

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

Integer-Coordinate Crystalline Meshes

by

Vasiliki Chatzi

B. S., University of Patras, 1992

Sc. M., Brown University, 1995

**A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University**

Providence, Rhode Island

May 2000

UMI Number: 9987740

UMI[®]

UMI Microform 9987740

Copyright 2000 by Bell & Howell Information and Learning Company.


**All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.**

**Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346**

© Copyright 1998,1999,2000 by Vasiliki Chatzi

This dissertation by Vasiliki Chatzi is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

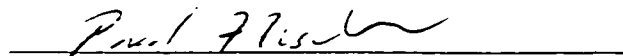
Date 5-12-00



Franco P. Preparata, Director

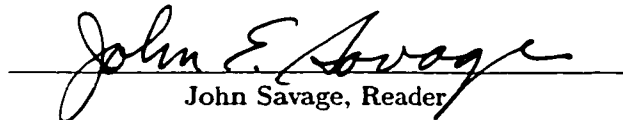
Recommended to the Graduate Council

Date 5/12/00



Paul Fischer, Reader
(MSC, Argonne National Laboratory)

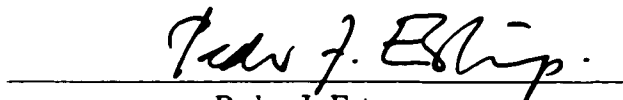
Date 5/12/00



John Savage, Reader

Approved by the Graduate Council

Date 5/18/00



Peder J. Estrup
Dean of the Graduate School and Research

Vita

Vitals

Vasiliki Chatzi was born on May 31, 1969 in Tripolis, Greece. She was admitted in the Computer Engineering and Informatics department at University of Patras in 1987, and graduated in 1992. She spent one year working as a software engineer in Patras, Greece, before entering the Ph.D. program in the Computer Science Department of Brown University.

Education

Ph.D. in Computer Science, May 2000.

Brown University, Providence, RI.

M.Sc. in Computer Science, May 1995.

Brown University, Providence, RI.

B.A. in Computer Engineering and Informatics, June 1992.

University of Patras, Greece

References

Refereed Conference Articles

Vasiliki Chatzi and Franco P. Preparata *Integer-Coordinate Crystalline Meshes* . Proceedings of the Swiss conference on CAD/CAM. Neuchatel, Switzerland, February 1999

Vasiliki Chatzi *Finding Basis Functions for Pyramidal Finite Elements*. Proceedings of the 3rd CGC Workshop on Computational Geometry. Providence, RI, October 1998.

Acknowledgements

I would like to thank everybody who helped me with the preparation of this thesis. Especially, I would like to thank my advisor Franco P. Preparata for leading me into the world of research, for spending countless hours discussing with me research problem and particularly for being there for me to provide help and support whenever I needed it. I also would like to thank Paul Fischer, for teaching me everything I know about the Finite Element Method and numerical computation in general and John Savage, for his support and advise. Finally, I want to thank José Castaños for a long and fruitful collaboration.

My stay at Brown was a very pleasant one, and this is mainly thanks to my friends. I want to especially thank Kostantina Trivisa, Hagit Shatkay and Zornitsa Athanasova for the help and support they gave me during my first years at Brown, when I needed it the most and Dimitris Michailidis, for being there to answer technical questions and give advise. Many thanks to Mike Benjamin, for tolerating me for so many years (not an easy thing to do, I realize). I would like to thank Manos Renieris, for his help with Latex and many other technical problems, but especially for believing in me. Many thanks go to Galina Shubina, for being the best friend and roommate I could wish for. Finally, I want to deeply thank the following people for their support: Laurent Michel, Costas Bush, Luis Ortiz, Ioannis Tsochantaridis, Constantinos Evangelinos and Johanna Quattrucci.

A big thank you to my parents Aggeliki and Panagiotis for teaching me the value of education early and for supporting me during my undergraduate studies. Most of all, I want to thank my brother Kostas and my sister Irini for helping me in so many ways.

Contents

List of Tables	x
List of Figures	xi
1 Introduction	1
2 The Finite Element Method	5
2.1 Introduction to the Finite Element Method	5
2.2 Mesh Generation	7
2.3 Basis Functions and Element Types	8
2.4 Constructing and Solving the System of Equations	12
2.5 Error Estimates and Mesh Adaptation	13
3 Finite Element Meshes	15
3.1 Introduction	15
3.2 High-Quality Meshes	16
3.3 Conformal vs. Non-conformal Meshes	17
3.4 Structured vs. Unstructured Meshes	18
3.5 Numerical errors	20
3.6 Conclusions	22
4 Integer-Coordinate Crystalline Meshes	23
4.1 General Description	23
4.2 Definitions	26
4.3 2D Crystalline Meshes	30
4.4 3D Crystalline Meshes	31

5	Basis Functions for Pyramidal Finite Elements	34
5.1	Using Pyramidal Elements in Mixed Meshes	34
5.2	Lagrangian Pyramid	35
5.3	Template Pyramid and Linear Transformation	36
5.4	Basis Functions for the Order-1 Pyramid	38
5.5	Generalizing to the Order- k Pyramid	41
5.6	Summary of Results	49
5.7	Automating the Process	51
6	Mesh Generation	52
6.1	Problem Description	52
6.2	Input Format	52
6.3	Generating the Internal Mesh	54
6.4	Obtaining an Approximation of the Boundary	59
6.5	Boundary Approximation in 2D	64
6.6	Boundary Approximation in 3D	69
6.7	Obtaining a Legal Mesh	74
6.8	Estimating the Quality of the Mesh	78
7	Mesh Adaptation	81
7.1	Introduction	81
7.2	General Refinement Technique	82
7.3	Complexity of Refinement	83
7.4	2D Mesh Refinement	90
7.5	3D Mesh Refinement	99
7.6	Non-standard Refinement for 3D Meshes	105
7.7	Coarsening	107
8	Experimental Results	112
8.1	Implementational Issues	112
8.2	Benchmarks and Error Metrics	113
8.3	Results	115
9	Conclusions and Future Work	119
9.1	Conclusions	119

9.2 Future Work	120
A Basis Functions	123
A.1 A Simple Example	123
B Basis Functions for Higher-Order Pyramids: Examples	126
B.1 Order-2 Pyramid	126
B.2 Order-3 Pyramid	127
B.3 Generating Basis Functions - Program	130
C Non-Standard Refinement of Non-Conformal Cubes	146
Bibliography	151

★ Parts of this thesis were previously published as [19, 20]

List of Tables

5.1	Numbering of vertices on the template pyramid.	36
5.2	Numbering of edges on the template pyramid.	37
5.3	Numbering of faces on the template pyramid.	37
5.4	Reduction of basis functions on faces.	40
8.1	Performance results for structured triangular meshes.	116
8.2	Performance results for structured quadrilateral meshes.	116
8.3	Performance results for unstructured triangular meshes obtained using Rivara refinement.	117
8.4	Performance results for crystalline meshes.	117
A.1	Basis functions associated with nodal points.	124

List of Figures

2.1	A domain and the mesh associated with it.	7
2.2	Fourth order Lagrangian triangle and square.	10
2.3	Lines composing Lagrangian basis function.	11
3.1	Conformal and non-conformal three dimensional meshes.	17
3.2	Examples of conformal and non-conformal two-dimensional meshes.	17
3.3	Two structured meshes	19
3.4	An unstructured mesh	20
3.5	Illustration of point placement errors.	21
4.1	Example two dimensional and three dimensional crystalline meshes.	24
4.2	Element capsules.	27
4.3	Level values.	28
4.4	Rectilinear, conformable and legal meshes	29
4.5	Possible neighboring arrangements for a square.	30
4.6	Possible neighboring arrangements for a triangle.	31
4.7	Transitional elements for 3D crystalline meshes.	31
4.8	3D neighboring arrangements not exhibiting the Delaunay property	32
5.1	Conformal transition between cubes of different sizes.	34
5.2	Low order Lagrangian pyramids.	35
5.3	Template pyramid.	36
5.4	The deformed pyramid and the template pyramid.	37
5.5	First order template pyramid.	38
5.6	First order basis functions for a square and a triangle.	40
5.7	A k th order pyramid.	42

5.8	Numbering of points on the base.	44
5.9	Nonequivalent points on the base	44
5.10	Mapping of points after rotations.	45
5.11	Reduction of ξ_i^L and ξ_i^R on the base of the pyramid	45
5.12	Reduction of Ξ^L and Ξ^R on the base of the pyramid	46
5.13	Reduction of the quadric on the faces	47
5.14	Reduction of $\phi_{i,j}$ on the base.	47
6.1	First steps of mesh generation	56
6.2	A case where Assumption 6.3.1 is violated	59
6.3	Exact representation of the boundary.	60
6.4	Projecting the vertices onto the boundary.	60
6.5	2D elements intersected by the boundary.	64
6.6	Ignoring intersection points produced by tangent lines.	65
6.7	Possible configurations after snapping.	66
6.8	Cases where the refinement does not give a better approximation	67
6.9	Successive mesh approximations.	68
6.10	Boundary intersecting a simple element.	69
6.11	Possible 3D configurations after snapping.	71
6.12	Ambiguous configuration and three possible partitions.	71
6.13	Another ambiguous configuration and two possible partitions.	72
6.14	3D partitions that do not result in better approximations.	73
6.15	Shapes of boundary elements	73
6.16	Boundary shapes 1–5 and associated partitioning.	75
6.17	Boundary shapes 6 and 7 and associated partitioning.	76
6.18	Two new tetrahedra used for approximating the boundary.	76
6.19	Example of conformable and rectilinear mesh.	77
6.20	Regions that are covered either by the mesh or by the domain but not by both. . . .	78
6.21	Getting a good upper bound on $\ s\ $	79
7.1	Flow diagram of the refinement process	83
7.2	A rectilinear mesh and its associated graph.	84
7.3	Possible neighboring arrangements in conformable mesh.	86
7.4	Path in $G(M)$ containing two weight-2 arcs.	86

7.5	Bounding boxes for elements in the refinement path starting from element x .	88
7.6	2D illustration of the refinement process.	90
7.7	Non-standard refinement for non-conformal elements.	91
7.8	Bounding the number of elements at distance n from x .	92
7.9	Illustration of the refinement algorithm on a 2D mesh.	93
7.10	Worst case level differences in neighboring elements.	95
7.11	Examples of how neighbors of x with higher level value are processed by the algorithm.	97
7.12	Elements at distance n from x to which the refinement can propagate.	99
7.13	Cubes with different level values sharing a vertex.	101
7.14	Standard patterns according to which faces of cubes are partitioned.	102
7.15	Examples illustrating why edges will be conformal after the last step.	103
7.16	Examples, illustrating why faces will be conformal after the last step.	104
7.17	A non-conformal cube and the corresponding standard patterns.	106
7.18	The faces of a non-conformal cube, marked with the appropriate standard patterns.	106
7.19	Example of non standard refinement	108
7.20	Non standard refinement of a non conformal cube shown in compact form.	109
7.21	Flow diagram of the coarsening process	110
7.22	2D illustration of the coarsening process.	111
8.1	Plot of the solution of the problem used as benchmark.	114
8.2	Crystalline meshes obtained using adaptive refinement for the benchmark problem.	118
A.1	First-order one-dimensional element.	123
A.2	Combining the basis functions of all elements.	124
A.3	The solution of the problem combines all basis functions.	125
B.1	Second order pyramid	126
B.2	Third order pyramid	128

Chapter 1

Introduction

Problems in physics and engineering defined on a given physical domain are often modeled by a system of partial differential equations, together with appropriate boundary conditions. This data completely models a physical system. If we were able to analytically solve the differential equations we would obtain the values of the physical quantities of interest in the defined region. However, obtaining a closed form solution of the equations is often impossible. In these cases we have to settle for using a numerical method to obtain an approximation to the solution.

The finite-element method is one of the tools commonly used today to approximately solve partial differential equations associated with a physical region. It was first used by engineers in the late fifties to obtain the numerical solution of partial differential equations in structural engineering [2, 24]. It soon became clear that the same technique could be used for the simulation of a wide range of physical problems. As computers became more powerful and readily available, the finite element method also gained popularity. Today it is widely used to solve problems not only in structural engineering [92, 84] but also in fluid mechanics [3], electromagnetism [46], heat transfer [27], metal forming [48] etc.

The first step of the method consists of discretizing the region by partitioning it into a set of simple shapes called *elements*. The resulting collection of elements is called a *finite element mesh*. Then, nodal points are defined on each of the elements, and a basis function is associated with each nodal point. The basis functions are used to construct a system of linear equations and, finally, the system is solved to obtain an approximation to the solution of the partial differential equation. The mesh used plays an important role in the entire computation since it affects not only how accurate our solution is, but also how fast it can be computed.

Although this is still an active research area, there is general agreement on some properties a high-quality mesh should have. First of all, since the problem is associated with a physical domain, the union of all the elements in the mesh should closely approximate the domain. We also would like to be able to cover different regions of the domain with elements of different size. In general, smaller elements give us a better approximation of the solution. However, using smaller elements

also implies that we need more elements to cover the same region of the domain, which results in a larger and harder to solve system of equations. So, we would like to cover regions of the domain where the solution changes rapidly (“regions of interest”) with smaller elements, while using larger elements for regions where the phenomenon is relatively stable. Moreover, the transition between elements of different sizes should be *smooth* [37], that is, the ratio of areas (or volumes) of two adjacent elements should be bounded by a small constant. Finally, it is important that the resulting system of equations be well-conditioned, and therefore easy to solve. It is not entirely clear what properties of the mesh necessarily result in a well-conditioned system; however, several theoretical and experimental results show that elements with small angles have to be avoided.

Another highly desirable, but not necessary, property is *conformity*. An element is said to be *conformal* if its intersection with any other element in the mesh is empty, or a vertex, or an edge, or a face. A mesh is conformal if all its elements are conformal [11, 76]. A point is called *non-conformal* if it is the vertex of one element, but is positioned in the interior of an edge of an other element. Non-conformal points create numerical problems which can only be solved by using complicated specialized techniques, and therefore should be avoided whenever possible.

Meshes have fallen into two broad categories: *structured* meshes, consisting of virtually identical elements, and *unstructured* meshes, whose geometry is arbitrary within some broad topological constraints. Structured meshes are easy to construct and result in systems of equations that are easy to generate and to solve. However, they are inappropriate for problems defined over a complex region and for problems exhibiting regions of interest. Unstructured meshes are more flexible, and are therefore used to model these problems. However, the resulting systems of equations are harder to generate and solve.

Another major problem associated with unstructured meshes is that of accumulating numerical errors. One of the standard operations done while generating or modifying the mesh is computing the coordinates of new points. This is done by performing operations on the coordinates of already existing points. Of course, when using this process repeatedly, the numerical errors propagate and accumulate. Sometimes, the errors are so large that the resulting mesh is not topologically consistent any more. In this case, we are forced to use complicated and time-consuming methods to compensate for this behavior.

Another type of numerical error occurs when assembling the system of linear equations. During this process, precomputed element matrices have to be modified to account for the presence of deformed elements. This operation introduces numerical errors in the system of equations, which, of course, reduces the accuracy of the obtained solution. Structured meshes don't exhibit this problem. Since all the elements are the same, element matrices can be precomputed to the desired accuracy.

In this work [20] we present *integer-coordinate crystalline meshes* which were designed to combine the advantages of both structured and unstructured meshes. A mesh is called *crystalline* if it consists of regular elements whose vertices are positioned on a regular grid. The regular elements we use in our meshes are squares in 2D and cubes in 3D. Cubes were chosen because there are indications

that they outperform tetrahedra for some problems [4]. Crystalline meshes allow us to use elements of different sizes in different regions of the domain, which makes them suitable for problems with regions of interest. Conformal transitions between regions containing d -cubes of different sizes are realized using elements belonging to a small set of distinct, non-cubical shapes. This allows us to precompute the element matrices used in the computation and ensure a very accurate system of linear equations. Moreover, it guarantees that the angles of all the elements will be bounded comfortably away from zero, resulting in a system of linear equations that is easier to solve.

If all the vertices in a crystalline mesh are placed on integer coordinates, the mesh is an *integer-coordinate crystalline mesh*. Using only integer coordinates allows us to use integer arithmetic during the mesh generation and adaptation phases, which alleviates the problem of accumulating numerical errors that is common in other types of meshes. Notice that by appropriately scaling the domain, integer coordinates can always be used.

Integer-coordinate crystalline meshes allow for different element sizes in different regions of the domain and are conformal. They only use elements belonging to a small set of distinct shapes, which allows us to construct the system of linear equations both fast and accurately. Their use minimizes numerical errors during the mesh generation and adaptation phases and simplifies adaptive computations. All these advantages, as well as empirical evidence that the systems of equations generated from crystalline meshes are well conditioned, suggest that crystalline meshes are appropriate for problems where structured meshes cannot be used.

While isolated examples of such meshes have been occasionally reported in the literature for 2D problems (for example, the cover of [10] shows a 2D crystalline mesh), our emphasis is on the far more complex and significant 3D setting. We will focus on the characteristics of crystalline meshes and on the procedures intended to adapt the element density as required to achieve high-quality solutions.

There is some similarity between crystalline meshes and meshes generated using quadtree or octree based methods [94, 7, 37, 75, 77] in that both meshes consist mainly of squares or cubes (at least at some point during the mesh generation phase). However, octree meshes do not use integer coordinates for their vertices, which results in points that are non-conformal in practice. Moreover, the last stages of the octree method are typically a triangulation phase, where the obtained cubes are cut into tetrahedra, and a smoothing phase, where the vertices of the tetrahedra are moved to obtain elements with larger angles. So the output meshes consist of irregular triangles or tetrahedra. This makes the assembling of the system of linear equations both computationally expensive and inaccurate.

Cartesian meshes (see for example [25]) are also somewhat similar to crystalline meshes in that they also use points with integer coordinates, and the elements of the mesh are squares in 2D, and cubes in 3D. However, the exclusive use of d -cubes results in non-conformal points in the transition regions between elements of different sizes. This forces researchers using Cartesian meshes to develop complicated and computationally expensive techniques to deal with the singularities arising from

the introduction of non-conformal points [5].

Some work has also been done on meshes that have guaranteed angle bounds [6, 60]. However, these results concentrate on triangular and tetrahedral meshes. Moreover, they seem to have more theoretical than practical value, since the meshes generated using the proposed algorithms have a much higher number of elements than needed, to achieve the desired accuracy. We are not aware of any work concentrating on hexahedral or mixed meshes.

The three-dimensional crystalline meshes we designed use pyramids to guarantee conformal transitions between regions of the domain that contain cubes of different size. Pyramids are very convenient for this purpose since they have one square face, which can be conformally attached to a cube, and four triangular faces which can be attached to tetrahedra. However, researchers avoid using pyramids, because finding basis functions for them is difficult. As part of this work, we overcame this well-known obstacle and present a method that can be used to construct basis functions for pyramids of order k for any $k > 0$.

In order for crystalline meshes to be useful, we need to have an automated procedure that will create them from the description of the domain. It turns out that the hardest part of the process is approximating the boundaries. A large portion of this thesis is devoted to discussing how this can be done efficiently.

Another important property of crystalline meshes is that they are suitable for adaptive computations, where the simulated system evolves over time. Creating meshes that facilitate adaptive computation is one of our main goals, and we will spend a large part of this thesis describing how mesh adaptation for crystalline meshes can be done efficiently.

The rest of this thesis is organized as follows. Chapter 2 contains a general introduction to the Finite Element method. Chapter 3 takes a closer look at Finite Element meshes and discusses their general characteristics and properties. Chapter 4 gives the characterization of integer-coordinate crystalline meshes. Chapter 5 is dedicated to the description of the method that can be used to find basis functions for pyramids. Chapter 6 presents the algorithm for crystalline mesh generation. Chapter 7 discusses the adaptive features. Chapter 8 addresses some implementation issues and gives some experimental results. Finally, Chapter 9 contains some open problems and gives a summary of the results and the conclusions.

Chapter 2

The Finite Element Method

2.1 Introduction to the Finite Element Method

The *Finite Element Method* is a mathematical technique for obtaining approximate solutions to the partial differential equations that predict the response of physical systems to external influences. It has been designed to facilitate automated computation and there exist several commercial software tools that implement it.

The Finite Element Method is mainly used for problems in engineering, physics and applied mathematics. Applications include, but are not limited to, problems in solid mechanics, heat transfer, fluid mechanics, acoustics and electromagnetism. Lately, much emphasis has been placed on problems that consist of a combination of these areas (for example, simulating devices that have an electronic as well as a mechanical component) and the special difficulties that arise in these situations [57, 41, 56].

The physical system we wish to simulate, and the external influences, can be described by presenting:

1. The *domain* of the problem, which is typically the physical region occupied by the system. For time-dependent problems the domain also consists of a time interval.
2. The *physical characteristics* of the system. This includes other relevant information about the system, such as specifications of the materials the system is made of.
3. The *governing equations* of the system, which are partial differential equations that characterize the physical process we want to simulate. For example, if the problem is a fluid mechanics problem, its governing equations are the Navier-Stokes equations that describe fluid flow. Some of these equations might also express a conservation or variation principle, such as the conservation of energy.
4. The *loading conditions*, which describe the external forces that interact with the system. Forces

that act on the interior as well as the exterior of the system (for example gravity) are modeled as part of the governing equations. Forces that act only on the exterior of the system are called *boundary conditions* and are modeled separately.

The information described above completely specifies the physical system. The Finite Element Method can then be used to obtain the values of the physical quantities of interest. Specifically, we get an approximation to the solution that is a linear combination of a set of basis functions that have been previously defined on the domain. This linear combination can be used to obtain an approximation of the value of the physical quantity at any point in the domain.

Let us now give a general overview of the Finite Element Method. It consists of several discrete steps:

1. The domain of the problem is partitioned into a set of simple shapes, called *elements*. The collection of all the elements is called a *finite element mesh*, and the process of creating the mesh from the domain is called *mesh generation*.
2. Nodal points are selected on the element. Then, a function is associated with each one of the nodal points. The set of functions defines a basis that spans the domain in which we seek the approximation, so these functions are called basis functions. The solution obtained will be a linear combination of these basis functions.
3. The basis functions are used to construct a system of linear equations. The system is then solved to obtain the coefficients by which the basis functions should be multiplied in order to obtain an approximation of the solution.
4. Numerical methods are used to obtain an estimate of the approximation error at each point of the domain. If the error exceeds some specified quantity, the mesh is modified and the entire process is repeated. This last step is not necessarily performed. When it is, we have an *adaptive* finite element computation.

Notice that the Finite Element Method, as described above, is a general approach to finding an approximation to the solution and not a specific algorithm, since, in many of the steps, we have several choices. For example, during the mesh generation step we can choose the shapes of the elements, and the governing criterion will be efficiency, since we want to use simple enough elements to facilitate subsequent steps of the computation. There are several different ways of placing nodal points in the mesh. Also, we can choose the functions used, as long as they satisfy some basic requirements. In the next sections of this chapter we will examine each of these choices in more detail.

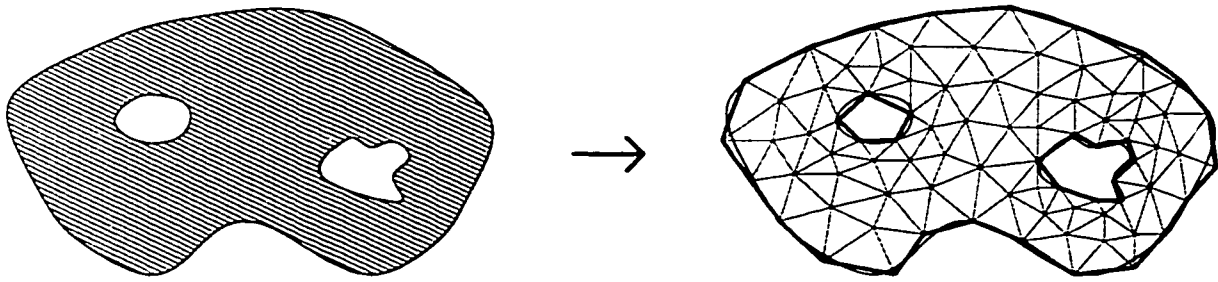


Figure 2.1: The physical domain of the problem and a mesh associated with it. Notice that the vertices are not placed exactly on the boundary (because of limitations in the accuracy of the computer), and the boundary of the mesh is an approximation of the boundary of the domain consisting of straight line segments.

2.2 Mesh Generation

Mesh generation is the first step of the Finite Element Method and can be formally defined as follows:

Definition 2.2.1 *The process of partitioning the domain of the problem into a set of simple shapes is called mesh generation.*

The process can be done manually, but for complicated, large meshes it is automated. The software tools used are called *mesh generators*. Very few of the existing mesh generators are fully automated. Most require user supervision to produce meshes for complex domains.

The simple shapes into which the domain is partitioned are called *elements* and the collection of all the elements is a *finite element mesh*. Figure 2.1 shows a 2D domain and a mesh associated with it. Notice that, in this example, all elements are triangles.

Although the definition specifies that the mesh should be a partition of the domain, this is not always possible for two reasons:

1. The accuracy of point placement is limited by the representation used by the computer for points. We may not be able to place points exactly on the boundary of the domain.

Figure 2.1 illustrates this problem. Notice that several of the points on the boundary of the mesh are not placed exactly on the boundary of the domain but in the interior or exterior of the domain (although they are close to the boundary). This happens because the computer representation used for point coordinates is not accurate enough to represent points *on* the boundary.

2. We are restricted to using specific shapes as elements (usually chosen on the basis of *simplicity*). This, in general, means that we are restricted to using elements that have straight line edges in 2D, or planar faces in 3D. If the boundaries of the domain are general curves, the best we can hope for is a good approximation of the curves by straight line segments or planar faces.

The domain shown in Figure 2.1, for example, has three boundaries that are general curves. The boundary of the mesh associated with the domain is a collection of cycles of straight line segments.

Mesh generators take as input a description of the domain. Typically, only the boundaries of the domain are given, together with information about the material that each part of the system consists of. This description can be in several different formats, depending on the application. For example, the boundary can be described by a set of partial functions. In cases where less accuracy is required, or where the shape of the domain is simple, the input might be a set of line segments (in 2D) or triangulated surfaces (in 3D).

The output of a mesh generator consists of a set of vertices and a set of elements. Each vertex can have a set of variables associated with it that describe some of its properties. For example, it is crucial to be able to distinguish vertices that are on the boundary of the domain from vertices placed in the interior of the domain, and so boundary vertices should be clearly marked by the mesh generator. Element entries also have other type of data associated with them, for example, information about the material the element is made of.

The mesh created at this step is used throughout the Finite Element computation. It affects both the accuracy of the solution obtained, and the time needed for the computation. We will talk about finite element meshes in more detail in Chapter 3.

2.3 Basis Functions and Element Types

After the mesh has been obtained, we define nodal points on each of the elements. Nodal points can be placed both on the boundaries of the elements, and in their interior. Nodes placed on edges or faces that are shared by two or more elements belong to all these elements. We will use n to denote the number of points placed on the entire mesh.

In the next step, a function is associated with each nodal point. Let ϕ_i ($0 \leq i < n$) be the function associated with the point p_i . These functions are a basis of the space in which we seek the approximation of the solution, and are therefore called *basis functions*. Specifically, when using the Finite Element Method, we solve a system of linear equations to obtain a vector \bar{x} . The approximation to the solution is then (for a 3D problem) a function $f(x, y, z)$ defined as:

$$f(x, y, z) = \sum_{i=1}^n \bar{x}_i \phi_i(x, y, z)$$

This approximation allows us to compute the value of the physical quantity of interest at any point in the domain. For 2D problems, the function $f(x, y)$ is obtained in a similar manner.

From the above discussion it is easy to see that, in general, placing more nodal points on the mesh will give us a better approximation to the solution. It will also result in a larger system of equations that is harder to solve.

It has been established (e.g. [95]) that in order for the Finite Element Method to work correctly, the basis functions should have the following properties:

- **Property 1.** *Each basis function ϕ_i associated with a nodal point p_i evaluates to 0 on all the points in the physical domain lying outside the elements containing p_i .* This requirement is easy to achieve, by constructing ϕ_i as a piecewise defined function which is identically 0 outside the relevant elements.
- **Property 2.** *The function ϕ_i evaluates to 1 on p_i and to 0 on all other nodal points.* Notice that nodal points outside the elements containing p_i do not pose a problem, since the function has been defined to evaluate to 0 outside these elements. Thus, the function has to be carefully constructed so that it also evaluates to 0 on nodal points that belong to the elements containing p_i .
- **Property 3.** *The function ϕ_i is continuous.* Since solutions of the partial differential equations are typically continuous it is reasonable to require that they be approximated by continuous functions. The derivatives of the basis functions, however, are not required to be continuous. This type of continuity, where the functions, but not their derivatives, are continuous is known as C_0 -continuity.

Property 3 in conjunction with Property 1 requiring that ϕ_i evaluate to 0 outside of elements containing p_i , yields the following (nonindependent) property:

- **Property 4.** *For each element containing p_i , ϕ_i must be 0 on the portions of the element's boundary not containing p_i .*

These four properties have to hold for all basis functions, regardless of the dimensionality of the problem. Specifically, they have to hold for both 2D and 3D basis functions. In this context, Property 1 and Property 2 yield:

- **Property 5.** *In 3D, for each element's face containing p_i , ϕ_i must reduce to a correct 2D basis function for that face.*

A common practice when designing basis functions for the nodal points defined on a mesh is to restrict our attention to one element at a time. Every time an element e is processed, appropriate basis functions for all nodal points in the element are defined. These basis functions are valid only within the element e . When using this policy, however, we have to be careful so that the resulting basis functions exhibit C_0 -continuity. Nodal points placed on shared vertices, edges or faces will have several partial functions associated with them, one for each element they belong to. These functions should have the same value on every point on the shared edges and faces. For an example of how basis functions defined on different elements can be combined, and how they can be used to reconstruct the approximation to the solution, see Appendix A.

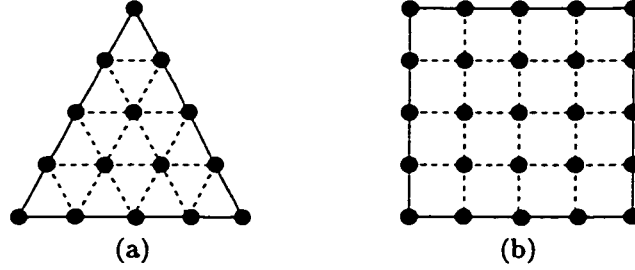


Figure 2.2: Standard 2D Lagrangian elements of order 4. (a) Triangle. (b) Square. Dashed lines denote the grid used for the placement of the internal nodal points.

Mesh elements are characterized by both the placement of nodal points on them, and the basis functions associated with the nodal points. The elements most commonly used today are elements of the Lagrange family, named so because the associated basis functions can be expressed as the product of 2 (3 for 3D elements) Lagrange interpolation polynomials.

2D Lagrangian elements of order k have $k + 1$ nodal points placed on each of their edges at regular intervals. These points define a grid in a natural way, and additional nodal points are placed on the vertices of this grid. Let n' be the number of nodal points placed in the element. Each one of these nodal points is associated in a standard way with an integer i' ($0 \leq i' < n'$), called its *local index*, that uniquely identifies the nodal point in the element. Because of the equidistant placement of nodal points on the edges, Lagrangian elements of very high order are not numerically stable, and in these cases, elements that have a Chebichev distribution of nodal points on the edges are normally used [39].

There are two types of standard 2D Lagrangian elements: triangles and squares. Lagrangian triangles of order k contain $\frac{(k+1)(k+2)}{2}$ nodal points in total. Lagrangian squares of order k contain $(k+1)^2$ nodal points. Figure 2.2 shows the node placement on a Lagrangian triangle and a Lagrangian square both of order 4. Notice that we can use order k Lagrangian triangles and squares in the same mesh without any difficulty, since the node placement on shared edges is not ambiguous.

The basis functions associated with each one of the nodal points are easy to construct. One method for constructing them involves constructing the Lagrangian polynomials and computing their product. Several other fast and accurate methods for obtaining the functions can be found in the literature (e.g. [10, 95]). Each of the resulting basis functions correctly reduces to a k -th degree polynomial on each of the edges of the element (since it must evaluate to 0 in k points of the edge).

The basis functions associated with nodal points placed on Lagrangian triangles are polynomials containing all monomials of the form $x^i y^j$ with $0 \leq i, j$ and $0 \leq i + j \leq k$. The basis functions for Lagrangian squares are polynomials containing all monomials of the form $x^i y^j$ with $0 \leq i, j \leq k$.

Standard 3D Lagrangian elements have faces that are valid 2D Lagrangian elements. Nodal points on the faces define a grid in the interior of the elements in a natural way, and internal points are placed on the vertices of this grid. Nodal points in 3D Lagrangian elements also have a unique

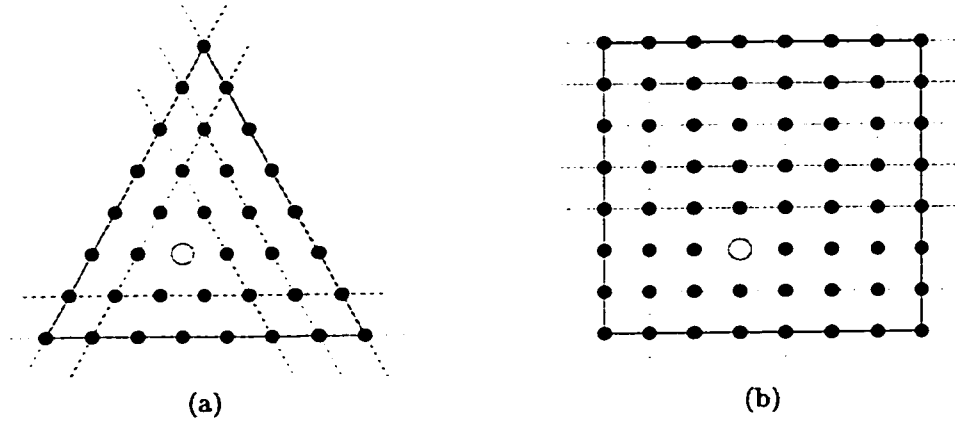


Figure 2.3: The lines composing the basis functions of a nodal point in the interior of a triangular (a) and a square (b) Lagrangian element.

local index associated with them.

Standard 3D Lagrangian elements include the tetrahedron and the cube. A tetrahedron of order k contains $\frac{1}{6}(k+1)(k+2)(k+3)$ nodal points, while a cube of order k contains $(k+1)^3$ points. In the 3D case too, the requirement that all element faces should be standard 2D Lagrangian elements ensures that elements with different shapes but of the same order can be placed in mixed meshes without conflicts involving the placement of the nodes on shared faces and edges.

Basis functions for each of the nodal points can be constructed by using generalizations of the methods used for 2D elements. An important property of the resulting basis functions is that the reduction of the function ϕ_i on one of the faces of the 3D element is identically 0 if p_i is not located on that face and, otherwise, it coincides with the valid 2D Lagrangian basis function for p_i on the face. This automatically ensures C_0 -continuity on the element boundaries.

Basis functions for nodal points on Lagrangian tetrahedra of order k are polynomials containing all monomials of the form $x^i y^j z^l$ with $0 \leq i, j, l$ and $0 \leq i + j + l \leq k$. The basis functions for nodal points on Lagrangian cubes of order k are polynomials containing all monomials of the form $x^i y^j z^l$ with $0 \leq i, j, l \leq k$.

Another important characteristic of standard Lagrangian elements is that the basis function ϕ_i associated with the nodal point p_i can also be expressed as a product of functions describing lines (for 2D elements) or planes (for 3D elements) when equated to 0. With a slight abuse of language we shall say that ϕ_i is respectively “a product of lines” or “a product of planes”. These geometric primitives cover all nodal points in the element besides p_i , thereby ensuring that the function evaluates to 0 on every nodal point p_j with $i \neq j$, in the element. A normalizing factor ensures that the function evaluates to 1 on p_i . Figure 2.3 shows the lines used in the basis functions associated with a point in the 7th-order triangle and square.

Lagrangian elements are widely used today because of computational simplicity, guaranteed C_0 -continuity, and good numerical performance. For this reason, in this study, we choose to work with Lagrangian elements, unless otherwise stated.

2.4 Constructing and Solving the System of Equations

The next step of the Finite Element Method is the construction of the system of equations. One unknown is associated with each of the nodal points, so that if the number of nodal points is n , the resulting system is $n \times n$.

The process starts by assigning to each of the nodal points a global number. This allows us to associate the point p_i with the unknown x_i . Then the system of linear equations is constructed. Of course, both the matrix of the system and the right hand side depend on the partial differential equation that underlies the physical problem.

Let A be the matrix of the system. Then $a_{i,j}$ is the integral of a function that depends on ϕ_i and ϕ_j (the basis functions associated with p_i and p_j) over the entire domain. For example, if the underlying problem involves the computation of the displacement of an elastic structure when a set of forces are acting on it, the matrix A is the *stiffness matrix* of the system. The stiffness matrix is used in several other physical problems (like heat conduction or flow problems), is commonly denoted by K , and can be defined as follows, for 2D problems:

$$K_{i,j} = \int_{\Omega} \left(\frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} \right) d\Omega \approx \int_M \left(\frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} \right) dM$$

where Ω is the domain of the problem and M is the mesh approximating it. We can rewrite this as:

$$K_{i,j} = \sum_{T \in M} \int_T \left(\frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} \right) dT$$

where T is an element of the mesh M . The above equation simply means that the integral can be computed by processing each of the elements in the mesh separately and adding up the results. Notice that since basis functions are defined so that they evaluate to 0, outside elements that contain the nodal points they are associated with, only integrals from elements containing both p_i and p_j contribute to the sum. That is:

$$K_{i,j} = \sum_{\substack{T \in M \\ p_i, p_j \in T}} \int_T \left(\frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} \right) dT$$

The above formula suggests the following procedure for computing K : For each element T in the mesh that contains n' nodal points, compute an $n' \times n'$ matrix K^e , containing the values of the integrals that involve basis functions associated with the points in the element. Let i', j' be the local indices of p_i and p_j (that is, the standard indices that nodal points are given within the element). Then:

$$K_{i',j'}^e = \int_T \left(\frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} \right) dT$$

These matrices are called element matrices. $K_{i,j}$ can then be computed by summing up all the corresponding entries in all the matrices for elements containing both p_i and p_j .

This approach has an added benefit. It is not necessary to compute all different element matrices separately. Instead, for each different *shape* of element occurring in the mesh, we compute one matrix, associated with a *template* element. This process needs to be done only once, and can be done off-line. The element matrices associated with the other elements of the same shape in the mesh can be computed by scaling. So, assembling the matrix of the system from the smaller element matrices is much more efficient than computing the integral separately for every entry, and therefore, this is the process commonly used. For a more detailed explanation on how element matrices are computed and used, refer to [95].

Another important thing to note is that the matrix A is very sparse. If p_i and p_j do not belong to the same element, $a_{i,j}$ is 0. So, we are guaranteed to have only $O(n)$ nonzero entries in the $n \times n$ matrix.

After A has been assembled, it has to be adjusted so that the resulting system conforms to the given boundary conditions. For example, if we are given Dirichlet boundary conditions, the value of the physical quantity we are trying to compute is known for the boundary nodes. In this case, the variables associated with the boundary nodes can be eliminated from the system.

The final step is computing the solution of the system of linear equations. Since, as we have already mentioned, the system is very sparse, direct solvers are rarely used, not even in cases where the same system has to be solved repeatedly with different right-hand sides. In most cases, we use appropriate iterative solvers, for efficiency reasons. After the solution vector has been obtained, the value of the physical quantity we are interested in at any point in the domain (not just the nodal points) can be approximated by computing the value of the linear combination of the basis functions at that point.

2.5 Error Estimates and Mesh Adaptation

After solving the system of linear equations, we obtain an approximation to the solution. Of course, it is also important to know how good this approximation is. For some partial differential equations we have a priori error estimates and, in these cases, we have a guaranteed error bound before applying the method. In most cases, however, these error bounds can only be computed by using specially developed numerical methods after the approximation has been obtained. All these methods give us “local” error bounds, that is, we can compute a different bound for every point in the domain.

It is well known that if a region of the domain is covered with smaller elements, applying the method will result in a better approximation in that region. However, using smaller elements also means that we will need more elements to cover the region and that it will contain more nodal points. That results in a larger system of equations, which is harder to solve. In order to optimize the use of our computational resources, we would like to cover the regions where the solution is easy

to approximate with larger elements, and use small elements only in regions where they are needed.

If the error bounds that we obtain are not satisfactory, a new system has to be generated and solved. The process starts by creating a new mesh with more appropriate element densities. Then, nodal points and basis functions are defined, and the system is assembled and solved. Of course, if the error is still too high, the process can be iterated as many times as necessary.

