBM				
	error	ratio	time(second)	iterations	unknowns
64×64	1.593569e-04	N/A	1.697e-02	32	861
128×128	3.670301e-05	2.118	9.923e-02	60	3338
256×256	8.686625e-06	2.099	6.992e-01	116	13160
512×512	2.134996e-06	2.074	6.024e+00	242	52056
Mesh Size	RT0				
	error	ratio	time(second)	iterations	unknowns
64×64	1.011978e-03	N/A	1.934e-01	74	2642
128×128	5.121661e-04	0.982	8.366e-01	108	10141
256×256	2.651845e-04	0.950	5.723e+00	219	39751
512×512	1.353009e-04	0.971	4.234e+01	462	156715
Mesh Size	BDM1				
	error	ratio	time(second)	iterations	unknowns
64×64	1.329723e-04	N/A	4.758e-01	87	5286
128×128	3.715907e-05	1.839	3.132e+00	171	20284
256×256	9.794951e-06	1.924	1.928e+01	306	79504
512×512	2.538575e-06	1.948	1.410e+02	597	313432
Mesh Size	RT1				
	error	ratio	time(second)	iterations	unknowns
64×64	1.701312e-05	N/A	8.074e-01	87	5286
128×128	4.628462e-06	1.878	4.472e+00	172	20284
256×256	1.215990e-06	1.928	2.458e+01	305	79504
512×512	3.125208e-07	1.960	1.638e+02	607	313432
Mesh Size	BDM2				
	error	ratio	time(second)	iterations	unknowns
64×64	4.598191e-07	N/A	1.243e+00	100	7929
128×128	4.169450e-08	3.463	6.774e+00	191	30426
256×256	4.824547e-09	3.111	4.000e+01	317	119256
512×512	2.865473e-09	0.751	3.366e+02	756	470148

Table 4.2: Detailed timing of RT0 (unit: second)

Mesh Size	RT0		
	mesh	matrix setup/solve	interpolation
64×64	0.083814	1.0178e-01	4.258e-03
128×128	0.257736	5.4728e-01	1.665e-02
256×256	0.983702	4.6136e+00	7.303e-02
512×512	3.849851	3.7933e+01	3.436e-01

rate but faster for given mesh size. BDM2 has the highest order accuracy of all five methods. For the same order of accuracy, the fastest method is BDM2, then EBM/RT1/BDM1/RT0.

Table 4.2 gives the timing of the RT0 method for the mesh generation, the matrix setup/solve and the interpolation of the solution. Note that RT0/BDM1/RT1/BDM2 use the same mesh. Therefore their mesh generation time is the same. Their timing differences lie only in the matrix setup/solve step. Here we find that the time spent on generating the mesh is only a small part of the total time when the mesh size is large. Most of the time are spent solving the algebraic equation (timing for the matrix setup is comparable with that of the interpolation step). It is more apparent when the mesh size is increased. For example, the ratio of time spent on the matrix setup/solve step compared to the mesh generation step is about 1.21 when the 64×64 mesh is used. The same ratio increases to 9.85 when the 512×512 mesh is used.

In the following, we use the EBM and MFEM to solve the same problem but using a more complicated boundary as shown in Fig. 4.3. The errors are still measured by the L_2 norm of $\nabla\varphi$ defined by (4.2) and the max tolerance is $1e^{-9}$. The mesh is more refined in order to resolve the boundary. Table 4.3 shows the convergence and timing results of the five methods. The general

conclusion is the same as for the first test. The RT0 is 1st order accurate, the EBM/BDM1/RT1 method are 2nd order and BDM2 is 3rd order accurate in flux. The EBM method is still the fastest method for the same mesh size. For the same accuracy, we have BDM2, then EBM/RT1/BDM1/RT0 in decreasing order of speed.

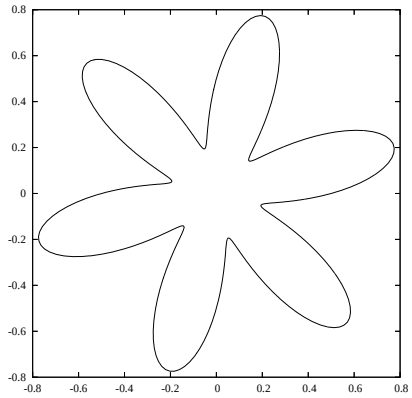

Figure 4.3: The boundary for the second test

Figs. 4.4, 4.5, 4.6, 4.7, 4.8 show the errors for $|\nabla\phi|_2$ using a 128^2 mesh for solving the boundary of Fig. 4.3.

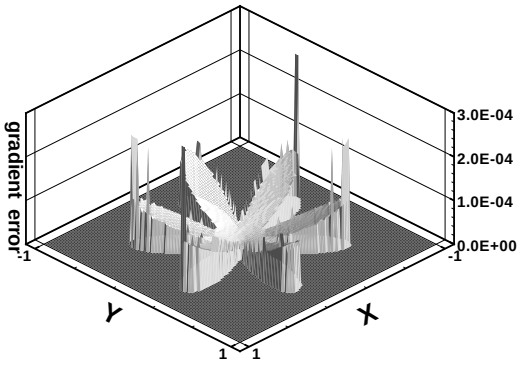

Figure 4.4: Norm of the gradient error by EBM using the 128×128 grid

Table 4.3: Convergence and Timing Study for the Boundary in Fig. 4.3

Mesh Size	EBM				
	error	ratio	time(second)	iterations	unknowns
64×64	2.110753e-04	N/A	2.232e-02	43	1008
128×128	5.779287e-05	1.869	1.641e-01	91	4008
256×256	1.472989e-05	1.920	1.439e+00	209	15738
512×512	3.641386e-06	1.952	1.040e+01	360	61967
Mesh Size	RT0				
	error	ratio	time(second)	iterations	unknowns
64×64	1.806532e-03	N/A	2.833e-01	115	3177
128×128	1.142133e-03	0.661	1.624e+00	218	12341
256×256	6.140341e-04	0.895	1.124e+01	415	47870
512×512	3.166839e-04	0.955	7.936e+01	770	187229
Mesh Size	BDM1				
	error	ratio	time(second)	iterations	unknowns
64×64	1.695040e-04	N/A	8.330e-01	151	6354
128×128	5.857866e-05	1.533	5.361e+00	278	24682
256×256	1.641488e-05	1.835	3.363e+01	461	95740
512×512	4.311759e-06	1.929	3.206e+02	1185	374458
Mesh Size	RT1				
	error	ratio	time(second)	iterations	unknowns
64×64	2.203143e-05	N/A	1.212e+00	151	6354
128×128	7.323467e-06	1.589	7.185e+00	296	24682
256×256	2.032626e-06	1.849	4.539e+01	549	95740
512×512	5.312876e-07	1.936	3.121e+02	1055	374458
Mesh Size	BDM2				
	error	ratio	time(second)	iterations	unknowns
64×64	6.651279e-07	N/A	1.917e+00	176	9531
128×128	7.259815e-08	3.196	1.219e+01	323	37023
256×256	9.293894e-09	2.966	9.087e+01	660	143610
512×512	3.087661e-09	1.590	6.074e+02	1162	561687