At each of these iterations we can generate the new mesh by using the mesh generator program with the appropriate parameters, but this is not necessary. It is much more efficient to create the new mesh by modifying the already existing mesh. This is called mesh adaptation and involves two reciprocal processes: *refinement* and *coarsening*.

In *refinement* elements located inside a specified region of the domain are partitioned into smaller elements. The process results in higher element density in the region and is used when the local error is too high. In other regions of the domain, the error bound might be much smaller than the bound we want to achieve. This indicates that the element density of the current mesh in that region is higher than the one actually needed. In this case, we can speed up the computation and (hopefully) still achieve the required error bound by combining several smaller elements to form a larger element. This process is called *coarsening*. Performing the refinement and coarsening operations is challenging because, while modifying the mesh we also want to retain all its characteristics that ensure that the next step of the computation is correct and efficient. By doing successive iterations of adaptive computation we are guaranteed to obtain an approximation within the specified error bound.

Chapter 3

Finite Element Meshes

3.1 Introduction

We have seen that the first step of the Finite Element Method consists of creating a mesh from the description of the domain. Formally, a finite element mesh can be defined as follows:

Definition 3.1.1 *A mesh M of a domain Ω is a set of simple shapes whose union approximates the domain.*

These simple shapes can be, for example, triangles and quadrilaterals for 2D domains or tetrahedra, pentahedra and hexahedra for 3D domains. Ideally, M should be a partition of the domain Ω . For technical reasons, which were discussed in Section 2.2, this many times is impossible. In these cases we would like M to be a partition of a good approximation of the domain. The difficulties associated with representing the domain exactly are the reason why the definition of a mesh does not require that the union of the elements equals the domain.

Definition 3.1.2 *Each one of the simple shapes in M is called an element.*

The process of creating the mesh given the description of the domain is called *mesh generation*. The mesh generated in this stage plays a crucial role in the rest of the computation. It affects the placement of nodal points, the choice of basis functions and the conditioning of the system of linear equations. Using a good mesh enables us to quickly obtain a close approximation to the solution of the partial differential equations. In the rest of this chapter we will talk about the geometric properties of Finite Element meshes and how these properties affect the computation.

3.2 High-Quality Meshes

When creating the mesh M for a given domain Ω we have two goals in mind. First of all, the approximation we will obtain using the mesh should be a good approximation to the actual solution of the partial differential equations, that is, the error bound should be small. Then, the system of linear equations constructed using the mesh should be well-conditioned, and therefore easy to solve. These two properties would ensure that we can obtain the solution both quickly and accurately.

Definition 3.2.1 *A mesh M is a high-quality mesh if it ensures both a well-conditioned system of linear equations and a good approximation to the solution.*

It is not entirely clear what geometric properties a mesh should have in order to be a high-quality mesh, and there is ongoing research in this area. However, experts agree [10, 37] that the mesh should at least have the following characteristics:

- *The union of all the elements must be a good approximation of the problem domain Ω .* This requirement is crucial, since we want the solution to be a good approximation of the solution of the partial differential equation. In particular, the boundary of M should be a good approximation of the boundary of Ω . However, defining what exactly constitutes a good approximation of the boundary is non-trivial and depends on the underlying physical problem.
- *We should be able to have different element density in different regions of the domain.* In general, physical problems tend to have “regions of interest”, where the solution varies significantly. We would like to cover these regions with small elements and use larger elements in regions where the solution is smooth. This contributes both to an accurate solution and to a manageable system of linear equations.
- *The variation in the size between two adjacent elements must be progressive.* In other words, the ratio of the measures of two adjacent elements should belong to a well-controlled range of values. Having neighboring elements that vary too much in size creates numerical problems and results in ill-conditioned systems of linear equations.
- *The elements of the mesh should not have small angles between adjacent facets,* that is, all angles in the mesh should be bounded away from 0. Several theoretical and experimental results show that elements with sharp angles should be avoided because they result in systems that are hard to solve. Specifically, the matrices of these systems have one or more eigenvalues so close to 0, that they cannot be solved reliably using numerical methods. We will call such systems “almost singular”.

In the rest of this work whenever we refer to a mesh as a high-quality mesh we mean that it exhibits the four properties discussed above.

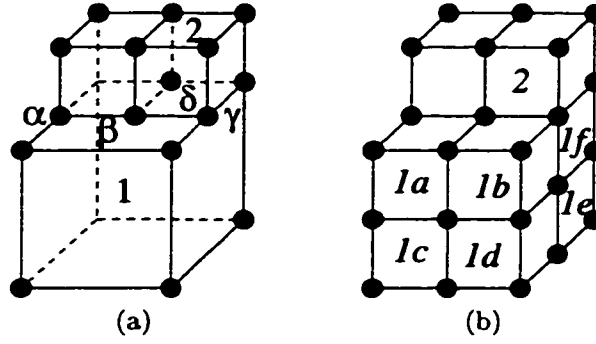


Figure 3.1: Three-dimensional illustration pertaining to the notion of conformity. (a) This mesh is non-conformal, because the intersection of the element marked 1 with the element marked 2 contains only part of the face of element 1. Element 1 is non-conformal and the points marked with α , β and γ are non-conformal points. The point marked with δ (which is in the back of the mesh) is also non-conformal. (b) Partitioning the non-conformal element 1 into eight cubes results in a conformal mesh.

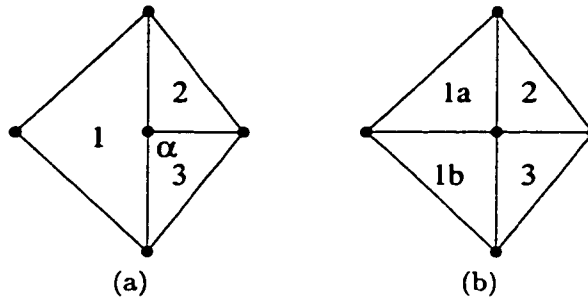


Figure 3.2: Conformity in two dimensions. (a) This mesh is non-conformal, because the intersection of the elements marked 1 and 2 contains only part of the edge of 1. Element 1 is non-conformal and the point marked α is a non-conformal point. (b) In this case also we can make the mesh conformal by partitioning the non-conformal element.

3.3 Conformal vs. Non-conformal Meshes

Another, not necessary but highly desirable, property is *conformity*. We will only give the definition for the three dimensional case, since the corresponding definition for the two dimensional case is easy to infer.

Definition 3.3.1 *M is a conformal mesh of Ω if the following conditions hold:*

1. *All elements of M have a non-empty interior.*
2. *The intersection of two elements in M is either:*
 - (a) *empty*
 - (b) *a point*

(c) an edge

(d) a face

Figure 3.1(a) shows a non-conformal 3D mesh and Figure 3.1(b) shows a conformal mesh covering the same domain. Figure 3.2 shows a two dimensional example.

Definition 3.3.2 *An element is called conformal if its intersection with any one of the other elements of the mesh is one of the four items listed in Definition 3.3.1 and non-conformal otherwise.*

For example, in Figures 3.1(a) and 3.2(a), only the elements marked with the number 1 are non-conformal. The other elements in the two meshes are conformal.

Definition 3.3.3 *We will call a point non-conformal if it is the vertex of one of the elements, but belongs to the interior of an edge or a face of another element of the mesh. Otherwise, we will call the point conformal.*

For example, the point marked α in Figure 3.1(a) is non-conformal, since it is a vertex of element 2 and it is located in the interior of an edge of element 1. Point β on the same figure is also non-conformal because it is a vertex of element 2 and belongs to the interior of a face of element 1. The rest of the points, shown highlighted in the figure, are conformal points.

Both in the two dimensional and in the three dimensional case, non conformal meshes can be made conformal by partitioning non-conformal elements in order to eliminate non-conformal points. (Figures 3.1(b) and 3.2(b)). This operation is non-trivial for several reasons. First of all, partitioning elements arbitrarily might create angles that are too small. In this case, the resulting mesh is not a high-quality mesh any more. Moreover, partitioning elements might create additional non-conformal points that have to be eliminated.

Conformity is not a required property for high-quality meshes, but conformal meshes are preferable to non-conformal meshes. Non-conformal points create mathematical problems, that have to be dealt with by using specialized mathematical techniques. There is ongoing research in this area [26, 5, 29], but all techniques available today are both complex and time-consuming. It is simpler and more efficient to work with conformal meshes whenever possible.

3.4 Structured vs. Unstructured Meshes

Finite Element meshes can be partitioned into two broad categories, *structured* and *unstructured*. To be more precise:

Definition 3.4.1 *A mesh M is called structured if the connection between its vertices is of the finite difference type.*

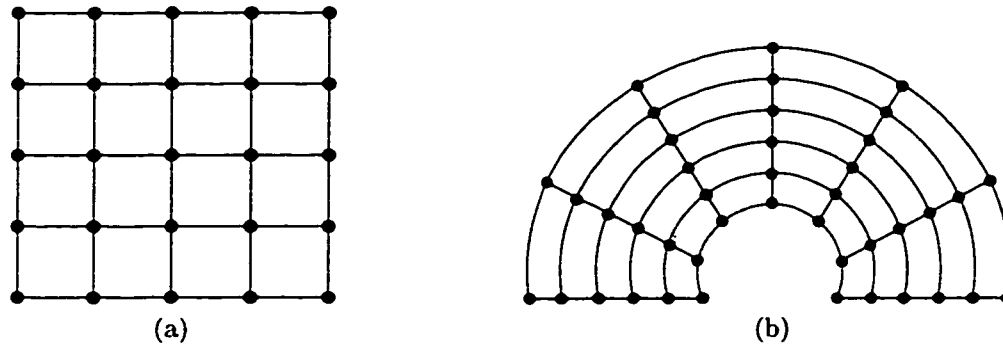


Figure 3.3: (a) A structured mesh on a square domain. (b) This mesh is also structured although it contains elements that have different sizes and shapes. It can however be mapped to a very regular mesh using a simple equation.

The above definitions mean that a structured mesh consists of elements that have the same shape, the same size and are very regularly placed in the domain. Figure 3.3(a) shows an example of a structured mesh. A mesh is also structured if it can be mapped to a very regular mesh using a simple transformation. For example, the mesh in Figure 3.3(b) is structured, although it contains elements with different shapes and sizes.

The biggest advantage of structured meshes is their simplicity. Generating the mesh, given the description of the domain, is trivial. The positions where the vertices of the mesh are placed can be calculated using a simple formula and the indices of the vertices corresponding to an element (as well as their coordinates) can be trivially computed using the index of the element. This implies that information about the elements and vertices does not have to be explicitly stored, since they can be computed as needed. Therefore, using structured meshes results in significant space savings.

Moreover, since the elements all have the same size and shape, they also have the same element matrices associated with them, which we therefore need to compute only once. So, generating the system of equations takes a small fraction of the time needed for the same task if a more complicated mesh is used. And, since element matrices are computed only once, the computation can be done much more accurately. We need to compute the matrices for a template element, but the scaling operation (which is likely to introduce numerical errors) is omitted. So, the final assembled system is more accurate, and therefore, we expect that the obtained approximate solution is closer to the true solution of the partial differential equation.

Structured meshes however also have significant disadvantages. First of all, problems commonly found in real life do not, in general, have a very regular domain. In these cases, structured meshes cannot be used, since they make it impossible to satisfactorily approximate the boundary. Moreover, the requirement that all elements should have the same size and shape is very limiting. Most problems exhibit regions of interest, where the solution changes much more rapidly than in the rest of the domain. If we want to use a structured mesh in such a case, we have to cover the whole domain with a mesh fine enough to give a satisfactory solution at the areas of interest. This, however, results

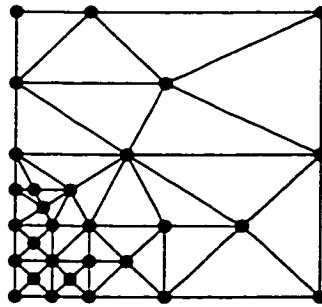


Figure 3.4: An unstructured mesh on the same domain as the structured mesh shown in Figure 3.3(a). Notice that the element density at the lower left-hand corner is much higher than at the rest of the mesh.

in a system with many more unknowns than necessary, which is difficult to solve because of its size.

The most important drawback however, is that when we are presented with a problem, we generally do not know if it has regions of interest or not (since we do not have the solution) or how fine the mesh should be in order to give us a good approximation. Therefore, even in cases where the problem is suitable for a structured mesh, we have no way of knowing it.

So, structured meshes are appropriate only for problems with very regular domains, for which we are sure that they do not exhibit regions of interest. For all other problems, *unstructured* meshes have to be used, i.e. meshes not complying with Definition 3.4.1.

Unstructured meshes consist, in general, of elements that can have different sizes and shapes. For example, an unstructured mesh for a 2D problem might have both triangles and quadrilaterals. Figure 3.4 shows an unstructured mesh covering a square domain. This mesh consists entirely of triangles, but the elements have different sizes and shapes. Notice that the element density at the lower left-hand corner is much higher than at the rest of the mesh. The mesh shown in Figure 2.1 is also an unstructured mesh.

Unstructured meshes are suitable for closely approximating any type of boundary. Moreover, they can be finer within the regions of interest, giving a good solution without giving many more unknowns than necessary. However, the elements and the corresponding vertices can no longer be identified by their indices. Special data structures representing the mesh have to be built and maintained. Another significant disadvantage is that the element matrices have to be computed one by one (since all the elements are different) and therefore, building the system of equations is both more expensive and less accurate.

3.5 Numerical errors

Every step of the Finite Element Method is likely to introduce different types of numerical errors. In this section, we will discuss errors resulting from the mesh used in the computation.

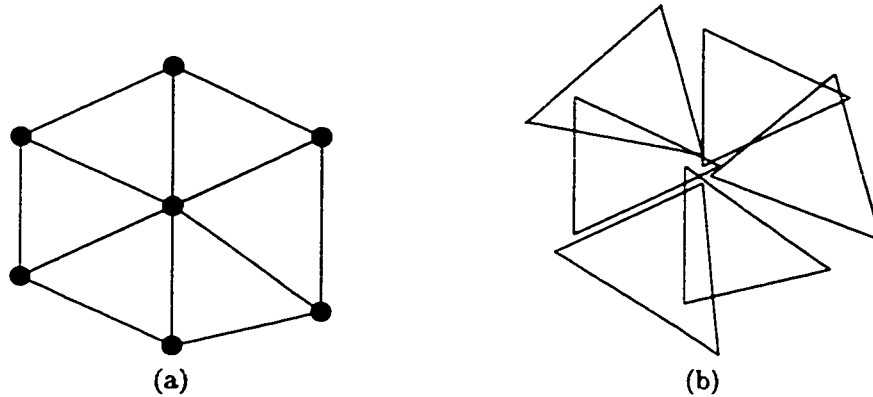


Figure 3.5: Illustration of point placement errors occurring during the mesh generation or mesh adaptation phases. (a) The mesh that would be generated in the absence of numerical errors. (b) The mesh actually used in the computation. This is a result of point placement errors. Notice that this mesh is not topologically consistent.

One type of mesh-related error is the *point placement error* that occurs when the mesh is constructed or modified. In these phases, one of the basic operations performed is constructing new points. The coordinates of these points are calculated by doing operations on the coordinates of already existing points. Doing operations on derived points repeatedly can result accumulating numerical errors. In some cases, the errors can be so large that the resulting mesh is not topologically consistent any more.

Figure 3.5 shows an example. In the absence of numerical errors the mesh constructed by the mesh generator would be the one shown in Figure 3.5(a). Due to point placement errors however, the mesh stored and used in the computation may be the one shown in Figure 3.5(b). Obviously, the solution obtained using this mesh is far less accurate than the one we would obtain by using a topologically consistent mesh.

This problem is so severe that researchers have developed methods that allow us to use meshes with point placement errors. These methods, however, add one more level of complexity to the Finite Element Method and are computationally expensive. The best solution to the problem would be to avoid point placement errors completely.

Another type of numerical error occurs while constructing the system of linear equations. In Section 2.4 we have described how the matrix of the system can be assembled from the smaller element matrices. We also mentioned that all element matrices for elements of the same shape can be obtained by scaling matrices associated with a template element. The matrices of the template element have to be computed only once. This operation can be done off-line and the results are very accurate. The scaling operations however, have to be done separately for each element in the mesh and therefore have to be reasonably fast. These operations introduce numerical errors into the system of equations and affect the accuracy of the solution.

The third source of numerical errors originating from the mesh used is related to the approximation of the domain Ω . Of course, having a “good” approximation of the domain, and specifically of its boundary is important. But, as explained in Section 2.2, the mesh generation step is bound to introduce error in the computation. Moreover, there is no concrete characterization of what a good approximation of the boundary is. We will discuss this issue in Section 6.8.

3.6 Conclusions

From the above discussion it is obvious that we would like to use Finite Element meshes that:

- are high-quality meshes.
- are conformal.
- combine the flexibility of unstructured meshes with the precision and speed obtained when using structured meshes.
- minimize the numerical errors that occur when generating and modifying the mesh (point placement errors).
- minimize the numerical errors that occur when assembling the system of linear equations.

In the next chapter we will present a new type of mesh, which we call *integer-coordinate crystalline mesh* which exhibits all these characteristics.

Chapter 4

Integer-Coordinate Crystalline Meshes

4.1 General Description

Integer Coordinate Crystalline meshes are a new type of mesh designed with two goals in mind. First, we wanted to minimize the mesh related numerical errors in the computation. This includes point-placement errors as well as errors occurring while constructing the system of linear equations. The second goal was to facilitate adaptive computation. We shall return to the latter issue in Chapter 7.

Of course, we also wanted the resulting mesh to be high-quality. Moreover, we wanted to be able to guarantee that the mesh will remain of high-quality after several adaptive iterations. Finally, we wanted to develop a general framework that would allow us to construct and modify both two-dimensional and three-dimensional meshes using the same general approach.

Crystalline meshes are formally defined as follows:

Definition 4.1.1 *A Finite Element mesh is a crystalline mesh if it consists of regular elements whose vertices are positioned on a regular grid.*

The above definition is very general and can accommodate several different types of two-dimensional and three-dimensional meshes. The crystalline meshes we are using have the following additional features:

- *The basic element type is a d -cube, that is, a square for the 2D meshes and a cube for the 3D meshes. We chose to use squares and cubes instead of triangles and tetrahedra (which are another very frequent choice) because they have been shown to perform better for several types of problems [4].*

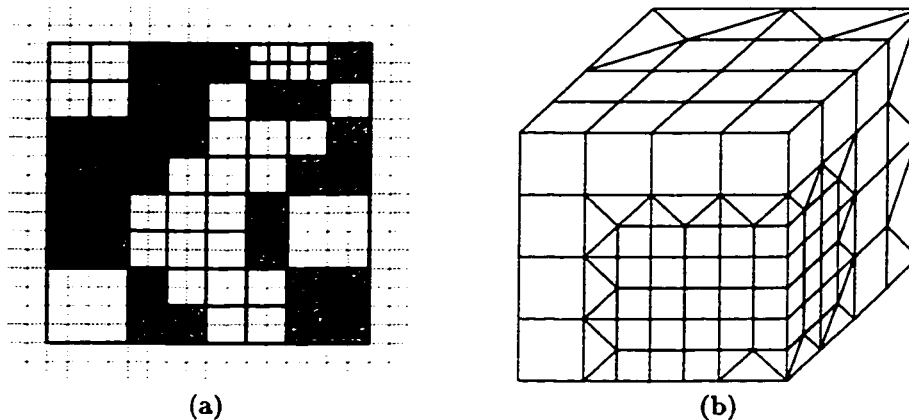


Figure 4.1: Example crystalline meshes. (a) Two-dimensional crystalline mesh. The dotted lines denote the regular grid supporting the mesh. Notice that it exhibits different element density in different regions and that all elements are either squares or right angle isosceles triangles. Transitional elements are shaded. (b) The boundary of a three-dimensional crystalline mesh. In this mesh we also have different element densities in different regions. Both meshes are conformal.

- *We can have basic elements of different size in different regions of the domain.* This allows us to cover “regions of interest” with smaller elements, while using larger elements to cover regions where the solution is smooth.
- *The meshes exhibit smooth transitions in element sizes.* Specifically, if two elements touch (formally, if they share a vertex, an edge or a face) the ratio of their measures belongs to a well-controlled range of values.
- *The meshes are conformal.* For the definition of conformity and a discussion of the advantages of conformal meshes see Section 3.3.
- *The elements used to realize conformal transitions have very few distinct shapes.*

We have already mentioned that we want the elements to be squares (or cubes), that we want to have elements of different sizes in different regions of the domain, and that we want the mesh to be conformal. Conformity implies that in two dimensions, two square elements of different sizes cannot share an edge, since this would create at least one non-conformal point. The same is true for two cubes of different sizes in three dimensions.

We solve this problem as follows: whenever we have two adjacent regions of the domain that are covered with squares or cubes of different sizes, we insert between them a buffering layer that contains elements of different shapes. This layer is responsible for realizing conformal transitions between regions requiring different element density. Since we want our mesh to consist mostly of d-cubes, we would like this buffering layer to be thin. The elements in these layers have only a few distinct shapes.

Figure 4.1 shows two examples: a two dimensional and a three dimensional crystalline mesh. The dotted lines in Figure 4.1(a) represent the regular grid supporting the mesh. They have been omitted in Figure 4.1(b) to avoid cluttering the picture. Both meshes are conformal and exhibit different element densities in different regions of the domain. Notice that in the two dimensional mesh many elements are d-cubes. Elements that have different shapes occur only in the regions between d-cubes of different sizes, in order to guarantee conformity. The same is true (although less visible) for the three dimensional mesh.

We will need the following definitions:

Definition 4.1.2 *An element in a crystalline mesh is a standard element if it is a square (for 2D meshes) or a cube (for 3D meshes).*

Definition 4.1.3 *An element in a crystalline mesh is a transitional element if it is not a standard element.*

In Figure 4.1(a), transitional elements are shaded.

Recall that in Section 3.2 we defined high-quality meshes as those that exhibit four specific properties: good approximation of the shape of the domain, potentially variable element density, controlled variation of size between adjacent elements, and element angles that are bound away from 0. It is clear from the above high level description that crystalline meshes exhibit at least two of these properties. Moreover, the use of only a few pre-specified shapes in crystalline meshes also guarantees that all angles will be bound well away from 0.

We are left with the issue of approximating the domain Ω . It is easy to see that the use of only a few shapes in the mesh restricts the types of boundaries that can be represented exactly. However, it is possible to approximate any type of boundary as closely as needed by using smaller elements next to the boundary. We will return to this issue in Chapter 6.

From the discussion above it is clear that the crystalline meshes described are high-quality meshes. Moreover we can guarantee that they will remain of high-quality after any number of adaptive iterations. Chapter 7 discusses how we can achieve this.

Definition 4.1.4 *A crystalline mesh is called integer-coordinate if the vertices of all its elements have integer coordinates.*

This requirement is not as restrictive as it seems. If the domain Ω is not suitable for such treatment, it can always be scaled appropriately. Using only integer coordinates allows us to use integer arithmetic during the mesh generation and mesh adaptation phases, which alleviates the problem of accumulating numerical errors (point placement errors) that are common in other types of meshes. This guarantees that the resulting meshes will be topologically consistent.

Recall that the other type of mesh-related numerical errors are those that occur during the assembly of the system of linear equations. Their source is the operations used to modify the

element matrices of the template element to account for elements that are deformed or scaled with respect to that template element. Since crystalline meshes only contain elements that have a few pre-specified shapes, we can assemble the matrix using a procedure that minimizes (and in some cases fully eliminates) numerical errors.

Specifically, we have one template element for each shape appearing in the crystalline mesh. The element matrices of the template elements can be computed off-line with very high accuracy. Notice that now no element in the mesh is deformed with respect to the template elements, so operations accounting for that can be completely eliminated. We are left with the operations accounting for scaling. Some of them don't need to be performed at all. For example, all elements of the same shape have the same stiffness matrix regardless of their size. So if the matrix of our system is the stiffness matrix it can be assembled using only addition. In other types of element matrices (for example, in the also very useful mass matrix [95]) we need to account for elements with different sizes. But even in these cases the matrix entries only need to be multiplied or divided by a power of 2. These operations can both be implemented efficiently using shifts. The hope is that they will introduce fewer numerical errors than the operations used to compute the matrices of deformed elements.

Using this technique gives us another advantage. If we are careful when creating or modifying the mesh we can ensure that elements that have both the same shape and the same size are grouped together. In this case, elements belonging to the same group will have exactly the same element matrices. So we can generate the element matrices just once for each group, thereby eliminating many unnecessary operations and taking advantage of the computers cache in an optimal way.

In the next sections we will present different types of crystalline meshes and discuss separately the 2D and the 3D cases.

4.2 Definitions

In this section we will give several definitions that will prove useful when discussing the mesh generation and mesh adaptation steps for crystalline meshes. The notions defined here apply to both 2D and 3D meshes. We will illustrate them using exclusively 2D examples because they are easier to visualize.

Definition 4.2.1 *A mesh cube is a d -cube whose boundary edges belong to the mesh.*

For example the two regions shown shaded in Figure 4.2 are mesh cubes.

Definition 4.2.2 *The capsule $c(x)$ of an element x is the smallest mesh cube containing x .*

Since standard elements are d -cubes whose boundary belongs to the mesh, every standard element coincides with its capsule. The capsule of each transitional element is a d -cube, strictly larger than the element that contains it. We will call capsules containing transitional elements *transitional capsules*.

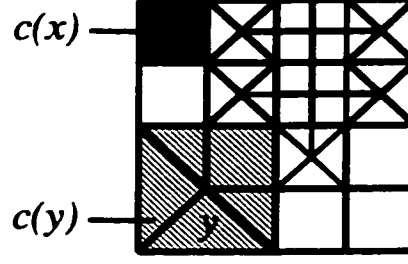


Figure 4.2: 2D illustration of the notion of capsules. The capsules of the elements x and y are shown shaded. x is a standard element, so it coincides with its capsule. y is a transitional element, so its capsule is the smallest enclosing mesh cube.

Figure 4.2 illustrates the notion of capsules. Element x is a standard element (that is, a square) therefore it coincides with its capsule, $c(x)$ (the region shown shaded). Element y is a transitional element (a triangle), so its capsule $c(y)$, which is shown striped here, is strictly larger than y and contains it.

With each of the elements in a crystalline mesh we associate a number, called its *level*. Let L be the length of the largest edge appearing in the mesh. If l denotes the length of the side of a generic mesh cube, the *level* of this cube is $\log_2(L/l) + 1 > 0$. Because of the way crystalline meshes are generated and modified, we can guarantee that this number will always be an integer. We define the level of a standard element to be the level of its capsule, whereas the level of a transitional element is conventionally the level of its capsule incremented by $\frac{1}{2}$.

Note that the level of the elements allows us to distinguish between standard and transitional elements, since only standard elements have level values that are integers. Moreover, the level uniquely identifies the size of a standard element, in the sense that standard elements that belong to the same mesh and have the same level number also have the same size. In general, if we know the level values of two standard elements in the same mesh we also know how their sizes compare, as stated in the following lemma:

Lemma 4.2.1 *Let x, y be two standard elements in the same crystalline mesh with corresponding edge lengths l and l' .*

$$\text{level}(x) = \text{level}(y) + i \text{ if and only if } 2^i l = l', \quad \forall i \in \mathbb{N}$$

Proof: The proof is immediate from the definition of level.

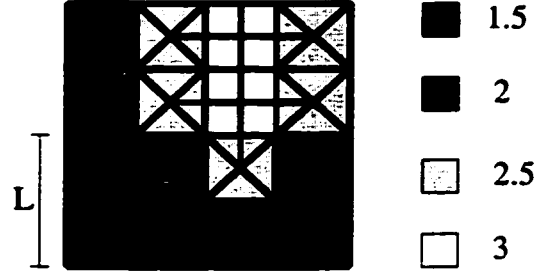


Figure 4.3: The elements in this figure are color coded according to their level values. L denotes the length of the largest edge in the mesh. Notice that the level value distinguishes standard and transitional elements. Also, it uniquely identifies the size of standard elements. This does not hold for transitional elements.

$$\begin{aligned}
 \text{level}(x) &= \text{level}(y) + i \Leftrightarrow \\
 \log_2\left(\frac{L}{l}\right) + 1 &= \log_2\left(\frac{L}{l'}\right) + 1 + i \Leftrightarrow \\
 \log_2(L) - \log_2(l) &= \log_2(L) - \log_2(l') + i \Leftrightarrow \\
 \log_2(l') &= \log_2(l) + i \Leftrightarrow \\
 l' &= 2^i l
 \end{aligned}$$

□

Figure 4.3 shows a crystalline mesh in which the elements have been color-coded according to their level values. L denotes the length of the largest edge in the mesh. Notice that standard elements that have the same level values also have the same size. This is not true for transitional elements.

We are now ready to define different types of crystalline meshes.

Definition 4.2.3 *A rectilinear mesh is a (not necessarily conformal) crystalline mesh inductively defined as follows:*

1. *The coarsest crystalline mesh (consisting of all standard elements of level 1) is rectilinear.*
2. *The mesh resulting from replacing one element of a rectilinear mesh with 2^d standard elements (4 squares in 2D or 8 cubes in 3D) is rectilinear.*

From the above definition it is easy to see that a rectilinear mesh is not necessarily conformal and does not necessarily exhibit smooth transitions in element sizes, since elements with very different sizes can be adjacent. Figure 4.4(a) shows an example of a rectilinear mesh.

We have seen that smooth transitions in element sizes is one of the required properties for high-quality meshes. In order to be able to formally talk about this property we introduce the notion

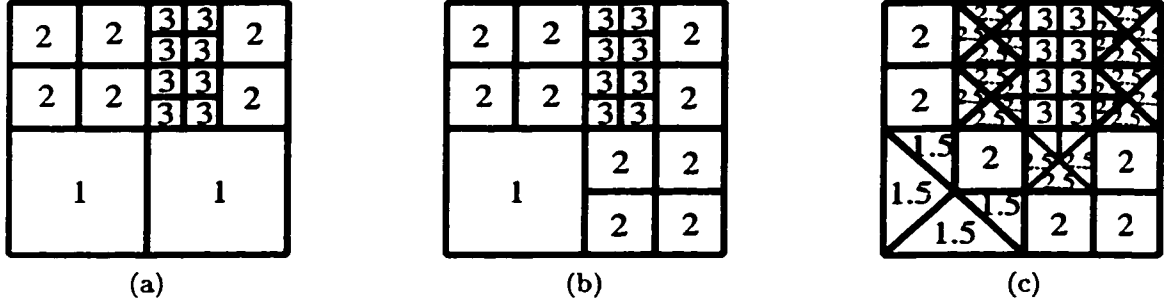


Figure 4.4: Different types of crystalline meshes. Numbers denote the levels of the elements. (a) Rectilinear mesh. Note that one of the level-1 squares has a level-3 neighbor, therefore the mesh is not conformable. (b) A conformable mesh. (c) A legal mesh. This is the mesh that is actually used in the computation.

of *neighboring* elements. We can think of neighboring elements as elements that are placed close together in the domain and share at least one point. The exact definition of the concept is different for 2D and 3D crystalline meshes and we will postpone presenting it until we talk specifically about each of these cases ¹.

Definition 4.2.4 *A conformable mesh is a rectilinear mesh such that the level of each element in the mesh differs by at most 1 from the levels of its neighbors.*

A conformable mesh might be non-conformal but is guaranteed to exhibit smooth transitions in element sizes. Figure 4.4(b) shows a conformable mesh. The mesh shown in Figure 4.4(a) is not conformable since an element with a level value 1 shares part of its edge with two elements with level value 3. Notice that we can always obtain a conformable mesh from a rectilinear mesh by repeatedly partitioning elements that have neighbors that are too small.

Definition 4.2.5 *We say that a crystalline mesh is legal if:*

1. *it is conformal,*
2. *the level of a standard element differs by at most $1/2$ from the levels of its neighbors, and*
3. *the levels of two transitional elements belonging to adjacent capsules differ by at most 1.*

It is easy to see that the second and third condition ensure that neighboring elements will not have sizes that differ by too much. A legal crystalline mesh is conformal and exhibits smooth transitions in element sizes. The meshes constructed during the mesh generation step, as well as the meshes resulting from each adaptation phase are legal, and are the ones used in the actual finite element computations.

¹Specifically, neighbors in 2D are elements that share an edge, while in 3D are elements that share a vertex, for reasons that will be discussed in Chapter 7.

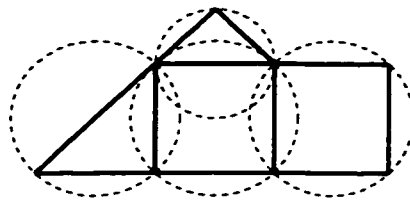


Figure 4.5: A square and all the different neighbors it can have in a crystalline mesh. The circum-circles are shown in dashed lines. Notice that the Delaunay property always holds.

In the example given in Figure 4.4 the legal mesh can be obtained from the conformable mesh by carefully partitioning all non-conformal elements so as to eliminate non-conformal points. This is true in general. Any conformable mesh can be made into a legal mesh by using a one-step procedure that identifies and partitions non-conformal elements.

4.3 2D Crystalline Meshes

We have already seen that in 2D crystalline meshes the standard elements are squares. We need only one type of transitional element: right-angle isosceles triangles. In a 2D crystalline mesh two elements are *neighbors* if they share an edge. For an example of a 2D crystalline mesh see Figure 4.1(a).

It is easy to see that the smallest angle present in any 2D crystalline mesh is 45° and it occurs in the transitional elements. Another property which is very important for 2D meshes is the Delaunay property, which holds when the interior of the circumcircle of any element does not contain any vertices of the mesh [28, 68]. 2D meshes that exhibit the Delaunay property are highly desirable, because the system of equations that results from them is easy to solve.

We can prove that 2D crystalline meshes (that is, conformal 2D meshes that consist only of squares and right-angle isosceles triangles) are Delaunay, which gives us one more reason to believe that they perform well in practice.

Bellow, we will prove this claim:

Lemma 4.3.1 *All two-dimensional crystalline meshes have the Delaunay property.*

Proof: We can prove the lemma by exhaustively examining all possible neighboring configurations. We have already seen that the mesh will consist only of squares and right-angle isosceles triangles, and that it is conformal. We consider each of the shapes and all possible neighbors that the shape could have. If all possible neighboring arrangements have the Delaunay property, the whole mesh will also have the Delaunay property.

Let us first look at the square elements. The only possible neighbors a square could have in a crystalline mesh are either a square of the same size or triangles of two different sizes. Figure 4.5

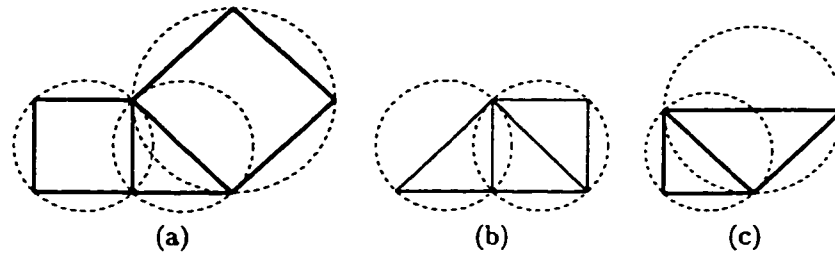


Figure 4.6: All possible neighbors of a right angle isosceles triangle in a crystalline mesh. (a) Squares of two different sizes. (b) Triangles of the same size sharing either the hypotenuse or one of the other sides. (c) Triangles of two different sizes in a side-hypotenuse adjacency. Dashed lines denote the circumcircles. The Delaunay property holds in all cases.

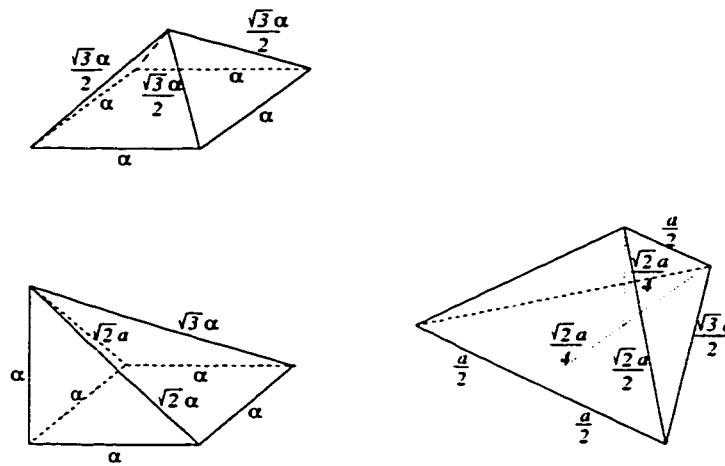


Figure 4.7: The two different types of pyramids and the tetrahedron used as transitional elements.

shows all possible neighbors of the square and the circumcircles of all the elements. It is easy to verify that the Delaunay property holds.

An element that is a right-angle isosceles triangle can have squares of two different sizes as neighbors (Figure 4.6(a)). It can also have triangles of the same size sharing either the hypotenuse or one of the two other sides, or a triangle of different size with a side-hypotenuse adjacency. All possible configurations are shown in Figure 4.6(b) and (c), and we can verify that the Delaunay property holds in all cases. \square

4.4 3D Crystalline Meshes

In three-dimensional crystalline meshes, the standard elements are cubes. Transitional elements are of three types: two distinct types of pyramids and one tetrahedron. The three types of transitional elements are shown in Figure 4.7. 3D crystalline meshes are conformal, and can have elements of

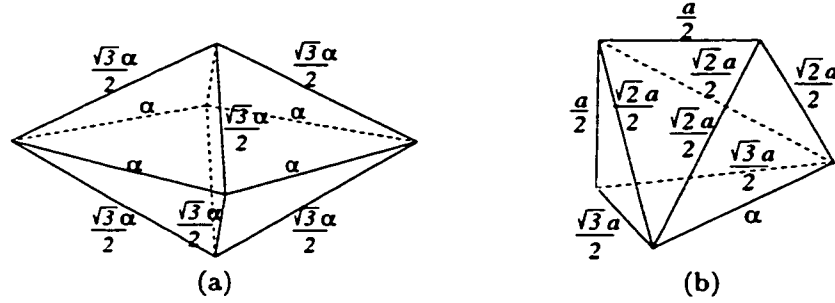


Figure 4.8: The two neighboring arrangements not exhibiting the Delaunay property. (a) Two pyramids sharing the square face. If α denotes the length of the side of the square face, the length of the sides shared between triangular faces is $\frac{\sqrt{3}\alpha}{2}$ and the height of the entire arrangement is α . (b) Two tetrahedra sharing the face without a right angle. We use α to denote the length of one arbitrarily chosen side. The length of the other sides of the arrangement are also shown.

different size in different regions of the domain. We can talk more about how we can guarantee these properties in Chapters 6 and 7 where we discuss mesh generation and mesh adaptation for crystalline meshes. Figure 4.1(b) shows the boundary of a 3D crystalline mesh. Notice that the boundary itself is a legal 2D crystalline mesh.

For the three-dimensional case we chose to slightly modify the definition of a neighbor. Specifically, we say that two elements are *neighbors* if they share at least one *vertex*. This new definition results in meshes containing slightly more elements but allows us to develop simpler algorithms for mesh generation and refinement. Moreover, its use greatly reduces the number of distinct shapes of transitional elements needed to realize conformal transitions between regions with different sizes. One of our earlier attempts to generate conformal 3D crystalline meshes, using the definition that neighbors are elements that share edges, resulted in more than ten different transitional element shapes.

The use of pyramids in 3D meshes is unusual. The main reason for this is that good basis functions for pyramidal finite elements are hard to find. In order to facilitate the use of 3D crystalline meshes we have developed a method for generating Lagrangian basis functions for order- k pyramids, for any $k > 0$ [19]. The method is inductive and can be easily automated. The resulting functions are compatible with the functions used for Lagrangian cubes and tetrahedra [95, 76]. The basis functions and the method used to generate them will be presented in detail in Chapter 5.

By examining all four shapes occurring in a 3D crystalline mesh we can establish that the smallest planar angle is 35° while the smallest dihedral angle is 45° . Moreover, every 3D crystalline mesh will have the Delaunay property almost everywhere: out of 15 possible neighboring arrangements, 13 exhibit the Delaunay property. This result was established by exhaustively examining all possible neighboring configurations by computer, in a way similar to the proof of the corresponding claim for the 2D case. The two non-Delaunay neighboring arrangements are shown in Figure 4.8. Let us note here that the Delaunay property is not considered as important for 3D meshes as it is for

2D meshes, so we expect that having regions in the mesh where it does not hold will not affect the properties of the system of linear equations significantly.

In the next chapter we will talk more about 3D crystalline meshes. We will discuss why we chose to use pyramids and explain how to generate Lagrangian basis functions for them.