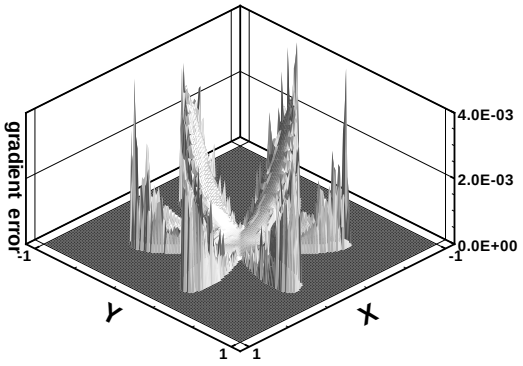

Figure 4.5: Norm of the gradient error by RT0 using the 128×128 grid

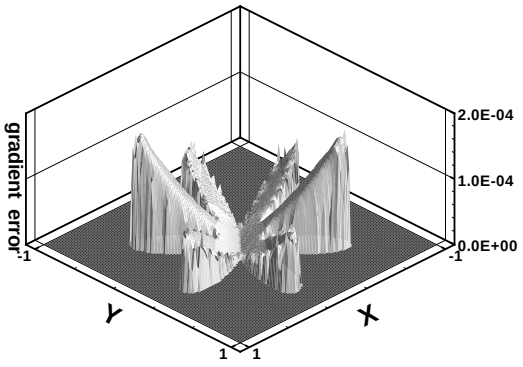

Figure 4.6: Norm of the gradient error by BDM1 using the 128×128 grid

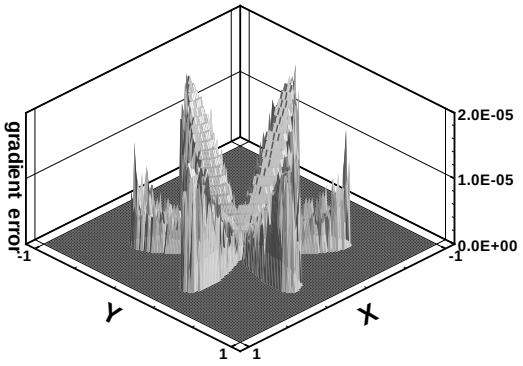

Figure 4.7: Norm of the gradient error by RT1 using the 128×128 grid

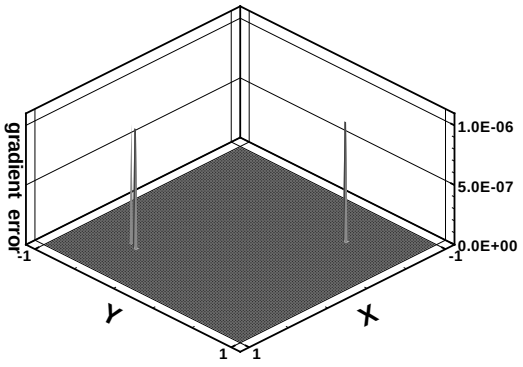

Figure 4.8: Norm of the gradient error by BDM2 using the 128×128 grid

Table 4.4: Maximum gradient errors on the boundary by different methods

Size	EBM	RT0	BDM1	RT1	BDM2
64×64	2.459720e-03	9.367834e-03	7.927028e-04	3.102073e-04	4.988912e-05
128×128	6.567893e-04	6.797467e-03	1.274594e-04	4.007023e-05	1.038527e-06
256×256	1.755489e-04	3.626596e-03	2.486447e-05	1.321007e-05	1.033665e-07
512×512	4.614643e-05	1.754357e-03	6.351849e-06	2.751578e-06	2.310828e-08

In Table 4.4, we show the maximum gradient errors on the boundary by different methods. From this table, we see that the order of accuracy of the maximum gradient errors on the boundary for the five methods are comparable to the L_2 norm on the whole domain.

4.3 Conclusion

In this chapter, we have used the embedded boundary method and the mixed finite element method to solve the elliptic boundary value problem in 2D. We compared convergence and timing results.

Since the embedded boundary method uses a structured cartesian grid, it is easier to implement. It is much harder to write the mesh generation program. But after the mesh is given, the discretization is simpler for the mixed finite element method. And it is easier to use the mixed finite element for the elliptic interface problem since the interface is in fact an internal boundary. However, the EBM method must be modified to solve an elliptic interface problem. To save computational resources when solving large problems, we could use EBM with automatic mesh refinement, which is one important part of our mesh generation method.

The EBM has the advantage of fewer unknowns with the same mesh size

compared with the MFM. There are two reasons for this. One reason is that the EBM uses a structured grid and the finite volume/central finite difference has super convergence in the mesh. The MFM uses an unstructured grid, and to achieve the same order of accuracy, a higher order basis function space is needed, which means more unknowns. The other reason is that the unknowns for EBM are cell centered and those for the MFM are edge centered. Since the approximate ratio of the vertices to faces to edges is 1:2:3 for a simple large triangle mesh, we know the ratio of the unknowns for the EBM, RT0, BDM1, RT1, BDM2 is approximately 1:3:6:6:9. Thus the EBM problem is smaller, which explains why it is much faster. However, for a given accuracy, the fastest method is BDM2 which is 3rd order accurate in flux, and then EBM/RT1/BDM1/RT0.

Chapter 5

A Ghost Fluid Method

The main idea of the embedded boundary method is to use the standard central finite difference method inside the computational domain while using the boundary condition to setup equation for the ghost cells and cells around the boundary. The interpolation is along the normal direction of the boundary.

Do we have a more simplified approach to do the interpolation? Can we do the interpolation along the x and y direction instead of the normal direction?

Yes, we can in some degree by using the ghost fluids method. The ghost fluids method was first proposed by Glimm, et al. [52]. Liu, et al. [48] used the ghost fluid method and level set method to solve the elliptic interface problem through the boundary condition capturing.

In this chapter, we will extend their idea and use the front tracking method instead of level set method. We investigate the convergence rate through examples.

We first show the formulation in 1D. Then it is straight forward to extend it to 2D since all we need to do is to use the formula in each direction independently.

5.1 Formulation

Using notation in [48], consider the 2D poisson equation 5.1. The interface jump condition can be $[u]_T = J_0(x_T)$ and $[\beta u_n]_T = J_1(x_T)$.

$$(\beta u_x)_x + (\beta u_y)_y = f(x, y) \tag{5.1}$$

Let $\vec{n} = (n^1, n^2)$ be the unit normal and $\vec{t}$ be the tangent. Let u_x, u_y, u_n, u_t be the derivatives along the $\vec{x}, \vec{y}, \vec{n}, \vec{t}$. We have the following identity from calculus [48]

$$u_n = u_x n^1 + u_y n^2, \tag{5.2}$$

$$u_t = u_x n^2 - u_y n^1, \tag{5.3}$$

and solving the above two equations gives

$$u_x = u_n n^1 + u_t n^2, \tag{5.4}$$

$$u_y = u_n n^2 - u_t n^1. \tag{5.5}$$

Multiplying the above equations by β and taking the jump across the interface, we have

$$[\beta u_n] = [\beta u_x n^1] + [\beta u_y n^2],$$

$$[\beta u_t] = [\beta u_x n^2] - [\beta u_y n^1],$$

$$[\beta u_x] = [\beta u_n n^1] + [\beta u_t n^2],$$

$$[\beta u_y] = [\beta u_n n^2] - [\beta u_t n^1],$$

which simply say that the jump along $\vec{n}, \vec{t}$ determine the jump along the $\vec{x}, \vec{y}$. Equivalently, the jump along $\vec{x}, \vec{y}$ can determine the jump along $\vec{n}, \vec{t}$.

Liu, etc., [48] assumes that the jump along the tangent $\vec{t}$, $[\beta u_t]_T$, is zero and then extend their ghost fluid method to 2,3 dimension using level set to

capture the jump. We will use their approach here. By doing this, the jump along the normal direction is satisfied while the jump along the tangential direction is smeared.

5.1.1 1D ghost fluid method

For simplicity, assume that we have a uniform grid. We assume that coefficient β is constant in each component. The variable u is defined on the grid point. We use standard 3 points central finite difference method to discretize equation 5.6 inside the computational domain without interface. The difficulty lies in how to discretize the equation where there is an interface crossing between two grid points. Fig. 5.1.1 shows such a case, where there are three grid points x_{-1}, x_0, x_1. Point x_{-1}, x_0 has the same component which is different from that at point x_1. The interface crosses the segments between x_0 and x_1 at T. We assume we know the distance between x_0 and T.

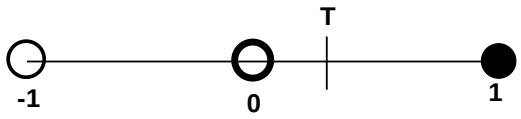

Figure 5.1: 1D mesh with two components

1st order ghost fluid method

For the 1st order ghost fluid method, we only need two point. In Fig. 5.1.1, we need point x_0 and x_1. Denote the u_{-1}, u_0 as the values defined at point x_{-1}, x_0 and U_1 as the value at point x_1. If the three points x_{-1}, x_0, x_1 has the same components, the standard central finite difference for 5.6 is

$$(\beta u_x)_x = f(x) \tag{5.6}$$

$$\frac{\beta\frac{u_1-u_0}{dx} - \beta\frac{u_0-u_{-1}}{dx}}{dx} = f(x), \tag{5.7}$$

or

$$\beta\frac{u_1-u_0}{dx^2} + \beta\frac{u_{-1}-u_0}{dx^2} = f(x), \tag{5.8}$$

where we have denoted U_1 as u_1. Thus we only need to consider the discretization of the "flux" βu_x at the center of two grid point $x_{-1/2}$ or $x_{1/2}$. This greatly simplify the formulation. Since they are symmetric, we only consider one of them at $x_{1/2}$.