Chapter 5

Basis Functions for Pyramidal Finite Elements

5.1 Using Pyramidal Elements in Mixed Meshes

In 3D crystalline meshes the basic element type is the cube. The mesh can have different element densities in different regions of the domain and is conformal (see Definition 3.3.1). Notice that since we want the mesh to be conformal, cubes of different size cannot share a face or an edge. Therefore, if the mesh contains two regions with different element density, we have to insert a transitional layer of non-cubical elements between them. Since we want the vast majority of elements in the mesh to be cubes, we would like this transition layer to contain as few elements as possible. Pyramids are ideal shapes to be used as transitional elements, because they have a square face, which can be conformally attached to a cube, and four triangular faces that can be attached to tetrahedra.

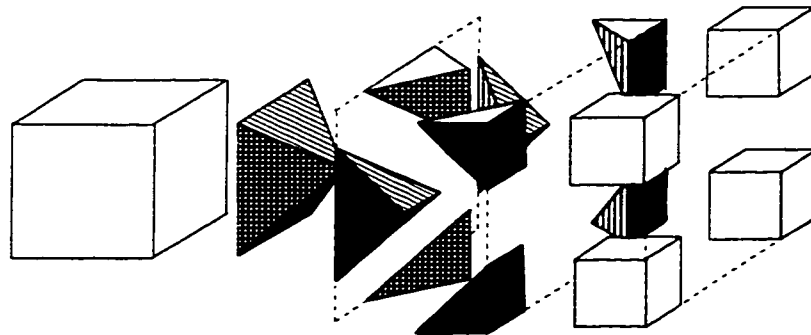


Figure 5.1: Realizing a conformal transition between cubes of two different sizes with five pyramids and four tetrahedra. The resulting surface is partitioned into four square faces, that can be shared with cubes of the next level.

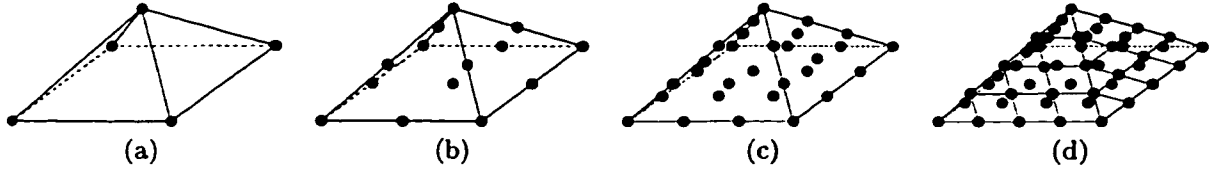


Figure 5.2: Low order Lagrangian pyramids : (a) first order (b) second order (c) third order (d) fourth order.

Assume, for example, that the cube shown in Figure 5.1 to the left, whose edge length is l , is adjacent to a region of the domain that is covered with smaller cubes, whose edges have length $\frac{l}{2}$. We can realize a conformal transition between the two regions as follows. First, we attach the square face of a pyramid to the face of the cube. Then, we add four tetrahedra, each one sharing one of its faces with one of the triangular faces of the pyramid. Finally, we use four more pyramids to fill in the spaces between the tetrahedra. Now we have four square faces whose edges have length $\frac{l}{2}$. We can conformally attach the smaller cubes to these faces. Notice that we realized a conformal transition between the two different sizes of cubes by using a buffering layer which has depth $\frac{l}{2}$.

Pyramids are very useful not only in crystalline meshes, but in any type of mixed 3D meshes that are conformal and contain both parallelepipeds and tetrahedra. However, computational scientists normally avoid using them, due to the difficulty of finding suitable and simple basis functions. In particular, to our knowledge, there is no previous work describing how to construct basis functions for pyramids, so that they can be used in a mixed mesh, together with Lagrangian cubes and tetrahedra. In this chapter we will present an easily automated method that can be used to construct these basis functions.

5.2 Lagrangian Pyramid

The pyramids that we consider in this paper are polyhedra, one face of which is a parallelogram and the other four faces are triangular. The node placement on the faces is the same as on Lagrangian parallelograms and triangles. Since internal nodes are not required for C_0 continuity, we chose not to use any internal points. This simplifies both the actual basis functions and the procedure used in constructing them. Figure 5.2 shows the node placement on Lagrangian pyramids of order 1, 2, 3 and 4.

Using this information, it is easy to count the number of nodes a k th order pyramid has. The base of the pyramid has $(k+1)^2$ nodes, and each of the triangular faces has $\frac{(k+1)(k+2)}{2}$ nodes. We evaluate the total number of nodes by inclusion-exclusion. If we add the above quantities, we count each node on an edge twice, so we have to subtract $(k+1)$ nodes for each edge. Finally, we have to add the five nodes associated with the vertices of the pyramid. Therefore, the total number on nodal points on a k th order pyramid is:

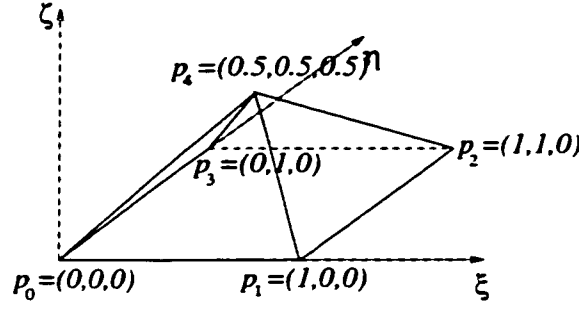


Figure 5.3: The pyramid chosen as a template. Next to each vertex we denote its number and its coordinates.

Vertex number	(ξ, η, ζ)
0	$(0, 0, 0)$
1	$(1, 0, 0)$
2	$(1, 1, 0)$
3	$(0, 1, 0)$
4	$(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$

Table 5.1: The numbering of the vertices of the template pyramid, and their coordinates.

$$(k+1)^2 + 4 \frac{(k+1)(k+2)}{2} - 8(k+1) + 5 = 3k^2 + 2 .$$

5.3 Template Pyramid and Linear Transformation

A common practice in constructing basis functions is to use a standard element (*template*). This element is placed in a different coordinate system (whose axis are labeled ξ , η and ζ) and its vertices have fixed coordinates. Then a bijective linear transformation is provided that maps any element of the same shape (but possibly different size or orientation) to the template element. The reverse transformation can be used to map any basis function for a nodal point of the template to a valid basis function for the corresponding nodal point on any mesh element. The mapping allows us to concentrate on finding basis functions for the template element, since the transformation can be used to obtain basis functions for all other elements corresponding to a given shape.

The template pyramid we chose to use is shown in Figure 5.3. The same figure also shows the standard numbering assigned to each vertex, and the coordinates of the vertex. This information is summarized in Table 5.1. The edges of the pyramid are also numbered according to convention. The numbering, as well as the equations of the line defined by each edge, are shown in Table 5.2. Each face of the template pyramid defines a plane. The equations of these planes are shown in Table 5.3.

In crystalline meshes there are only two types of pyramids (shown in Figure 4.7). For reasons of

Edge num.	Vertices on edge	Equation of the line
0	0,1	$\zeta = 0$ and $\eta - \zeta = 0$
1	1,2	$\zeta = 0$ and $\xi + \zeta - 1 = 0$
2	2,3	$\zeta = 0$ and $\eta + \zeta - 1 = 0$
3	3,0	$\zeta = 0$ and $\xi - \zeta = 0$
4	0,4	$\eta - \zeta = 0$ and $\xi - \zeta = 0$
5	1,4	$\eta - \zeta = 0$ and $\xi + \zeta - 1 = 0$
6	2,4	$\xi + \zeta - 1 = 0$ and $\eta + \zeta - 1 = 0$
7	3,4	$\eta + \zeta - 1 = 0$ and $\xi - \zeta = 0$

Table 5.2: The numbering of the edges on the template pyramid, the vertices that define them, and the equations defining the line associated with them.

Face number	Vertices defining face	Equation of the plane
0	0,1,2,3	$\zeta = 0$
1	0,1,4	$\eta - \zeta = 0$
2	1,2,4	$\xi + \zeta - 1 = 0$
3	2,3,4	$\eta + \zeta - 1 = 0$
4	3,0,4	$\xi - \zeta = 0$

Table 5.3: The numbering of the faces of the template pyramids, the vertices defining each one of them, and the equation of the plane defined by the face.

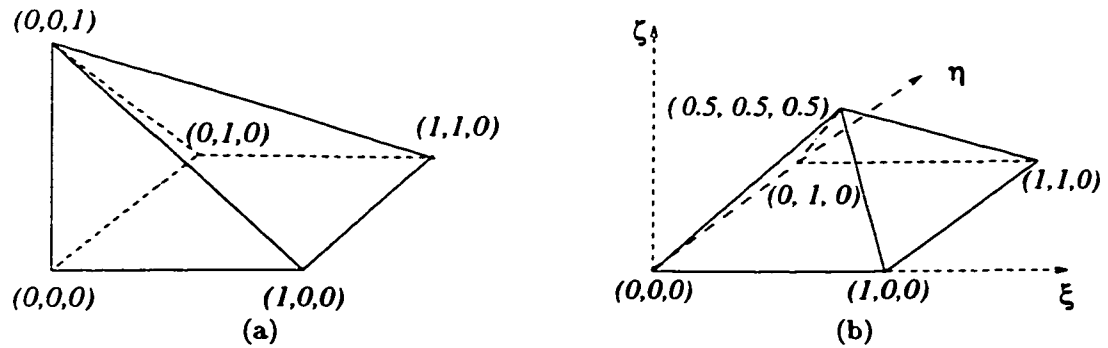


Figure 5.4: (a) A deformed pyramid in the coordinate system (x, y, z) . (b) The template pyramid in the coordinate system (ξ, η, ζ)

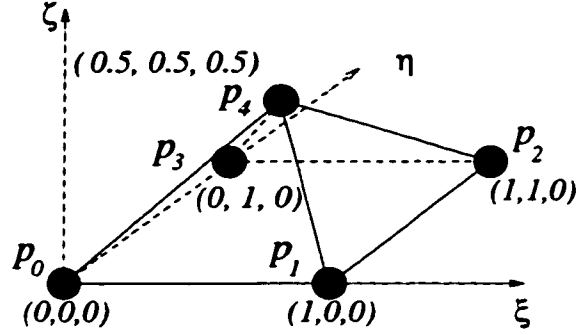


Figure 5.5: First order template pyramid.

efficiency and accuracy we chose to use two template elements, one for each distinct pyramid shape in the mesh. Both of these templates are shown in Figure 5.4. For the rest of this chapter we can focus on the template shown in Figure 5.4(b), since results obtained for it can be extended to the modified one by using the bijection:

$$\begin{aligned}\xi &= x + \frac{1}{2}z, \\ \eta &= y + \frac{1}{2}z, \\ \zeta &= \frac{1}{2}z.\end{aligned}$$

Once we have basis functions for the two template elements, we can obtain basis functions for any other pyramid in the mesh by using standard scaling, translation and rotation operations¹.

5.4 Basis Functions for the Order-1 Pyramid

Now that we have described the template pyramid and the mappings that can be used to find basis functions for any pyramid, given basis functions for the template, we will try to find basis functions for the template pyramid.

Since all Lagrangian elements we have seen so far have polynomial basis functions, it is natural to first try to find polynomial basis functions for the pyramid. Let us consider the first order pyramid (Figure 5.5). By using a method analogous to the one used for the standard Lagrangian elements, we can try to construct the basis functions as a product of planes. However, regardless of which planes we use, the resulting functions do not fulfill one or more of the requirements, and therefore cannot

¹In general, for every pyramid P with a parallelogram base, there is a bijection that maps it to the template pyramid and can be used to compute basis functions

be used as basis functions. This is a property that characterizes pyramidal elements. Specifically, we can prove the following theorem:

Theorem 5.4.1 *It is impossible to construct polynomial basis functions for the first order Lagrangian pyramid.*

Proof: In order to prove the theorem, it suffices to show that it is impossible to construct a basis function for one of the nodal points. We arbitrarily choose p_0 . Based on the discussion in Section 2.3, we know that ϕ_0 should:

- reduce to 0 on the two triangular faces that do not contain p_0 .
- reduce to a linear function on the two triangular faces containing p_0 (since the basis functions for the nodal points of first-order triangles are linear).
- reduce to a bilinear function on the square face (since the basis functions for nodal points on first-order squares are bilinear).

Since the polynomial should reduce to 0 on the two triangular faces not containing p_0 , it should contain the factors $(\xi + \zeta - 1)$ and $(\eta + \zeta - 1)$ which, when equated to 0, are the equations of the planes defined by the two faces. On the triangular face containing the vertices 0, 1 and 4, which defines the plane $\eta - \zeta = 0$ this term reduces to:

$$(\xi + \eta - 1)(2\eta - 1) = 2\xi\eta - \xi + 2\eta^2 - \eta - 2\eta + 1 = 2\xi\eta - \xi + 2\eta^2 - 3\eta + 1$$

which contains a square term. Since the function is required to reduce to a linear function on this plane, this is enough to prove that a polynomial function for p_0 does not exist. \square

However, there is no reason to restrict the basis functions to be polynomials. In order to enforce C_0 continuity, we only have to ensure that the basis functions reduce to 0 on the faces not containing the corresponding nodal point, and to the correct 2D basis function on the faces containing it. Indeed, if we allow the basis functions to be ratios of polynomials, we can construct basis functions for a template pyramid of order k , for any integer $k > 0$.

We will start by finding basis functions for the first order pyramid (Figure 5.5). This pyramid has five nodal points, each one of which coincides with one of its vertices. Let $\phi_i(\xi, \eta, \zeta)$ be the function associated with the point $p_i = (\xi_i, \eta_i, \zeta_i)$. Then $\phi_i(\xi, \eta, \zeta)$ should satisfy all the general requirements for 3D Lagrangian basis functions. In particular, it should:

- evaluate to 1 on p_i and to 0 on all nodal points p_j with $i \neq j$.
- reduce to 0 on the faces not containing p_i .
- reduce to the appropriate 2D first order basis function on the faces containing p_i .

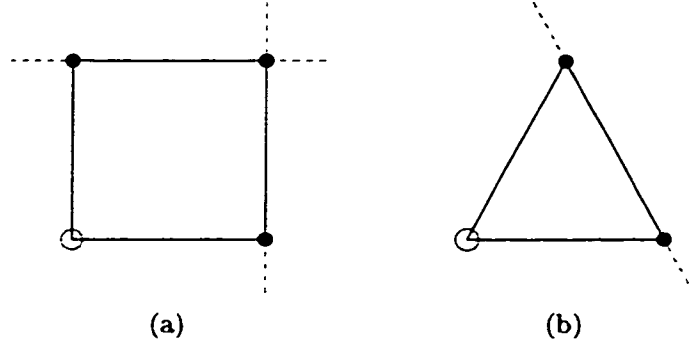


Figure 5.6: Lagrangian basis functions for (a) the first order square and (b) the first order triangle.

basis functions	face 0 $\zeta = 0$	face 1 $\eta - \zeta = 0$	face 2 $\xi + \zeta - 1 = 0$	face 3 $\eta + \zeta - 1 = 0$	face 4 $\xi - \eta = 0$
$\phi_0(\xi, \eta, \zeta)$	$(\xi - 1)(\eta - 1)$	$\xi + \eta - 1$	0	0	$\xi + \eta - 1$
$\phi_1(\xi, \eta, \zeta)$	$\xi(\eta - 1)$	$\xi - \eta$	$\xi - \eta$	0	0
$\phi_2(\xi, \eta, \zeta)$	$\xi\eta$	0	$\xi + \eta - 1$	$\xi + \eta - 1$	0
$\phi_3(\xi, \eta, \zeta)$	$(\xi - 1)$	0	0	$\xi - \eta$	$\xi - \eta$
$\phi_4(\xi, \eta, \zeta)$	0	η	$\xi - 1$	$\eta - 1$	ξ

Table 5.4: Functions to which each basis function should reduce on each of the faces.

In Section 2.3 we have mentioned that Lagrangian basis functions for triangles and squares are polynomials that can be factored into k and $2k$ linear terms, respectively. For ease of description, and with a slight abuse of language, we shall refer to geometric constructs (lines, planes) rather than to “linear terms describing geometric constructs” for the rest of this chapter. If $k = 1$, the basis functions associated with nodal points on a triangle are lines. The basis functions for the nodal points on the square are the product of two lines (Figure 5.6).

From the information given above, we can deduce all the properties the basis functions of the first order pyramid should have. For example, the basis function $\phi_0(\xi, \eta, \zeta)$ (which is associated with $p_0 = (0, 0, 0)$) should evaluate to 0 on faces $p_1p_2p_4$ and $p_2p_3p_4$, and reduce to a line passing through p_1 and p_4 on face $p_0p_1p_4$ and to a line passing through p_3 and p_4 on face 4. On the square face of the pyramid, ϕ_0 should reduce to the product of two lines, one passing through p_1 and p_2 and the other passing through p_2 and p_3 . The requirements for ϕ_1 , ϕ_2 and ϕ_3 are similar. Finally ϕ_4 should reduce to 0 on the square face of the pyramid, and to a line passing through the two points on the base on each of the triangular faces. A summary of what each of the basis functions should reduce to on each of the faces can be found in Table 5.4. Notice that the reduction of the function might be multiplied by a constant, which is used to force the function to evaluate to 1 on the associated nodal point.

Now that the requirements for each of the basis functions are clear, we are ready to state the following theorem:

Theorem 5.4.2 *There exist basis functions for the first order Lagrangian pyramid which are ratios of polynomials.*

Proof: The following functions exhibit all the required properties:

$$\phi_i(\xi, \eta, \zeta) = (-1)^{i+1} \frac{(\xi + (-1)^{\xi_i} \zeta - 1 + \xi_i)(\eta + (-1)^{\eta_i} \zeta - 1 + \eta_i)}{2\zeta - 1}, \quad i = 0..3$$

$$\phi_4(\xi, \eta, \zeta) = 2\zeta$$

□

Each of the functions for the nodal points on the base of the pyramid is a ratio of polynomials. The two terms on the numerator are the two equations of the planes (defined by two triangular faces of the pyramid) where the function should evaluate to 0. The term in the denominator ensures that they will reduce to a *linear* function on the triangular faces containing the point. Notice that the denominator evaluates to 0 when $\zeta = \frac{1}{2}$, and that the only point on the pyramid where this happens is its fourth vertex. However, on this point, the value of the numerator goes faster to 0 than the value of the denominator. So the value of the functions at this point is 0. In fact, for example:

$$\begin{aligned} \lim_{\zeta \rightarrow 1/2} \phi_0\left(\frac{1}{2}, \frac{1}{2}, \zeta\right) &= \lim_{\zeta \rightarrow 1/2} \left(-\frac{(\frac{1}{2} + \zeta - 1)(\frac{1}{2} + \zeta - 1)}{2\zeta - 1} \right) \\ &= \lim_{\zeta \rightarrow 1/2} \left(-\frac{(\zeta - \frac{1}{2})(\zeta - \frac{1}{2})}{2\zeta - 1} \right) \\ &= \lim_{\zeta \rightarrow 1/2} \left(-\frac{(\zeta - \frac{1}{2})(\zeta - \frac{1}{2})}{2(\zeta - \frac{1}{2})} \right) \\ &= \lim_{\zeta \rightarrow 1/2} \left(-\frac{1}{2} \left(\zeta - \frac{1}{2} \right) \right) \\ &= -\frac{1}{2} \lim_{\zeta \rightarrow 1/2} \left(\frac{1}{2} - \frac{1}{2} \right) \\ &= 0 \end{aligned}$$

5.5 Generalizing to the Order- k Pyramid

We now describe an inductive construction process, which, taking as basis the order-1 basis functions, and assuming as a premise the order- $(k-1)$ functions, produces the desired order- k basis functions. We begin by classifying the nodal points of the order- k pyramid according to their common requirements. Specifically, we shall distinguish and treat separately:

1. points not on the base (the simpler case);

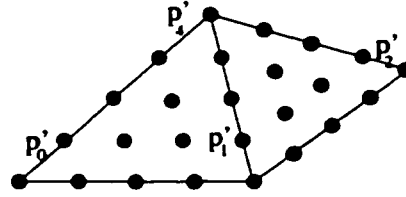


Figure 5.7: A k th order pyramid for $k = 4$. The nodal points not on the base form a order- $(k - 1)$ pyramid. The nodes that are vertices of the order- $(k - 1)$ pyramid are shown marked.

2. points on the base.

We begin with the points not on the base.

Lemma 5.5.1 *Given basis functions for the $(k-1)$ -st order pyramid which are ratios of polynomials, we can construct basis functions, also ratios of polynomials, for nodal points not on the base of the order- k pyramid.*

Proof: The nodal points not lying on the base (that is, the points for which $\zeta \neq 0$) form a (non-template) order- $(k - 1)$ pyramid (see Figure 5.7). We can map to the present spatial configuration, as indicated in Section 5.3, the basis functions for the order- $(k - 1)$ template pyramid (inductively assumed to be available).

In order to compute the mappings, we need the coordinates of three points on the base of the pyramid (p'_0, p'_1, p'_3), and the coordinates for the fourth point at the apex of the pyramid (p'_4). Using the fact that points on the edges are placed at regular intervals and the equations of the planes defining each of the edges (Figure 5.7), we can easily compute the coordinates of these four points to be:

$$\begin{aligned} p'_0 &= \left(\frac{1}{2k}, \frac{1}{2k}, \frac{1}{2k} \right) \\ p'_1 &= \left(\frac{2k-1}{2k}, \frac{1}{2k}, \frac{1}{2k} \right) \\ p'_3 &= \left(\frac{1}{2k}, \frac{2k-1}{2k}, \frac{1}{2k} \right) \\ p'_4 &= \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2} \right) \end{aligned}$$

Substituting these coordinates to the equations defining the mapping we obtain:

$$\begin{aligned}
x &= \frac{1}{2k} + \left(\frac{k-1}{k}\right) \xi \\
y &= \frac{1}{2k} + \left(\frac{k-1}{k}\right) \eta \\
z &= \frac{1}{2k} + \left(\frac{k-1}{k}\right) \zeta
\end{aligned}$$

So, we can obtain the basis functions for the points on the k th order pyramid by substituting:

$$\begin{aligned}
\xi' &= \frac{2k\xi - 1}{2k - 2} \\
\eta' &= \frac{2k\eta - 1}{2k - 2} \\
\zeta' &= \frac{2k\zeta - 1}{2k - 2}
\end{aligned}$$

for ξ , η and ζ respectively.

The functions so obtained have most of the required properties: they reduce to 0 on the triangular faces not containing the associated nodal point, evaluate to 1 on the latter, and to 0 on all the other nodal points lying above the base. However, they have two shortcomings: they don't reduce to 0 on the base of the pyramid, and, on the triangular faces containing the associated point they reduce to a polynomial that is the product of $k - 1$ (instead of k) lines. Fortunately, this inadequacy can be remedied by multiplying the functions by ζ . Since this multiplication causes the basis function ϕ_i to evaluate to ζ_i on its associated point p_i , the function must be normalized by the factor $\frac{1}{\zeta_i}$. \square

We now turn our attention to the more complex case of nodal points on the pyramid's base. Recall that the placement of these points is identical to the placement of points on the k th order square. Specifically, the points are placed on the vertices of a grid that is formed by connecting nodal points located on opposite edges of the square. The base of the pyramid contains $(k + 1)^2$ nodal points in total.

In order to take advantage of the very regular placement of nodal points on the base, we will use a slightly different terminology. Specifically, we will use $p_{i,j}$ ($0 \leq i, j \leq k$) to denote the nodal point that is located in the i th column and j th row of the nodes (Figure 5.8). Obviously,

$$p_{i,j} = \left(\frac{i}{k}, \frac{j}{k}, 0\right)$$

The basis function $\phi_{i,j}(\xi, \eta, \zeta)$ (corresponding to the point $\phi_{i,j}$), when set equal to 0, defines a surface in three-dimensional space, which we shall specify by means of its intersections with the pyramid faces that contain the given point (either in their interior or on their boundary). Due to

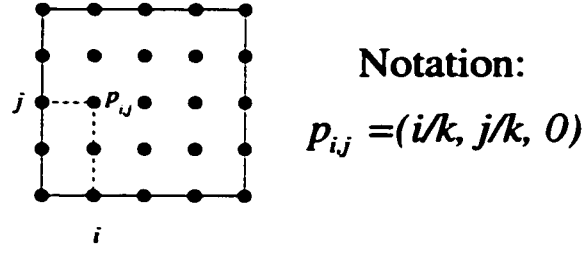


Figure 5.8: We will use $p_{i,j}$ to denote the nodal point with coordinates $(\frac{i}{k}, \frac{j}{k}, 0)$.

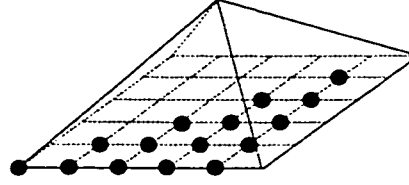


Figure 5.9: Set of nonequivalent points on the base of a 5th order pyramid.

the symmetries of the square, we shall limit our considerations to nonequivalent points. These are points with $0 \leq j \leq i < k$, as illustrated in Figure 5.9. The basis functions for the rest of the points can be obtained by using the mapping given in Section 5.3 to rotate the pyramid.

Specifically, the following equations hold for all points on the base of the pyramid ($0 \leq i, j \leq k$):

$$\begin{aligned}\phi_{i,j}(\xi, \eta, \zeta) &= \phi_{j,k-i}(\eta, 1 - \xi, \zeta) \\ \phi_{i,j}(\xi, \eta, \zeta) &= \phi_{k-i,k-j}(1 - \xi, 1 - \eta, \zeta) \\ \phi_{i,j}(\xi, \eta, \zeta) &= \phi_{k-j,i}(1 - \eta, \xi, \zeta)\end{aligned}$$

Figure 5.10 shows the three rotations we will use to obtain the basis functions associated with nodal points that are equivalent to the ones shown in Figure 5.9. For example, the basis functions of the points on the edge $p_{k,0}p_{k,k}$ can be obtained by using the first rotation shown in Figure 5.10(b). For the points on edge $p_{k,k}p_{0,k}$ we can use the rotation shown in Figure 5.10(c).

Our objective is to design functions $\phi_{i,j}(\xi, \eta, \zeta)$ that are ratios of polynomials. Their numerators, therefore, will contain as factors polynomials (called *primitives*), each of which, equated to 0, defines a three-dimensional surface (referred to as a *primitive construct*). By Property 5 of basis functions and the specific properties of Lagrangian elements, the intersection of any such primitive construct with a surface of the pyramid must be a line where the reduction of $\phi_{i,j}$ has value 0. We shall first describe the various primitive constructs to be used.

Definition 5.5.1 For each i , $i = 0, 1, \dots, k$, let

$$\xi_i^L = \xi + \zeta - \frac{i}{k} \quad \text{and} \quad \xi_i^R = \xi - \zeta - \frac{i}{k}.$$

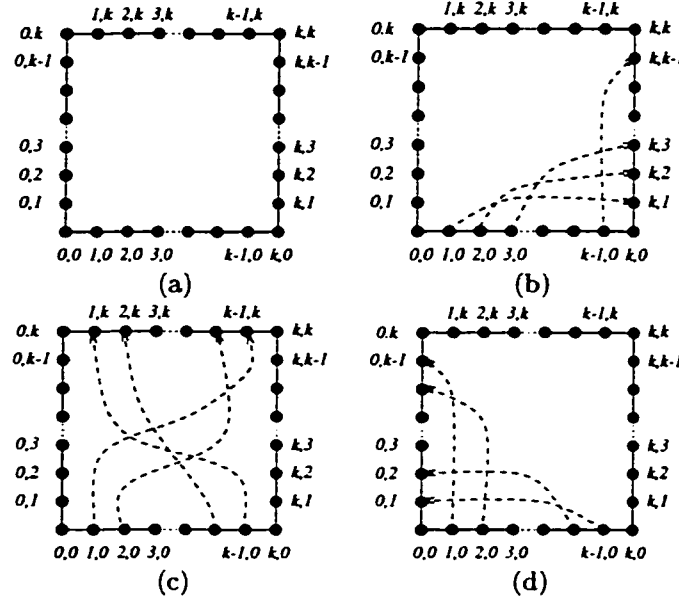


Figure 5.10: Mapping of the points on the edges. The points that are mapped to each other are connected by a dashed line. (1) Numbering of the points on the edges. (b) Mapping resulting in from the first rotation. (c) Mapping resulting in from the second rotation. (d) Mapping resulting in from the third rotation.

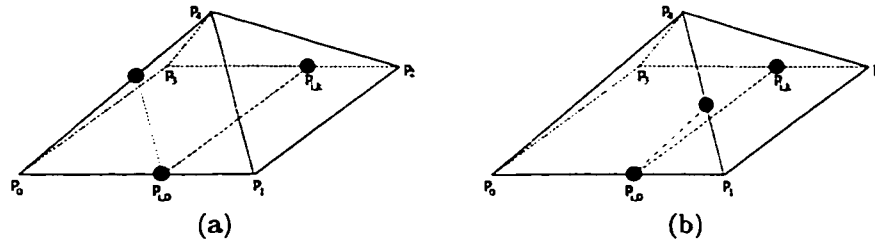


Figure 5.11: The functions ξ_i^L and ξ_i^R define planes. This figure shows the lines that the functions reduce to on the relevant faces of the pyramid. (a) Reduction of ξ_i^L on the base and the triangular face $p_0 p_1 p_4$. (b) Reduction of ξ_i^R on the base and the triangular face $p_0 p_1 p_4$. Notice that both functions reduce to the same line on the base of the pyramid. Moreover, all three lines are primitives used in the construction of the basis functions for 2D Lagrangian triangles and squares.

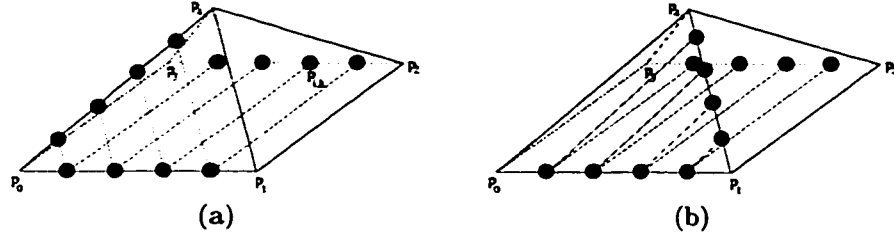


Figure 5.12: (a) Reduction of Ξ^L on the square and front triangular faces of the pyramid (faces 0 and 1). (b) Reduction of Ξ^R on the same faces. Both functions result in a set of parallel lines on both faces.

Both $\xi_i^L = 0$ and $\xi_i^R = 0$ describe planes and reduce on the base to the line $\xi - \frac{i}{k} = 0$ passing by $p_{i,0}$ and $p_{i,k}$. On face $p_0p_1p_4$ of the pyramid (in the plane $\eta - \zeta = 0$) they reduce to the lines $\xi + \eta - \frac{i}{k} = 0$ and $\xi - \eta - \frac{i}{k} = 0$, respectively. We conclude that ξ_i^L and ξ_i^R are valid primitives, since the latter lines, each of which is parallel to one of the edges of the triangular face, are primitives for 2D Lagrangian basis functions. Figure 5.11 shows the lines $\xi_i^L = 0$ and $\xi_i^R = 0$ reduce to, on the square and front triangular faces of the pyramid.

By the symmetry of the ξ and η directions, we analogously define:

Definition 5.5.2 For each j , $j = 0, 1, \dots, k$, let

$$\eta_j^L = \eta + \zeta - \frac{j}{k} \quad \text{and} \quad \eta_j^R = \eta - \zeta - \frac{j}{k}.$$

Adapting the preceding discussion to this situation, both η_j^L and η_j^R are verified to be valid primitives.

Convenient additional constructs (corresponding to sheaves of contiguous parallel planes) are given in the following definition:

Definition 5.5.3 Let

$$\Xi^L = \prod_{i=1}^{k-1} \xi_i^L, \quad \Xi^R = \prod_{i=1}^{k-1} \xi_i^R, \quad H^L = \prod_{j=1}^{k-1} \eta_j^L, \quad \text{and} \quad H^R = \prod_{j=1}^{k-1} \eta_j^R.$$

Figure 5.12 shows the reduction of Ξ^L and Ξ^R on the relevant faces of the pyramid. The reduction for H^L and H^R is analogous.

Our catalog of primitive constructs is completed by a set of quadric surfaces:

Definition 5.5.4 For each i , $0 \leq i \leq k$ let

$$\begin{aligned} q_i &= 2\zeta(\xi + \eta - \frac{i}{k}) - (\xi + \zeta - \frac{i}{k})(\eta + \zeta - \frac{i}{k}) \\ &= -\zeta^2 - \xi\eta + \xi\zeta + \eta\zeta + \xi\frac{i}{k} + \eta\frac{i}{k} - \frac{i^2}{k^2} \end{aligned}$$

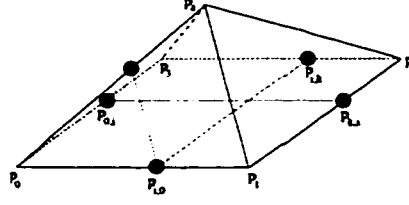


Figure 5.13: The quadric q_i reduces (intersects the pyramid) to two perpendicular lines in the base. On the two triangular faces containing p_0 it reduces to a line.

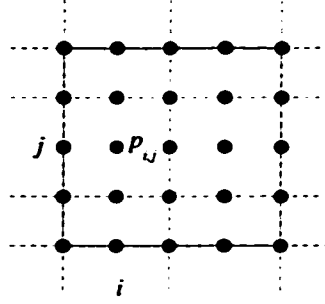


Figure 5.14: The lines resulting from factoring the reduction of $\phi_{i,j}$ on the plane $\zeta = 0$.

Notice that $q_i = 0$ belongs to the pencil of quadrics sharing four lines in 3D space, specifically the lines containing $p_{i,0}p_{i,k}$ and $p_{0,i}p_{k,i}$ in the base plane, the line by $p_{i,0}$ parallel to edge p_1p_4 on face $p_0p_1p_4$, and the line by $p_{0,i}$ parallel to edge p_3p_4 on face $p_0p_4p_3$. All four lines are primitives in the corresponding Lagrangian squares and triangles, so that the quadric is a valid primitive construct. Whereas the generators of the pencil are degenerate quadrics (pairs of planes), $q_i = 0$ is a hyperbolic paraboloid. Figure 5.13 shows the lines q_i reduces to on three of the faces of the pyramid (square base, and the two triangular faces containing $p_{0,0}$).

Using the defined primitives we can now tackle the construction of the functions $\phi_{i,j}(\xi, \eta, \zeta)$.

Lemma 5.5.2 *There exist basis functions that are ratios of polynomials for nodal points lying on the base of the order- k pyramid.*

Proof: We shall restrict the analysis to nonequivalent points, since, from the basis function for each of the chosen representatives, by standard linear transformation of coordinates, we can generate the basis function for any point on the pyramid's base.

We distinguish three cases:

1. *Points in the interior of the base ($p_{i,j}$, $0 < j \leq i < k$). Such points belong to a single face (the base) of the pyramid.*

By Property 4 given in Section 2.3, the basis function $\phi_{i,j}$ must reduce to 0 on all triangular

faces of the pyramid and, therefore, it must contain the four factors $\xi_0^R, \xi_k^L, \eta_0^R, \eta_k^L$ corresponding to the four planes of the lateral faces. By Property 5 (Section 2.3), on the base it must reduce to the appropriate two-dimensional basis function for the corresponding nodal point on the Lagrangian square. As discussed in Section 2.3 the latter is a product of $2k$ lines, covering all the columns and rows defined by the nodal points different from $p_{i,j}$, i.e., to the product

$$\left(\prod_{\substack{l=0 \\ l \neq i}}^k \xi_l^R \prod_{\substack{m=0 \\ m \neq j}}^k \eta_m^R \right)_{\zeta=0}$$

It follows that a valid expression for ϕ_{ij} is:

$$\phi_{i,j} = c_{i,j} \xi_0^R \xi_k^L \eta_0^R \eta_k^L \frac{\Xi^R H^R}{\xi_i^R \eta_j^R}$$

where $c_{i,j}$ is a normalizing factor enforcing the value 1 at $p_{i,j}$. The exact value of the factor can be easily computed as being:

$$c_{i,j} = (-1)^{i+j} \frac{k^{2k}}{i!(k-i)!j!(k-j)!}$$

2. *Points in the interior of an edge* (p_{i0} , $i > 0$). Such points belong to two faces, the base and the face on plane $\xi - \zeta = 0$. By Property 4, $\phi_{i,0}$, $i > 0$, must reduce to 0 on each of the three triangular faces not containing the edge, i.e., it must contain the factors $\xi_0^R = \xi - \zeta$, $\xi_k^L = \xi + \zeta - 1$, and $\eta_k = \eta + \zeta - 1$. By Property 5, on the triangular face containing the point, the function reduces to the product of k lines, represented by the expression:

$$\left(\prod_{j=0}^{i-1} \xi_j^R \prod_{j=i+1}^k \xi_j^L \right)_{\eta=\zeta}$$

and, on the base, to the product of $2k$ lines, represented by the expression:

$$\left(\prod_{j=0}^{i-1} \xi_j^R \prod_{j=i+1}^k \xi_j^L \eta_k^L \prod_{j=1}^k \eta_j^R \right)_{\zeta=0}$$

The union of these two sets of factors satisfy the requirements of both faces. However in the reduction to face $p_0 p_1 p_4$ ($\eta = \zeta$), factor η_k^L yields an unwanted $(2\eta - 1)$ factor. This term, however, can be cancelled by a divisor $(2\zeta - 1)$, which does not affect the reduction to the base. We conclude that the following expression satisfies all requirements:

$$\phi_{i,0}(\xi, \eta, \zeta) = c_{i,0} \frac{\xi_0^R \xi_k^L \eta_k^L}{2\zeta - 1} H^R \prod_{j=1}^{i-1} \xi_j^R \prod_{j=i+1}^{k-1} \xi_j^L$$

where, again, $c_{i,0}$ is a normalizing constant with value:

$$c_{i,0} = (-1)^{i+1} \frac{k^{2k-1}}{((k-1)!)^2}$$

3. *Point on the vertex* ($p_{0,0}$). Such points belong to three faces, the base and the two lateral faces on planes $\xi - \zeta = 0$ and $\eta - \zeta = 0$. By Property 4, $\phi_{0,0}$ must reduce to 0 on each of the two triangular faces not containing the point, i.e, it must contain the factors ξ_k^L and η_k^L . By Property 5, it must also reduce to a product of $2k$ lines on the base of the pyramid and to a product of k lines on each of the triangular faces containing the point. The defined quadrics can be used to jointly satisfy these requirements, due to their straight-line intersections with the surface of the pyramid. The union of the enumerated factors is the product:

$$\xi_k^L \eta_k^L \prod_{j=1}^{k-1} q_j$$

which, for exactly the same reason as in case 2 above, is to be divided by the factor $(2\zeta - 1)$. We conclude that the following expression satisfies all requirements:

$$\phi_{0,0}(\xi, \eta, \zeta) = c_{0,0} \frac{\xi_k^L \eta_k^L}{2\zeta - 1} \prod_{j=1}^{k-1} q_j$$

$c_{0,0}$ being the normalizing factor with value:

$$c_{0,0} = (-1)^k \frac{k^{2k-1}}{i!(k-i)!(k-1)!}$$

□

From Theorem 5.4.2. and Lemmas 5.5.1 and 5.5.2, we draw the following conclusion:

Theorem 5.5.3 *For any $k > 0$ there exist basis functions for the order- k Lagrangian pyramid, which are ratios of polynomials.*

5.6 Summary of Results

We have now presented the techniques that can be used to obtain basis functions for all nodal points on the order- k pyramid. In this section we summarize the results.

The following equations give the basis functions for all nodal points on the order-1 pyramid:

$\phi_0(x, y, z)$	$= -\frac{(x+z-1)(y+z-1)}{2z-1}$
$\phi_1(x, y, z)$	$= \frac{(x-z)(y+z-1)}{2z-1}$
$\phi_2(x, y, z)$	$= -\frac{(y-z)(x-z)}{2z-1}$
$\phi_3(x, y, z)$	$= \frac{(y-z)(x+z-1)}{2z-1}$
$\phi_4(x, y, z)$	$= 2z$

When generating the basis functions for points on higher-order pyramids we use the following primitive constructs:

$\xi_i^L = \xi + \zeta - \frac{i}{k}$	$\xi_i^R = \xi - \zeta - \frac{i}{k}$	$\eta_i^L = \eta + \zeta - \frac{i}{k}$	$\eta_i^R = \eta - \zeta - \frac{i}{k}$
$\Xi^L = \prod_{i=1}^{k-1} \xi_i^L$	$\Xi^R = \prod_{i=1}^{k-1} \xi_i^R$	$H^L = \prod_{i=1}^{k-1} \eta_i^L$	$H^R = \prod_{i=1}^{k-1} \eta_i^R$
$q_i = -\xi^2 - \xi\eta + \xi\zeta + \eta\zeta + \xi\frac{i}{k} + \eta\frac{i}{k} - \frac{i^2}{k^2}$			

The basis functions are given by the following equations:

$\phi_i(\xi, \eta, \zeta) = \frac{1}{\zeta_i} \zeta \phi'_{m(i,k)}(\frac{2k\xi-1}{2k-2}, \frac{2k\eta-1}{2k-2}, \frac{2k\zeta-1}{2k-2})$	$\zeta_i > 0$
$\phi_{0,0}(\xi, \eta, \zeta) = c_{0,0} \frac{\xi_k^L \eta_k^L}{2\zeta-1} \prod_{j=1}^{k-1} q_j$	$c_{0,0} = (-1)^k \frac{k^{2k-1}}{i!(k-i)!(k-1)!}$
$\phi_{k,0}(\xi, \eta, \zeta) = \phi_{0,0}(\eta, 1-\xi, \zeta)$	
$\phi_{k,k}(\xi, \eta, \zeta) = \phi_{0,0}(1-\xi, 1-\eta, \zeta)$	
$\phi_{0,k}(\xi, \eta, \zeta) = \phi_{0,0}(1-\eta, \xi, \zeta)$	
$\phi_{i,0}(\xi, \eta, \zeta) = c_{i,0} \frac{\xi_0^R \xi_k^L \eta_k^L}{2\zeta-1} H^R \prod_{j=1}^{i-1} \xi_j^R \prod_{j=i+1}^{k-1} \xi_j^L$	$c_{i,0} = (-1)^{i+1} \frac{k^{2k-1}}{((k-1)!)^2}$ $0 < i < k$
$\phi_{k,i}(\xi, \eta, \zeta) = \phi_{i,0}(\eta, 1-\xi, \zeta)$	$0 < i < k$
$\phi_{i,k}(\xi, \eta, \zeta) = \phi_{k-i,0}(1-\xi, 1-\eta, \zeta)$	$0 < i < k$
$\phi_{0,i}(\xi, \eta, \zeta) = \phi_{k-i,0}(1-\eta, \xi, \zeta)$	$0 < i < k$
$\phi_{i,j}(\xi, \eta, \zeta) = c_{i,j} \xi_0^R \xi_k^L \eta_0^R \eta_k^L \frac{\Xi^R}{\xi_i^R} \frac{H^R}{\eta_j^R}$	$c_{i,j} = (-1)^{i+j} \frac{k^{2k}}{i!(k-i)!j!(k-j)!}$ $0 < j \leq i < k$
$\phi_{i,j}(\xi, \eta, \zeta) = \phi_{k-i,k-j}(1-\xi, 1-\eta, \zeta)$	$0 < i < j < k$

The only thing we still need to clarify is the first line in the above table. The function $m(i, k)$ computes the local index of a nodal point on the order- $(k-1)$ pyramid that corresponds to the nodal point with local index i on the order- k pyramid. The function $\phi'(\xi, \eta, \zeta)$ is the order- $(k-1)$ basis function. The first entry gives the rule for constructing the basis functions for all points that are not located on the base of the pyramid. The rest of the entries refer to points on the base.

For an example of how these equations are applied, see Appendix B.

5.7 Automating the Process

The proofs of Lemmas 5.5.1 and 5.5.2 immediately suggest a method that can be used to automatically generate the basis functions associated with the nodal points on the order- k pyramid. The process is recursive, with base case $k = 1$, given by Theorem 5.4.2.

We are still missing two pieces:

- *A procedure that generates the nodal points.* This procedure should take k as an input and generate the coordinates of all nodal points on the order- k pyramid. Given the regular placement of the nodal points on the pyramid this procedure is easy to implement.
- *A procedure that implements $m(i, k)$.* The input to this procedure is the local index i of a nodal point and the order of the pyramid k . If the point is located above the base of the pyramid, the procedure returns the local index of the point corresponding to i on the order- $(k - 1)$ pyramid. The mapping is crucial in finding the basis functions for points above the base. Again the regular placement of points on the pyramid makes the procedure easy to implement.

Appendix B.3 presents the code that can be used to generate the basis functions. We chose to use MAPLE, but any other tool that can do symbolic computations would be equally appropriate.

Chapter 6

Mesh Generation

6.1 Problem Description

The problem we address in this chapter can, at a very high level, be stated as follows:

Given the description of a domain Ω , generate a legal integer-coordinate crystalline mesh approximating it.

From the above description, it is apparent that any algorithm for solving this problem should ensure that:

- *all vertices of the elements are placed on integer coordinates*, at least during the mesh generation step. If the domain has been scaled, it should also provide a mapping that would allow us to compute the actual coordinates of a point whenever needed during the rest of the finite element computation.
- *the resulting mesh is a legal crystalline mesh*.
- *the resulting mesh is a good approximation of the input domain Ω* . Specifically, the user should be able to provide a bound on the approximation error, and the mesh generator should guarantee that the bound is achieved.

In this chapter will discuss how each of these goals can be achieved. Before talking about the mesh generator, however, we need to specify what exactly the input to the mesh generator will be. We will do this in the next section.

6.2 Input Format

In general, mesh generators take as input several different data items that can be divided into two broad categories:

- Data that describes the geometry of the problem. This data is usually given in a file and contains a precise mathematical definition of the domain Ω .
- Data that describes properties that the resulting finite element mesh should have. This information is usually given via a user interface.

The geometric data should be sufficient to provide a precise definition of the domain of the problem. A standard format for this type of data does not exist. Most mesh generators require the geometric data to be given in a data file that is written in a simple language. It seems that each existing mesh generator defines its own language for describing the geometry of the domain [1, 8, 36, 69, 86]. However, different input data formats can be divided into two broad categories:

- *Explicit (closed form)*: The boundary of the domain is given as a set of functions with associated ranges.
- *Constructive (discrete)*: In this case, the curves defining the boundary are described using points or tangent lines, that uniquely define the curve. For example, a straight-line segment is described by its two endpoints, a circle by three points or two tangent lines etc.

It is easy to see that the two types of data are equivalent. If one representation is considered preferable, the conversion of data given in another format can be done during a preprocessing step. There are some tools available that have a graphical user interface and let users interactively create the geometry of the domain of the problem. The tool then outputs the data in the format required by a specific mesh generator.

There are several ways of describing the desired properties of the output mesh, and, unfortunately, a standard format does not exist for this type of data either. For most applications, a good description can be achieved using only 2 parameters:

- l : a variable that describes how large the largest element in the resulting mesh should be. Among others, this could be a bound on the length of the edges, the area of elements (in 2D) or faces (in 3D) or on the volume of the elements (in 3D). If the problem has regions of interest that need a higher resolution and are known in advance, l can be different for different regions of the domain.
- ϵ : A variable that denotes how closely the boundary has to be approximated. We will talk more about this measure in Section 6.8.

Since no standard format for the input data exists, we would like to abstract away the specific data representation and give algorithms that do not depend on them. Specifically we will assume that, as a way of describing the domain, we are given the following:

- A bounding box bb . We assume that the domain is entirely contained in the given box.

- A subroutine $\text{IN}(p)$ that given a point p , described by its coordinates, returns *true* if p is inside the domain and *false* if it is outside.
- A subroutine $\text{INTERSECTS}(ls)$ that given a line segment ls , described by its endpoints, returns the number of times it is intersected by the boundaries of the domain, as well as all the intersection points.

The bounding box of the domain is easy to compute from any of the usual data formats described above, during a preprocessing step. For the subroutine INTERSECTS we can use the description of the boundary and known formulas of analytic geometry (see, for example, [61, 81]). The subroutine IN can be implemented by using the INTERSECTS subroutine to count the number of times a line segment between p , and a point outside the bounding box bb , is intersected by the boundary, and returning true if the number is odd, and false otherwise.

Let us note here, however, that if the boundary can be given in the form of general functions, finding the bounding box, or intersection points might be quite complex.

For the second type on input data (variables that describe the desired properties of the output mesh) we specify that:

- l is a bound on the length of the edge of the largest square (or cube) allowed in the mesh. This means that the bound on the largest edge allowed in the mesh is $l\sqrt{2}$ for the 2D case, and $l\sqrt{3}$ for the 3D case. Moreover, the bound on the area of the elements in the 2D case is l^2 and the bound on the volume for the 3D case is l^3 .
- ϵ : A bound on the approximation error in a metric that we will define at the end of this chapter. It suffices to say that this metric penalizes global as well as local inaccuracies and is normalized by the size of the domain.

Now we are ready to give the mesh generation algorithm.

6.3 Generating the Internal Mesh

The mesh generation algorithm that constructs a crystalline mesh has a very simple high level structure. Its outline is given in Algorithm 6.1.

Initially, we extend the bounding box, if necessary, to obtain a square (in 2D) or a cube (in 3D). In the next step, we map the resulting bounding box to a different set of coordinates, to ensure that the entire mesh generation computation can be performed using integer arithmetic. This can be done by using a translation operation followed by scaling. The translation ensures that the center of the bounding box coincides with the origin of the axes. The scaling operation places the vertices of the bounding box on coordinates that are powers of 2. The choice of the exact values of the coordinates may depend on l , but we can always take full advantage of the range of integers supported by the

```

GENERATE_MESH(bb, l,  $\epsilon$ )
1  Adjust bb so that all its sides are of equal length
2  Translate and scale the domain. Compute bound  $l''$ 
3  Create a mesh element coinciding with bb
4  edge_length  $\leftarrow$  bb.edge_length
5  while edge_length >  $l'$ 
6  do for each element el in the mesh
7      do STANDARD_REFINE(el)
8      edge_length  $\leftarrow$  edge_length/2
9  for each element el in the mesh
10 do if IS_EXTERNAL(el)
11     then remove el from the mesh
12 APPROXIMATE_BOUNDARIES(mesh,  $\epsilon$ )
13 Partition as many elements as necessary obtain a legal mesh.

```

Algorithm 6.1: High level outline of the mesh generation algorithm

computer used. We multiply l with the same scaling constant to obtain l'' , which is the bound on the edge length of any element in the scaled domain.

We can now construct a mesh element that completely coincides with the bounding box of the domain. Then we repeatedly partition all the elements in the current mesh into equal parts: squares are partitioned into 4 squares and cubes into 8 cubes. The process stops when all elements in the mesh have edge length less than l' . In the next step, all elements are tested, and elements that are completely external to the domain are removed from the mesh. The fact that we only test elements that have edge length less than l'' to determine if they are outside the domain allows us to design the IS_EXTERNAL() subroutine so that it is guaranteed to give the correct results.

We are left with a mesh that completely covers the domain and consists entirely of d-cubes. The boundaries of this mesh are only a very rough approximation of the boundaries of the original domain. In order to obtain a better approximation we use the subroutine APPROXIMATE_BOUNDARIES().

Example 6.3.1 *We will demonstrate how the algorithm works by using a simple 2D example. Assume that the domain given is the one shown in Figure 6.1(a). The bounding box is $x = 10..250$, $y = 30..165$ and $l = 9$. During the first step we make the bounding box a square, by extending it along the y direction. The new bounding box shown in Figure 6.1(b) has the same x values and $y = -22.5..217.5$.*

In the next step we perform a translation to ensure that the center of the new bounding box coincides with the axis origin. The new coordinate system x' , y' is defined as:

$$(x', y') = (x - 130, y - 97.5)$$

Obviously, the bounding box is now in the range $-120..120$ in both dimensions.

Finally, we scale the domain so that the vertices of the bounding box are placed on points whose coordinates are powers of 2. In this case, we decided to map the box in the range $-2^{15}..2^{15}$. The

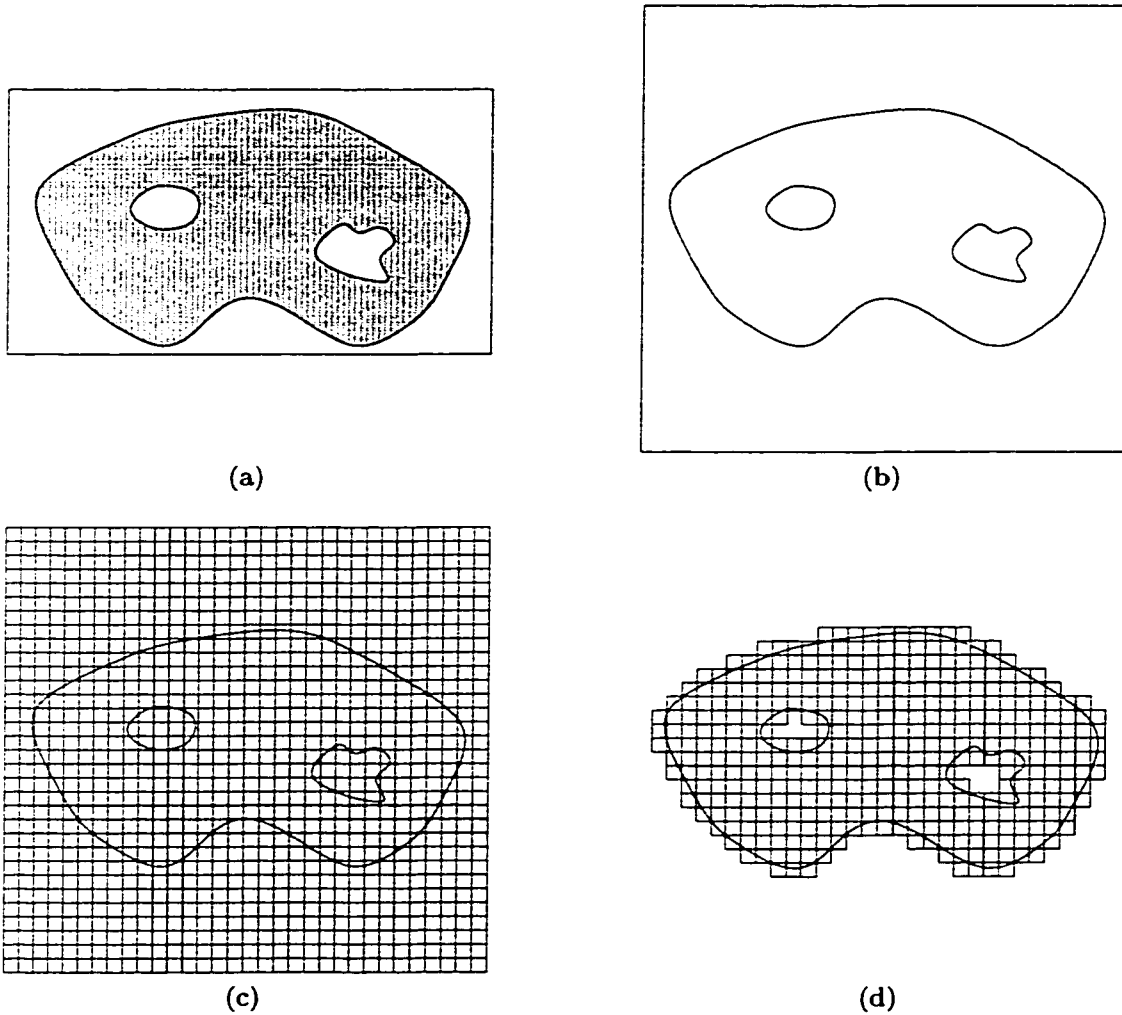


Figure 6.1: A simple example that demonstrates the first steps of the mesh generation algorithm. (a) The domain and the bounding box are given as input. (b) The bounding box has been extended to form a square. (c) The elements are refined until all edges in the mesh are shorter than l''' . (d) Elements that are completely outside the domain are removed.

new coordinates of the points are given as:

$$(x'', y'') = \left(\frac{2^{15}}{120} x', \frac{2^{15}}{120} y' \right)$$

Obviously, we can compute the original coordinates of a point $p'' = (x'', y'')$ in the current system using:

$$p = \left(\frac{120}{2^{15}} x'' + 130, \frac{120}{2^{15}} y'' + 97.5 \right)$$

Notice that the algorithm needs to output only a scaling constant and a translation parameter for each dimension. Using the same scaling parameter we can compute $l'' = \frac{2^{15}}{120} l \approx 2457.6$.

Now we can create a square element that completely coincides with the bounding box and repeatedly divide all elements in the mesh until we obtain elements with edge length less than l'' . This happens after 5 iterations. The resulting mesh is shown in Figure 6.1(c). Finally, we remove all the squares that are located completely outside the domain. We are left with a mesh whose boundary is a very coarse approximation of the boundary of the domain, as shown in Figure 6.1(d).

The algorithm given above can be easily modified for the case where different regions of the domain require different mesh density. In this case, l is not a number, but a default value that is a bound on edge lengths on the entire mesh and a list of pairs (*length, subdomain_description*). These pairs specify a bound on the length of the edges in the corresponding subdomain. In this case, the first while loop of the algorithm (lines 5–8) will use the default length as l . After this, another loop will be executed that examines all the resulting elements, to determine if they fall (partially or completely) inside one of the subdomains described in the input. If they do, they are refined until the specified bound on edge length is achieved. Since the crystalline meshes generated will be used in combination with an adaptive technique, we do not need the full expressive power of general functions for describing the subdomains. It is enough to specify them as 2D or 3D boxes. The adaptive algorithm will make all other necessary adjustments during the computation.

For Algorithm 6.1 to work, the input must satisfy several conditions. First of all, the bounding box around the domain should be correct. If portions of the domain fall outside the bounding box, the algorithm might not generate a mesh that covers them.

Of even greater importance, is the value of l . Notice that we perform the `IS_EXTERNAL()` test on each element of the mesh only after their edges have become shorter than l . We should be able to implement the `IS_EXTERNAL()` subroutine using only the two subroutines that were given as a description of the domain: `IN(p)` and `INTERSECTS(ls)`. Of course, we can use the `IN()` subroutine to test as many points in the interior of the element as we like, but this is not acceptable. Not only can this approach be very time consuming, it also will never give us a conclusive answer, if all the points we test happen to be outside the domain. Therefore, we are forced to make the following assumption:

Assumption 6.3.1 *No feature of the given domain is smaller than l in any of its dimensions.*

The above assumption ensures that if the edges of the element are not intersected by a boundary, the element is either entirely inside or entirely outside the domain. So, testing one point in the interior of the element is enough to give a conclusive answer.

```

IS_EXTERNAL(el)
1  for each edge e of el
2  do if INTERSECTS(e)
3      then return false
4  c ← CENTER(el)
5  if IN(c)
6      then return false
7  else return true

```

Algorithm 6.2: Subroutine that returns *true* if an element is entirely external to the domain, and *false* otherwise.

How the procedure IS_EXTERNAL() can be implemented using the two basic subroutines that describe the domain is shown in Algorithm 6.2. During the first loop, each one of the edges of the element is tested for intersections with the boundary, using the INTERSECTS() subroutine. If it intersects, the element cannot be completely external to the domain, so we can safely return *false*. If no edges of the element intersect the boundary, and Assumption 6.3.1 holds, the element is either completely inside or completely outside the domain. For reasons of symmetry, we test the center of the element using the IN() subroutine, and return the corresponding answer.

Assumption 6.3.1 might seem too restrictive, but it only states that the user should be able to estimate the size of the features of the domain (in each region) that have to be captured by the mesh, and give the appropriate information to the mesh generator. For most applications, this should not pause a problem. Moreover, an assumption similar to Assumption 6.3.1 is necessary if we want to use only the two subroutines mentioned above as a way of describing the domain. Finally, *l* could be automatically computed in a preprocessing step, from the description of the domain.

This is demonstrated in Figure 6.2. The square is an element of a 2D mesh and we want to determine if it is external to the domain (that is, if it does not contain any portion of the domain). We can test many line segments for intersections with the boundaries and many points in the element to determine if they fall inside or outside the domain. However, if features of the domain are allowed to be arbitrarily small, an adversary can always place a portion of the domain inside the square, in such a way as to ensure that the portion is not intersected by any of the tested line segments, and does not contain any of the tested points. Only a bound on the size of the features will allow us to develop a strategy for testing, that will always give correct results. While any bound on the size would be sufficient for developing the strategy, we chose *l* for simplicity.

After the process of discarding elements that are external to the domain, the mesh generation algorithm proceeds by finding a better approximation to the boundaries. To do that, the procedure APPROXIMATE_BOUNDARIES() is used. This is by far the most complex task in the entire mesh generation process and we will devote the following sections to discussing how a good approximation

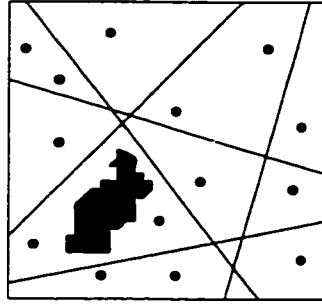


Figure 6.2: Demonstration why Assumption 6.3.1 is necessary if we want to have a test that always determines correctly if an element is completely external to the domain. The square is a mesh element. The lines and points shown are lines that have been tested for intersections with the boundary and points that have been tested using the `IN()` subroutine. The shaded region is a portion of the domain placed by an adversary so that it cannot be detected by the tests that have been performed so far.

can be obtained, and how the quality of the approximation can be quantified.

6.4 Obtaining an Approximation of the Boundary

One feature that every good mesh generator should have is the ability to closely approximate the boundaries of the domain. This is because the shape and the position of the boundaries affect the solution of some partial differential equations in an almost chaotic manner. This means that small changes in the boundaries might result in a big, unpredictable change in the solution. Ideally, we would like to be able to represent the boundaries exactly. However, most interesting problems are defined in domains whose boundaries consist not only of straight-line segments and planes, but also of curves and curved surfaces. This leaves us with two choices:

- If the problem we are trying to model is very sensitive to the shape of the boundary, we can introduce elements with curved edges and faces, that will allow us to represent the boundary exactly. Unfortunately, introducing “general” shapes in the mesh creates mathematical problems. The systems of linear equations associated with such meshes are harder both to generate and to solve. Because of these difficulties, we use elements with general shapes only next to the boundary where they are absolutely necessary.

We can use the procedure `APPROXIMATE_BOUNDARIES()` to produce such a mesh. One method that can be used is replacing all elements that are intersected by the boundary by elements with general shapes. In this case, we obtain a mesh like the one shown in Figure 6.3.

There is an alternative, and more popular method to obtain the general-shape boundary elements. We start by removing all elements that intersect with the boundary. Now we have a mesh that is contained in the domain. We can process this mesh and find its boundary,

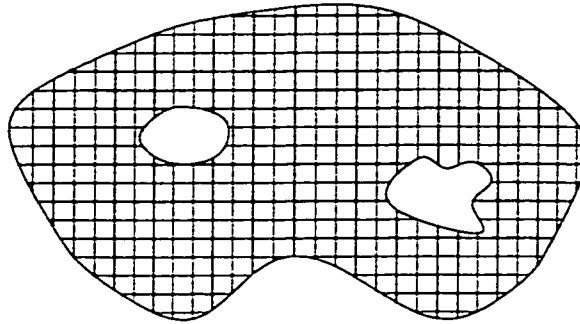


Figure 6.3: Elements with general shapes are used as boundary elements. In this case, the boundary can be represented exactly.

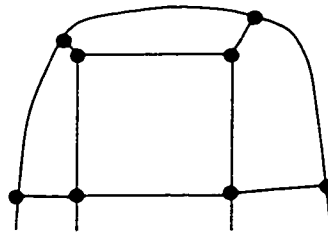


Figure 6.4: A 2D example illustrating a technique to obtain boundary elements of general shapes. We start by projecting the vertices of the boundary of the mesh onto the boundary of the domain. In this case, we project all four vertices of a square onto the boundary of Ω and obtain three elements with general shapes.

which is a collection of straight-line edges in 2D or square faces in 3D. Each of the vertices on the boundary of the mesh is then projected onto the boundary of the domain, using some local criterion. Of course, having the projection of the vertices on the boundary of Ω , we can easily identify the projection of a straight-line edge or a face. The boundary elements are then formed to contain both the edge (face) on the boundary of the mesh and its projection on the boundary of the domain. Figure 6.4 shows a 2D example. A square element next to the boundary is shown, together with the three general-shaped elements resulting when projecting its vertices on the boundary of Ω .

Alternatively, we can remove all the elements intersected by the boundary and create the general-shaped elements by projecting the boundary of the resulting mesh on the boundary of the domain. This process is more time consuming, but produces better quality meshes.

- If the problem is not very sensitive to the shape of the boundary, we can approximate curved lines by straight line segments and curved surfaces by planar faces. This allows us to approximate the boundary using a finite number of different element shapes. Of course, we still want the boundary of the mesh to be as close as possible to the boundary of the domain. In the rest of this section we will describe how this can be achieved.

In the crystalline mesh generator, the procedure `APPROXIMATE_BOUNDARIES()` is the one responsible for building a good approximation of the boundaries. The mesh that it is given as input has only a very rough approximation of the boundaries, which has been created just by removing all elements that are external to the domain. After the procedure finishes, the boundaries have been approximated up to an approximation error ϵ , where ϵ is a parameter given by the user. A high level description of the subroutine is given in Algorithm 6.3.

```

APPROXIMATE_BOUNDARY(mesh,  $\epsilon$ )
1  Boundary_Elements  $\leftarrow$  FIND_BOUNDARY_ELEMENTS(mesh)
2  SIMPLIFY(Boundary_Elements, mesh)
3  for each element bel in Boundary_Elements
4  do COMPUTE_APPROXIMATION(bel)
5  error  $\leftarrow$  COMPUTE_APPROXIMATION_ERROR(mesh)
6  while error  $>$   $\epsilon$ 
7  do IMPROVE_APPROXIMATION(Boundary_Elements)
8     error  $\leftarrow$  COMPUTE_APPROXIMATION_ERROR(mesh)

```

Algorithm 6.3: High level description of the subroutine approximating the boundaries.

The algorithm first calls the subroutine `FIND_BOUNDARY_ELEMENTS()`, which returns a set containing all mesh elements that are intersected by the boundary (that is, all elements that are not completely internal to the domain). These elements are the ones that have to be processed in order to obtain a better approximation of the boundary.

In order to explain what the rest of the algorithm does, we will need the following definition:

Definition 6.4.1 *A boundary element is simple if any of its edges is intersected by the boundary at most once.*

Having to deal only with simple boundary elements significantly reduces the number of cases that we have to consider when computing the approximation. So we call the subroutine `SIMPLIFY()`, that processes non-simple elements and ensures that all elements passed to the subroutine `COMPUTE_APPROXIMATION()` are simple. Next the subroutine `COMPUTE_APPROXIMATION()` is used. It examines all boundary elements and replaces some of them by elements with different shapes that approximate the boundary better.

After the approximation is computed, we calculate the approximation error. If it is larger than the error specified by the user, we repeatedly find a closer approximation to the boundaries (by calling the `IMPROVE_APPROXIMATION()` subroutine) and compute the new error, until the bound is achieved.

There are several subroutines used in Algorithm 6.3 that have to be explained further. We start with the subroutine `FIND_BOUNDARY_ELEMENTS()`, which has a simple high-level structure (pseudocode for it is given in Algorithm 6.4). Each element in the mesh is examined in turn, to determine if it contains a portion of the boundary. If it does, it is added to the set of elements that are returned by the subroutine.

Notice that in order to test for boundary elements, we use the subroutine `INTERSECTS()` that we assume was given as part of the input. This subroutine, depending on the description of the domain, might be quite complex. For a discussion of this topic, see 6.2.

```

FIND_BOUNDARY_ELEMENTS(mesh)
1  Boundary_Elements  $\leftarrow \emptyset$ 
2  for each element el in the mesh
3  do if IS_BOUNDARY_ELEMENT(el)
4      then Boundary_Elements  $\leftarrow$  Boundary_Elements  $\cup$  {el}
5  return Boundary_Elements

IS_BOUNDARY_ELEMENT(el)
1  for each edge ed of el
2  do if INTERSECTS(ed)
3      then return true
4  return false

```

Algorithm 6.4: High level description of the subroutine that finds the boundary elements in the mesh.

The procedure `SIMPLIFY()` (given in Algorithm 6.5) is the one that ensures that no boundary element has an edge that is intersected by the boundary more than once. It is implemented by keeping two separate sets of boundary elements. One consists of elements that are simple. The other, contains elements that have to be tested and, if there is a need, simplified. The algorithm terminates when the second set is empty. Every time an element is examined and we discover that

it is not simple, we partition it into 2^d standard elements. The hope is that by splitting all the edges of the element into two, intersections that fall on the same edge will be separated. Assumption 6.3.1 ensures that this is very likely to happen. The elements resulting from the refinement are then examined and divided into three categories: external elements, that are discarded, internal elements that are added to the mesh, and boundary elements, that are added to the set of elements to be simplified.

```

SIMPLIFY(Boundary_Elements, mesh)
1  unprocessed  $\leftarrow$  Boundary_elements
2  simple  $\leftarrow \emptyset$ 
3  while unprocessed  $\neq \emptyset$ 
4  do remove element el from unprocessed
5     is_simple  $\leftarrow$  true
6     for all edges ed of el
7     do if INTERSECTS(ed).count > 1
8        then is_simple  $\leftarrow$  false
9     if is_simple
10    then simple  $\leftarrow$  simple  $\cup$  {el}
11    else PARTITION(el)
12        for each one of the resulting elements child_el
13        do if IS_BOUNDARY_ELEMENT(child_el)
14           then unprocessed  $\leftarrow$  unprocessed  $\cup$  {child_el}
15           else if IS_EXTERNAL(child_el)
16              then DISCARD(child_el)
17              else mesh  $\leftarrow$  mesh  $\cup$  {child_el}
18        remove el from the mesh
19  Boundary_elements  $\leftarrow$  simple

```

Algorithm 6.5: The procedure that ensures that no boundary element edge will intersect the boundary more than once.

This leaves us with three questions that have to be answered:

- (a) Given a simple boundary element, how do we approximate it?
- (b) Given an approximation of a boundary element, how do we compute a better approximation?
- (c) How do we compute the error of the current approximation so that we can stop when the error bound specified by the user is satisfied?

In the rest of this section we will look into questions (a) and (b). Section 6.8 is completely devoted to answering (c).

In order to find an approximation to the boundary elements, we will use a very simple technique. First of all, remember that all boundary elements are simple, so that no edge intersects with the boundary more than once. These simple elements are examined for intersections. Every time an intersection point p is found (using the INTERSECTS() subroutine), its distance from the two end-points of the edge is calculated and it is identified with the closest one. The portion of the boundary

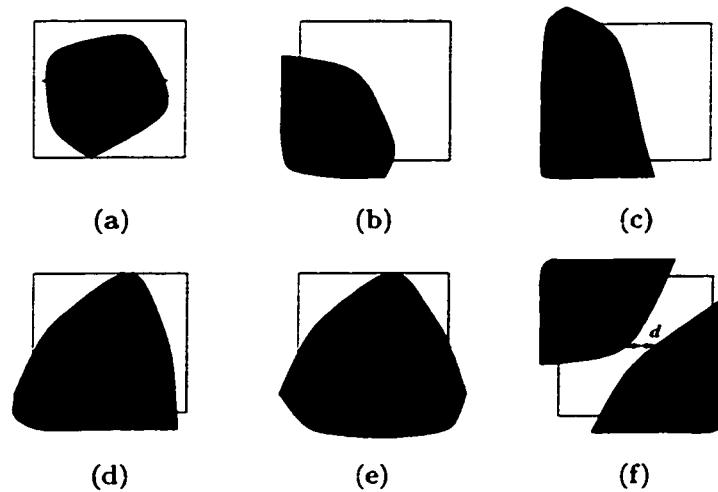


Figure 6.5: The possible ways an element can be intersected by the boundary. (a) One intersection. (b) Two intersections on adjacent sides. (c) Two intersections on opposite sides. (d) Three intersections, of which the two “real” ones are on contiguous sides (e) Three intersections, the two “real” ones being on opposite sides (f) Four intersections. The only valid configurations are (b) and (c).

between two consecutive intersection points is then approximated by a straight-line segment. This process is usually called “snapping”. Snapping reduces the number of possible boundary element configurations so that we can do a case-by-case analysis.

To obtain an approximation that fits the boundary better than the one we already have we use an iterative process, based on a similar concept. Each boundary element is partitioned into 2^d standard elements and an approximation is computed for each one of the resulting elements. If the approximation computed by this process is better than the old one, we keep the refinement. Otherwise, we keep the old approximation, since we don’t want to unnecessarily introduce new elements to the mesh.

We will now look at the possible boundary element approximations both for the 2D and the 3D case.

6.5 Boundary Approximation in 2D

Before the approximation procedure is executed, the mesh contains only square elements. After the elements have been simplified, each edge is intersected by the boundary at most once. So an element can have between one and four intersection points. Figure 6.5 shows all possible intersection configurations. Notice that:

- In cases (a) and (f), the domain contains a feature that has one dimension with length d less than l (as indicated in the figure). This violates Assumption 6.3.1. In practice, if these

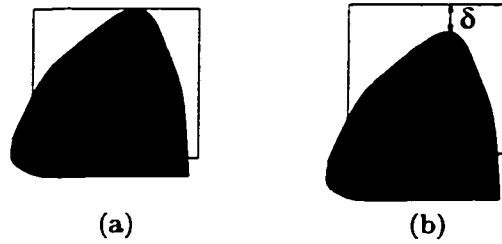


Figure 6.6: (a) Boundary that has one tangent intersection point with the top element edge. (b) The boundary has been moved down by a distance δ so that no intersection point can be detected.

configurations are encountered, we can conclude that the value of l , is not sufficiently small to allow us to approximate the domain well. Both of these special configurations can be easily detected, since these are the only configurations having one and four intersection points respectively.

- In cases (d) and (e) we have one intersection point that is not a “real” intersection. Instead, the edge of the element is a tangent of the boundary at that point. If we move the tangent line an infinitesimal distance the intersection point will either disappear or become two intersection points. We can also assume that these configurations do not occur. In practice, this means that whenever such a configuration is encountered, we will ignore it.

It is easy to see that this practice will not greatly affect the quality of the boundary approximation, and that any quality bound that can be proven with tangent points taken into account, will also be valid after the modification. Consider, for example, the two boundary elements shown in Figure 6.6. In both cases, we have the same boundary, only in (b) the boundary has been moved a small distance δ so that no intersection point will be detected on the top edge. As the distance δ goes to 0, the two cases become identical. Therefore, whatever quality bound holds for case (b), will also hold for case (a), if the tangent intersection point is ignored.

This leaves us with cases (b) and (c) shown in Figure 6.5 as the only valid configurations.

During the first step of the approximation algorithm, all intersection points of the element are identified. Next, the distance between the intersection point and each one of the edge endpoints is computed, and the point is identified with the closest one. Finally, the curve between two consecutive intersection points is approximated by a line segment. We will call this process *snapping*.

If we use the standard numbering of the vertices of a square, each configuration that can result from the snapping process performed on an element with k intersection points can be described by a k bit number. We interpret a 0 in this number to mean that the corresponding intersection point has been snapped to the lowest numbered vertex, while an 1 means that it has been snapped to the highest numbered vertex. This enables us to do a case analysis easily. Figure 6.7 shows all the possible configurations after snapping for the two valid cases (b) and (c). Notice, that after snapping

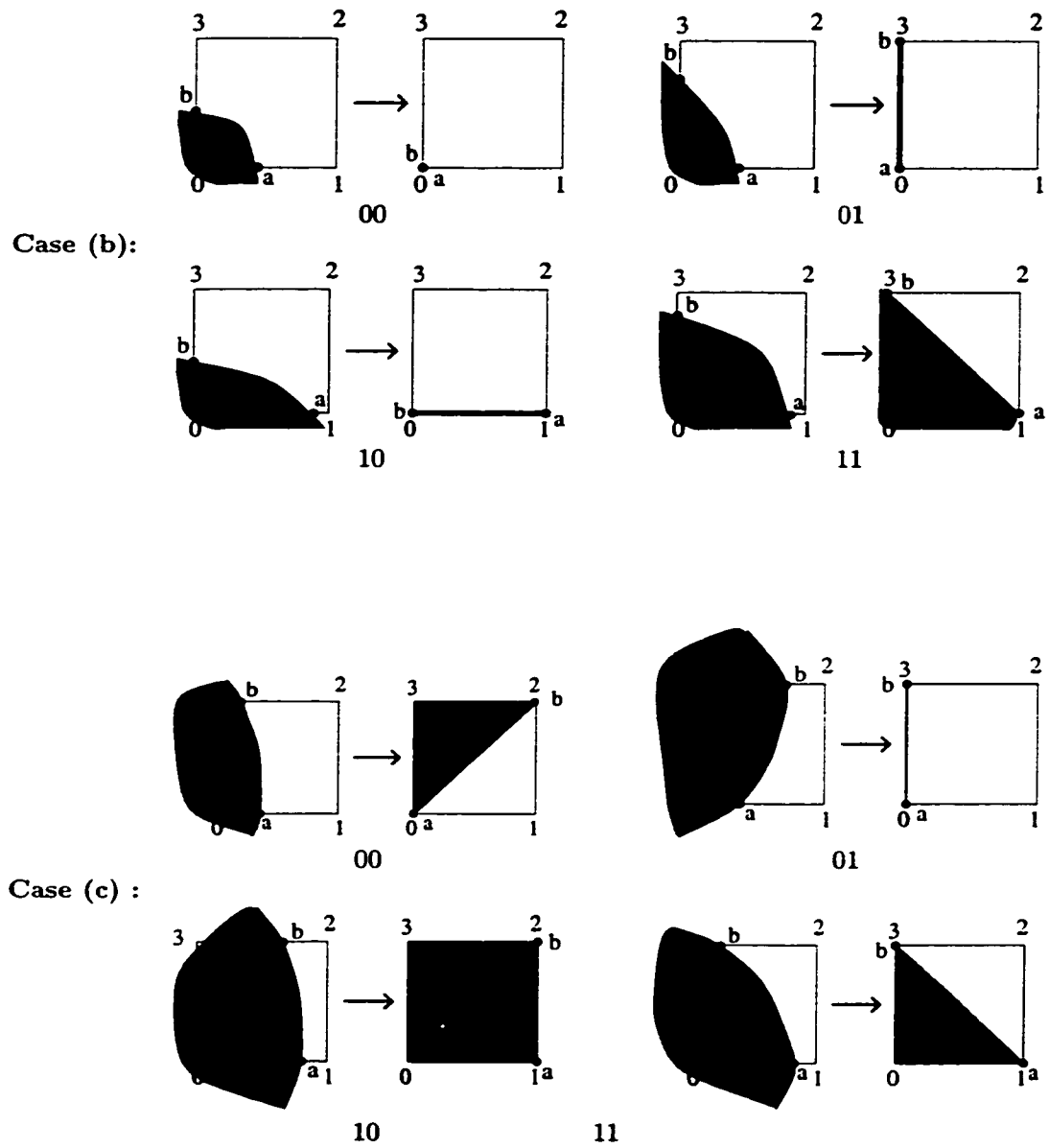


Figure 6.7: Possible configurations, after snapping for cases (b) and (c).

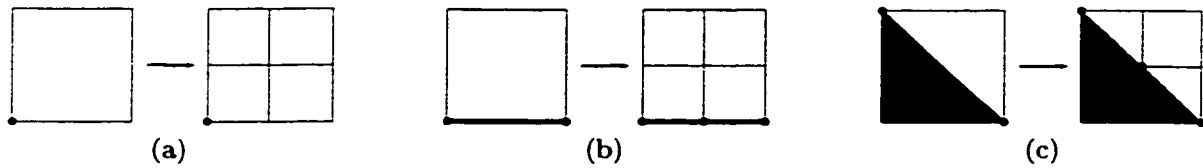


Figure 6.8: Cases where the refinement does not give a better approximation. (a) In both cases, the boundary reduces to a vertex. (b) The boundary reduces to the same edge. (c) The boundary reduces to the same diagonal.

a boundary element will reduce to one of the following:

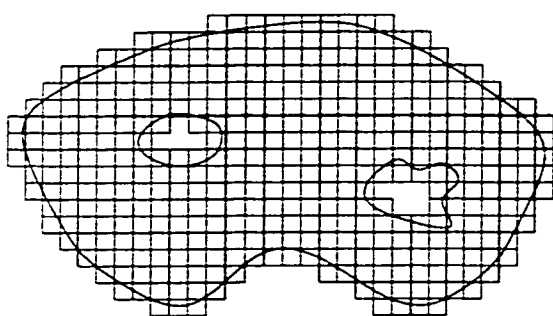
- (1) a vertex of the element.
- (2) a line coinciding with an edge of the element.
- (3) a line coinciding with a diagonal of the element.