If there is a interface between point x_0, x_1 as in Fig. 5.1.1, we can not directly use central finite difference because of the jump conditions

$$[u]_T = J_0(x_T)$$

and

$$[\beta u]_T = J_1(x_T)$$

. We need two ghost fluids, u_1 for fluid from the left, and U_0 for the fluid from the right. Now, we can use the four values u_0, u_1, U_0, U_1 to setup equations so that they satisfy the jump conditions to first order. First we use linear interpolation $u_T = (1-\theta)u_0 + \theta u_1$ and $U_T = ((1-\theta)U_0 + \theta U_1)$ to approximate the value at the interface. u_T, U_T have to satisfy the first jump condition $U_T - u_T = J_0$. This is the first equation 5.9. Then we can use u_T, u_0 to approximate the left derivative $u_x|_T = \beta_0 \frac{u_T - u_0}{\theta dx}$ and similarly for the right side derivative $U_x|_T = \beta_1 \frac{U_1 - U_T}{(1-\theta)dx}$. They have to satisfy the second jump $U_x|_T - u_x|_T = J_1$ which gives the second equation 5.10.

$$((1-\theta)U_0 + \theta U_1) - ((1-\theta)u_0 + \theta u_1) = J_0, \tag{5.9}$$

$$\beta_1 \frac{U_1 - ((1-\theta)U_0 + \theta U_1)}{(1-\theta)dx} - \beta_0 \frac{((1-\theta)u_0 + \theta u_1) - u_0}{\theta dx} = J_1, \tag{5.10}$$

where θdx is the distance between x_0 and T. Solving the two equations for the ghost fluids u_1, U_0, we have

$$u_1 = c_0 u_0 + c_1 U_1 + rhs,$$

where

$$c_0 = \frac{(-1+\theta)(\beta_0 - \beta_1)}{(-1+\theta)\beta_0 - \theta\beta_1} u_0,$$

$$c_1 = \frac{-\beta_1}{(-1+\theta)\beta_0 - \theta\beta_1},$$

$$rhs = \frac{(dx - dx\theta)J_1 + J_0\beta_1}{(-1+\theta)\beta_0 - \theta\beta_1}.$$

Note that the ghost fluids are a linear combination of the values u_0, U_1

and the jump J_0, J_1.

The above formulation is first order because linear interpolation are used. Although the formulas seem to be complex, it is very easy to derive them using some symbolic software, such as Mathematica.

2rd order ghost fluid method

We can also use quadratic interpolation to approximate values u_T, U_T at the interface. Now we need four points instead of 2 points, x_{-1}, x_0, x_1, x_2. Thus we have four values u_{-1}, u_0, U_1, U_2 and two ghost fluids u_1, U_0. Since the formula derivation can be automated by using Mathematica, we only give the Mathematica code in the following.

First, we find the quadratic function using u_{-1}, u_0, u_1.

$$T[x\_] := ax^2 + bx + c;$$

$$eq0 = T[-\Delta] == u_{-1};$$

$$eq1 = T[0] == u_0;$$

$$eq2 = T[\Delta] == u_1;$$

$$sol = Solve[eq0, eq1, eq2, a, b, c];$$

$$f[x\_] = T[x]/.sol[[1]];$$

$$df[x\_] = D[T[x], x]/.sol[[1]];$$

Next, we find the quadratic function using U_0, U_1, U_2.

$$S[x\_] := ax^2 + bx + c;$$

$$eq0 = S[0] == U_0;$$

$$eq1 = S[\Delta] == U_1;$$

$$eq2 = S[2\Delta] == U_2;$$

$$sol = Solve[eq0, eq1, eq2, a, b, c];$$

$$F[x\_] = S[x]/.sol[[1]];$$

$$dF[x\_] = D[S[x], x]/.sol[[1]];$$

Finally, we can set up the two equations for the jump conditions.

$$eq0 = F[\theta\Delta] - f[\theta\Delta] == J_0;$$

$$eq1 = dF[\theta\Delta] - df[\theta\Delta] == J_1;$$

$$sol = Solve[eq0, eq1, u_1, U_0];$$

We have the following relation for the ghost fluid u_1 as the linear combination of u_{-1}, u_0, U_1, U_2.

$$u_{-1} = c_{-1}u_{-1} + c_0u_0 + c_1U_1 + c_2U_2 + rhs,$$

where

$$c_{-1} = -\frac{(-1+\theta)((2-5\theta+2\theta^2)\beta_0 + (3-2\theta)\theta\beta_1)}{(2+\theta-5\theta^2+2\theta^3)\beta_0 + \theta(3+\theta-2\theta^2)\beta_1},$$

$$c_0 = \frac{2(-1+\theta)(2(-2+\theta)\theta\beta_0 + (3+\theta-2\theta^2)\beta_1)}{(2+\theta-5\theta^2+2\theta^3)\beta_0 + \theta(3+\theta-2\theta^2)\beta_1},$$

$$c_1 = \frac{2(-2+\theta)^2\beta_1}{(2+\theta-5\theta^2+2\theta^3)\beta_0 + \theta(3+\theta-2\theta^2)\beta_1},$$

$$c_2 = \frac{2(-1+\theta)^2\beta_1}{(2+\theta-5\theta^2+2\theta^3)\beta_0 + \theta(3+\theta-2\theta^2)\beta_1},$$

$$rhs = \frac{-2\Delta(2-3\theta+\theta^2)J_1 + 2(-3+2\theta)J_0\beta_1}{(2+\theta-5\theta^2+2\theta^3)\beta_0 + \theta(3+\theta-2\theta^2)\beta_1}.$$

The Discretization

After we have the formula for the ghost fluids, we can easily discretize the elliptic equation. If we want to use the 1st order method to discretize equation (5.8) at point x_0, we have the ghost fluid $u_1 = c_0 u_0 + c_1 U_1 + rhs$. Therefore, the flux at point $x_{1/2}$ is

$$\beta_0 \frac{u_1 - u_0}{dx^2} = \beta_0 \frac{(c_0 - 1)u_0 + c_1 U_1 + rhs}{dx^2} \tag{5.11}$$

The flux at point $x_{-1/2}$ doesn't need to change.

If we need to discretize at point x_1, we have the ghost fluid formula $U_0 = C_0 u_0 + C_1 U_1 + Rhs$ and

$$\beta_1 \frac{U_0 - U_1}{dx^2} = \beta_1 \frac{C_0 u_0 + (C_1 - 1)U_1 + Rhs}{dx^2}, \tag{5.12}$$

where the coefficients C_0, C_1, Rhs are calculated from the 1st order ghost fluid

formula considering point x_1, x_0 here as point x_0, x_1.

We can also use the 2rd order ghost fluid formula in the similar way. However, it must be noted that in order to use the 2rd order ghost fluid formula, the components for x_{-1}, x_0 must be the same and components for x_1, x_2 must also be the same. Otherwise, we need to switch back to the 1st order ghost fluid formula which does not have such restriction.

It is important to be careful how to choose the coefficient β in the discretization. If the discretization position is x_0, we must use β_0 from the the same component. If the position is x_1, we need to use β_1 instead.

5.1.2 2D ghost fluid method

The best part of the ghost fluid method is that we can extend the 1D formula straight forwardly to 2D (or 3D). In 2D, if the interface has derivative a jump in the normal (and tangential) directions, we can calculate the jump in $\vec{x}, \vec{y}$ direction. Then we discretize the derivative for x, y separately. Thus 2D (or 3D) problem is reduced to 1D problem.

5.2 Example

Now we show a 2D example. The computational domain is $[-1, 1] \times [-1, 1]$. There is a circle with radius 0.3 and center (0,0). Inside the circle, $\beta = 1$, and outside the circle, $\beta = 0.2$. Let the analytical solution be $u(x, y) = e^{\frac{x^2+y^2}{2}}$ inside the circle and $u(x, y) = sin(x^2+y^2)$ outside the circle. Substitute the solution into the equation to get the right hand function $f(x, y)$. The jump

conditions J_0, J_1 at the interface is calculated using the exact solution. The boundary condition is also calculated using the exact solution. Fig. 5.2 shows the approximate solution using the 2nd order accurate ghost fluid method. From the figure, it is clear that the solution has very accurate jump profile.

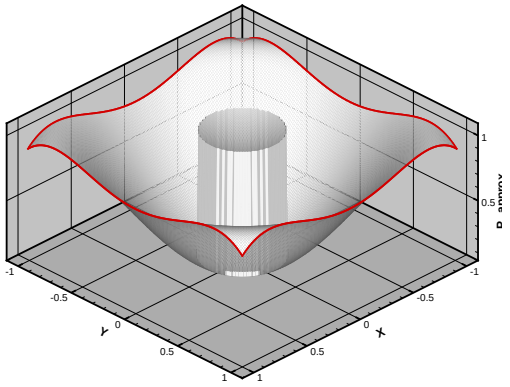

Figure 5.2: a 2D solution calculated using the 2rd order ghost fluid method

Fig.5.2 shows the approximate x derivative. We also have a very accurate gradient profile.

It should be noted that in order to calculate the approximate derivatives, we use central finite difference and the ghost fluid formula if needed. If the 2rd order ghost fluid formula is used to discretize the PDE, we also need to use the 2rd order ghost fluid formula when calculating the approximate derivatives after solving the equations.

In Table 5.2, we show the convergence rate of the two ghost fluid method. We use the max norm for the potential u and L_2 norm for the gradient as in previous chapter.

However, it seems that the ghost fluid method does not always give good

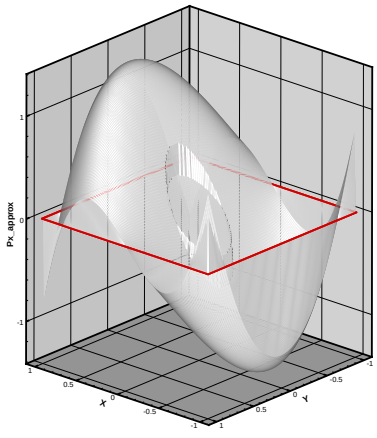

Figure 5.3: approximate x derivative calculated using the 2rd order ghost fluid method

Table 5.1: Convergence Study for the Ghost Fluid method

Mesh Size	1st Ghost Fluid Method			
	potential error	ratio	gradient error	ratio
10×10	7.834047e-02	N/A	7.804623e-02	N/A
20×20	3.288107e-02	1.25	8.119717e-02	-0.06
40×40	6.322738e-03	2.38	1.100779e-02	2.88
80×80	3.933982e-03	0.69	6.429411e-03	0.78
160×160	1.442078e-03	1.45	2.636914e-03	1.29
320×320	4.253379e-04	1.76	8.321577e-04	1.66
640×640	1.254099e-04	1.76	3.055100e-04	1.45
Mesh	2rd Ghost Fluid Method			
Size	potential error	ratio	gradient error	ratio
10×10	3.406400e-02	N/A	7.158676e-02	N/A
20×20	7.255979e-03	2.23	1.921226e-02	1.90
40×40	1.815689e-03	2.00	4.702095e-03	2.03
80×80	4.427846e-04	2.04	1.182949e-03	1.99
160×160	1.091516e-04	2.02	2.968873e-04	1.99
320×320	2.744184e-05	1.99	7.450425e-05	1.99
640×640	6.682062e-06	2.04	1.867109e-05	2.00

Table 5.2: The Second Convergence Study for the Ghost Fluid method

Mesh Size	1st Ghost Fluid Method			
	potential error	ratio	gradient error	ratio
10×10	1.422432e-01	N/A	2.567303e-01	N/A
20×20	5.787385e-02	1.30	2.092508e-01	0.30
40×40	1.505660e-02	1.94	1.231668e-01	0.77
80×80	9.784978e-03	0.62	8.861856e-02	0.48
160×160	3.700755e-03	1.40	6.228199e-02	0.51
320×320	2.252100e-03	0.72	4.295575e-02	0.54
Mesh	2rd Ghost Fluid Method			
Size	potential error	ratio	gradient error	ratio
10×10	6.255407e-02	N/A	2.510799e-01	N/A
20×20	2.485743e-02	1.33	7.830582e-01	-1.64
40×40	1.026261e-02	1.28	1.279421e-01	2.61
80×80	7.590548e-03	0.44	1.062941e-01	0.27
160×160	7.330523e-03	0.05	8.587813e-02	0.31
320×320	7.305210e-03	0.05	5.668032e-02	0.60

results. The previous example has a solution which is symmetric around the (0,0). In the following test, we still use the same domain and interface. The only difference is that we change the analytical solution to be $u(x,y) = e^{\frac{(x-1)^2+y^2}{2}}$ inside the circle and $u(x,y) = sin(x^2 + (y-1)^2)$ outside the circle. The convergence rates are shown in Table 5.2. We can see that the methods have difficulty in converging.

Considering that it is very simple to implement the ghost fluid method, we conclude that the ghost fluid method is much better than the standard central finite difference method which might not even converge.

It should be noted that, our ghost fluid method is closely connected with the Matched Interface and Boundary method (MIB) by Zhou, et al. [53]. Our ghost fluid method captures only the gradient jump in the normal direction. The MIB method transforms the normal and tangential jump into jumps in

the x and y coordinates. Then it use interpolation and jump conditions to setup equation for the ghost fluids. While our ghost fluid method uses the gradient jump either in the x or y direction only.

Although the MIB method can be very high order accuracy, the disadvantage of the MIB method is that in order to obtain high order accuracy, a large stencil is required. This means that it is very hard to solve complex interface problem without mesh refinement.

Chapter 6

Incompressible Flow

In this section, we do a simple 2D Rayleigh-Taylor instability test using the FronTier-Lite package [10]. The Frontier-Lite package is extracted from the FronTier code. As a geometry package for interface tracking, it can provide the component and the shortest interface point of a given non interface point, the interface normal at the crossing between two points with different components. It can also change the interface dynamically using given velocity field.

We use the projection method for the incompressible flow [35–37]. We use the approach by Tryggvason, et al . [8, 9] and Kang [54] to smooth the discontinuous coefficient using the Heaviside function. Namely, for discontinuous coefficient, for example, the density, we smooth them by using [48]

$$\rho(r) = r^- + (r^+ - r^-)H(r)$$

where

$$H(r) = \begin{cases} 0, & r < -\varepsilon \\ \frac{1}{2} + \frac{r}{2\varepsilon} + \frac{1}{2\pi} sin(\frac{\pi r}{\varepsilon}), & -\varepsilon \le r \le \varepsilon \\ 1, & \varepsilon < r \end{cases} \tag{6.1}$$

ε is a given constant in the order of the grid spacing.

For simplicity, we assume constant density in each component. The Navier-Stokes equation has the following form

$$\nabla \cdot u = 0$$

$$u_t + (\vec{V} \cdot \nabla)u + \frac{p_x}{\rho} = \frac{(2\mu u_x)_x + (\mu(u_y + v_x))_y}{\rho}$$

$$v_t + (\vec{V} \cdot \nabla)v + \frac{p_y}{\rho} = \frac{(\mu(u_y + v_x))_x + (2\mu(v_y))_y}{\rho} + g$$

The algorithm we used is the following.

Step 1. Advance the interface using the old velocity, return the time step dt used. This step is needed in the first place because it often happens that the interface has to reduce the given time step to advance properly.

Step 2. Set up the components and smooth the density and viscosity using the Heaviside function.

Step 3. Solve the advection equation. We simply used the central finite difference method for the advection equation. See also [9].

Step 4. Solve the diffusion equation using the Crank-Nicolson method. This method can obtain stable solution with bigger time step. Although the Adams-Bashford integration scheme is preferred by many authors [9], we do not use it here. The reason is that we can not guarantee that the interface can be advanced using a given time step, especially when the interface is complex.

Step 5. Compute the Projection by solving the pressure equation. Proper boundary condition for the pressure should be used. Here we simply used

$$\frac{\partial p}{\partial n} = 0$$

Step 6. Update the New velocity using the pressure gradient.

Now we give the results for a 2D Rayleigh-Taylor instability test. Our computational domain is $[0, 1] \times [0, 4]$. A sine wave interface is used as the initial interface as fig. 6. For simplicity, we have used $\mu = \pi/50$ for both fluids.

The density is 1 for the upper fluid and 0.1 for the other fluid. The force is set as $g = -100$ to speed up the simulation. Fig. 6 shows the interface around time 0.51 using a grid size 50×200. Fig. 6 shows the interface calculated using a grid size 100×400.

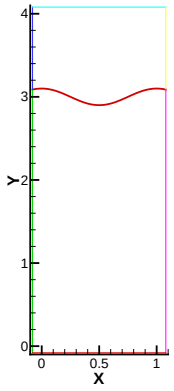

Figure 6.1: The initial interface is chosen as a sine wave

From the two figs, we have similar interface profile under mesh refinement for the RT instability using the FronTier-Lite packages and the projection method.