In cases (1) and (2), no edges divide the boundary element, so we approximate the boundary by either including the entire element in the mesh or by discarding it. We determine which one of these two options is the closest approximation by testing the center of the square to determine if it is in the interior of the domain. In case (3), the boundary has been snapped to a diagonal of the element. So we divide the square element into two triangular elements (along the appropriate diagonal), one of which will be included and the other excluded. We can find out which should be included by testing the center of gravity of one of the triangles to determine if it is inside the domain. Notice that the elements resulting from this procedure are all, either squares or, isosceles right-angled triangles. These elements are legal for the 2D crystalline mesh, so no further processing is necessary.

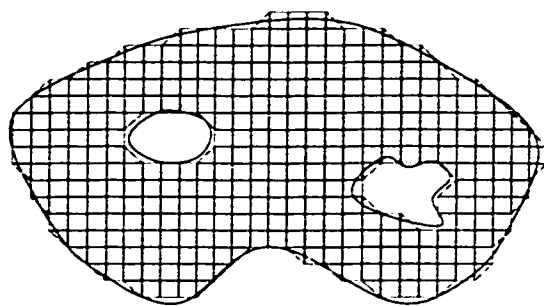
Once we have an approximation of the boundary, we can find a better one by a simple procedure. Each one of the original boundary elements is partitioned into four squares. An approximation for each of the resulting squares is computed, by using the same approximation procedure. If this gives us the same approximation as the original one, that is, if the boundary is approximated by the same line segments in both cases and the same portions are included and excluded (see Figure 6.8), we keep the old approximation. Otherwise, we keep the newest one, which is more accurate.

Figure 6.9 shows a typical example of how the subroutines `COMPUTE_APPROXIMATION()` and `IMPROVE_APPROXIMATION()` work. In (a), we see the mesh after all elements completely external to the domain have been removed. In (b), we see the first approximation that will be computed. Figures (c) and (d) show how the mesh will look like after calling `IMPROVE_APPROXIMATION()` once and twice, respectively. Notice that every time the `IMPROVE_APPROXIMATION()` subroutine is called, the quality of the approximation increases.

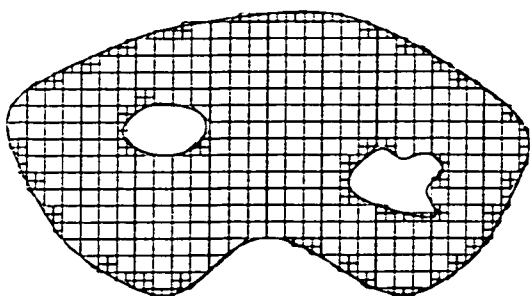
It is clear that by using this approach we can represent only boundaries that consist of straight lines at 0, 45, 90 and 135 exactly. Other straight lines and curves have to be approximated.



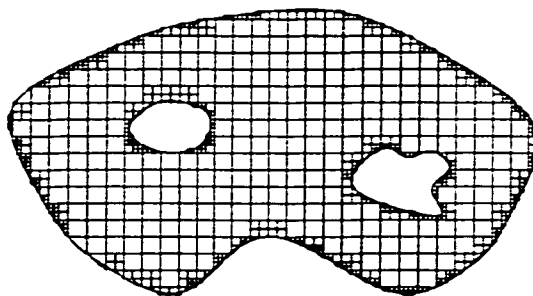
(a)



(b)



(c)



(d)

Figure 6.9: Successive approximations to the domain. (a) After all external elements have been removed. (b) First approximation. (c) Better approximation obtained after one iteration. (d) Approximation obtained after 2 iterations.

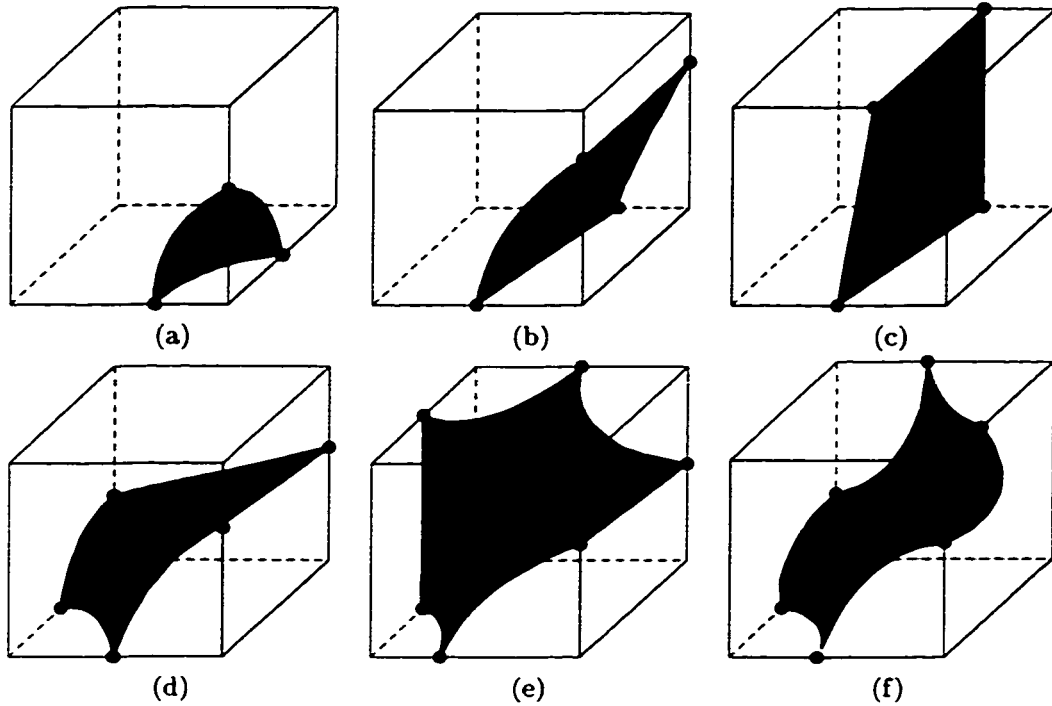


Figure 6.10: Possible ways the boundary can pass through the cubes (for simple elements)

6.6 Boundary Approximation in 3D

In the 3D case, the elements constructed by the crystalline mesh generator are cubes. Up to this point, no elements with other shapes have been introduced. Moreover, the `SIMPLIFY()` subroutine ensures that no elements have edges intersected by the boundary more than one time. It is easy to see that the intersection points on a face of a boundary elements will be placed in one of the configurations shown in Figure 6.5.

We can use the same reasoning we used in the 2D case to claim that faces can have only two different intersection configurations, both with exactly two intersection points on the edges of the face. One case is when the two intersection points are on adjacent edges (Figure 6.5(b)), the other, when the intersection points are on opposite edges (Figure 6.5(c)). Under these assumptions, there are only 6 possible ways the boundary can pass through a cubic element. All of them are shown in Figure 6.10.

After the intersection points are detected, a snapping procedure is performed. The distance of every intersection point p to each of the endpoints of the edge is computed and p is identified with the closest endpoint. The portion of the boundary that is inside the cube (and which we infer from the intersection points) is then approximated by one or more planar surfaces.

In order to find all the possible states of a cube after snapping, we need to do a case analysis.

As in the 2D case, each element with k intersection points is associated with a k -bit number. We then examine all 2^k possible values of this number, interpreting 0 to mean that the corresponding intersection point has been snapped to the lower numbered vertex on the edge, while 1 means that it has been snapped to the higher numbered vertex.

By looking at Figure 6.10 we see that we have one cube with 3 intersection points, one cube with 5 intersection points, two distinct cubes with 4 intersection points and two distinct cubes with six intersection points. Therefore, the number of different cases that need to be examined is:

$$2^3 + 2 \cdot 2^4 + 2^5 + 2 \cdot 2^6 = 200$$

Of course, we cannot show all of these cases here, for reasons of space. Fortunately, after examining these cases, we notice that they reduce to only eleven ways in which the boundary can intersect a simple element after snapping.

Three of these cases are trivial:

1. The boundary reduces to a vertex.
2. The boundary reduces to an edge.
3. The boundary reduces to a face.

In all these cases the boundary can be approximated correctly by either including or excluding the entire cube. We distinguish between these two cases by using the `IN()` subroutine to check if the center of the cube is inside or outside the domain.

This leaves us with eight non-trivial cases, which are shown in Figure 6.11. Notice that the only real information we have about the intersection of the boundary with the cube are the intersection points (which are the points where the edges of the cube and the boundary intersect). Using this information we can accurately perform the snapping operation and deduce the line segments which correctly approximate the intersection of the boundary with the faces of the cube. In most cases these line segments give us enough information to compute a good approximation of the portion of the boundary in the interior of the cube (cases (a)-(e)).

Obtaining configuration (h) implies that the boundary of our domain has a feature of size less than l in one of its dimensions. Using assumption 6.3.1, we can conclude that this configuration will not occur. In practice, this means that if this configuration is encountered, the value of l given by the user was incorrect.

This leaves us with configurations (f) and (g). Both of these configurations are ambiguous and need further processing. Specifically, the cuts obtained on the faces of the cube in configuration (f) are consistent with three different partitions of the cube, as shown in Figure 6.12. Fortunately, we can easily distinguish among these cases by doing some more processing. We start by partitioning the cube into eight smaller cubes. Let us denote each one of these cubes by an integer, corresponding to the index of the vertex of the original cube included in it (also shown in Figure 6.12). We then use

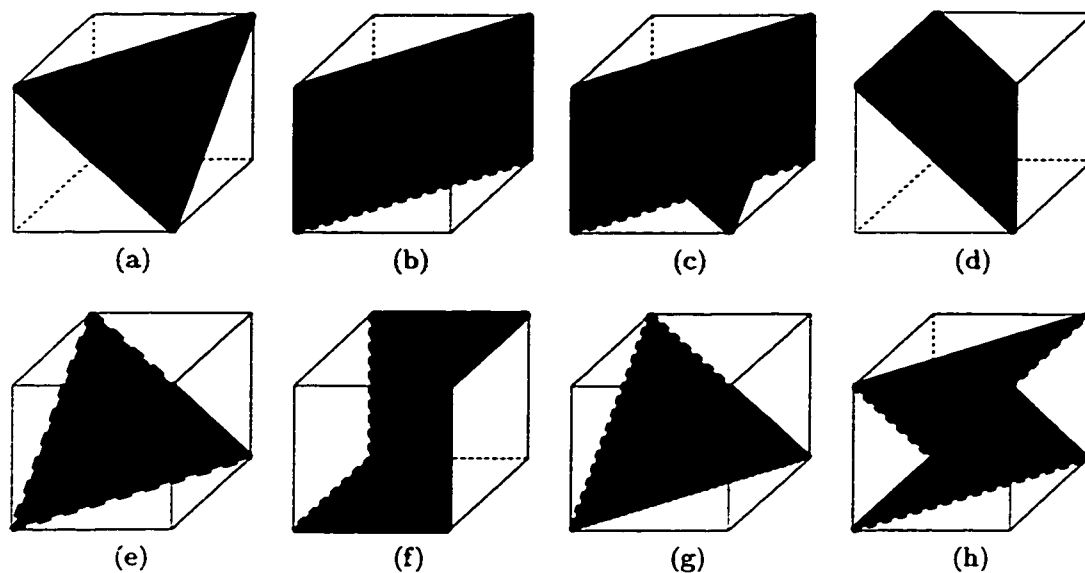


Figure 6.11: Possible ways the boundary can intersect a cube after snapping. The planar surfaces that are used to approximate the boundary are shown in gray. The intersection of the boundary with the faces of the cube is reduced to a set of straight line segments, that are shown here in bold. From these segments we can (in most cases) deduce the correct approximation of the surface.

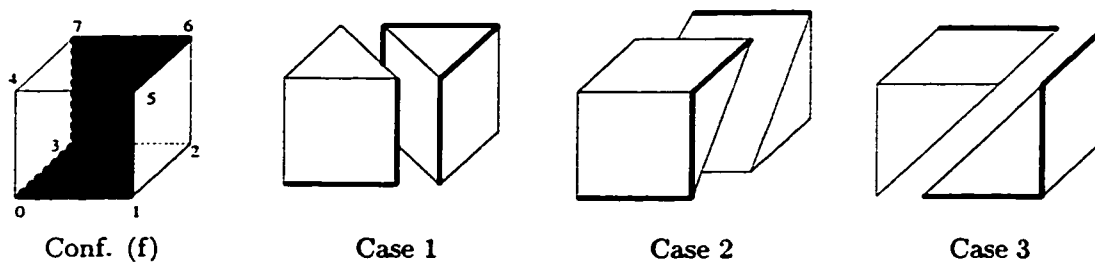


Figure 6.12: Configuration (f) is ambiguous, because there are three different partitions of the cube that are consistent with the cuts on the faces.

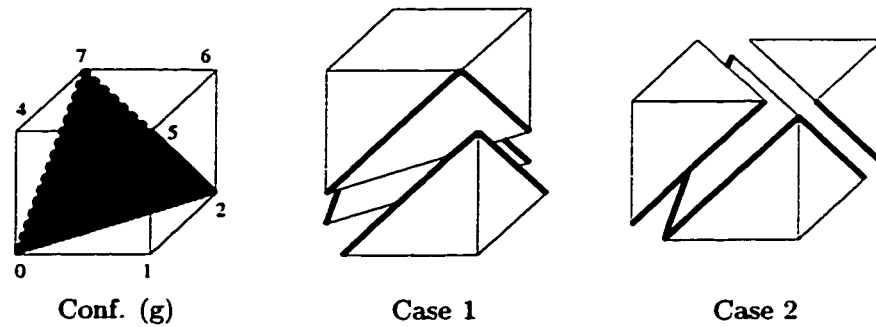


Figure 6.13: Configuration (g) is consistent with two different partitions of the cube.

the same procedure to obtain an approximation of the boundary for each of the smaller cubes. It is easy to see that if we are dealing with Case 1, the new approximation will not partition cube 0, while if we are dealing with either Case 2 or Case 3, it will. In the same manner, the new approximation will not partition cube 5 or cube 1 if we are dealing with Case 2 or Case 3 respectively. So we can distinguish between these three cases by checking which one of the three smaller cubes doesn't get divided by the partition.

Configuration (g) is also ambiguous. Figure 6.13 shows two possible partitions of the cube that are consistent with the cuts on the faces. In this case too, we can determine which one of the two partitions is the better approximation of the boundary by partitioning the cube into eight cubes and computing the approximation for them. Clearly, in Case 1 cubes 0 and 2 will be classified as belonging to configuration (g), while cubes 5 and 7 will be classified as belonging to configuration (a). The opposite is true for Case 2. So we can completely determine which configuration we are dealing with by checking one of these cubes, for example cube 0.

The procedure used for finding a better approximation of the boundary is identical to the one used in the 2D case. We partition each of the boundary cubes into eight smaller cubes and use the same process of finding the intersection points and snapping. In this case too, we use the resulting cubes only if we obtain a better approximation. If we obtain the same approximation as with the larger cube, the larger cube is used. In this way, we avoid introducing elements if they are not useful. Figure 6.14 shows the cases where partitioning the cube does not give a better approximation. Notice that if the cube belongs to case (f) or (g) we choose to always partition it, since these two configurations are ambiguous.

There is one more issue we need to address. By looking at the configurations in Figure 6.11, we realize that when the boundary cubes are partitioned, the resulting shapes are not necessarily valid for a crystalline mesh. In Configuration (a), for example, the cube is partitioned into one tetrahedron and one heptahedron. Figure 6.15 shows all the shapes that might occur at the boundary. These shapes need to be further partitioned to obtain simpler shapes.

While designing the partitions we want to ensure that:

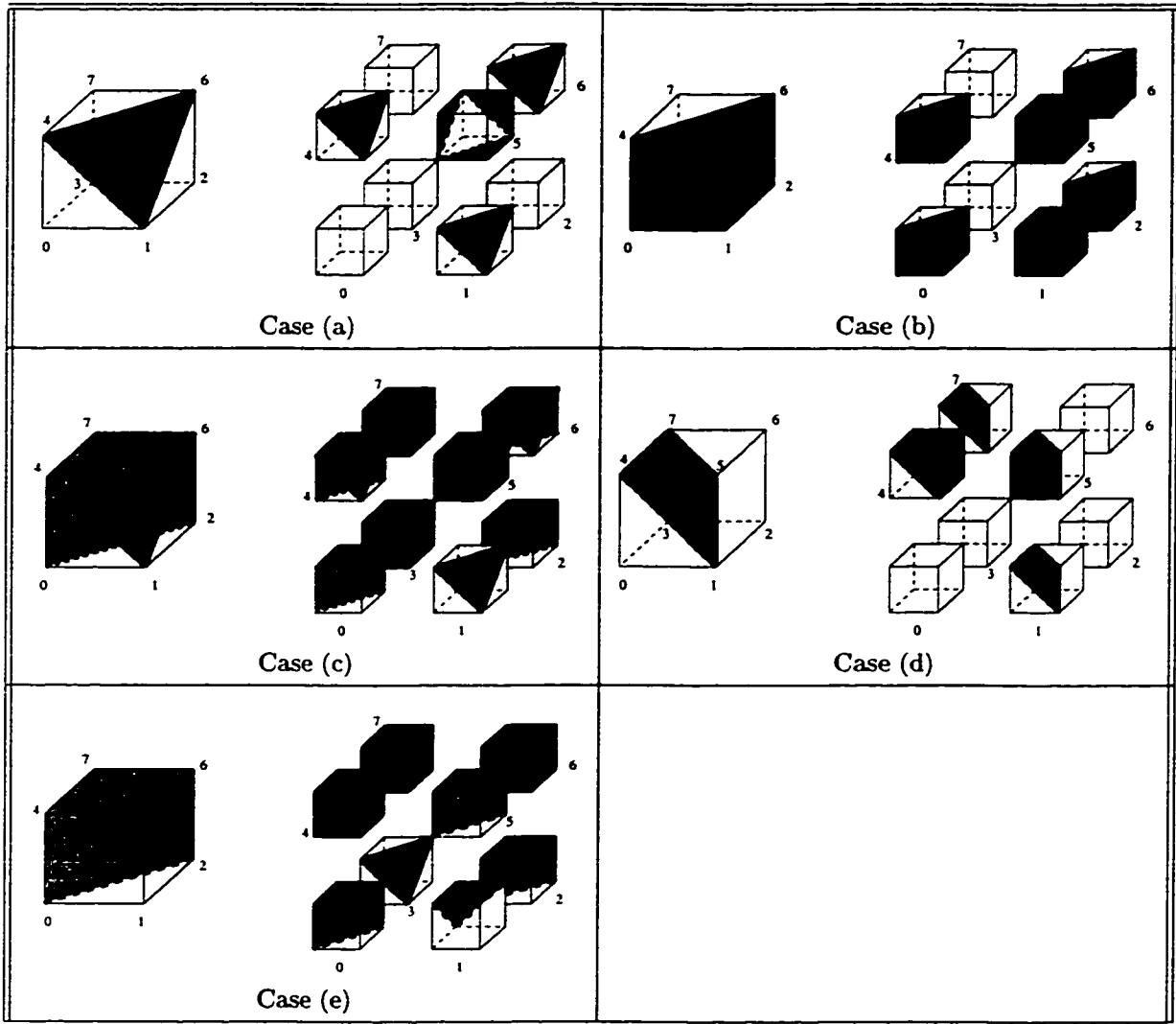


Figure 6.14: Cases where the partitioning of the cube does not result in a better approximation. Only cases (a)–(e) from Figure 6.11 are shown. In processing cubes belonging to case (f) or (g) we always keep the partition.

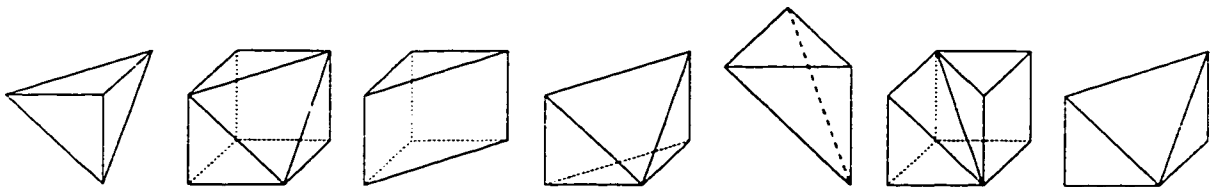


Figure 6.15: Shapes of boundary elements. Each one of these shapes has to be further partitioned to obtain simpler shapes that can be used easily.

1. faces and edges that are shared between boundary elements will be partitioned so that the resulting mesh is conformal.
2. the resulting shapes are simple, so that it is easy to generate basis functions for them. Ideally, we would like to use only cubes, the two types of pyramids and the tetrahedron that are allowed in crystalline meshes.

In order to achieve our first goal we require that every triangular face of the elements that is part of a face of the original cube should be partitioned into two triangles, while square faces should not be partitioned. Notice that faces that are not part of the faces of the original cube, but were created by the process that approximates the boundary, are not shared with any other elements. Therefore, we can partition them as we like without worrying about conformity.

The exact partitioning we chose for each one of the boundary shapes is shown in Figures 6.16 and 6.17. Notice that in some cases we obtain elements that are different from the transitional element shapes shown in Figure 4.7. Shape 1, for example, is partitioned into 4 tetrahedra, none of which is of the same shape as the tetrahedron used in the interior of the mesh. By looking at all the partitions, we observe that they introduce two different tetrahedra (used when partitioning shapes 1, 2, 4 and 7), shown in detail in Figure 6.18. Fortunately, both new shapes are very regular and without sharp angles. Therefore, they can be treated the same way as the other elements in the crystalline mesh. Moreover, since these shapes occur only at the boundaries, we expect only a very small percentage of the elements in the mesh to have these shapes.

6.7 Obtaining a Legal Mesh

In the next step, we modify the mesh to obtain a legal mesh, that can be used in the computation. Before discussing the process, let us define the level value for transitional boundary elements. Notice that the definition used for transitional elements in the interior of the cube will not work here, since it is based on the level of the element's capsule. Capsules, however, are defined as mesh cubes containing the element. Transitional elements on the boundary of the mesh might not be contained in any mesh cube. In order to eliminate this difficulty, we give a new definition of capsules for transitional boundary elements.

Definition 6.7.1 *The capsule $c(x)$ of a transitional boundary element x is the cube that was partitioned to obtain x .*

This definition is very similar to Definition 4.2.2 given for all mesh elements, the only difference being that in this case, $c(x)$ is not a mesh cube, since part of its boundary is located outside the mesh. By stretching the definition slightly, we can assign a level value to $c(x)$, by specifying that it is the same as the level value of a mesh cube y that has the same edge length as $c(x)$. Specifically, if l is the edge length of $c(x)$ and L is the largest edge in the mesh, $level(c(x)) = \log_2(L/l) + 1$.

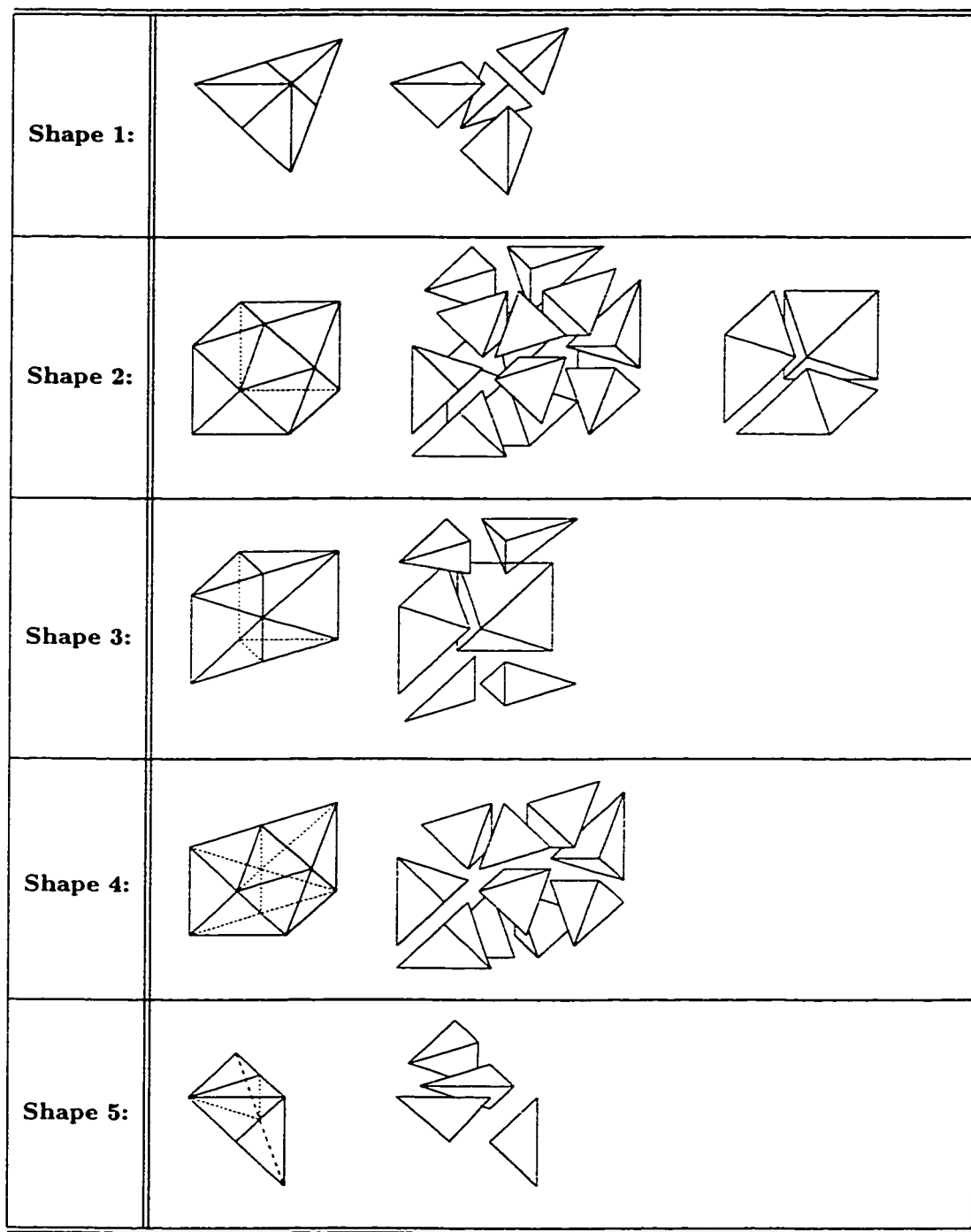


Figure 6.16: Boundary shapes 1–5 and associated partitioning. Notice that when partitioning shapes 1, 2 and 4 we obtain two types of tetrahedra that are different from the transitional tetrahedron used in the interior of the mesh.

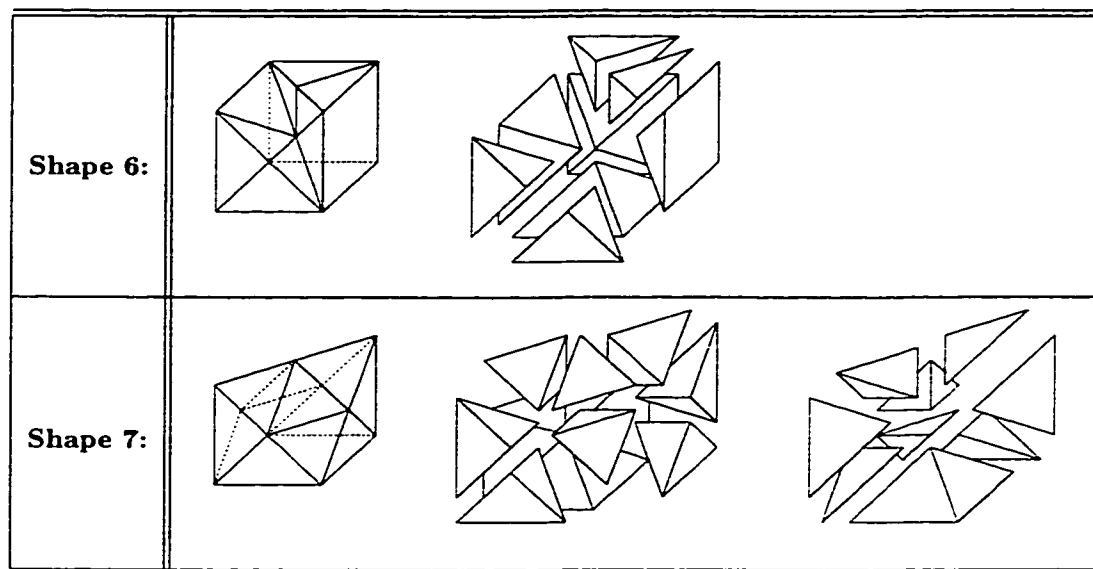


Figure 6.17: Boundary shapes 6 and 7 and associated partitioning. Shape 7 also results in two types of tetrahedra not used in the interior of the mesh.

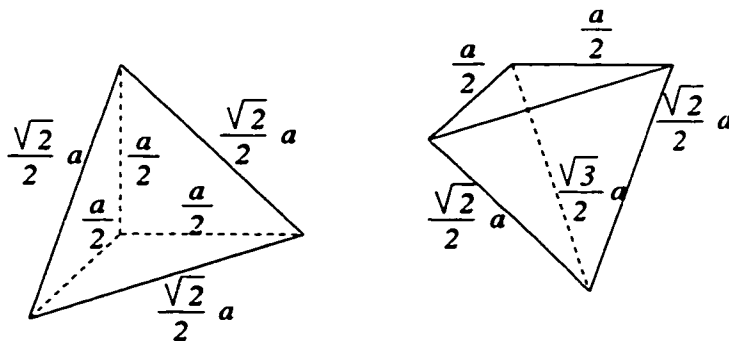


Figure 6.18: Two new tetrahedra used for approximating the boundary.

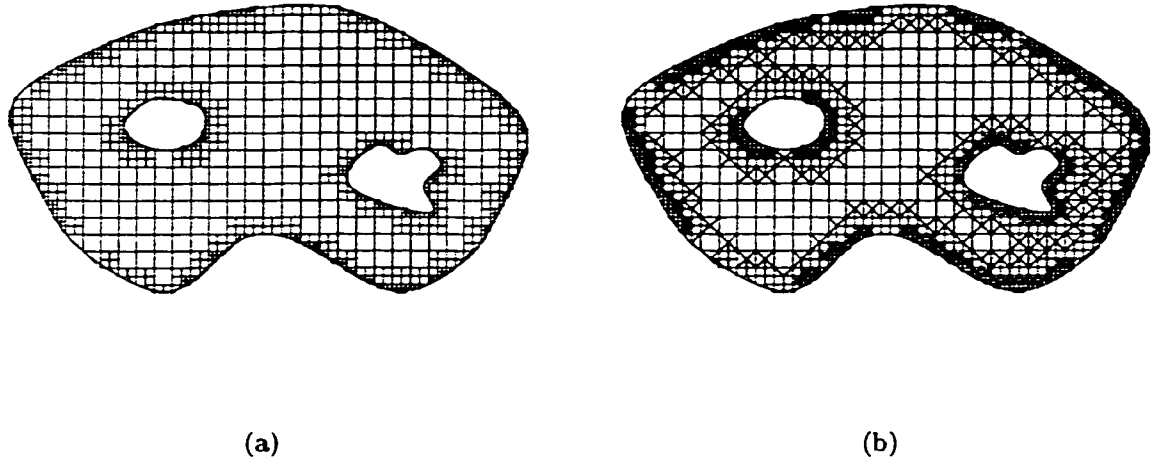


Figure 6.19: Steps used to obtain a legal mesh. (a) All square elements that have neighboring elements that are too small (more than 1 level difference) are partitioned. (b) Non-conformal elements are partitioned in order to eliminate non-conformal points.

We define the level of element x to be the level of its capsule incremented by $1/2$. This is the same definition used for transitional elements in the interior of the mesh.

The mesh the algorithm has generated up to this point is not legal for two reasons:

- It is not necessarily conformal.
- Neighboring elements might differ very much in level values. This can happen either because the user has specified different element density in different regions of the domain, or because elements that are close to the boundary have been repeatedly partitioned to obtain a better approximation. The example mesh shown in Figure 6.9(d) shows this problem.

We create the legal mesh using a two-phase process. We first ensure that no neighboring elements differ in level values by more than one. This can be done by partitioning all standard elements that have neighbors that are too small into 2^d standard elements (that is, into 4 elements for 2D problems and into 8 elements for 3D problems) iteratively, until the condition holds for the entire mesh. An example of the mesh at the end of this process is given in Figure 6.19(a). This process is called *refinement propagation* and plays a crucial role during both mesh generation and mesh adaptation. We will discuss it extensively in Chapter 7. In the same chapter, we will give an efficient alternative implementation.

In the second phase, we make the mesh conformal, by partitioning each non-conformal element in order to eliminate non-conformal points. The previous step ensures that there are only a finite number of different non-conformal configurations in the entire mesh. We assign a specific partitioning

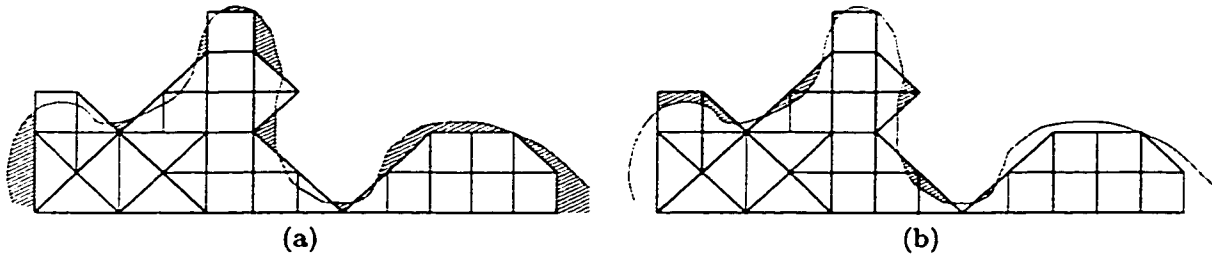


Figure 6.20: A part of the domain and the mesh covering it. (a) Regions in $\Omega - M$ are shown shaded. (b) Regions in $M - \Omega$ are shown shaded.

to each one of them, that ensures that the resulting mesh is conformal. This process is also used for mesh adaptation and we will talk more about it in the associated chapter.

6.8 Estimating the Quality of the Mesh

In order to be able to estimate how good the mesh generated by the above procedure is, we also need to be able to quantify how well it approximates the domain. We have already seen that the required approximation highly depends on the underlying physical problem. There is no automatic way to determine that the generated mesh is good enough for the associated problem, so we need to have a specification of how close the approximation needs to be, supplied by the user. We have used the term ϵ to denote the approximation error.

We need to develop a way to quantify the quality of the approximation. Specifically, we want a metric for the approximation error that exhibits the following characteristics:

1. It should be well behaved. When comparing two different approximations of the same domain the one that is intuitively better should give a smaller error value. Moreover, if the mesh and the domain coincide, the approximation error should be 0.
2. It should scale well. If we scale both the domain and the mesh associated with it by the same factor, the value of the error should remain the same.
3. It should penalize local inaccuracies, as well as global inaccuracies. Many error metrics give a small error value if most of the boundary is approximated well, while a small region is approximated badly. Since such a mesh is likely to give us a bad approximation of the solution of the partial differential equation, we want such meshes to have a high approximation error.
4. Given the description of the domain and the description of the mesh, we should be able to compute the error value, or at least a good upper bound of it.

Let Ω denote the problem's domain and M the mesh associated with it. We also use $\Omega - M$ to denote the region covered by the domain but not by the mesh while, symmetrically, $M - \Omega$ denotes

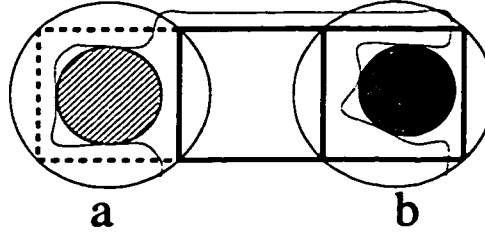


Figure 6.21: Portions of the boundary of a domain and the associated mesh (shown with heavy lines). Two circles are shown, one in $\Omega - M$ (striped circle labeled a) and the other in $M - \Omega$ (dark circle labeled b). In both cases, the circumcircle of the square (shown in solid lines) is an upper bound on the area of the circle.

the region covered by the mesh but not by the domain. Figure 6.20 illustrates these two concepts. In (a), regions in $\Omega - M$ are shown shaded while in (b), regions in $M - \Omega$ are shown shaded. $\|\Omega\|$ denotes the measure of domain Ω (its area if it is a 2D domain, or its volume if it is a 3D domain). We define the approximation error of M as follows:

Definition 6.8.1 *Let s be a maximal sphere located completely within $\Omega - M$ or $M - \Omega$, and $\|s\|$ be its measure. Then the approximation error of M is:*

$$\frac{\|s\|}{\|\Omega\|}$$

This error metric fulfills all the above requirements, as long as the boundary is sufficiently smooth. We believe that requiring that the derivatives of the functions describing the domain are continuous should suffice, however this topic needs further investigation. However, as long as this “smoothness” condition holds, the error is lower when the approximation is intuitively better. Moreover, it is 0 when Ω and M coincide and does not change when both the domain and the mesh are scaled. Since s is a maximal sphere, local inaccuracies are penalized.

We still need to discuss how we can obtain a bound of the error to be used in the mesh generation algorithm for crystalline meshes. Of course, a bound can be computed very accurately, using computationally expensive techniques, like numerical integration and extensive sampling. However, we can compute a less accurate bound if we have a lower bound on $\|\Omega\|$ and an upper bound on $\|s\|$. Both of these quantities can be easily obtained.

Specifically, we can obtain a lower bound on $\|\Omega\|$ by using the mesh created after step 11 of the `GENERATE_MESH()` algorithm (that is the mesh that contains only cubes and has all elements external to the boundary removed). A good approximation of $\|\Omega\|$ is obtained by summing up the measures of all elements in the mesh that are not boundary elements. Notice that since all elements at this are d -cubes, the computation is easy.

Moreover, the measure of the circumsphere of the capsule of any boundary element is a good upper bound for $\|s\|$, because of the way the mesh was constructed. Figure 6.21 demonstrates this. Notice that it is not possible for either $\Omega - M$ or $M - \Omega$ to be larger than the circumcircle, otherwise

the entire element would have been included in (or symmetrically, excluded from) the mesh, resulting in a region that belongs to both M and Ω (or to neither). Of course, when computing this, measure elements that have been partitioned and then restored because the partition did not give us a better approximation, are not taken into account. We use the smaller elements (in the partition) instead in our estimates. This not only gives us an easy way to compute an upper bound on the error, it also helps us determine how the error bound changes with successive applications of the `IMPROVE_APPROXIMATION()` procedure.

Let ϵ_o be the error bound computed on a mesh M and ϵ_n be the same bound computed on the mesh M' , obtained from M by applying the `IMPROVE_APPROXIMATION` procedure once. Since the capsules of the boundary elements in M' have edge lengths which are half of the corresponding edge length in M , it follows that:

$$\begin{aligned} \epsilon_n &\leq \frac{1}{4}\epsilon_o && \text{in the 2D case} \\ \epsilon_n &\leq \frac{1}{8}\epsilon_o && \text{in the 3D case} \end{aligned}$$

The above bounds guarantee that the number of iterations needed to obtain the error specified by the user is small.

Chapter 7

Mesh Adaptation

7.1 Introduction

To obtain a high-quality solution, we need a mesh with a high element density within the regions of interest. However, given the description of a physical problem and its domain, we generally cannot establish *a priori* the regions of interest. In such cases, the finite element computation proceeds as follows. Given an initial mesh over the domain, a system of equations is generated and solved. Then, bounds on the errors associated with each element are computed and used to determine in which subregions the mesh density must be modified. Finally, a new mesh is created based on this information. The procedure is applied iteratively until a satisfactory error bound is achieved over the whole domain [11, 47, 62]. This process can be significantly facilitated and accelerated by mechanisms that automatically modify the existing mesh. Such modification is called *mesh adaptation*, and involves two reciprocal processes, *refinement* and *coarsening*. Formally, the two processes can be defined as follows:

Definition 7.1.1 *Refinement is the process of partitioning the elements of the mesh M that are located in a selected region R to obtain smaller elements.*

Definition 7.1.2 *Coarsening is the process of combining the elements of the mesh M that are located in a selected region R to obtain larger elements.*

Of course refinement increases the element density in R , while coarsening decreases it. A good mesh adaptation process should have the following characteristics:

1. Retains the quality of the mesh. That is, if the adaptation is applied to a high-quality mesh, the resulting mesh should also be high-quality. If the quality of the mesh degrades with each adaptive iteration, we have to insert quality improving steps (for example smoothing or selective remeshing) after several adaptive iterations.