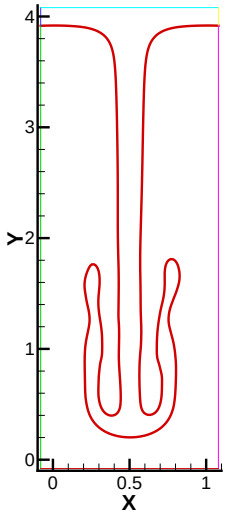

Figure 6.2: The interface is calculated using grid size 50×200 around time 0.51

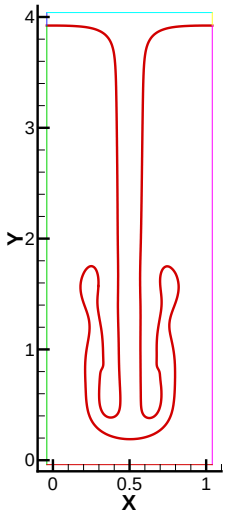

Figure 6.3: The interface is calculated using grid size 100×400 around time 0.51

Appendix A

Coding Issues

In this section, we briefly show our coding method. We write the program in C++ and used several techniques which are not available in the C language: virtual functions, templates and the Standard Template Library. After that, we discuss the structure of the code for mesh generation and the finite element discretization.

A.1 Virtual Function, Template and Standard Template Library

We write our program in C++. Although C++ might be slower than Fortran or C, it is an object oriented programming (OOP) language and code written in C++ is much easier to be read and reused. With new C++ compilers and careful implementation, codes written in C++ are not much slower than those written in Fortran/C.

We use mainly two C++ techniques which are not available in C: the virtual function and template.

A.1.1 Virtual Function

A C++ virtual function is a special member function of a base class. It could be over-ridden in its derived classes. We will show its benefit through the following example. The FronTier code uses function pointers instead of virtual functions.

In our code, we frequently need to solve algebraic equations. There are a lot of packages which are publicly available. We might even have our own

code. More often than not, they have different invoking interface. It is often the case that in the begin, we are not sure which package to use. Thus we write a base class called SOLVER. A simplified version of the class SOLVER is shown in Table A.1.

Table A.1: The definition of a simplified class SOLVER

```
class SOLVER{
public:
    SOLVER();
    virtual Create(int ilower, int iuppter, int d_nz, int o_nz);
    virtual void Set_A(int i, int j, double val); // A[i][j] = val;
    virtual void Set_b(int i, double val); // b[i] = val;
    virtual void Get_x(double *x); // x=inv(A)b

    virtual void Solve(void);
}
```

Then we derive classes from SOLVER for a Gaussian elimination method, PETSc [11], Hypre [12], and LAPACK [13]. Then when defining class for the finite element (FEM) discretization, we define the class FEM as in Table A.2.

Table A.2: The definition of a simplified class FEM

```
class FEM{
SOLVER *m_solver;
.....
}
```

The member functions in the FEM function properly without knowing which package it is calling. Thus when we want to use a different package to solve the algebraic equations, we only need to implement a derived class

from SOLVER and then pass the pointer to FEM without any modification in the finite element code. In this way, the independent parts of a big code are interconnected in the minimal way.

A.1.2 Template and Standard Template Library

A template is like a mold for new code. After a template code for an abstract object is written, similar code for other objects could be generated automatically. We used several structures in the Standard Template Library of the C++ language: vector, link, priority_queue, map and the sort algorithm. It is true that we could write our own generic code for these standard structures and algorithm without using templates. But we need the same structure for different objects. Without a template, we have to write different codes for different objects. It is just a waste time and when the code for one object is modified, codes for other objects have to modified also. It would not be easy to maintain the code.

A.2 Code Structure

For our mesh generation and finite element code, we have written several classes: FRONTIER2D/FRONTIER3D giving the data for the boundary of the computational domain, QUADTREE/OCTREE for generating the QUADTREE/OCTREE structure, MESH2D/MESH3D for the housekeeping of the mesh data structures and FEM2D/FEM3D for the finite element methods, SOLVER for solving algebraic equations. FRONTIER2D/FRONTIER3D,

QUADTREE/OCTREE, FEM2D/FEM3D and SOLVER all have derived classes giving different implementation. For example, the mixed finite elements using RT0, RT1, BDM1, BDM2 are implemented as derived classes from FEM2D. Hypre, Hypre_GMRES, PETSc, GAUSSIAN are derived classed from SOLVER for different packages or methods for solving matrix.

Their relation are the following (for 2D code, 3D code is similar).

$$FRONTIER2D \leftarrow QUADTREE \Leftrightarrow MESH2D \leftarrow FEM2D \rightarrow SOLVER$$

where $A \leftarrow B$ means that class B has a pointer pointing to class A. QUADTREE uses information from FRONTIER2D to generate the quadtree and then generate the mesh to store inside MESH2D. QUADTREE helps MESH2D to start point location. FEM2D needs the mesh from MESH2D and uses SOLVER to solve the algebraic equation.

Appendix B

A Compact Finite Difference Method for Curvature Calculation

B.1 Introduction

The compact difference method [55] is a high order finite difference method. We first need a grid to use the finite difference. For the interval [0,1], define $x_i = i \times h, i = 0, 1, ..., N$, where $h = 1/(N-1)$ is the grid length and N is the number of grid points. Then for a function $f(x)$, its second order of derivative with x can be computed by the following equation:

$$f_i^{''} = \frac{a_1 f_{i-1} + a_2 f_i + a_3 f_{i+1}}{h^2}, \tag{B.1}$$

where a_1, a_2 and a_3 are undetermined parameters and $f_i = f(x_i)$ and $f^{''} = f_{xx}$. Using Taylor expansion of $f(x)$ at the point x_i, the highest order of accuracy (B.1) could have is second order by the central difference method:

$$f_i = \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2} + O(h^2). \tag{B.2}$$

The compact finite difference [55] for the second order of derivative with x using only three point stencil could be written as:

$$c_1 f_{i-1}^{''} + f_i^{''} + c_2 f_{i+1}^{''} = \frac{a_1 f_{i-1} + a_2 f_i + a_3 f_{i+1}}{h^2}, \tag{B.3}$$

where c_1, c_2 and a_1, a_2, a_3 are undetermined parameters. Using Taylor expansion, the highest order of accuracy B.3 could have is given by the following 4th order scheme:

$$\frac{1}{10} f_{i-1}^{''} + f_i^{''} + \frac{1}{10} f_{i+1}^{''} = \frac{\frac{6}{5} f_{i-1} - \frac{12}{5} f_i + \frac{6}{5} f_{i+1}}{h^2} + O(h^4). \tag{B.4}$$

It is worth noting that the function constructed by the cubic spline also satisfies a similar relation. The compact finite difference for the first order of derivative with x can also be derived in the same ways [55].

B.2 Compact finite difference for surface tension

Now we use the compact finite difference method to approximate the curvature of a curve defined by a finite number of connected points. Our method is similar to the method by Fyfe, etc [56] who used a cubic spline function to approximate the curvature. We assume that the curve is closed and planar. The extension to 3D curve is straight forward. One of the reasons that we are interested in the approximation of the curvature is that the surface tension of an interface between two fluid phases could be written as

$$\gamma = \sigma\kappa, \tag{B.5}$$

where σ is a known coefficient and κ is the curvature. Thus the approximation of the surface tension becomes the approximation of curvature.

From differential geometry, we know that

$$\vec{r}(s)_{ss} = \kappa\vec{\beta}, \tag{B.6}$$

where s is arclength, $\vec{r}(s)$ is a position vector for the curve, κ is the curvature and $\vec{\beta}$ is the unit normal vector perpendicular to the tangent of the curve. We only need to approximate $\vec{r}(s)_{ss}$, from which we obtain the curvature

$\kappa = \|\vec{r}(s)_{ss}\|$ and the normal $\vec{\beta} = \frac{\vec{r}(s)_{ss}}{\kappa}$. Since $\vec{r}(s)$ is a vector function, let us denote it as $\vec{r}(s) = (r_1(s), r_2(s))$. Thus, in order to approximate $\vec{r}(s)_{ss}$, we need to approximate the second derivative with respect to s of two scalar functions $r_1(s)$ and $r_2(s)$. Considering that the grid length is often non-uniform, we can not use equation (B.4) directly. Instead we use the following compact scheme on a non-uniform grid:

$$c_1 f''(x_i - h) + f''(x_i) + c_2 f''(i + \alpha h) = \frac{a_1 f(x_i - h) + a_2 f(x_i) + a_3 f(x_i + \alpha h)}{h^2}, \tag{B.7}$$

By using Taylor expansion, we get a 3rd order accurate scheme for f'' with

$$c_1 = -\frac{-\alpha - \alpha^2 + \alpha^3}{(1+\alpha)(1+3\alpha+\alpha^2)},$$

$$c_2 = -\frac{1 - \alpha - \alpha^2}{(1+\alpha)(1+3\alpha+\alpha^2)},$$

$$a_1 = \frac{12\alpha}{1 + 4\alpha + 4\alpha^2 + \alpha^3},$$

$$a_2 = -\frac{12}{1 + 3\alpha + \alpha^2},$$

$$a_3 = \frac{12}{1 + 4\alpha + 4\alpha^2 + \alpha^3}.$$

B.2.1 First algorithm

Our first algorithm is the following. Approximating the arc length by the length of the segments connecting two consecutive points and using the formula (B.7), we set up two systems of equations for the unknowns: $r_1''(s_i)$

and $r_2^{''}(s_i)$. The two matrices for the two systems of equations are the same and only the right hand sides are different.

B.2.2 Second algorithm

The above approach gives the curvature with 2nd order accuracy since the arc length is approximated by a linear function. We will also show this by an example in the next section. If 2nd order accuracy is not sufficient, we improve the approximation of the arc length and then use the more accurate arc length in equation (B.7). The new arc length S_i from the i^{th} point to the $(i+1)^th$ point on the curve can be approximated by a cubic polynomial curve $(f(t), g(t))$, where $f(t)$ is determined by the following four equations:

$$f(0) = r_1(s_i),$$

$$f(s_i) = r_1(s_{i+1}),$$

$$f^{''}(0) = r_1^{''}(s_i),$$

$$f^{''}(s_i) = r_1^{''}(s_{i+1}),$$

and $g(t)$ is determined similarly. Also $f^{'}(t)$ is given by:

$$f^{'}(t) = -\frac{(6f(0) - 6f(s_i) + 2s_i^2 f^{''}(0) - 6s_i t f^{''}(0) + s_i^2 f^{''}(s_i) + 3t^2(f^{''}(0) - f^{''}(s_i)))}{6s_i}. \tag{B.8}$$

Now the new arc length can be integrated numerically using the expres-

sion

$$s = \int_0^{s_i} \sqrt{f'(t)^2 + g'(t)^2} dt. \tag{B.9}$$

If needed, the above process can be iterated to obtain a more accurate approximation for the arc length as in [56].

B.3 Example

Now we use our scheme on a simple test problem. Our test problem is a circle with radius 1 and its center at (0,0). The grid points on the circle are generated by random perturbation of an uniform grid. The sample code is given in Table B.1.

Table B.1: The code for generating the grid points on the circle

```
// the following is written in C language
// npoints is the number of points on the circle
dtheta = 2*PI/(npoints);
for(i=0; i¡npoints; i++)
{
/* random number from [0,1]*/
tmp = ((double)rand())/RAND_MAX;
/* the ith points lies on */
/* the circle between angle [(i-0.25)*dtheta, (i+0.25)*dtheta]*/
alpha = i*dtheta + (tmp-0.5)*dtheta*1.0/2;

P[i][0] = 0 + r*cos(alpha);
P[i][1] = 0 + r*sin(alpha);
}
```

The Table B.2 gives curvature errors with different number of grid points for a particular computation. Since the grid points are generated randomly,

each computation with a fixed number of points gives different results.

Table B.2: Maximum curvature errors

	maximum curvature errors	
number of points	max err using 1^{st} method	max err using 2^{nd} method
4	2.375e-01	1.025e-1
8	6.350e-02	1.030e-2
16	1.873e-02	8.118e-4
32	5.355e-03	7.387e-5
64	1.540e-03	3.876e-6
128	3.635e-04	2.810e-7
256	1.005e-04	1.968e-8
512	2.333e-05	1.676e-9
1024	6.219e-06	2.114e-10

From the Table B.2 we can see that the curvature approximation for the first method is about 2nd order accurate as having been predicted. The second method is at least 3rd order accurate.

The compact finite difference method is very simple to use. However, it can not be easily extended to 2D functions as is needed to define surface curvature. Instead we are considering using the Bezier triangles (Bezier splines on triangles) to approximate the curvature of triangulated surfaces.

Appendix C

Nonreflecting Boundary Conditions for an Elliptic Problem

C.1 Introduction

In this section, we present an algorithm for solving an elliptic problem on a rectangle domain without given boundary condition. Our problem is the following equation for gravitational potential:

$$\Delta V(x) = 4\pi G\rho(x), x \in \Omega, \tag{C.1}$$

where Δ is the Laplacian operator, $V(x)$ is the gravitational potential, G is a gravitational constant, and $\rho(x)$ is the mass density. The computational domain Ω is a finite domain. Thus it is assumed that there are mass only inside Ω. The specialty of this problem is that no boundary condition is given.

If the right hand side of equation (C.1) is zero, $V(x)$ is radially symmetric and the general solution ([50]) is

$$V(x) = \begin{cases} c_1 + c_2 \log(r), \text{2D}, \\ c_1 + c_2 \frac{1}{r}, \text{3D}. \end{cases} \tag{C.2}$$

It is well known (for example, see [57]) that by using a fundamental solution $K(x)$ which satisfies:

$$\Delta K(x) = \delta(x), \tag{C.3}$$

$V(x)$, the solution to (C.1), could be written as:

$$V(x) = \int_\Omega K(x-\xi)\Delta V(\xi)d\xi = 4\pi G \int_\Omega K(\xi)\rho(\xi)d\xi. \tag{C.4}$$

Noting the equation (C.3) is radially symmetric, it is easy to find the fundamental solution (see [50]):

$$K(x) = \begin{cases} \frac{1}{2\pi}\log(r), \text{2D}, \\ \frac{1}{4\pi r}, \text{3D}. \end{cases} \tag{C.5}$$

There are several ways for solving (C.1). The first method is to use a very large domain Φ to cover Ω, for example a rectangle in 2D or a box in 3D. Since generally $V(r) = O(\frac{1}{r})$ (r is the distance of x to the center of Ω), we know that if the boundary of Φ is large enough, $V(r)$ is approximately zero. Thus we could approximate the value of $V(x)$ at the boundary of Φ by 0. Then we solve a finite difference approximation to the Laplace operator with these zero boundary conditions on the domain Φ. The advantage of such an approach is that it is easy to program. The disadvantage is that since $V(x)$ goes to zero slowly in the order of $\frac{1}{r}$, a large domain is needed if high accuracy is needed, leading to many unknown variables.

The second method is to solve (C.1) using directly the formula (C.4). Thus for any point $x \in \Omega$, we obtain the solution $V(x)$ by quadrature. However, since the integral is a volume integration, if there are N grid points inside the computational domain Ω where we need to evaluate $V(x)$, we need approximately $O(N^2)$ computation. Thus the brute force approach is not applicable, especially when the problem size is large. The fast multipole method [50] could be used to reduce the computation to $O(N)$. However, the corresponding constant is very large.

The third method is to use a nonreflecting boundary condition on the

boundary of Ω [58, 59], where generally the boundary condition consists of a local or non-local operator.

Our method combines the second and third approach. The idea is simple. We first use the formula (C.4) to compute the value of $V(x, y)$ on the boundary (which could be regarded as a nonreflecting boundary condition) and then use a finite difference method with this boundary condition.

In the following, we first give our method and then give the discussion.

C.2 Method

In this section, we would explain our method using a 3D cylindrical symmetric solution as example. In a cylindrical coordinate system, with (r, ϕ, z) as coordinates, the Laplacian operator is written as:

$$\Delta = \frac{1}{r}\frac{\partial}{\partial r}(r\frac{\partial}{\partial r}) + \frac{1}{r^2}\frac{\partial^2}{\partial \phi^2} + \frac{\partial^2}{\partial z}. \tag{C.6}$$

When the solution is cylindrical symmetric, the equation (C.1) becomes

$$\frac{1}{r}\frac{\partial}{\partial r}(r\frac{\partial V(r,z)}{\partial r}) + \frac{\partial^2 V(r,z)}{\partial z} = 4\pi G\rho(r,z). \tag{C.7}$$

For simplicity, we assume the computation domain is a rectangle for (r, z), $[0, 1] \times [0, 1]$. We partition the domain into a uniform grid as shown in Figure C.1. Let δr and δz be the width of the grid. Denote $r_i = i \times \delta r$, $z_i = i \times \delta z$ and $f_{i,j}$ the value of $f(r_i, z_j)$.