2. Does not generate unnecessary elements. We should try to minimize the number of additional elements created outside the region that need processing.
3. Is efficient.
4. Retains the characteristics of the original mesh. For example, if we start with a quadrilateral mesh, the resulting mesh should also be quadrilateral. If the initial mesh is conformal, the resulting mesh should also be conformal.

Adaptive computation has become very important [53], and many researchers work in this area. Of course, the adaptation methods that are developed are different for each type of input mesh.

Most of the results obtained involve triangular and tetrahedral meshes. For these problems we can distinguish two different approaches: the first type of algorithm operates on meshes that are Delaunay triangulations and create a mesh that is also Delaunay [32, 38, 42, 59, 73, 78, 93]. These procedures focus on calculating an optimal position where new points can be inserted. Then, a portion of the mesh around the new point is destroyed and remeshed. Another family of methods focuses on partitioning the current elements. The popular Rivara refinement method [71, 70], for example, falls in this category. Several other methods for partitioning have been developed that use different criteria to choose how each element should be refined [51, 52, 58, 66, 67]. Some work has also been done on quadrilateral [42, 80], hexahedral [50] and mixed meshes [40, 82].

In this chapter we will describe how adaptive computation can be done for crystalline meshes¹. We will focus on the process of refinement and discuss the coarsening only briefly, since the ideas and techniques used are analogous.

7.2 General Refinement Technique

The problem of refinement for crystalline meshes can be stated as follows: we are given a legal crystalline mesh M and a (not necessarily continuous) region R to be refined. We should refine the elements in the region in such a way that the resulting mesh is also legal. We can achieve this by doing refinement in a very controlled way. Figure 7.1 is a flow diagram showing an initial approach for the refinement algorithm. In the first step (clean-up phase) we replace all transitional, as well as standard elements contained in a transitional capsule, by their capsule, thereby obtaining a conformable mesh. Then, all the elements of the resulting mesh that fall completely within the selected region are refined. Standard elements with edge length l are partitioned into standard elements with edge length $l/2$, so that a standard element of level j will be replaced by 2^d elements of level $j + 1$.

Definition 7.2.1 *The process of partitioning a standard element x of a crystalline mesh with edge length l into 2^d standard elements with edge length $\frac{l}{2}$ is called standard refinement.*

¹Parts of this work were previously published as [20]

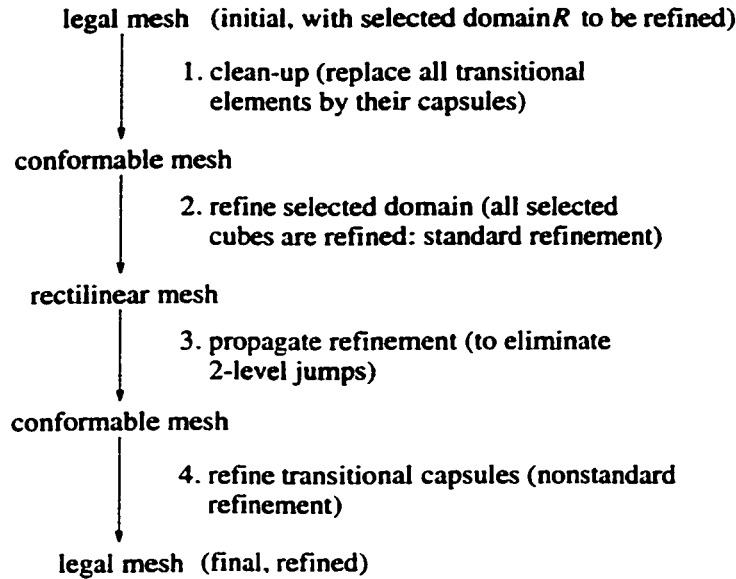


Figure 7.1: Flow diagram of the refinement process.

At the end of this step, the mesh may contain neighboring² elements that have level values that differ by 2. In such case, the resulting mesh is no longer conformable, so the refinement is propagated in order to obtain a conformable mesh. The propagation step also identifies the elements that will be transitional capsules in the output mesh. In the final step we refine each transitional capsule appropriately. This refinement depends upon the levels of the capsule's neighbors, and is referred to as *nonstandard refinement*.

7.3 Complexity of Refinement

We would like to know the inherent complexity of the refinement step. That is, we would like to count the number of elements that are different in the input and output meshes. Of course, all elements in R need to be modified during the refinement. Elements outside R may be modified as a result of the refinement propagation step.

We now analyze the refinement propagation step (step 3). At an abstract level, we identify each element of the rectilinear mesh under consideration with the node of a node- and edge-weighted directed graph G . The weight of a node is its level. We establish an arc (v_x, v_y) from node v_x to node v_y , if the corresponding elements x, y are neighbors and $level(x) > level(y)$. We assign to (v_x, v_y) the weight $w(v_x, v_y) = level(x) - level(y)$ (Figure 7.2).

It is clear that G is acyclic, and that if l is the maximum level of an element in the mesh, the maximum length of a path in G will be l . A path in G is said to be *smooth* if all the edges in the

²For a discussion of the meaning of neighbors and definitions of other technical terms used, see Section 4.2.

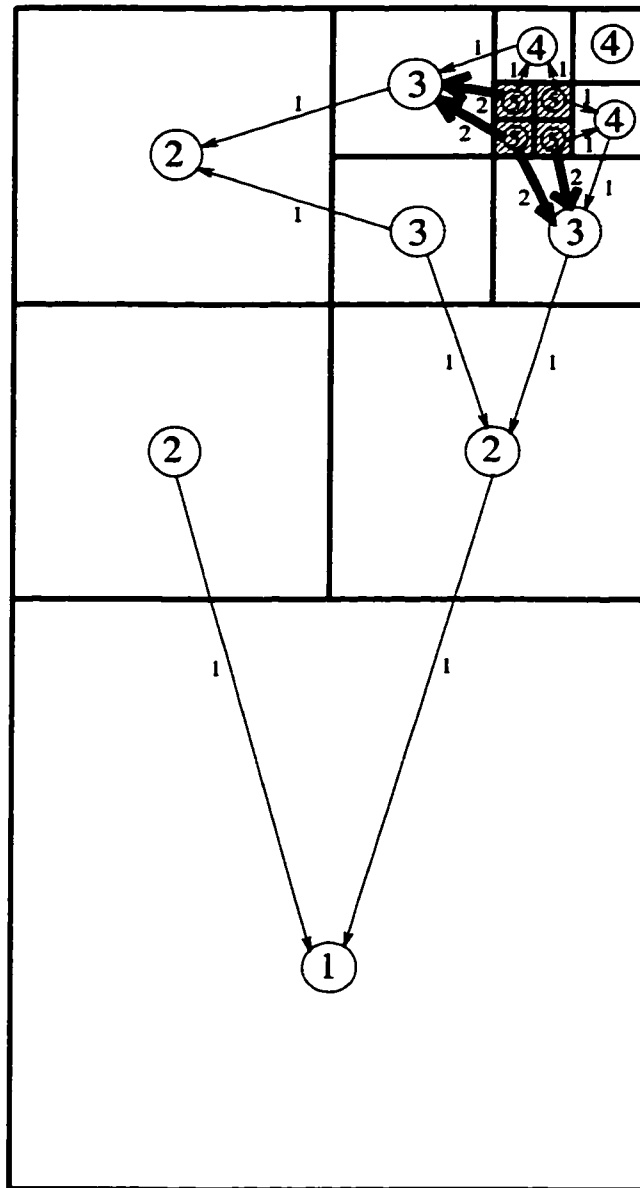


Figure 7.2: A rectilinear mesh and its associated graph. The heavy arcs have weight larger than 1. The region that was refined during the last step is shown shaded.

path have weight 1. The mesh is conformable if and only if all paths of G are smooth.

By the action of the standard refinement (step 2), the weights of a subset of arcs of the (conformable) mesh are increased by 1, so that 2 is the largest weight of arcs of the resulting rectilinear mesh M . Consider any path $p = (v_1, v_2, \dots, v_s)$ in the graph $G(M)$ of M . We claim:

Lemma 7.3.1 *In any nonsmooth path p in $G(M)$, there is at most one arc of weight 2.*

Proof: We will start with a case analysis. Consider two arbitrarily chosen neighboring elements x and y in the original conformable mesh (the mesh obtained after the clean-up phase). By the definition of a conformable mesh we know that either the two elements have the same level value, or their level values differ by 1. In the second case we can assume, without loss of generality, that $level(x) = level(y) + 1$ (see Figure 7.3, case 2). During the refinement phase, none, or one, or both of the elements are to be refined. This gives us eight distinct cases, shown in Figure 7.3. In the same figure we also show the subgraph of G that is obtained in each of the cases, where bold arrows indicated arcs with weight 2. Notice that if an arc (v, w) has weight 2, we can conclude that the element associated with v was created during the last refinement step, while the element associated with w was not refined (Case 2(c)).

Using this case analysis, we can prove the Lemma by contradiction. Suppose that in p there are two arcs (v_i, v_{i+1}) and (v_j, v_{j+1}) with $(1 \leq i < j \leq s - 1)$ of weight 2, with no intermediate weight-2 arc (Figure 7.4).

From the case analysis we know that since v_j has an outgoing weight-2 arc, the element associated with it has been created during the last refinement step. We also know that v_j has an incoming weight-1 arc (v_{j-1}, v_j) . We see that the only case where an element that was refined during the last step has an incoming weight-1 arc, is Case 2(d), and this implies that the element corresponding to v_{j-1} was also created during the last refinement. We can follow the path from v_{i-1} to v_j backwards, applying the same reasoning at each arc, concluding that all nodes on the path from v_{i+1} to v_j have been created during the last refinement step. In particular, the element corresponding to v_{i+1} has been created during the last refinement step. However, from the case analysis we know that no element created during the last refinement step can have a node associated with it with an incoming weight-2 arc, a contradiction which proves the Lemma. \square

Definition 7.3.1 *A refinement path of M is a path whose first arc has weight-2 and is followed by a run of weight-1 arcs.*

Refinement propagation occurs exactly on refinement paths. Lemma 7.3.1 proves that refinement propagation can be done in one pass. Assume that x, y are two neighboring elements in the mesh we obtained after step 2, that v, w are the two nodes associated with them, and (v, w) is a weight-2 arc. We start by refining y , thereby eliminating the level-2 jump between x and y . If w has outgoing arcs, we follow them and refine the elements associated with them. We continue following outgoing arcs and refining the corresponding elements until we reach the end of the refinement path. Lemma

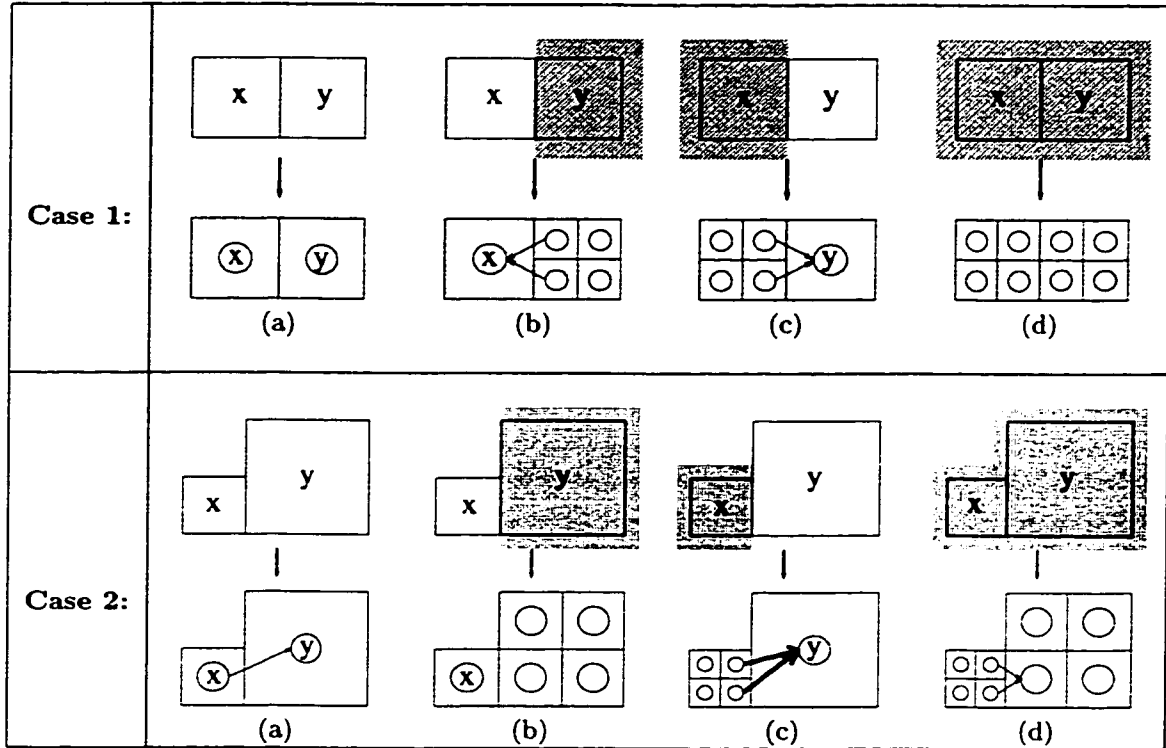


Figure 7.3: A 2D illustration of the possible neighboring arrangements in the original conformable mesh and in the mesh resulting after the refinement. Elements selected to be refined are shown shaded. Together with the elements resulting from the refinement we also show the subgraph associated with them. Case 1: The neighbors originally have the same level value. Case 2: The level values of the neighbors differ by 1. Notice that Case 2(c) is the only one resulting in arcs with weight 2 (shown heavier).



Figure 7.4: The path used in the proof. The arcs (v_i, v_{i+1}) and (v_j, v_{j+1}) have weight 2. All other arcs shown have weight 1.

7.3.1 proves that, once an element is refined, the resulting elements will not be refined again during the refinement propagation.

In order to obtain an upper bound on the number of elements that will be modified during the refinement propagation step, we also need to estimate how many refinement paths, we have and what their length is. Using the definition of the refinement paths we can easily prove the following theorem:

Theorem 7.3.2 *If $l > 0$ is the maximum level occurring in the mesh, the depth of refinement propagation is at most l .*

The following lemmas can be used to bound the number of refinement paths:

Lemma 7.3.3 *Let (v, w) be the first arc in a refinement path in $G(M)$ and x, y be the two mesh elements corresponding to v and w respectively. Then x is an element on the boundary of R and y is outside R (where R is the region that was chosen to be refined).*

Proof: Since (v, w) is the first arc in a refinement path, it has weight 2 (by definition). By using the case analysis shown in Figure 7.3, we conclude that x was created during the last refinement step, while y was not refined. So, y is outside R , while x is inside R , and has a neighbor (y) that is outside R . This means that x is on the boundary of R . \square

The above lemma proves that if $m(\delta(R))$ is the number of elements on the boundary of R , there can only be $m(\delta(R))$ nodes in the graph from which refinement paths start. However, if a refinement path can branch out in an arbitrary way, one may suspect that the number of elements on a refinement path is potentially exponential in l . Theorem 7.3.4 proves that this is not the case.

In order to prove this theorem we need to distinguish between the distance between two nodes in the refinement graph and the physical distance between two points placed on the domain. Following established practice, we will use the following definition:

Definition 7.3.2 *The distance $d(u, v)$ between two nodes u and v in a graph G is the minimum number of edges in any path from u to v .*

For clarity, let us introduce some more notation:

Definition 7.3.3 *Let x be an element in the rectilinear mesh obtained after the standard refinement phase, and let v be the node in $G(M)$ associated with it. The refinement set $S(v)$ of v is defined as:*

$$S(v) = \{w \in G(M) : w \text{ is on a refinement path starting from } v\}$$

Intuitively, $S(v)$ contains all nodes associated with elements that have to be refined because v was created. Notice that the definition also implies that if no refinement paths start from v , $S(v) = \emptyset$.

Before we are ready to state the theorem, we need one more definition:

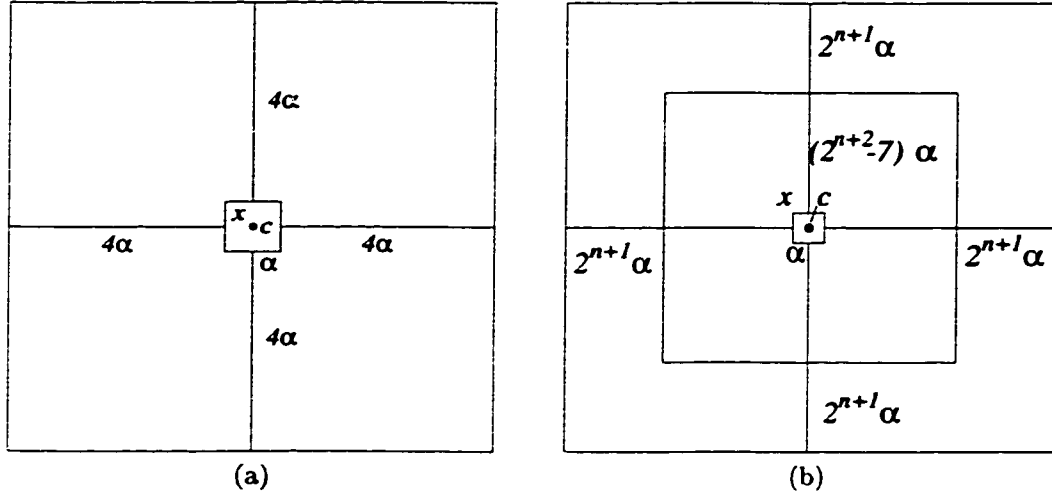


Figure 7.5: Two-dimensional illustration of Lemma 7.3.5. (a) All elements that have distance 1 from x are completely included in the bounding box with center c and edge length 9α . (b) All elements that have distance n from x are completely included in a box with center c and edge length $(2^{n+3} - 7)\alpha$.

Definition 7.3.4 Let v be a node in $G(M)$ associated with element x . For each $n > 0$

$$S_n(v) = \{w \in G(M) : w \in S(v) \text{ and } d(v, w) = n\}$$

Therefore, $S_n(v)$ contains all nodes in the refinement set of v that are exactly n edges away from it.

Theorem 7.3.4 Let x be an element in the rectilinear mesh obtained after the standard refinement phase, and let v be the node in $G(M)$ associated with it. Then, $\|S_i(v)\|$ is bounded by a constant, for any $i > 0$.

We will prove this theorem separately for the 2D and the 3D case, after the definition of neighboring elements for the two cases has been presented. For now, we will prove the following lemma, which will be used in the proof of Theorem 7.3.4.

Lemma 7.3.5 Let x be an element in the rectilinear mesh obtained after the standard refinement phase, with edge length α and center c and let v be its associated node in $G(M)$. All elements associated with nodes in $S_n(v)$, for all $n > 0$, are completely contained in a box with center c and edge length $(2^{n+3} - 7)\alpha$.

Proof: We will prove the lemma by using induction on the length of the path i .

- **Base case:** $i = 1$

Since x is an element at the beginning of the refinement path, all nodes in $S_1(v)$ are connected to it by a weight-2 arc. So the level value of the elements associated with them is $level(x) - 2$ and therefore their edge length is 4α . They also have to be adjacent to x (Figure 7.5(a)). This means that they have to be completely contained in a box with center c and edge length:

$$\alpha + 2 \cdot 4\alpha = 9\alpha = (2^{1+3} - 7)\alpha$$

- **Induction hypothesis:** $i = n - 1$

We assume that all elements associated with nodes in $S_{n-1}(v)$ are completely contained in a box with center c and edge length $(2^{n+2} - 7)\alpha$.

- **Inductive step:** $i = n$

All elements associated with nodes in $S_n(v)$ have a level value of $level(x) - n - 1$. This implies that their edge length is $2^{n+1}\alpha$. Using the induction hypothesis we know that they have to be adjacent to elements that are completely contained in a box with center c and edge length $(2^{n+2} - 7)\alpha$ (Figure 7.5(b)). So, the edge length of the new box is:

$$2 \cdot 2^{n+1}\alpha + (2^{n+2} - 7)\alpha = 2^{n+2}\alpha + 2^{n+2}\alpha - 7\alpha = (2^{n+3} - 7)\alpha$$

This proves the lemma. □

The proof for Theorem 7.3.4 uses Lemma 7.3.5 and the fact that elements that have distance n from x (in a graph-theoretic sense) have edge lengths equal to $2^{n+1}\alpha$. This allows us to count how many elements can have distance n from x and discover that it is a constant.

By combining Theorem 7.3.2, Lemma 7.3.3 and Theorem 7.3.4 we can trivially prove the following:

Corollary 7.3.6 *Let $m(\delta(R))$ be the number of elements on the boundary of R . The number of elements that need to be modified during the refinement propagation step is $O(l \cdot m(\delta(R)))$.*

After the refinement propagation, we obtain a conformable mesh. Now we can, in one pass, locate all non-conformal elements and refine them in order to obtain a conformal mesh. This happens during Step 4 of the refinement procedure, and the non-conformal elements located are (standard) elements that have neighbors with a higher level value. The refinement used during this step is called *nonstandard* refinement and will result in some transitional elements, at the boundaries between regions having different element density.

Knowing the length of the refinement path allows us to compute the complexity of the refinement algorithm. It is given in the following Corollary of Theorems 7.3.2 and 7.3.4.

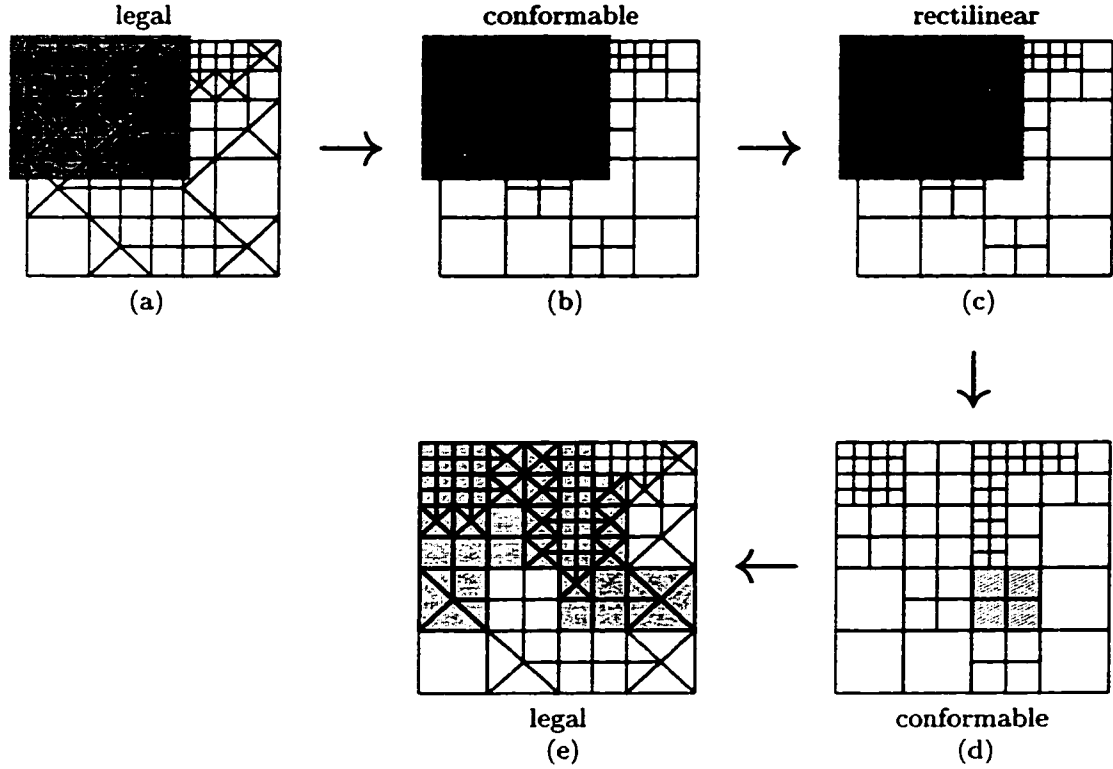


Figure 7.6: (a) A region R has been selected for refinement (shown shaded). (b) The non-standard elements are removed. (c) Squares that fall completely within R are regularly refined. (d) The refinement is propagated to neighboring elements (striped element). (e) Non-conformal elements are refined. The striped region contains all the elements that were modified by the refinement. Notice that it almost coincides with R .

Corollary 7.3.7 *Let R be the region to be refined, and let $m(R)$ and $m(\delta(R))$ denote respectively, the numbers of elements in R and on its boundary. Then the number of elements that need to be modified during the refinement is $O(m(R) + l \cdot m(\delta(R)))$. If $m(\delta(R)) = o(m(R))$, the complexity of the refinement is $\Theta(m(R))$.*

Note that condition $m(\delta(R)) = o(m(R))$ is commonly met, and in this case, the complexity of refining the region R dominates the complexity of the refinement propagation. In the following chapters we present efficient refinement techniques for crystalline meshes in 2D and 3D.

7.4 2D Mesh Refinement

The general refinement process discussed in Section 7.2 is illustrated in Figure 7.6 for a 2D mesh. We start with a legal crystalline mesh and a region R that needs to be refined (shown shaded in Figure 7.6(a)). During the clean-up phase, we coarsen all transitional elements, thereby obtaining a non-conformal mesh that consists only of squares (Figure 7.6(b)). Clearly, the resulting mesh

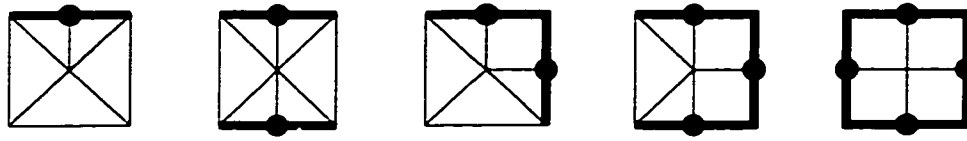


Figure 7.7: All possible non-conformal configurations for elements in a 2D conformable mesh. Bold sides denote boundaries shared with higher level elements.

is conformable. We then use standard refinement to refine all elements within the selected region (Figure 7.6(c)). The resulting mesh is rectilinear but might not be conformable. In order to make it conformable we propagate the refinement to eliminate all two-level jumps (Figure 7.6(d)). Finally, we use non-standard refinement to refine all non-conformal elements, in order to obtain a legal mesh (Figure 7.6(e)), which will be used in the computation. All shaded elements in Figure 7.6(e) are those altered by the refinement process.

Refinement propagation ensures that the resulting mesh is conformable. Each element in the conformable mesh will have edges that are either conformal or have a vertex in the middle. This means that there are only five different configurations (modulo right-angle rotations) a non-conformal element can have, depending on the subset of sides it shares with higher level elements. All possible configurations are shown in Figure 7.7, with emphasis given to the non-conformal vertices and the sides containing them. In the same figure we show an appropriate non-standard refinement for each configuration. Notice that all non-conformal edges are divided in two, while all conformal edges are not divided. Therefore, the mesh will be conformal (and legal) after the last step of the algorithm has been executed.

Going back to the example of Figure 7.6, we realize that the refinement procedure, as described so far, is not very efficient. For example, notice that the transitional elements in the lower right-hand corner of the mesh are removed during the clean-up phase, resulting in a standard element. During the last phase of the refinement procedure, this element is refined, in order to obtain a conformal mesh. The resulting elements are exactly the same as in the original mesh. This suggests that we would obtain a more efficient algorithm if we could identify elements that are identical in the input and output meshes and avoid processing them. This becomes more important in cases where the input mesh consists of many elements, and the region R to be refined contains only a small portion of them. In general, we would like the refinement procedure to run in time proportional to the number of elements that need to be modified, and not to the number of elements in the entire mesh, and produce exactly the same meshes as the procedure shown in Figure 7.1.

Corollary 7.3.7 states that the number of elements that need to be modified by the refinement algorithm is $O(m(R) + l \cdot m(\delta(R)))$. We would like to design an algorithm for refinement that has this complexity. The algorithm would then be optimal. Let us start by stating and proving Theorem 7.3.4 for the 2D case.

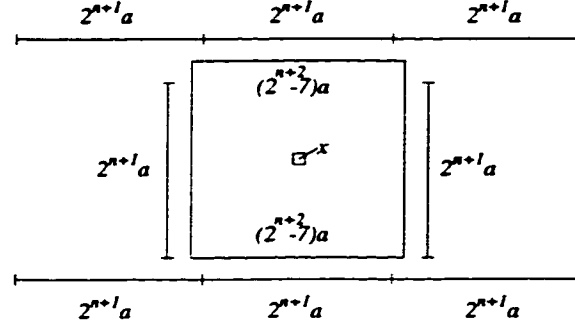


Figure 7.8: The maximum number of elements that are on a refinement path starting from x and have distance n to it is 8.

Theorem 7.4.1 *In the 2D case,*

$$\|S_n(v)\| \leq 8, \quad \forall n > 0$$

Proof: Lemma 7.3.5 proves that all elements associated with nodes in $S_{n-1}(v)$ are contained in a box with edge length $(2^{n+2} - 7)\alpha$, where α is the length of the edge of x . We also know that the level value of the elements associated with nodes in $S_n(v)$ is $level(x) - n - 1$, which implies that their edge length is $2^{n+1}\alpha$.

By the definition of neighbors, we know that the elements at distance n have to share an edge with the elements at distance $n - 1$. But since the length of the edges of the bounding box is less than twice the length of the edge of the elements, the maximum number of elements at distance n from x is 8 (see Figure 7.8). \square

The bound proved above is overly pessimistic. By using a more complicated method (not reported here) that takes into account the structure of crystalline meshes, we can prove a bound of 4. However, even the bound of 8 is enough to guarantee that the number of elements that need to be modified is $O(m(R) + l \cdot m(\delta(R)))$. An efficient algorithm should run in time proportional to this quantity.

In order to develop an efficient algorithm, we notice that all elements in R need to be modified. Moreover, elements that are not in R need to be modified only if at least one of their neighbors is modified. So we can avoid modifying elements that are not affected by the refinement procedure by keeping track of all modified elements and checking all their neighbors to see if modifications are in order.

The algorithm starts by replacing all transitional elements, whose capsule falls completely within the selected region R , with said capsule (Figure 7.9(b)). This is equivalent to the clean-up phase, but is performed only on elements in R . Next, all standard elements completely within R are refined, using standard refinement (Figure 7.9(c)). Now we have to perform the two last steps of the procedure, refinement propagation and non-standard refinement of non-conformal elements. The refinement propagation is accomplished by using the procedure `PROPAGATE_REFINEMENT()`, shown as pseudocode in Algorithm 7.1.

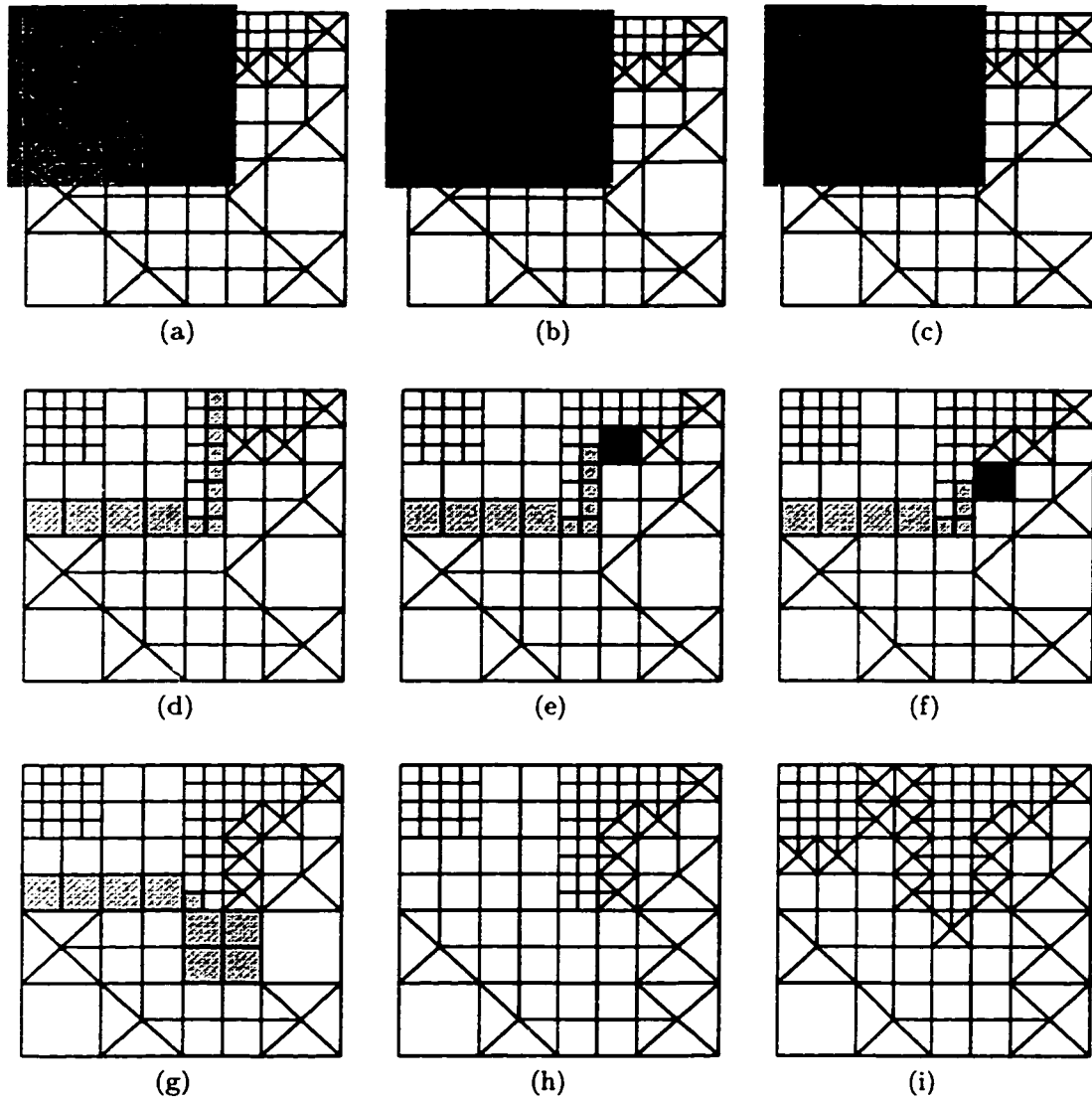


Figure 7.9: An example of the refinement procedure: (a) The initial mesh. The region R is shown shaded. (b) Non-standard elements in R are removed. (c) Elements in R are refined (standard refinement). (d) Elements on the boundary of R are inserted into Q (shown shaded). (e) Level difference of $1/2$: the neighbor is coarsened and its capsule is refined. (f) Level difference of 1, and neighbor does not belong to a transitional capsule: the neighbor is refined using non-standard refinement. (g) The neighbor belongs to a transitional capsule: the capsule is refined using standard refinement and the resulting elements are inserted in Q . (h) State of mesh when Q is empty. (i) Non-standard refinement is used to make the mesh conformal. The resulting mesh is legal.

```

PROPAGATE-REFINEMENT( $M, R$ )
1  for each element  $x$  on the boundary of  $R$ 
2  do insert  $x$  into  $Q$ 
3  while  $Q \neq \emptyset$ 
4  do  $x \leftarrow \text{head}[Q]$ 
5     for each neighbor  $y$  of  $x$  with  $\text{level}(y) < \text{level}(x)$  AND  $y \notin Q$ 
6     do  $\delta \leftarrow \text{level}(y) - \text{level}(x)$ 
7         switch
8             case  $\delta = \frac{1}{2}$  :
9                  $z \leftarrow \text{COARSEN}(y)$ 
10                NONSTANDARD-REFINE( $z$ )
11             case  $\delta = 1$  AND  $y \notin$  a transitional capsule :
12                NONSTANDARD-REFINE( $y$ )
13             case  $(\delta = 1$  AND  $y \in$  a transitional capsule ) OR  $\delta = \frac{3}{2}$  :
14                 $z \leftarrow \text{COARSEN}(y)$ 
15                STANDARD-REFINE( $z$ )
16                insert all resulting elements in  $Q$ 

```

Algorithm 7.1: The algorithm used to simulate a breadth-first search on the graph G . $\text{COARSEN}(y)$ replaces all elements belonging to the transitional capsule of y with the capsule z and returns it.

Since all elements within R have been refined, paths of G within R are, by construction, smooth. So, the refinement propagation will start from elements that are at the boundary of R and proceed outwards. We will simulate a breadth-first search of the graph G using a queue Q . Throughout the execution of the algorithm, Q contains elements that have been modified by the refinement procedure and whose neighbors need to be checked and maybe modified.

We start by inserting into Q all elements that are on the boundary of R (shown lightly shaded in Figure 7.9(d)). The algorithm then proceeds by extracting elements from the queue, examining all their neighbors, modifying them if necessary, and when appropriate, inserting new standard elements that need processing into the queue. The algorithm terminates when the queue is empty.

To gain the correct insight into the operation of the proposed algorithm, we will prove some facts about it.

Lemma 7.4.2 *During the execution of the algorithm PROPAGATE_REFINEMENT(), Q contains only standard elements that have been created during the current refinement step.*

Proof: We initialize Q by inserting all elements on the boundary of R . During the previous two steps of the refinement procedure all transitional elements in R have been removed and then all elements in R (which are now standard) have been refined using standard refinement. So, all elements in R are now standard elements that have been created during the current refinement step. Since the elements inserted in Q are in R , this is enough to prove that the lemma is true when Q is initialized.

The only other point where elements are inserted into Q is line 16. These elements are the result of a STANDARD_REFINE() operation (line 15). Therefore, these elements, too, are standard elements

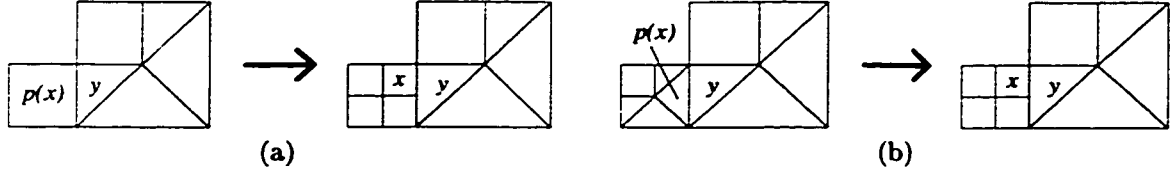


Figure 7.10: Examples showing the worst cases for Lemma 7.4.4: (a) x is created by refining a standard element. In this case, $level(x) - level(y) = \frac{3}{2}$. (b) x was created by first removing some transitional elements and then using standard refinement to refine their capsule. In this case also, $level(x) - level(y) = \frac{3}{2}$.

created during the current refinement step. □

Lemma 7.4.3 *The procedure PROPAGATE_REFINEMENT() terminates.*

Proof: The procedure terminates when the queue Q is empty. We have already proven that only standard elements that have been created during the current refinement step are inserted into Q . Moreover, Lemma 7.3.1 proves that once an element has been refined, none of the resulting element will be refined again during the refinement propagation step.

Combining these two facts, we conclude that once an element has been removed from the queue, it will not be refined again, and no elements resulting from it can be inserted into the queue. This proves that only a finite number of elements will ever be inserted into Q , and therefore the procedure will terminate. □

Lemma 7.4.4 *Let x be an element that has just been extracted from the queue Q and y one of its neighbors. Then, $level(x) - level(y) \leq \frac{3}{2}$.*

Proof: First of all, notice that whenever an element is removed from the mesh by the refinement procedure, it is replaced by elements of the same or higher level value. Therefore, in order to prove the bound we can assume that y is an element that is present in the input mesh.

Since x was just removed from Q , we know that it is a standard element created during the current refinement step (Lemma 7.4.2). We can distinguish two cases, depending on how x was created:

- x was created by refining a standard element in the original mesh, using standard refinement (Figure 7.10(a)). Let $p(x)$ denote this standard element in the original mesh. Obviously, y was a neighbor of $p(x)$ in the input mesh, and since the mesh was legal:

$$level(p(x)) - level(y) \leq \frac{1}{2}.$$

Moreover,

$$level(x) = level(p(x)) + 1.$$

Combining these two facts we get:

$$level(x) - level(y) \leq \frac{3}{2}.$$

So, the lemma holds in this case.

- x was created by coarsening a transitional element, and then refining the resulting transitional capsule (Figure 7.10(b)). Let $p(x)$ be the transitional element in the input mesh that was a neighbor of y . Since the original mesh is legal:

$$level(p(x)) - level(y) \leq 1.$$

We also know that:

$$level(x) = level(p(x)) + \frac{1}{2}.$$

Combining these two facts we get:

$$level(x) - level(y) \leq \frac{3}{2}.$$

This is enough to prove that the lemma always holds. \square

Using the above lemma and the fact that level values are always either integers or *integer* + $\frac{1}{2}$ we can prove that:

Corollary 7.4.5 *Let x be an element that has just been extracted from the queue Q , and y one of its neighbors. Let $\delta = level(x) - level(y)$. If $\delta > 0$ then $\delta \in \{\frac{1}{2}, 1, \frac{3}{2}\}$.*

During the PROPAGATE_REFINEMENT() subroutine, and while we are examining the neighbors of x , we are only interested in neighbors with level values lower than $level(x)$, since the refinement cannot propagate to neighbors with the same or higher level value. We will now describe and justify the actions that need to be performed in each of the cases addressed in Corollary 7.4.5.