Our method consists of the following three steps:

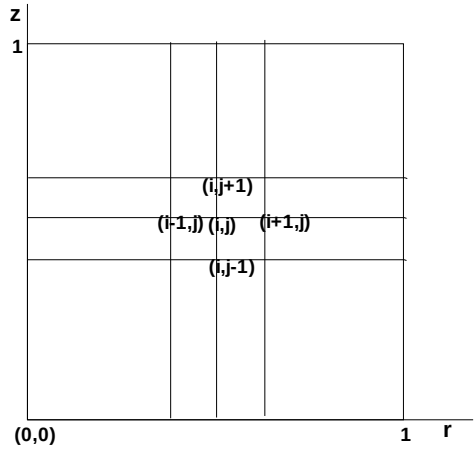

Figure C.1: computational domain, grid and the stencil for the finite difference method

1. compute the artificial boundary condition by using formula (C.4);

2. discretize the equation (C.7) using a central finite difference method (or any other appropriate method, for example, finite elements);

3. solve the matrix and get the solution $V(x)$ inside Ω; if the gradient of $V(x)$ is needed, we simply differentiate $V(x)$ to obtain it.

When using the finite difference methods (FD), the variables are either cell-centered or vertex-centered. The main difference between these methods lies near the boundary. When cell-centered FD is used, the discretization of the boundary conditions (both Dirichlet and Neumann boundary conditions) require ghost cells outside the computational domain. When vertex-centered FD is used, Dirichlet boundary condition needs no special treatment. However the Neumann boundary condition needs a one sided three point stencil finite difference which is 2nd order accurate so that when combined with the central

FD inside the domain, the whole scheme is 2nd order accurate. When the artificial boundary condition (the value of $V(x)$ on the boundary) is given, the above treatment of the boundary is standard, and we do not show the discretization of the boundary here.

To compute the artificial boundary condition using equation (C.4), we need to evaluate the integration numerically. For the cylindrical symmetric Laplacian, (C.4) becomes

$$V(x) = 4\pi G \int_{\Omega} K(\xi)\rho(\xi)d\xi = 4\pi G \int_{z=0}^{1}\int_{r=0}^{1}\int_{\phi=0}^{2\pi} \frac{1}{4\pi r}\rho(t,\phi,z)d\phi dr dz. \quad \text{(C.8)}$$

Quadrature rules can be used to solve the this integration. For example, the simplest midpoint rule gives:

$$V(x) = 4\pi G \sum_{i,j} \rho(r_i, z_j)\delta r \delta z,$$

where

$$\rho(r_i, z_j) = \sum_{k} \rho(r_i, z_j, \phi_k)\delta\phi.$$

The discretization of equation (C.7) inside the domain is simply the central finite difference method:

$$\frac{1}{r}\frac{r_{i+\frac{1}{2}}\frac{V_{i+1,j}-V_{i,j}}{\delta r} - r_{i-\frac{1}{2}}\frac{V_{i,j}-V_{i-1,j}}{\delta r}}{\delta r} + \frac{V_{i,j+1} - 2V_{i,j} + V_{i,j-1}}{\delta z^2} = 4\pi G\rho(r_i, z_i). \quad \text{(C.9)}$$

A similar method is to first take the derivative with respect to r in (C.7) and then use finite differences.

How fast is our algorithm? If Ω is a rectangle in 2D or cube in 3D and the grid is uniform, then there are approximately $O(\sqrt{N})$ points on the boundary in 2D and $O(N^{2/3})$ in 3D. Since computing $V(x)$ at each boundary point requires $O(N)$ computational work, the time complexity of this algorithm is $O(N^{3/2})$ in 2D and $O(N^{5/3})$ in 3D. The time complexity for solving the matrix derived using the finite difference is generally $O(Nlog(N))$ which is much smaller than $O(N^{3/2})$ or $O(N^{5/3})$. Thus, the most time consuming part of our algorithm is the first step to compute the boundary condition using the fundamental solution. To reduce the time of computation, we can use the fast multipole method [50]. However, we need only use it to compute the solution on the boundary. We estimate that the time complexity would be reduced to at most $O(Nlog(N))$ for this method.

Appendix D

Cell Boundary Element Method

In this section, we compare the cell boundary element method [60–63] with the mixed-hybrid finite element method. We will find that the cell boundary element method has a similar formulation with the mixed finite element method in that the fluxes on the edges of a element are a linear combination of the potential defined on the edges of the same element.

The cell boundary element method (CBEM) is a special boundary element method applied upon each element of the domain. The boundary element methods [64–66] are elements methods where the boundary, instead of the domain, is partitioned into elements and the unknowns are defined on the boundary and values inside the domain is computed by a boundary integration. The cell boundary element method is a combination of FEM and BEM. The computational domain is partitioned into finite elements as in FEM. Then the matrix is set up using integration along the boundary of each element just like the mixed FEM. In the following, we will show how to use the CBEM.

First, we need to show some basics from BEM. For more detail, refer

to [64–66]. As in [64], we develop the boundary element formulation for the Laplace's equation:

$$u_{,ii} = 0 \quad \text{in} \quad \Omega.$$

By using Gauss's theorem and integration by parts, we get the Green's second identity:

$$\int_\Omega u_{,ii} w d\Omega - \int_\Omega u w_{,ii} d\Omega = \int_\Gamma (u_{,i} w - u w_{,i}) n_i d\Gamma. \tag{D.1}$$

Now let u be the solution of the Laplace's equation and w be the fundamental solution. Substitute u, w into equation (D.1) and taking special care when evaluating the integral of the Dirac distribution on the boundary, we have

$$c(\xi)u(\xi) + \oint_\Gamma q^*(x,\xi)u(x)d\Gamma = \oint_\Gamma u^*(x,\xi)q(x)d\Gamma, \tag{D.2}$$

where $u^*(x,\xi)$ is the fundamental solution, $q_i = u_{,i}$, $q_i^* = u_{,i}^*$ and

$$c(\xi) = \begin{cases} 1 - \frac{\alpha}{2\pi} & \text{for} \quad \xi \in \Gamma, \\ 1 & \text{for} \quad \xi \in \Omega, \\ 0 & \text{others.} \end{cases}$$

where α is the angle of the boundary. The boundary element method use equation (D.2) to set up the matrix. Using some basis functions on the boundary, we could get a system of equation of the following

$$HU + B = GQ,$$

where U, Q are the unknowns and H, G are some matrix. Multiply both side

by G^{-1}, we have

$$Q = G^{-1}HU + G^{-1}B.$$

Notice the similarity of the above system of equations and those of equation (3.11). They all say that the flux on the boundary are linear combination of the potential. The cell boundary element method just uses the above equation and using the flux continuity condition (3.13) to set up the matrix like the mixed-hybrid finite element method.

Bibliography

[1] Mark A.Yerry and Mark S.Shephard. A modified quadtree approach to finite element generation. *IEEE Comput. Graph. Appl.*, 3(1):39–46, 1983.

[2] Mark A. Yerry and Mark S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal for Numerical Methods in Engineering*, 20:1965–1990, 1984.

[3] G. M. Nielson and J. Sung. Interval volume tetrahedrization. In *IEEE Visualization*, pages 221–228, 1997.

[4] Yongjie Zhang. *Boundary/Finite Element Meshing from Volumetric Data with Applications*. PhD thesis, The University of Texas at Austin, August 2005.

[5] Yongjie Zhang, Chandrajit Bajaj, and Bong-Soo Sohn. 3d finite element meshing from imaging data. *Communications in Numerical Methods in Engineering*, 194(48-49):5083–5106, 2006.

[6] R. Shekhar, E. Fayyad, R. Yagel, and J.F. Cornhill. Octree-based decimation of marching cubes surfaces. In *IEEE Visualization*, pages 163–169, 1996.

[7] S. Schaefer and J.D. Warren. Dual marching cubes: Primal contouring of dual grids. In *Pacific Conference on Computer Graphics and Applications*, pages 70–76, 2004.

[8] S. O. Unverdi and G. Tryggvason. A front-tracking method for viscous, incompressible, multifluid flows. *J. Comput. Phys.*, 100(1):25–37, 1992.

[9] G. Tryggvason, B. Bunner, A. Esmaeeli, D. Juric, N. Al-Rawahi, W. Tauber, J. Han, S. Nas, and Y.-J. Jan. A front-tracking method for the computations of multiphase flow. *J. Comput. Phys.*, 169:708–759, 2001.

[10] Jian Du, Brian Fix, James Glimm, and et al. A simple package for front tracking. *J. Comput. Phys.*, 213(2):613–628, 2006.

[11] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. Petsc home page. http://www.mcs.anl.gov/petsc, 2001.

[12] R. Falgout and et al. Hypre home page. http://www.llnl.gov/CASC/hypre/software.html, 2007.