1. $level(x) - level(y) = \frac{1}{2}$ (Figure 7.11(a)).

This means that y is a transitional element, whose capsule has a level value of $level(x) + 1$. In this case, the refinement does not have to be propagated, but y might not be conformal (because of the construction of x). So, we use the subroutine COARSEN() to replace y and all the elements in its capsule by the capsule, and refine the capsule again, using non-standard refinement. This process is guaranteed to make the boundary between x and the capsule of y conformal.

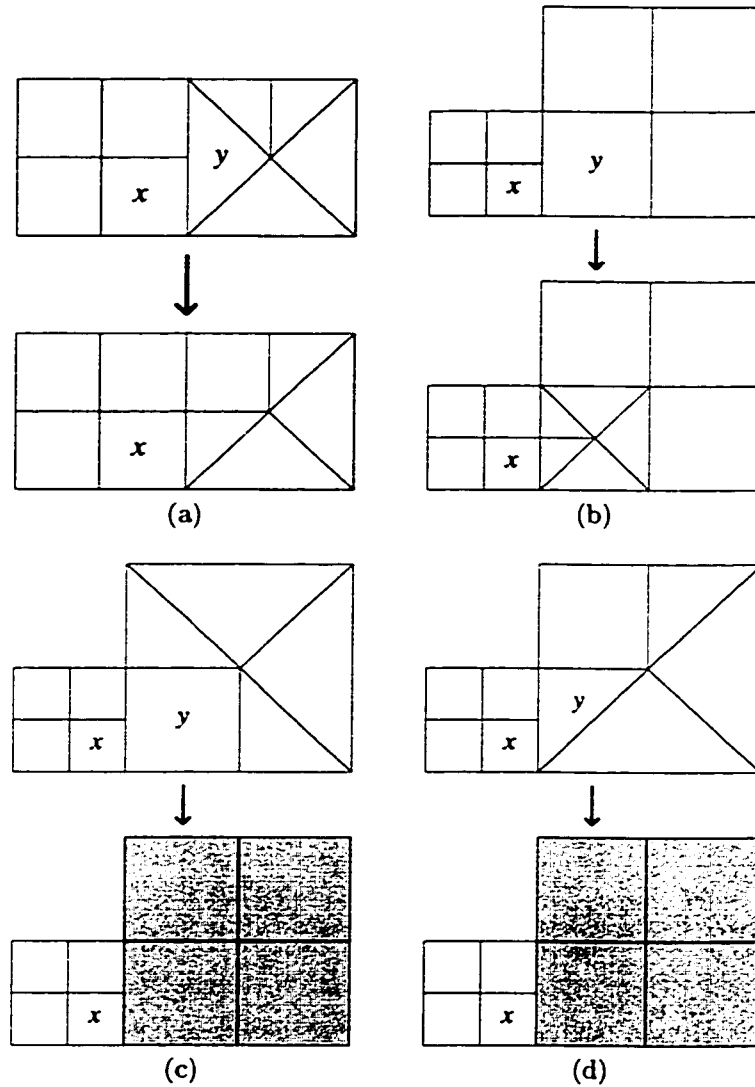


Figure 7.11: Examples of how neighbors of x with higher level value are processed by the algorithm. (a) $level(x) - level(y) = \frac{1}{2}$. The COARSEN() subroutine is used and the obtained transitional capsule is refined using non-standard refinement. (b) $level(x) - level(y) = 1$, and y does not belong to a transitional capsule. Non-standard refinement is used to refine y . (c) $level(x) - level(y) = 1$ and y belongs to a transitional capsule. The capsule is obtained by using the COARSEN() subroutine and then refined with standard refinement. The resulting standard elements, shown shaded, are inserted into the queue. (d) $level(x) - level(y) = \frac{3}{2}$. This case is handled in exactly the same way as case (c).

2. $level(x) - level(y) = 1$ (Figure 7.11(b) and 7.11(c)).

In this case, y is a standard element. We can again distinguish two cases.

In the first case (Figure 7.11(b)), y does not belong to a transitional capsule. This means that y has been created by a standard refinement operation. Therefore, it is enough to refine y , using non-standard refinement, in order to make the boundary between x and y conformal.

If y belongs to a transitional capsule (Figure 7.11(c)), it has been created by a non-standard refinement operation. The transitional capsule to which y belongs, and x , have a difference of two in level values. In order to understand what the appropriate action is in this case, we will look at how the original algorithm, presented in Figure 7.1, would handle this case. Since we want the two refinement algorithms to produce the same mesh, we should simulate its actions. The original algorithm starts by removing all transitional and standard elements contained in transitional capsules by their capsules. Therefore, y would have been removed, and the corresponding neighbor of x would be y 's transitional capsule $tc(y)$. Since $level(x) - level(tc(y)) = 2$, the refinement has to be propagated, and $tc(y)$ will be refined, using standard refinement. Finally, the refinement will be propagated, if necessary, to the neighbors of the newly created elements. We can achieve the same results by using the COARSEN subroutine to replace y and all the elements in the same transitional capsule by the capsule. Next, we refine the capsule using standard refinement. This entire process is a refinement propagation step. In order to ensure that the refinement will be propagated further, if necessary, we insert the newly created elements into the queue.

3. $level(x) - level(y) = \frac{3}{2}$ (Figure 7.11(d)).

This case is similar to the second case. The element y is a transitional element and its capsule has a level difference of two with x . Using the same reasoning as above, we conclude that the refinement has to be propagated. We coarsen y and then use standard refinement to refine its capsule. Finally, we insert all elements created by this refinement into the queue.

Notice that during the while-loop (lines 3-16), we never modify elements that are already in the queue. This is justified, because only elements created in this refinement step are placed in the queue (Lemma 7.4.2), and we know that these elements do not need to be refined again during the refinement propagation step (Lemma 7.3.1). So, the refinement propagation will be executed correctly. Figure 7.9(h) shows the mesh immediately after the queue has become empty and the execution of the subroutine PROPAGATE_REFINEMENT() is about to terminate.

As the last step of the refinement procedure, the remaining non-conformal elements are refined, using non-standard refinement. The resulting mesh is legal (Figure 7.9(i)) and is guaranteed to be the same as the one resulting from applying the procedure shown in Figure 7.1 to the same initial mesh with the same region R .

As was our objective, the new refinement procedure only takes time proportional to the number of elements that need to be modified. If the number of elements on the boundary of R is asymptotically

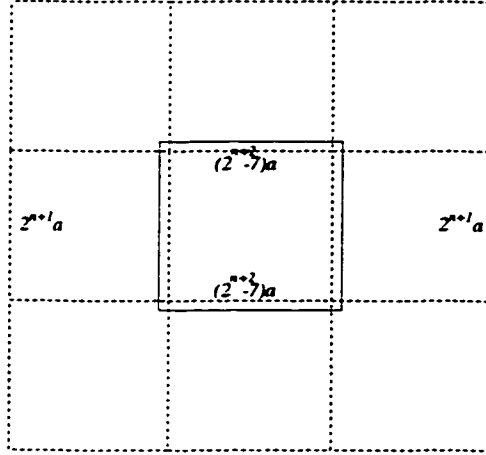


Figure 7.12: Up to 9 elements at level $level(x) - n - 1$ can share part of a face of the box containing all elements at distance $n - 1$ from x . The faces of the elements at distance n are shown dashed.

smaller than the number of elements in R , the entire procedure runs in time proportional to the number of elements in R .

7.5 3D Mesh Refinement

The refinement of 3D crystalline meshes conforms to the general philosophy of Section 7.2 and can be done using Algorithm 7.1, which was also used for 2D meshes. Theorem 7.3.4 still holds but with a different constant. Let us state and prove it for the 3D case.

Theorem 7.5.1 *In the 3D case*

$$||S_n(v)|| \leq 98, \quad \forall n > 0$$

Proof: As in the 2D case, the box containing all elements whose associated nodes are in $S_{n-1}(v)$ has edge length $(2^{n+2} - 7)\alpha$ (where α is the length of the edges of x). Moreover, the edge length of elements at distance n is $2^{n+1}\alpha$.

Using a very pessimistic estimate on the number of elements with level value $level(x) - n - 1$ that can share a face, edge or vertex with elements in the interior of the box we have:

- Up to 9 elements that share a face with each face of the box (Figure 7.12) resulting in 54 elements in total.
- Up to 3 elements that share an edge with each edge of the box, giving a total of 36 elements.
- Up to 8 elements sharing only a vertex with the box.

This gives us a bound of at most 98 elements that can have a distance n from x . □

The proof shown above does not use any information about the structure of crystalline meshes. By using a more complicated method that relies on the structure we can prove a bound of 26. Both bounds, however, are sufficient to prove that Corollary 7.3.7 holds.

So Algorithm 7.1 can be used and will have the same running time. However, there are some modifications due to the new definition of neighboring elements. According to this definition, a 3D mesh is conformable if elements that share a *vertex* have a level difference of at most one. Since we want the refinement propagation step to result in a conformable mesh, we have to propagate the refinement from an element x to an element y even if the two elements share only a vertex (remember that, by definition, a crystalline mesh is conformable if neighboring elements differ by at most one in level values, and that we defined neighbors to be elements that share at least a vertex in the 3D case). This means that, in general, more elements will be affected by the refinement, but the difference will not be significant. For example, if the region R to be refined has the shape of a parallelepiped, the number of additional elements that will be affected by the refinement is bounded from above by $8l$ (where l is the maximum level occurring in the mesh). Remember that according to Corollary 7.3.7, the number of elements affected by the refinement is $\Theta(m(R) + l \cdot m(\delta(R)))$.

We also have to specify how the last step of the algorithm (using non-standard refinement to refine non-conformal elements in order to obtain a conformal mesh) can be performed. In the 3D case, the operation is much more complicated than in the 2D case, since now an element can be non-conformal because of non-conformal faces, non-conformal edges or a combination of the two. This results in a very large number of possible non-conformal configurations, each one of which has to be treated separately. An added source of complexity comes from the requirement to keep faces shared between elements conformal, while one or both of the elements are refined. Finally, we would like the procedure for making the mesh conformal to work in one pass.

In order to deal with the complexity of the operation and ensure that the resulting mesh is conformal, we chose a different approach. Elements in the conformable mesh are examined and are chosen for refinement if they share at least one vertex with a higher level element. The type of non-standard refinement used for each element is determined by the number of vertices shared with higher level elements as well as their relative positions (modulo reflections and rotations).

We can count the number of distinct neighboring configurations as follows: Let us assume that the vertices of the cube that are shared with higher-order elements are colored black, while vertices not shared with higher-order elements are colored white. The number of distinct neighboring configurations for the cube is the same as the number of distinct colorings of the cube. We can count the number of distinct colorings, modulo reflections and rotations, by using Polya's theory. In this way we find that the number of distinct neighboring arrangements is 22. Another way of counting this is by realizing that the number of neighboring configurations is the same as the number of equivalence classes for 3-variable boolean functions (Harvard classes [44]), which is 22. Associated with each neighboring configuration is a specific refinement, and all refinements can be seen in Appendix C. The specific refinements were chosen because they result in fewer distinct element shapes and partition each cube into fewer elements (compared to several other configurations we tried).

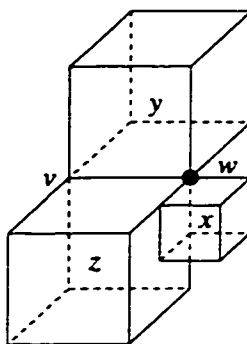


Figure 7.13: Example that helps demonstrate why elements that share only one vertex with higher-level cubes need to be refined. The element labeled y shares only vertex w with the higher level cube x . However, a third element z shares both an edge with y and part of its face with x . If we are not allowed to refine y , the conformal refinement of z becomes problematic.

Deciding which one of the 22 refinements is appropriate for each one of the cubes is relatively easy: we just have to identify neighbors with higher level value and decide in which class the cube belongs. Each of the 22 different refinements, however, has to be implemented separately, which makes this part of the implementation complex.

Figure 7.13 shows an example that can help motivate why we chose to refine elements that share only a vertex with higher level elements. The portion of the crystalline mesh shown is conformable. The elements x and y share only a vertex, and y is conformal. However, there is a third element, z , that shares an edge with y and part of its face with x . Clearly, z is not conformal and needs to be refined. If we choose not to refine y , we have to ensure that the edge (v, w) , which is shared between y and z , is not partitioned while refining z . Otherwise, y would not be conformal anymore and should be refined during a second pass. In principle, we could find an appropriate refinement for z that does not partition (v, w) . However, classifying the cubes according to how many faces are non-conformal *and* how many edges should not be partitioned (and their relative positions), results in many equivalence classes (more than 80). Each one of these cases has to be treated separately. Having so many equivalence classes complicates both the design and the implementation of the algorithm. Moreover, since cubes of different classes can be neighbors in the mesh, it makes it almost impossible to ensure that the mesh will be conformal after one pass of non-standard refinement. The situation becomes much simpler if we are also allowed to refine y .

As the above example shows, the method we use requires us to refine some of the elements that are conformal in the original conformable mesh, which means that refining them is not necessary. This implies that the proposed method may result in meshes that consist of more elements than absolutely necessary. In practice, however, the number of conformal elements refined is small, while the added benefits from the simplicity of both the algorithm and the implementation are quite large. For example, if the region R to be refined has the shape of a parallelepiped, only eight conformal elements will be refined (one corresponding to each of the eight vertices of R).

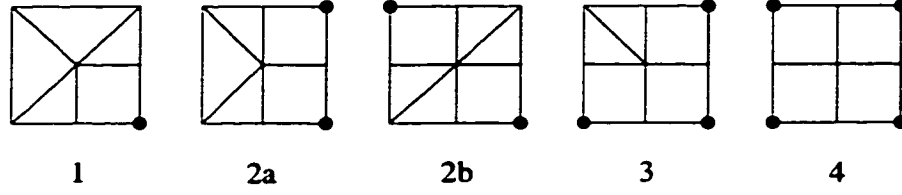


Figure 7.14: Patterns appearing on the faces of non-conformal cubes to be refined (standard patterns). Vertices shared with higher level cubes are shown as solid circles. Faces that do not share any vertices with higher level cubes are not partitioned.

Now we also have to describe how to refine each one of the resulting 22 configurations, so that we can guarantee that the resulting mesh is conformal. To further simplify the problem, we enforce the condition that the cubes we refine, using non-standard refinement, are partitioned so that their faces exhibit some specific patterns, depending on the number of vertices on the face that are shared with higher-level cubes and their relative positions. Any set of such patterns would guarantee conformity between elements that share faces or edges as long as it fulfills the following two requirements. First, it should be consistent, in the sense that the same pattern is assigned to all the faces that have the same number of vertices shared with higher order cubes in the same relative positions. Secondly, it should ensure that all edges that have some of their vertices shared with higher-level elements are partitioned in two equal parts, and edges that have none of their endpoints shared with higher-level elements are not partitioned.

Out of this large number of valid patterns, we would like to choose the ones that result in the smallest number of different shapes being generated during the non-standard refinement phase. We also would like to minimize the number of elements generated by each refinement and ensure that the elements resulting from them are well-behaved. For example, we would like to avoid generating elements with very sharp angles, or complicated shapes.

One set of patterns that seems to work very well according to all criteria mentioned above, is the set shown in Figure 7.14. We will call this set the *standard patterns*. Notice that an edge is partitioned into two edges with equal length if and only if at least one of its endpoints is shared with a higher-level cube. Edges that do not share any endpoints with a higher level cube are not partitioned. By looking at the pattern, we see that if a vertex v is shared with a higher-level element (these are the vertices shown as solid circles in the figure), the faces it belongs to are partitioned so that a square is formed, having v as one of its vertices.

We claim that if all elements that share at least one vertex with higher-level cubes are refined so that their faces are partitioned according to the standard patterns, then the resulting mesh is conformal, provided that the refinement does not generate any non-conformal elements in the interior of the cube. This claim can be formalized as follows.

Lemma 7.5.2 *Given a conformable 3D crystalline mesh, if all cubes in the mesh that share vertices with higher-level cubes are refined so that their faces bear a standard pattern, the resulting mesh*

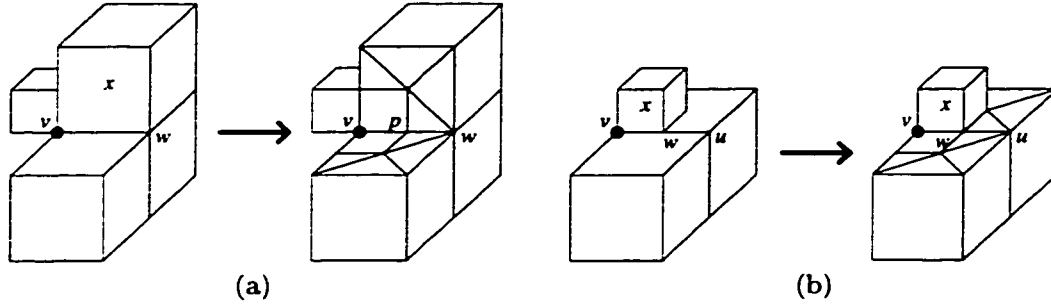


Figure 7.15: Examples, illustrating why edges will be conformal after the non-standard refinement step. (a) x shares (v, w) with elements having the same level value. Since v also belongs to an element with higher level value, all elements sharing (v, w) are refined. The resulting edges are conformal. (b) x shares (v, w) with elements having lower level values. These elements will be refined, resulting in two conformal edges.

will be conformal, provided that the partitioning does not introduce non-conformal elements in the interior of the cube.

Proof: In the 3D case, non-conformity can arise from either non-conformal edges or non-conformal faces.

We will first consider the edges of the elements, and will prove that after the refinement, all edges shared between elements will be conformal. Let x be an element in the conformable mesh and (v, w) one of its edges. The edge can be shared by up to four cubes in the mesh. We can distinguish two cases.

1. All elements sharing (v, w) have the same level value (Figure 7.15(a)).

If neither v nor w belongs to an element with higher level value, (v, w) will not be partitioned even if some of the elements sharing it are refined. If either v or w also belong to an element with higher level value, x will be refined so that a new vertex will be created in the middle of (v, w) . The same is true for all other elements sharing (v, w) . Therefore, after the refinement, the two edges resulting from splitting (v, w) will be conformal.

In the example shown, x shares the edge (v, w) with two other elements that have the same level value as x . However, vertex v also belongs to a higher level element. This means that all elements sharing (v, w) will be refined. By looking at the standard refinement patterns in Figure 7.14, we see that all elements will be refined so that a vertex p will be inserted at the midpoint of (v, w) . The resulting edges, (v, p) and (p, w) , will be conformal.

2. The elements sharing (v, w) have different level values (Figure 7.15(b)).

Since we assume that the elements share the entire edge, this can only mean that the elements that have a different level value than x will be larger, or equivalently, have a lower level value. The mesh is conformable, so the only valid level value for these elements is

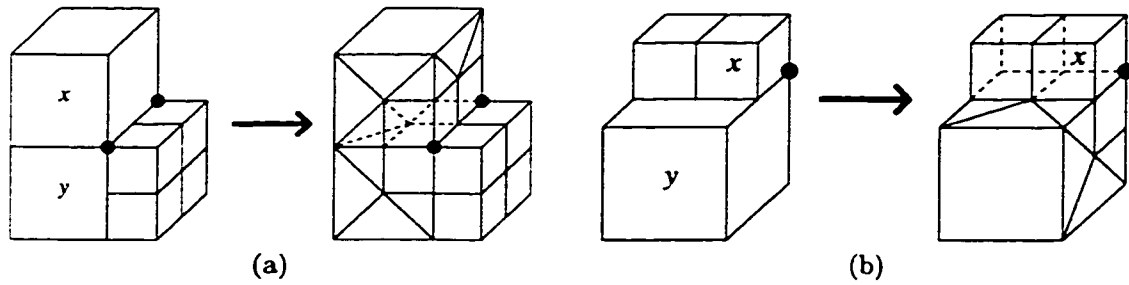


Figure 7.16: Examples, illustrating why faces will be conformal after the non-standard refinement step. (a) x and y share a face, that has two vertices belonging to higher level elements. Both x and y are refined, and the shared face is refined according to the same pattern. The dashed lines show how the shared face will be refined. (b) x shares f with the lower-level element y . y is refined so that one of the faces of the resulting elements exactly matches f .

$level(x) - 1$. This implies that the edges of the larger cubes will have a length which is twice the length of the edges of x . Moreover, by construction, either v or w is a vertex of these larger elements. Without loss of generality, we can assume that it is v . Let (v, u) be the edge of the larger cubes that contains w . Notice that w is the midpoint of (u, v) .

When the lower-level elements are examined by the algorithm, the fact that they share a vertex with a higher level element will be discovered, and they will be refined. Regardless of which other vertices of the elements are shared with higher-level elements, (v, u) will be divided into two edges, by inserting a vertex at its midpoint, which coincides with w . So, after all elements have been refined, both edges resulting from (v, u) will be conformal.

In the example shown, x is sharing (v, w) with two elements with level value $level(x) - 1$. Both of the larger cubes have v as one of their vertices. During the last step of the algorithm both larger elements will be refined. The refinement will divide (v, u) into two edges of equal size, by inserting a vertex at its midpoint. This vertex coincides with w . Both edges resulting from the refinement are conformal.

This proves that after the nonstandard refinement step, all edges in the mesh will be conformal.

Let us now consider the faces of the elements in the mesh. We will prove that after the non-standard refinement, all faces in the mesh will also be conformal. Since non-conformity can only arise from non-conformal edges or non-conformal faces, this will prove that the resulting mesh will be conformal.

Again, let x be an element in the conformable mesh, and f be one of its faces. We have two separate cases, depending on the level of the element x shares f with.

1. Both elements have the same level value (Figure 7.16(a)).

In this case all four vertices of the face are shared by both elements. If any of the four vertices is shared with a higher level element, both elements sharing f will be refined and f

will be partitioned during the refinement. However, both elements will partition f according to the same pattern, and therefore the resulting faces will be conformal. If none of the four vertices of f belongs to a higher level element, the element might still be refined (caused by other vertices that are not in f), but in any case f will not be partitioned and will be conformal in the resulting mesh.

In the example shown, elements x and y share a face. Two of the vertices of the shared face are also shared by higher level elements. Therefore, both elements are refined. In this case, the shared face is refined according to standard pattern 2a in Figure 7.14, and all five resulting faces are conformal.

2. The element with which x shares f has a different level value (Figure 7.16(b)).

Let this element be y . Using the same reasoning as in the case of the edges, we conclude that y should have a lower level value, and since the mesh is conformable, its level value is $level(x) - 1$. In this case, the two elements also share exactly one of their vertices. Therefore, y is chosen to be refined, since it shares a vertex with x which has a higher level value. The face of y that contains f will be refined according to the appropriate standard pattern. By looking at the standard patterns in Figure 7.14, we see that a square will be formed, that matches f . So, after the refinement f will be conformal.

In the example shown, x is sharing f with a lower-level element y . During the non-standard refinement phase, y is refined. In this specific instance, the face of y that contains f will be refined according to standard pattern 2a, since it contains two vertices that also belong to higher level elements, and both of these vertices are the endpoints of the same edge. One of the faces of the resulting elements matches f exactly.

Notice that for both the edge conformity and the vertex conformity proof we used the fact that the original mesh is conformable, since we assumed that elements sharing a vertex will not have a level difference of more than one. \square

Lemma 7.5.2 above proves that *any* refinement that cuts the faces of the cubes in a way consistent with Figure 7.14 will result in a conformal mesh. This simplifies the problem, but doesn't solve it completely. For each of the 22 possible neighboring arrangements, we have to describe exactly how the refinement should be performed. We will devote the next section to a discussion of this problem.

7.6 Non-standard Refinement for 3D Meshes

For each one of the 22 possible neighboring configurations, there are several ways to refine the element so that the standard patterns appear on the element's faces, and the elements generated from the refinement are conformal. All of these refinements would result in conformal meshes and are acceptable.

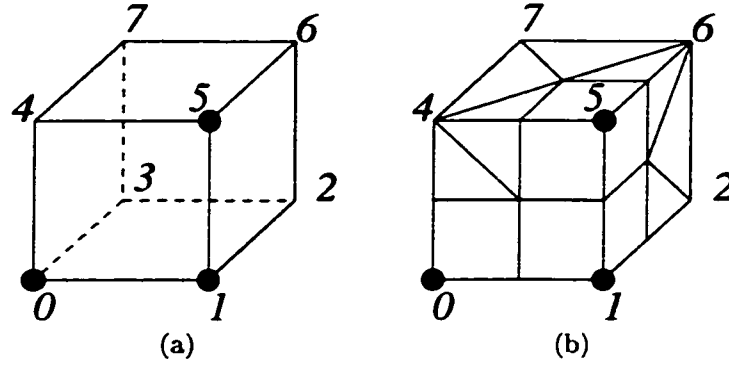


Figure 7.17: (a) A non-conformal cube that shares three vertices with higher-level cubes. These vertices appear as solid circles. Also shown is the standard numbering of the vertices. (b) The faces of the cube have been marked by using the appropriate standard pattern.

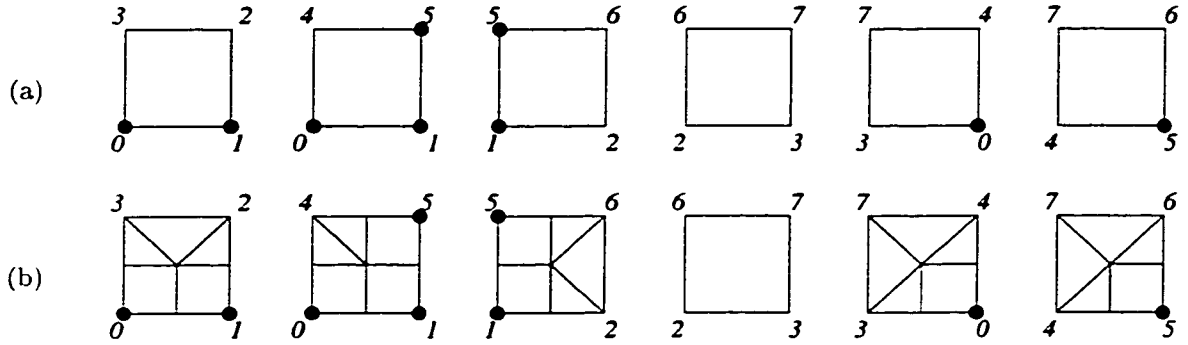


Figure 7.18: (a) All the faces of the non-conformal cube shown in Figure 7.17. (b) The faces of the cube marked with the appropriate standard patterns.

However, while choosing specific refinements for each of the 22 neighboring configurations we have two more goals in mind. First, we would like to minimize the number of elements generated by each non-standard refinement, since more elements imply that we will need more time to generate the system of linear equations. Moreover, we would like to minimize the number of distinct element shapes used in the crystalline mesh, for reasons of simplicity and efficiency.

For example, we could perform the non-standard refinement as follows. First we refine the faces of the cube according to the appropriate standard patterns. Then we create one element for each of the faces appearing on the cube by connecting each of its vertices to the center of the cube. This process is easy to describe and to automate, and will result in conformal elements. However, we found that it generates many more elements than necessary. Moreover, some of the transitional element shapes generated in this way have small angles, and therefore may result in ill-conditioned systems of linear equations. By carefully choosing the way we perform the non-standard refinement, we can generate meshes that are better conditioned and have fewer elements.

To illustrate the approach, we discuss in some detail one specific instance. Specifically, we

consider the case of a cube that shares three vertices, all placed on the same face, with higher-level cubes. This cube is shown in Figure 7.17(a), where vertices shared with higher-level cubes appear as solid circles. In the same figure, we show the standard numbering of the vertices of the cube. In Figure 7.17(b) the same cube is shown, where its three visible faces have been marked with the appropriate standard patterns as specified in Figure 7.14. In order to better explain how all the faces of the cube will be partitioned, we show all the faces of the cube in Figure 7.18(a), emphasizing the vertices that are shared with higher-level cubes. Figure 7.18(b) shows the appropriate standard pattern for each of the faces. Notice that if we were to use the procedure described above to refine this cube, we would generate 26 transitional elements. We will now show a much more efficient refinement.

The entire process is shown in Figure 7.19. We start by removing a cube, that is formed around vertex 5 of the original cube, which is shared with a higher-order element (2). Another cube, formed next to vertex 1, is removed next (3). We continue by removing two pyramids, whose square faces were formed when the first cube was removed (4 and 5). We continue removing elements from the cube in the same fashion, always keeping in mind that the resulting elements should divide the faces of the original cube into the appropriate standard patterns, that shared faces and edges should be conformal, and that only cubes and the transitional elements shown in Figure 4.7 should be used (6–15). We conclude that this element can be refined by using only 15 elements (three of which are cubes).

Figure 7.20 shows the same refinement, in a more compact way. Figure 7.20.1 shows the cube, the vertices shared with higher level elements and the appropriate standard patterns for each of the three visible faces. Figure 7.20.2 shows the refinement of part of the original cube. Each element has been marked with a number that denotes the last step in Figure 7.19 where the element is present. In the background we show the part of the cube that remains after removing all the elements shown. It is identical to the one shown in Figure 5.1.11. Finally, Figure 7.20.3 shows how the remaining part of the cube is refined, resulting in two pyramids and three tetrahedra.

In Appendix C we show all 22 possible neighboring configurations as well as an appropriate refinement for each one of them, which were generated in the same way as the one described here.

7.7 Coarsening

Coarsening is the reciprocal process of refinement. It is used during the adaptation phase in regions where the approximation error is much lower than the error bound that the computation has to achieve. In these cases, we can have a more efficient computation by decreasing the element density in these regions.

For crystalline meshes specifically, the coarsening problem can be stated as follows: we are given a legal crystalline mesh and a (not necessarily connected) region R to be coarsened. We should produce another legal crystalline mesh with lower element density in R .

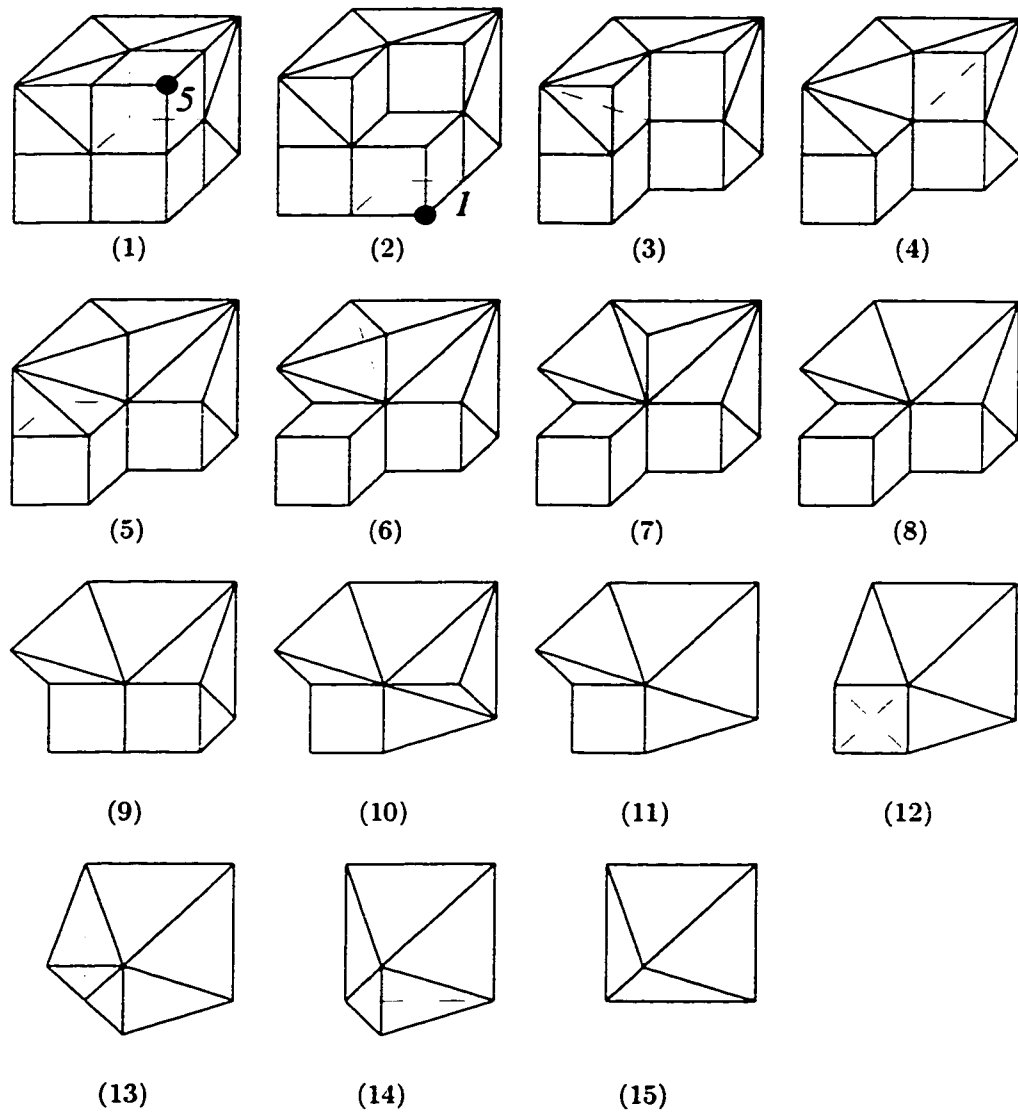


Figure 7.19: Non-standard refinement used for an element sharing three vertices, all on the same face, with higher-level cubes. Each figure can be obtained from the previous one by removing one element. Dashed lines show the “hidden” sides of the element that will be removed next. The refinement can be done using 15 elements.

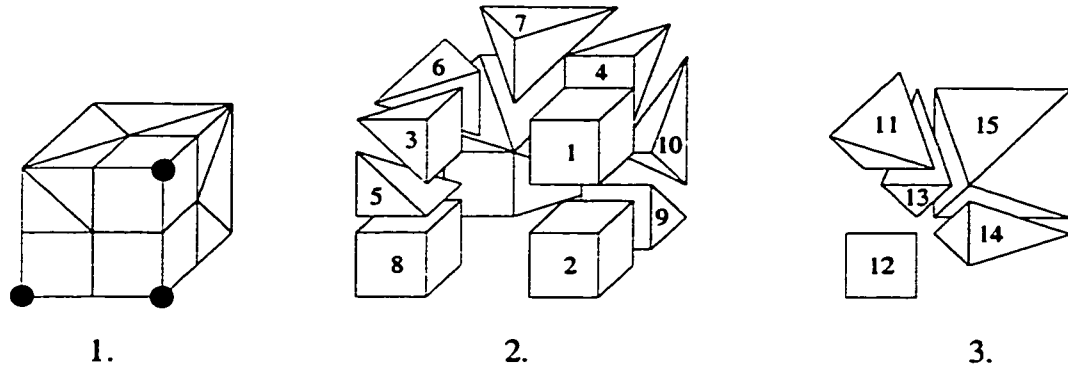


Figure 7.20: Another way of showing the same refinement. (1.) The faces of the cube are shown marked with the standard patterns. (2.) The refinement of part of the cube is shown. (3.) The refinement of the rest of the cube is shown. Elements are marked with a number corresponding to the step in Figure 7.19 where they appear for the last time.

The fundamental asymmetry between the process of refinement and coarsening originates from the fact that coarsening *decreases* the accuracy of the approximation, while refinement increases it. So, while the refinement algorithm is allowed to change the element density of the mesh outside R , the coarsening algorithm should be very conservative. The element density should decrease in R , if possible, but it should not change in regions outside R .

Figure 7.21 shows the flow diagram of the algorithm used in the coarsening computation. It is very similar to the diagram shown in Figure 7.1 for refinement.

Figure 7.22 illustrates the coarsening process on a 2D crystalline mesh. Again we start by eliminating all transitional elements from region R (Figure 7.22(b)). Then, elements that are completely within R are examined. Groups of 2^d elements in R are identified and combined to form larger elements. The resulting mesh is shown in (Figure 7.22(c)).

In order to obtain a conformable mesh we have to eliminate jumps of 2 levels. In principle, we can achieve this by propagating the coarsening outside R . This, however, would result in lower element density in regions outside R , which implies that we would obtain a less accurate solution in these regions. Therefore, this method cannot be used.

Instead, we eliminate 2-level jumps by *refining* elements that have neighbors that are too small. The process is identical to the refinement propagation described previously. In this case, however, the refinement paths originate in elements that are located outside R and continue completely within R (since elements in R were the only ones modified up to this point). So, the elements that are refined using standard refinement during this step are elements in R (Figure 7.22(d)).

During the final step, we use non-standard refinement to eliminate non-conformal points and obtain a legal mesh. Figure 7.22(d)) shows the legal mesh resulting from the coarsening computation. The elements that are different from the input mesh are shown shaded. Notice that not all elements in R were modified. Some of them could not be coarsened because they have neighbors that have

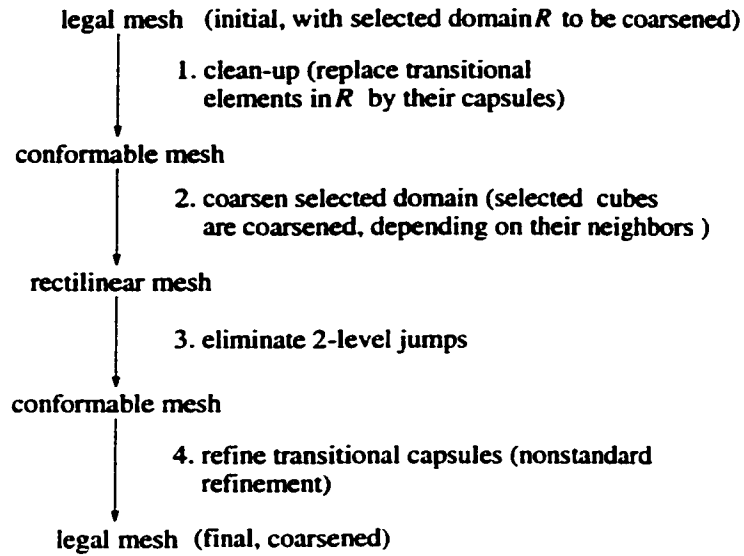


Figure 7.21: Flow diagram of the coarsening process.

too high level value and are located outside R . For example, the 8 elements at the upper right-hand corner of R remain unchanged. Some elements located outside R have also been changed. But closer inspection reveals that the only modification performed there is the replacement of one type of non-standard refinement by another type in order to keep the mesh conformal. Moreover, all elements modified for this reason are elements that have neighbors located in R . This modification should not affect the accuracy of the solution in these regions.

The flowchart shown in Figure 7.21 describes a rather inefficient process. However, we can use it as a starting point to obtain an efficient algorithm that constructs exactly the same meshes given the same input. The ideas used are the same as in the case of refinement: We perform the clean-up step only within R instead of in the entire mesh and we eliminate jumps of 2 levels by simulating a breadth-first search on the refinement paths.

By doing these modifications we obtain an algorithm with running time $O(m(R))$, which matches the inherent complexity of the coarsening step.