[13] Lapack home page. http://www.netlib.org/lapack/index.html, 2007.

[14] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, Computer Science Department, Carnegie Mellon University, May 1997.

[15] D. L. Marcum. Efficient generation of high-quality unstructured surface and volume grids. *Engineering with Computers*, 17(3):211–233, 2001.

[16] J.F. Thompson, B.K. Soni, and N.P. Weatherill. *Handbook of grid generation*. CRC Press, 1999.

[17] S. Owen. A survey of unstructured mesh generation technology. In *7th International Meshing Roundtable*, page 1998, 1997.

[18] J. Glimm and O. McBryan. A computational model for interfaces. *Adv. Appl. Math.*, 6:422–435, 1985.

[19] James Glimm, John W. Grove, Xiao Lin Li, Keh ming Shyue, Yanni Zeng, and Qiang Zhang. Three-dimensional front tracking. *SIAM Journal on Scientific Computing*, 19(3):703–727, 1998.

[20] I.-L. Chern, J. Glimm, O. McBryan, B. Plohr, and S. Yaniv. Front tracking for gas dynamics. *J. Comput. Phys.*, 62(1):83–110, 1986.

[21] O. McBryan. Elliptic and hyperbolic interface refinement in two phase flow. In J. Miller, editor, *Boundary and Interior Layers - Computational and Asymptotic Methods*, Dublin, 1980. Boole Press.

[22] J. Glimm, B. Lindquist, O. McBryan, and L. Padmanabhan. A front tracking reservoir simulator, five-spot validation studies and the water coning problem. In R. E. Ewing, editor, *Mathematics of Reservoir Simulation*, Philadelphia, 1983. SIAM.

[23] Wei Guo. *A Parallelized Point-Shifted Tetrahedral Grid for the Finite Element Method*. PhD thesis, SUNY at Stony Brook, May 2002.

[24] Yoon-Ha Lee. *Stochastic Error Analysis of Multiscale Flow Simulations: The Two-phase Oil Reservoir Problem*. PhD thesis, SUNY at Stony Brook, August 2005.

[25] M.D. Berg, M.V. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition edition, 1998.

[26] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Series In Computer Science, 1990.

[27] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithm*. MIT Press, second edition edition, 2001.

[28] W.E. Lorensen and H.E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[29] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999.

[30] Stanley J. Osher and Ronald P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.

[31] Jules Bloomenthal (Editor). *Introduction to Implicit Surfaces (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, 1997.

[32] HERBERT EDELSBRUNNER, LEONIDAS J. GUIBAS, and JORGE STOLFI. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, May 1986.

[33] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulations. *Int. J. Foundations of Computer Science*, 13(2):181–199, 2002.

[34] Max Langbein, Gerik Scheuermann, and Xavier Tricoche. An efficient point location method for visualization in large unstructured grids. In T. Ertl, B. Girod, G. Greiner, H. Niemann, Hans-Peter Seidel, E. Steinbach, and R. Westermann, editors, *Proc. 8th Int. Worksh. Vision, Modeling, and Visualization*. IOS Press, 2003.

[35] Alexandre J. Chorin. Numerical solution of the navier-stokes equations. *Math. Comput.*, 22:745–762, 1968.

[36] J. B. Bell, P. Colella, and H. M. Glaz. A second order projection method for the incompressible naviercstokes equations. *J. Comput. Phys.*, 85:257, 1989.

[37] David L. Brown, Ricardo Cortez, and Michael L. Minion. Accurate projection methods for the incompressible naviercstokes equations. *J. Comput. Phys.*, 168:464–499, 2001.

[38] Charles S. Perskin. The immersed boundary method. *Acta Numerica*, 11:479–517, 2002.

[39] R.J. LeVeque and Z.L. Li. The immersed interface method for elliptic equations with discontinuous coefficients and singular sources. *SIAM J. Numer. Anal.*, 31:1019C1044, 1994.

[40] H. Johansen and P. Colella. A cartesian grid embedding boundary method for poisson's equation on irregular domains. *J. Comput. Phys.*, 147:60C85, 1998.

[41] P.-A. Raviart and J.-M. Thomas. a mixed finite element method for second order elliptic problems. In I. Galligani and E. Magenes, editors, *Mathematical Aspects of Finite Element Methods*, pages 292–315. Springer-Verlag, 1977.

[42] Franco Brezzi and Michel Fortin. *Mixed and Hybrid Finite Element Methods.* Springer Series In Computational Mathematics 15, 1991.

[43] Alfio Quarteroni and Alberto Valli. *Numerical Approximation of Partial Differential Equations.* Springer Series in Computational Mathematics 23, 1997.

[44] G. Chavent and J.E. Roberts. A unified physical presentation of mixed, mixed-hybrid finite elements and standard finite difference approximations for the determination of velocities in waterflow problems. *Adv. Water Resources*, 14(6):329–348, 1991.

[45] A. Younes, R. Mose, P. Ackerer, and G. Chavent. A new formulation of the mixed finite element method for solving elliptic and parabolic pde with triangular elements. *J. Comput. Phys.*, 149(1):148–167, 1999.

[46] Douglas N. Arnold. Mixed finite element methods for elliptic problems. *Comput. Methods Appl. Mech. Engrg.*, 82:281–300, 1990.

[47] Catherine E. Powell. *Optimal Preconditioning for Mixed Finite Element Formulation of Second-Order Elliptic Problems*. PhD thesis, The University of Manchester, December 2003.

[48] X.-D. Liu, R. Fedkiw, and M. Kang. A boundary condition capturing method for poisson's equation on irregular domains. *J. Comput. Phys.*, 160:151–178, 2000.

[49] A. Mayo. The fast solution of poisson's and the biharmonic equations on irregular regions. *SIAM J. Numer. Anal.*, 21:285C299, 1984.

[50] A. Mckenney, L. Greengard, and A. Mayo. A fast poisson solver for complex geometries. *J. Comput. Phys.*, 118:348C355, 1995.

[51] Roman Samulyak, Jian Du, James Glimm, and Zhiliang Xu. A numerical algorithm for mhd of free surface flows at low magnetic reynolds numbers. *J. Comput. Phys.*, 2007.

[52] J. Glimm, D. Marchesin, and O. McBryan. *J. Comput. Phys.*, 39:179,200, 1981.

[53] Y.C. Zhou, S. Zhao, M. Feig, and G.W. Wei. High order matched interface and boundary method for elliptic equations with discontinuous coefficients and singular sources. *J. Comput. Phys.*, 213:1–30, 2006.

[54] M. Kang, R. Fedkiw, and X.-D. Liu. A boundary condition capturing method for multiphase incompressible flow. *J. Sci. Comput.*, 15(3):323–360, 2000.

[55] LeLe. Compact finite-difference schemes with spectral-like resolution. *J. Comput. Phys.*, 103(1):16–42, 1992.

[56] D.E. Fyfe, E.S. Oran, and M.J. Fritts. Surface tension and viscosity with lagrangian hydrodynamics on a triangular mesh. *J. Comput. Phys.*, 76:349–384, 1988.

[57] Robert C. McOwen. *Partial Differential Equations: Methods and Applications*. Prentice-Hall, 1991.

[58] Dan Givoli. High-order nonreflecting boundary conditions without high-order derivatives. *J. Comput. Phys.*, 170:849–870, 2001.

[59] Dan Givoli. High-order local non-reflecting boundary conditions: a review. *Wave Motion*, 39:319–326, 2004.

[60] Y. Jeon and D. Sheen. Analysis of a cell boundary element method. *Adv. Comput. Math.*, 22:201–222, 2005.

[61] Y. Jeon, E.-J. Park, and D. Sheen. A cell boundary element method for elliptic problems. *Numerical Methods for Partial Differential Equations*, 21:496–511, 2005.

[62] M. Tan, T. Farrant, and W.G. Price. A cell boundary-element method for viscous laminar flow solutions. *Proc. R. Soc. Lond. A*, page 4277C4304, 1999.

[63] T. Farrant, M. Tan, and W. G. Price. A cell boundary element method applied to laminar vortex-shedding from arrays of cylinders in various arrangements. *Journal of Fluids and Structures*, 20:375–402, 2000.

[64] L. Gaul, M. Kögl, and M. Wagner. *Boundary Element Methods for Engineers and Scientists: An Introductory Course with Advanced Topics.* Springer-Verlag, 2003.

[65] C.A. Brebbia and J. Dominguez. *Boundary Elements: An Introductory Course.* McGraw-Hill Inc, 1989.

[66] J. Raamachandran. *Boundary And Finite ELements.* Narosa Publishing House, 2000.