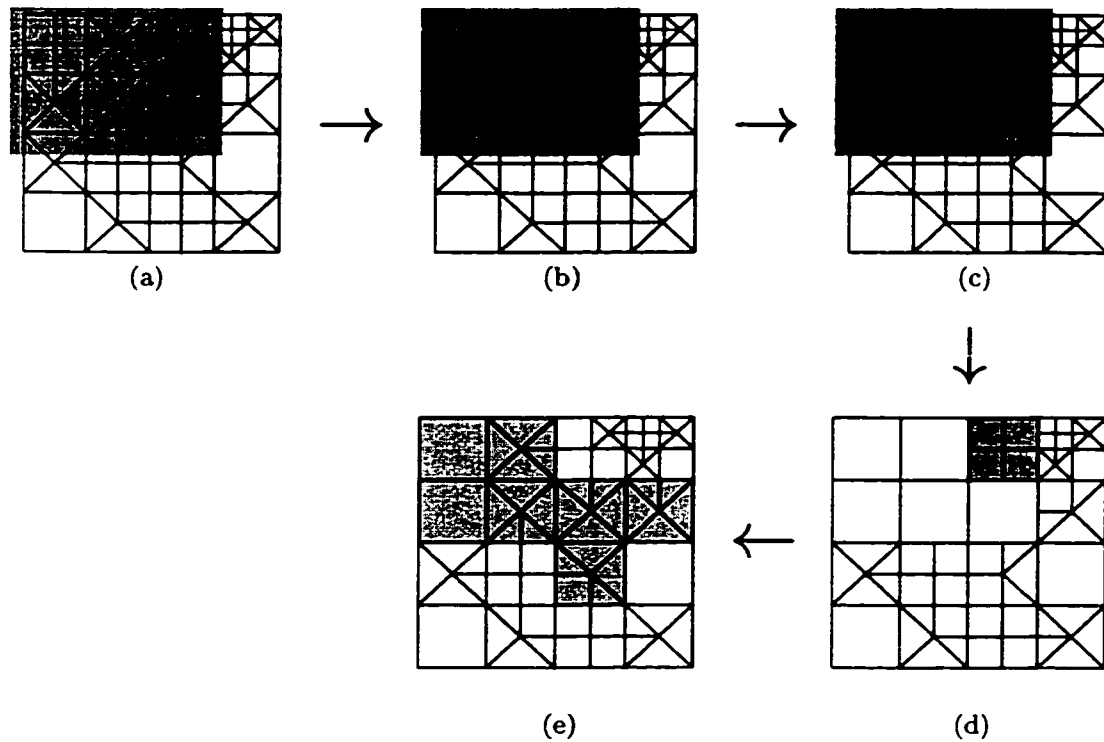


Figure 7.22: (a) A region R has been selected for coarsening (shown shaded). (b) The non-standard elements in R are removed. (c) Groups of 4 squares that fall completely within R are combined. (d) Level-2 jumps are eliminated (dark element). In this case the refinement paths originate outside R . (e) Non-conformal elements are refined. The striped region contains all the elements that were modified by the coarsening process. Notice only R has lower element density. Elements outside R were modified only in cases where one type of non-standard refinement was replaced by another.

Chapter 8

Experimental Results

8.1 Implementational Issues

In order to test how crystalline meshes perform in practice, we have developed a prototype program (in collaboration with J. Castaños and building on previous work of J. Castaños and J. Savage [14, 16, 13], whose focus was the parallelization of refinement) that uses the finite element method to solve Poisson's equation with Dirichlet boundary conditions in both 2D and 3D. In order to be able to make comparisons, we implemented separate procedures that can also handle structured meshes as well as unstructured non-crystalline meshes. We have chosen to use for experimentation a well known and trusted benchmark (see below) and we are well aware that extensive future experimentation is in order to fully assess the competitiveness of crystalline meshes.

One of the first issues we faced when we started the project was the choice of programming language. We had to decide between using FORTRAN, which is the language traditionally used for numerical computations, and using a more flexible, object-oriented language like C++. The vast majority of researchers working on the Finite Element Method use FORTRAN, mainly for three reasons:

- There is a large amount of numerical software publicly available that is written in FORTRAN. This software is well-tested, efficient and familiar. Researchers believe that they can take advantage of this software best by using FORTRAN for their own programs.
- Some researchers believe that object-oriented languages, and in particular C++, are inefficient, especially when it comes to numerical software.
- Researchers in this field are more familiar with FORTRAN and are more comfortable using it.

On the other hand, object-oriented languages provide many features that make software development both easier and less error-prone. Moreover, we could use the flexibility provided by C++ to

handle different types of equations, meshes and elements in a uniform manner. For these reasons we chose to write the prototype almost entirely in C++. Blas and Lapack subroutines were used for solving the systems of linear equations. We found that C++ gives us the flexibility needed to handle different types of meshes and different types of elements, and allows for easy and efficient memory management. Every important data structure in the program is handled through smart pointers and is automatically deleted when it is no longer needed, freeing the allocated memory [35]. This allows us to handle very large meshes (up to 100,000 points) without encountering memory problems.

The first interesting observation is that, contrary to the predictions, the final program is very efficient, achieving running times virtually identical to the running times of programs achieved by software written entirely in FORTRAN. The explanation for this behavior is that time-consuming numerical computations are done using the same efficient subroutines traditionally used in FE software. The time needed for the rest of the computations (that is, for operations that manipulate the mesh) is such a small fraction of the overall running time that even doubling it does not have a significant effect on the performance of the system.

Another issue we faced was the computation of element matrices. Many times element matrices are computed by using numerical integration. This approach is very general; however, some of the accuracy is necessarily lost. We chose to use a database of element matrices that were computed off-line using exact integration. This approach does not only allow for faster system assembly, but also minimizes numerical errors. Combined with the fact that the scaling operations used for elements in crystalline meshes are simple shifts, it guarantees that the numerical errors introduced during the phase of system assembly are negligible.

Finally, following the established practice, we use sparse matrix representation for the system of linear equations and we use iterative methods for solving them.

8.2 Benchmarks and Error Metrics

The system is capable of performing adaptive computation as follows. We start with a very coarse mesh that correctly approximates the problem's domain. Given a target error bound, the system of equations is assembled and solved, the regions of interest are found and, if the error is too large, refined. The process is repeated until the desired error bound is achieved. During the whole process the mesh is kept conformal. The entire refinement history of each of the elements is kept in the form of a tree. Elements in the current mesh are the leaves of the tree. They are also threaded to form the element list. When an element x is refined, the elements resulting from this refinement become the children of the data structure representing x in the refinement tree. Also, x is removed from the element list and its children are added in its place. However, x remains in the refinement history tree and its children are attached to it.

We employed the prototype to evaluate the performance of crystalline meshes when used in adaptive computation for problems with regions of interest and compare it to the performance of

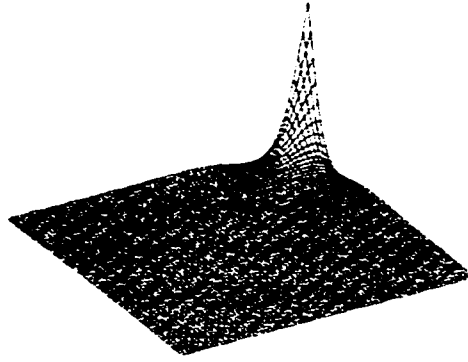


Figure 8.1: The solution to the problem used as a benchmark. The solution changes rapidly in the region next to one of the corners.

both structured and unstructured crystalline meshes. Specifically, we chose to use the standard benchmark problem that appears as Problem (3.61) in [72].

Formally, we want to solve Poisson's equation with Dirichlet boundary conditions, defined as follows:

$$\begin{aligned} -\Delta u &= 0 & \text{in } \Omega \\ u &= g(x, y) & \text{on } \Gamma = \partial\Omega \end{aligned}$$

with

$$g(x, y) = \cos(2\pi(x - y)) \frac{\sinh(2\pi(x + y + 2))}{\sinh(8\pi)}$$

and

$$\Omega = [-1, 1]^2$$

For this specific problem we have a closed-form solution,

$$u = g(x, y)$$

which allows us to accurately evaluate the performance of the method. The domain of the problem, as well as the solution, are shown in Figure 8.1. Notice that the solution is very smooth over almost the entire domain, but changes very rapidly next to the $(1, 1)$ corner. This makes the problem suitable for evaluating the performance of adaptive computation.

For this evaluation we used two different error norms that are commonly used for Finite Element computations. Let $e = [x_1, x_2, \dots, x_n]$ be our error vector (the vector of the residuals) of length n and assume that it has been produced using a mesh with m elements. Then, the L_∞ norm of the vector is:

$$\|e\|_\infty = \max_{1 \leq i \leq n} (|x_i|)$$

Using the L_∞ norm for an adaptive computation implies that we want the approximation computed to differ by at most the target error on all the vertices of the mesh.

The L_2 metric is considered by some researchers to be a better metric for finite element computation because it takes into account the area of the elements. In 2D, it is formally defined as:

$$\|e\|_2 = \left(\sum_{i=1}^m r_i^2 \right)^{1/2}$$

where j is a nodal point in element i (which will be denoted as x_i) and

$$r_i = \left(\sum_{j \in x_i} e_j^2 \right)^{1/2} \cdot \text{area}(x_i)$$

Of course, in the 3D case we are using the volume of the elements instead of the area. The idea is that we are willing to tolerate a higher error in elements that are very small, but we want the errors associated with elements that cover a large part of the domain to be smaller.

Having defined the benchmark problem and the metrics we used, we are ready to present the results. Since the benchmark problem exhibits regions of interest, we expect unstructured meshes to do better than structured meshes. We also expect adaptive refinement techniques to prove extremely useful.

8.3 Results

We used the benchmark to compare the performance of four different types of meshes: (i) structured meshes consisting of triangles or (ii) squares, (iii) unstructured meshes, consisting of triangles that have been adapted using the very popular Rivara refinement (presented in [70, 71]) and (iv) crystalline meshes. The experiments were performed on a Sun Ultra 10 computer with 256 Mb of memory. In all cases we have a target error and we compare the number of elements and vertices in the mesh that achieves it and the number of Conjugate Gradient iterations needed to solve the final system of linear equations. Since many times the target errors were achieved only after several iterations of the refinement process (where the system was solved, the selected elements refined and the new mesh did not achieve the target error, so that another round of refinement had to be

Target error L_∞ - norm	Structured Mesh - Triangles				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	512	289	53	4	< 1
0.005	1089	2048	107	2	1
0.001	8192	4225	213	2	9
0.0005	8192	4225	213	1	8
0.0001	32768	16641	422	2	71
0.00005	131072	66049	832	2	535

Target error L_2 - norm	Structured Mesh - Triangles				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	8	9	3	1	< 1
0.001	128	81	26	3	< 1
0.0001	512	289	53	2	< 1
0.00001	8192	4225	213	3	7
0.000001	32768	16641	422	2	71
0.0000001	131072	66049	832	2	535

Table 8.1: Results obtained when using structured meshes consisting of triangles. For each target error, we give the number of elements and vertices in the final mesh, the number of Conjugate Gradient iterations needed to solve the final system, the number of adaptive iterations and the time the entire adaptation process took (in seconds).

Target error L_∞ - norm	Structured Mesh - Quadrilaterals				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	256	289	38	4	< 1
0.005	1024	1089	76	2	< 1
0.001	4096	4225	152	2	3
0.0005	4096	4225	152	1	4
0.0001	16384	16641	300	2	42
0.00005	65536	66049	591	2	312

Target error L_2 - norm	Structured Mesh - Quadrilaterals				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	81	64	19	3	< 1
0.001	256	289	38	2	< 1
0.0001	1024	1089	76	2	< 1
0.00001	4096	4225	152	2	3
0.000001	16384	16641	300	2	41
0.0000001	65536	66049	591	2	308

Table 8.2: Results obtained when using structured meshes consisting of quadrilaterals to solve the problem.

Target error L_{∞} - norm	Unstructured Mesh - Rivara refinement				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	120	72	25	7	< 1
0.005	190	108	33	3	< 1
0.001	2172	1117	114	11	4
0.0005	5242	2663	176	4	8
0.0001	29536	14868	415	7	131
0.00005	67572	33919	627	6	391
Target error L_2 - norm	Unstructured Mesh - Rivara refinement				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	8	9	3	1	< 1
0.001	82	52	18	5	< 1
0.0001	350	193	45	4	< 1
0.00001	1132	593	84	3	< 1
0.000001	5194	2657	177	4	5
0.0000001	34438	17337	464	4	67

Table 8.3: Results obtained when using unstructured meshes consisting of triangles and the popular Rivara refinement method.

Target error L_{∞} - norm	Unstructured Mesh - Crystalline				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	141	122	31	4	< 1
0.005	231	195	40	2	< 1
0.001	868	700	83	3	< 1
0.0005	3113	2445	158	5	5
0.0001	29002	25691	523	4	105
0.00005	40039	34574	596	4	248
Target error L_2 - norm	Unstructured Mesh - Crystalline				
	elements	vertices	CG iter.	adaptive iter.	time (sec)
0.01	44	45	16	3	< 1
0.001	122	106	29	2	< 1
0.0001	321	272	48	2	< 1
0.00001	1058	933	94	2	< 1
0.000001	9508	8698	292	3	11
0.0000001	35753	33202	592	2	80

Table 8.4: Results obtained when using crystalline meshes and the refinement method suggested in this work.

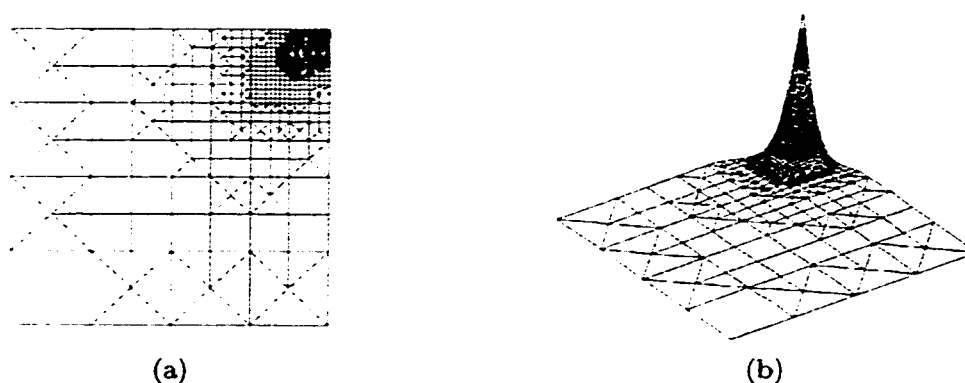


Figure 8.2: Crystalline mesh obtained by starting with a very coarse mesh and using adaptive refinement for the benchmark problem. (a) top view (b) the value of the solution is indicated by the height of each point. Notice that at the region of interest the element density is much higher than in other regions of the domain.

executed) we also present the number of adaptive iterations needed to obtain the final mesh and the time (in seconds) the whole adaptation process took. The results are presented in Tables 8.1–8.4.

As expected, structured meshes are outperformed by both the unstructured and the crystalline meshes. Also notice that, consistently with the results presented in [4], quadrilateral meshes are performing better than triangular meshes in this benchmark.

We now turn our attention to comparing the triangular meshes obtained by using the Rivara refinement with the crystalline meshes. We see that the two methods give comparable results in terms of mesh size and solving time. We also observe that crystalline meshes require fewer adaptive iterations to reach the solution. This is because the crystalline refinement algorithm is more generous in placing points than the Rivara refinement algorithm. Although further study is in order, it appears from these preliminary results that crystalline meshes are comparable to triangular unstructured meshes.

In Figure 8.2 we show one of the crystalline meshes that were generated by the adaptive refinement process. Notice how the element density, as expected, is much higher near the corner of the domain where the value of the solution changes rapidly.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this work we have presented integer-coordinate crystalline meshes and have argued that they give us the benefits of both structured and unstructured meshes, by offering the flexibility of different element densities in different regions, and the ability to generate the system of equations quickly and accurately. Moreover, we have shown how crystalline meshes can be used in both 2D and 3D domains, and how their properties allow us to treat both of these problem in a uniform fashion.

The crystalline meshes for 3D domains use pyramids to realize conformal transitions between regions that contain cubes of different sizes. Pyramids are rarely used in finite element meshes today, because finding basis functions for them is non-trivial. As part of this work, we present an easy-to-automate, iterative process that can be used to construct basis functions for the order- k ($k > 0$) Lagrangian pyramid. We hope that this result will encourage the widespread use of pyramidal finite elements in mixed meshes.

Of course, in order for crystalline meshes to be useful in practice, we need to have a way to automatically generate them from the description of the domain. We presented algorithms for 2D and 3D crystalline mesh generation that can be used to obtain an arbitrarily close approximation of any reasonably smooth domain, while still using only a finite number of distinct shapes. To evaluate our algorithm, we also defined a metric that can be used to quantify the quality of the approximation. We used this metric to prove that the approximation error resulting from the algorithm decreases rapidly with each iteration.

The main focus of our work is the use of crystalline meshes for adaptive computation. We have given a description of how mesh adaptation can be performed in a controlled way (focusing on mesh refinement). We have carefully analyzed the complexity of the proposed refinement and have given an optimal algorithm that can be used both for the 2D and the 3D case.

Moreover, we have described how the controlled mesh adaptation mechanism allows us to use

a one-pass method to obtain a conformal mesh after the refinement propagation step. We have described the different types of non-conformal elements that can be present in the mesh at this point of the computation. For the more difficult 3D case, we have given a general mechanism that when used guarantees conformality in the resulting mesh. We have also presented specific non-standard refinement for the 2D and the 3D case.

Finally, we have presented a working prototype, implemented in C++, that uses the finite element method to solve Poisson's equation with Dirichlet boundary conditions both in two and three dimensions. It has been programmed so that it can handle crystalline and non-crystalline unstructured meshes, and compute error bounds which are used to selectively refine or coarsen regions of the mesh. We used it to obtain comparative results and discussed them.

We hope that our work will inspire researchers in the Finite Element community to use crystalline meshes for problems with regions of interest, and will allow them to obtain more accurate results faster.

9.2 Future Work

In this work, we have given a definition of crystalline meshes and described how they can be used in an adaptive Finite element computation. However, many more issues need to be explored.

Many of the simulation problems that researchers are interested in involve complex domains and require very high resolution. For such problems, we need to use meshes that contain millions of elements and vertices (see, for example [85]). No sequential computer can keep such meshes in main memory, therefore it is not possible to do efficient operations on the mesh. When we consider that we also have to assemble and solve a system of linear equations, the problem becomes even more severe. For meshes like that, parallel computers have to be used. Parallelizing the Finite Element Method is non-trivial, and many researchers are working on the issues introduced by parallelism [29, 30, 87, 49, 45, 16, 14].

The traditional approach to the problem was to generate the mesh sequentially, partition it among the available processors and continue with the computation. Recently, however, the Finite Element community realized that the entire computation can be made much more efficient by generating the mesh in parallel [23, 65]. Of course, parallel mesh generation raises a new set of issues that have to be dealt with. Researchers in the area are working on obtaining efficient parallel mesh generation algorithms for 2D Delaunay and general triangulations [21, 33, 43, 63] and 3D tetrahedral meshes [22, 34, 65, 64, 74, 91]. Some results have also been obtained in generating quadrilateral [54] and hexahedral [55, 83] meshes in parallel.

The basic problem that any parallel mesh generation algorithm has to address is identifying vertices that have been created on the boundary between two or more partitions. These vertices should be consistently stored in more than one processor. This requires, in the general case, one communication round per element partitioning round. An important characteristic of crystalline

mesh generation is that it is deterministic and very local. During the element partitioning phase, we do not need any information about the neighbors of the element we are currently processing. Therefore, the entire phase can be done without any communication between processors. Preliminary results indicate that this would lead to an efficient mesh generation algorithm. We would like to investigate this problem further.

We also would like to look into the problem of parallel mesh adaptation. Crystalline meshes ensure that the refinement paths are acyclic. Therefore, we know that once an element has been refined during the propagation step, the resulting elements will not be refined during the same step. However, refinement paths starting in elements stored in processor p_i , may pass through elements stored in other processors and return to other elements in p_i . For a discussion of the problems that this might create and solutions that work well in practice see [14, 15]. We would like to investigate partitioning schemes that, when used together with crystalline meshes would minimize these problems.

Moreover, no matter what partitioning technique we use, regions of interest will probably not be uniformly distributed among processors. Therefore, any mesh adaptation step might (and most probably will) introduce load-balancing problems. They occur when many elements that reside in some of the processors are refined, while elements that reside on other processors remain mostly unchanged. To solve this problem, a mesh adaptation step often has to be followed by a load balancing step, where work load is more evenly distributed among the processors. Significant work has already been done on the load balancing step for general meshes [13, 17, 14, 90]. However, since crystalline meshes exhibit more structure than general meshes, we believe that we can exploit it in order to obtain a specific algorithm that works better on crystalline meshes than the general purpose algorithms.

Another very interesting area of research where the finite element method is extensively used today, is flow simulation for problems with very high Reynolds numbers. Finding the pressure and velocity of the air around an airplane flying at very high speed is an example of a problem with these characteristics. Meshes used in this type of problem need to have high resolution but only in specific areas and in a particular dimension (that changes from region to region) which is known in advance. In the example of the airplane, high resolution is needed only close to the surface of the airplane and perpendicular to it. For problems like that, meshes that contain elements with very high aspect ratios are commonly used. The areas of mesh generation and mesh adaptation for this type of problem is currently very active [89, 88, 18, 9, 79, 31, 12]. However, all the results that we are aware of, are using triangular or tetrahedral meshes. The problem with this approach is that high aspect ratios in triangles or tetrahedra necessarily imply sharp angles.

We would like to try a different approach, based on crystalline meshes. At first, crystalline meshes do not seem appropriate for these problems, since the use of squares and cubes ensures that the element density will be the same in all directions. However, there is a method that would allow us to use crystalline meshes as a starting point to create non-crystalline meshes that contain elements with high aspect ratio. It is based on slicing the elements in the regions of interest so that

the mesh has high density in the specified dimension. The slicing can be done after the crystalline mesh generation step has been completed. Of course, it has to be done carefully to ensure that no non-conformity is introduced. We want to further explore this idea and see when and under what circumstances it can be applied. We expect that the problem will be relatively easy for 2D meshes and, of course, become significantly harder for the 3D case.

Appendix A

Basis Functions

A.1 A Simple Example

In order to better understand how basis functions are used to construct the approximation of the solution of a partial differential equation, we will now present an example. For simplicity, we chose an one-dimensional problem.

Example A.1.1 *The domain of the problem we are considering is $[0..5]$. A mesh has been constructed on the domain, which partitions it into five elements. Each element is a line segment of length one. Two nodal points are placed on each element, one on each of its endpoints. So, each nodal point (besides the ones placed on 0 and 5) is shared between two elements.*

We want the final approximation to be a piecewise linear function, therefore our basis functions should also be piecewise linear. Let us focus on an arbitrarily chosen element e_i in the mesh, with endpoints p_i and p_{i+1} . It has two nodal points associated with it, so we should define two linear functions, one associated with p_i and the other with p_{i+1} . The requirement that the functions should evaluate to 1 on the point with which they are associated, and to 0 on the other point, completely defines them. The resulting functions for one element are shown in Figure A.1.

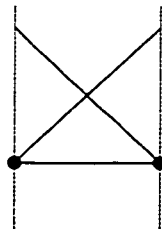


Figure A.1: A finite element for an one-dimensional problem, with two nodal points. The linear basis functions associated with the points are shown.

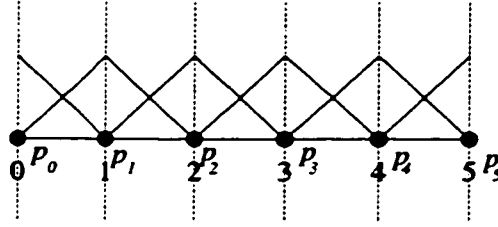


Figure A.2: The basis functions for all the points are obtained by combining the basis functions associated with each one of the elements. Notice that the resulting functions are continuous on the shared points.

$$\begin{aligned}\phi_0(x) &= \begin{cases} 1-x & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases} \\ \phi_i(x) &= \begin{cases} x-(i-1) & i-1 \leq x \leq i \\ (i+1)-x & i \leq x \leq i+1 \\ 0 & \text{otherwise} \end{cases} \quad 1 \leq i \leq 4 \\ \phi_5(x) &= \begin{cases} x-4 & 4 \leq x \leq 5 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

Table A.1: Basis functions associated with nodal points.

When combining the basis functions associated with the points while processing each element separately, we obtain the functions shown in Figure A.2. Notice that they exhibit C_0 continuity. The resulting functions are shown in Table A.1.

In the next step of the finite element method, these basis functions are used to build a system of linear equations. The system has six unknowns, one associated with each nodal point. Assume that by solving the system we obtain $x^T = [1, \frac{1}{3}, \frac{1}{3}, 1, \frac{7}{3}, \frac{13}{3}]$. Then, the approximation to the solution obtained by the method is:

$$f(x) = 1\phi_0(x) + \frac{1}{3}\phi_1(x) + \frac{1}{3}\phi_2(x) + 1\phi_3(x) + \frac{7}{3}\phi_4(x) + \frac{13}{3}\phi_5(x) =$$

$$\begin{cases} 1 - \frac{2}{3}x & 0 \leq x \leq 1 \\ \frac{1}{3} & 1 \leq x \leq 2 \\ \frac{2}{3}x - 1 & 2 \leq x \leq 3 \\ \frac{4}{3}x - 3 & 3 \leq x \leq 4 \\ 2x - \frac{7}{3} & 4 \leq x \leq 5 \\ 0 & \text{otherwise} \end{cases}$$

The resulting function is shown in Figure A.3. Notice that, like the basis functions, it is piecewise linear and exhibits C_0 continuity.

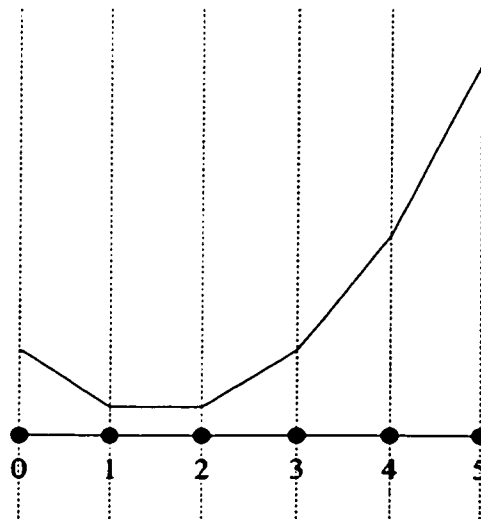


Figure A.3: The solution to the problem is a linear combination of the basis functions.

Appendix B

Basis Functions for Higher-Order Pyramids: Examples

In this appendix, we calculate, as an illustration of the method, the basis functions for order-2 and order-3 pyramids.

B.1 Order-2 Pyramid

The order-2 pyramid is shown in Figure B.1, where we have highlighted the non-equivalent nodal points whose basis functions are to be computed (the functions for all other points can be obtained by standard coordinate transformations).

Let $\phi_i(\xi, \eta, \zeta)$ denote the basis function for point p_i ($i = 0, 4, 5, 9, 13$). Functions ϕ_4 and ϕ_9 are obtained by scaling the corresponding basis functions ϕ'_4 and ϕ'_0 , respectively, of the order-1 pyramid. Since $k = 2$, the coordinate transformations are:

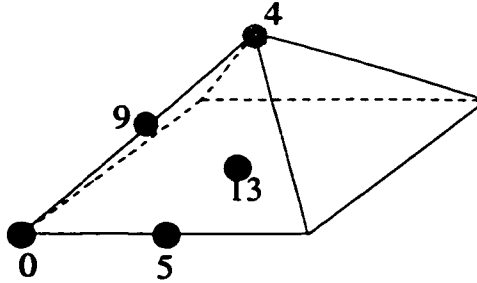


Figure B.1: Node placement on the second order Lagrangian pyramid. The standard numbering of the nodes is also shown. Only the set of non-equivalent nodes is shown. The nodes whose basis functions can be obtained from the basis functions of the order-1 pyramid are shown in gray.

$$\xi' = \frac{4\xi - 1}{2}, \quad \eta' = \frac{4\eta - 1}{2}, \quad \zeta' = \frac{4\zeta - 1}{2}$$

$$\begin{aligned} \phi_4(\xi, \eta, \zeta) &= \frac{1}{1/2} \zeta \phi'_4(\xi', \eta', \zeta') \\ &= 2\zeta(4\zeta - 1) \end{aligned}$$

$$\begin{aligned} \phi_9(\xi, \eta, \zeta) &= \frac{1}{1/4} \zeta \phi'_9(\xi', \eta', \zeta') \\ &= -8\zeta \frac{(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} \end{aligned}$$

Next we consider the basis functions of points $p_0 = p_{0,0}$, $p_5 = p_{1,0}$ and $p_{13} = p_{1,1}$ on the base. Specializing the general formula and noting that:

$$\Xi^L = (\xi + \zeta - \frac{1}{2}), \Xi^R = (\xi - \zeta - \frac{1}{2}), H^L = (\eta + \zeta - \frac{1}{2}), H^R = (\eta - \zeta - \frac{1}{2})$$

we obtain:

$$\begin{aligned} \phi_{0,0}(\xi, \eta, \zeta) &= c_{0,0} \frac{\xi_2^L \eta_2^L}{2\zeta - 1} \prod_{j=1}^1 q_j \\ &= 4 \frac{(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} \\ &\quad \left(-\zeta^2 + \xi\zeta + \eta\zeta - \xi\eta + \frac{1}{2}\xi + \frac{1}{2}\eta - \frac{1}{4} \right) \end{aligned}$$

$$\begin{aligned} \phi_{1,0}(\xi, \eta, \zeta) &= c_{1,0} \frac{\xi_0^R \xi_2^L \eta_2^L}{2\zeta - 1} H^R \prod_{j=1}^0 \xi_j^R \prod_{j=2}^i \eta_j^L \\ &= 8 \frac{(\xi - \zeta)(\xi + \zeta - 1)(\eta - \zeta)}{2\zeta - 1} (\eta - \zeta - \frac{1}{2}) \end{aligned}$$

$$\begin{aligned} \phi_{1,1}(\xi, \eta, \zeta) &= c_{1,1} \xi_0^R \xi_2^L \eta_0^R \eta_2^L \frac{\Xi^R}{\xi_1^R} \frac{H^R}{\eta_1^R} \\ &= 16(\xi - \zeta)(\xi + \zeta - 1)(\eta - \zeta)(\eta + \zeta - 1) \end{aligned}$$

since

$$c_{0,0} = (-1)^2 \frac{2^2}{(1!)^2} = 4, \quad c_{1,0} = (-1)^2 \frac{2^3}{1!1!1!} = 8, \quad c_{1,1} = (-1)^2 \frac{2^4}{1!1!1!1!} = 16$$

B.2 Order-3 Pyramid

We now turn our attention to the order-3 pyramid, illustrated in Figure B.2, with the nonequivalent points appropriately highlighted (points on the base, whose basis functions are not obtained by scaling are shown solid).

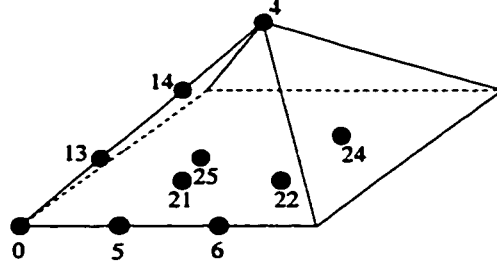


Figure B.2: Node placement on the third order Lagrangian pyramid. The standard numbering of the nodes is also shown. Only the set of non-equivalent nodes is shown. The nodes, whose basis functions can be obtained from the basis functions of the order-2 pyramid, are shown in gray.

Functions $\phi_4, \phi_{14}, \phi_{13}$ and ϕ_{25} are obtained by scaling the corresponding functions $\phi'_4, \phi'_9, \phi'_0$ and ϕ'_5 of the order-2 pyramid. With the coordinate transformation ($k = 3$) we obtain:

$$\xi' = \frac{6\xi - 1}{4}, \quad \eta' = \frac{6\eta - 1}{4}, \quad \zeta' = \frac{6\zeta - 1}{4}$$

$$\begin{aligned} \phi_4(\xi, \eta, \zeta) &= \frac{1}{1/2} \zeta \phi'_4(\xi', \eta', \zeta') \\ &= 2\zeta(4\zeta - 1)(2\zeta - 1) \end{aligned}$$

$$\begin{aligned} \phi_{14}(\xi, \eta, \zeta) &= \frac{1}{1/3} \zeta \phi'_9(\xi', \eta', \zeta') \\ &= -9\zeta(6\zeta - 1) \frac{(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} \end{aligned}$$

$$\begin{aligned} \phi_{13}(\xi, \eta, \zeta) &= \frac{1}{1/6} \zeta \phi'_0(\xi', \eta', \zeta') \\ &= 9\zeta \frac{(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} \\ &\quad (-9\zeta^2 + 9\zeta\xi - 9\zeta\eta + 9\eta\zeta + 3\xi + 3\eta - 2) \end{aligned}$$

$$\begin{aligned} \phi_{25}(\xi, \eta, \zeta) &= \frac{1}{1/6} \zeta \phi'_5(\xi', \eta', \zeta') \\ &= 54\zeta \frac{(\xi - \zeta)(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} (3\eta - 3\zeta - 1) \end{aligned}$$

Finally, we consider points $p_0 = p_{0,0}, p_5 = p_{1,0}, p_6 = p_{2,0}, p_{21} = p_{2,1}, p_{24} = p_{2,2}$.

Noting that:

$$\begin{aligned}\Xi^L &= \left(\xi + \zeta - \frac{1}{3}\right) \left(\xi + \zeta - \frac{2}{3}\right) \quad , \quad \Xi^R = \left(\xi - \zeta - \frac{1}{3}\right) \left(\xi - \zeta - \frac{2}{3}\right) \\ H^L &= \left(\eta + \zeta - \frac{1}{3}\right) \left(\eta + \zeta - \frac{2}{3}\right) \quad , \quad H^R = \left(\eta - \zeta - \frac{1}{3}\right) \left(\eta - \zeta - \frac{2}{3}\right)\end{aligned}$$

we have:

$$\begin{aligned}\phi_{0,0}(\xi, \eta, \zeta) &= c_{0,0} \frac{\xi_3^L \eta_3^L}{2\zeta - 1} \prod_{j=1}^2 q_j \\ &= -\frac{81}{4} \frac{(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} \\ &\quad (-\zeta^2 + \xi\zeta + \eta\zeta - \xi\eta + \frac{1}{3}\xi + \frac{1}{3}\eta - \frac{1}{9}) \\ &\quad (-\zeta^2 + \xi\zeta + \eta\zeta - \xi\eta + \frac{2}{3}\xi + \frac{2}{3}\eta - \frac{4}{9}) \\ \phi_{1,0}(\xi, \eta, \zeta) &= c_{1,0} \frac{\xi_0^R \xi_3^L \eta_3^L}{2\zeta - 1} H^R \prod_{j=1}^0 \xi_j^R \prod_{j=2}^2 \xi_j^L \\ &= \frac{243}{4} \frac{(\xi - \zeta)(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} (\eta - \zeta - \frac{1}{3}) \\ &\quad (\eta - \zeta - \frac{2}{3})(\xi + \zeta - \frac{2}{3}) \\ \phi_{1,1}(\xi, \eta, \zeta) &= c_{1,1} \xi_0^R \xi_3^L \eta_0^R \eta_3^L \frac{\Xi^R}{\xi_1^R} \frac{H^R}{\eta_1^R} \\ &= \frac{729}{4} (\xi - \zeta)(\xi + \zeta - 1)(\eta - \zeta)(\eta + \zeta - 1) \\ &\quad (\xi - \zeta - \frac{2}{3})(\eta - \zeta - \frac{2}{3}) \\ \phi_{2,0}(\xi, \eta, \zeta) &= c_{2,0} \frac{\xi_0^R \xi_k^L \eta_k^L}{2\zeta - 1} H^R \prod_{j=1}^1 \xi_j^R \prod_{j=3}^2 \xi_j^L \\ &= -\frac{243}{4} \frac{(\xi - \zeta)(\xi + \zeta - 1)(\eta + \zeta - 1)}{2\zeta - 1} (\eta - \zeta - \frac{1}{3}) \\ &\quad (\eta - \zeta - \frac{2}{3})(\xi - \zeta - \frac{1}{3}) \\ \phi_{2,2}(\xi, \eta, \zeta) &= c_{2,2} \xi_0^R \xi_3^L \eta_0^R \eta_3^L \frac{\Xi^R}{\xi_2^R} \frac{H^R}{\eta_2^R} \\ &= \frac{729}{4} (\xi - \zeta)(\xi + \zeta - 1)(\eta - \zeta)(\eta + \zeta - 1) \\ &\quad (\xi - \zeta - \frac{1}{3})(\eta - \zeta - \frac{1}{3})\end{aligned}$$

since

$$\begin{aligned}
c_{0,0} &= (-1)^3 \frac{3^4}{(2!)^2} = -\frac{81}{4} & c_{1,0} &= (-1)^2 \frac{3^5}{1!2!2!} = \frac{243}{4} \\
c_{1,1} &= (-1)^2 \frac{3^6}{1!2!1!2!} = \frac{729}{4} & c_{2,0} &= (-1)^3 \frac{3^5}{2!1!2!} = -\frac{243}{4} \\
c_{2,2} &= (-1)^4 \frac{3^6}{2!1!2!1!} = \frac{729}{4}
\end{aligned}$$

B.3 Generating Basis Functions - Program

The following pages contain a maple program that can be used to generate the basis functions for all the points on the order- k pyramid. Of course, the same procedure can be easily implemented using any other package that allows symbolic computation.

```

# *****
# *****
#
# Procedure that generates the nodal points of the order k pyramid.
# n is the number of nodal points.
# It returns p, which is an array containing all the points.
#
# *****
# *****

generatePoints := proc(k, n)
    local i,j, count, p;

    p := array(1..n);

    # Vertices, which are always numbered first.

    p[1] := [0,0,0];
    p[2] := [1,0,0];
    p[3] := [1,1,0];
    p[4] := [0,1,0];
    p[5] := [1/2,1/2,1/2];

    count := 6; # count is used to keep track of how many points we already have

    # Edge 0

    for i from 1 to k-1 do
        p[count] := [i/k, 0, 0];
    end do
end proc

```

```

    count := count + 1;
od;

# Edge 1

for i from 1 to k-1 do
  p[count] := [1, i/k, 0];
  count := count + 1;
od;

# Edge 2

for i from 1 to k-1 do
  p[count] := [(k-i)/k, 1, 0];
  count := count + 1;
od;

# Edge 3

for i from 1 to k-1 do
  p[count] := [0, (k-i)/k, 0];
  count := count + 1;
od;

# Edge 4

for i from 1 to k-1 do
  p[count] := [i/(2*k), i/(2*k), i/(2*k)];
  count := count + 1;
od;

# Edge 5

for i from 1 to k-1 do
  p[count] := [(2*k-i)/(2*k), i/(2*k), i/(2*k)];
  count := count + 1;
od;

# Edge 6

```

```

for i from 1 to k-1 do
  p[count] := [(2*k-i)/(2*k), (2*k-i)/(2*k), i/(2*k)];
  count := count + 1;
od;

```

Edge 7

```

for i from 1 to k-1 do
  p[count] := [i/(2*k), (2*k-i)/(2*k), i/(2*k)];
  count := count + 1;
od;

```

Face 0

```

for j from 1 to k-1 do
  for i from 1 to k-1 do
    p[count] := [i/k, j/k, 0];
    count := count + 1;
  od;
od;

```

Face 1

```

for i from 1 to k-2 do
  for j from 1 to k-i-1 do
    p[count] := [(i+2*j)/(2*k), i/(2*k), i/(2*k)];
    count := count+1;
  od;
od;

```

Face 2

```

for i from 1 to k-2 do
  for j from 1 to k-i-1 do
    p[count] := [(2*k-i)/(2*k), (i+2*j)/(2*k), i/(2*k)];

```

```

        count := count+1;
    od;
od;

# Face 3

for i from 1 to k-2 do
    for j from 1 to k-i-1 do
        p[count] := [(2*k-i-2*j)/(2*k), (2*k-i)/(2*k), i/(2*k)];
        count := count+1;
    od;
od;

# Face 4

for i from 1 to k-2 do
    for j from 1 to k-i-1 do
        p[count] := [(i)/(2*k), (2*k-2*j-i)/(2*k), i/(2*k)];
        count := count+1;
    od;
od;

RETURN(p);
end;

# *****
#
# The following functions return the primitive constructs.
#
# *****

xiL := proc(i,k)
local temp;

```

```

temp := xi + zeta - i/k;
RETURN(temp);
end;

```

```

XIL := proc(k)
local temp;

temp := product(xiL(j,k), j=1..k-1);
RETURN(temp);
end;

```

```

xiR := proc(i,k)
local temp;

temp := xi - zeta - i/k;
RETURN(temp);
end;

```

```

XIR := proc(k)
local temp;

temp := product(xiR(j,k), j=1..k-1);
RETURN(temp);
end;

```

```

etaL := proc(i,k)
local temp;

temp := eta + zeta - i/k;
RETURN(temp);
end;

```

```

ETAL := proc(k)

```

```

local temp;

temp := product(etaL(j,k), j=1..k-1);
RETURN(temp);
end;

etaR := proc(i,k)
local temp;

temp := eta - zeta - i/k;
RETURN(temp);
end;

ETAR := proc(k)
local temp;

temp := product(etaR(j,k), j=1..k-1);
RETURN(temp);
end;

quadric := proc(i,k)
local temp;

temp := expand(2*zeta*(xi+eta-i/k) - (xi+zeta-i/k)*(eta+zeta-i/k));
RETURN(temp);
end;

# *****
#
# This procedure returns the basis function associated with the point p on
# the base of an order-k pyramid.
# *****

basePoint := proc(p,k)

```

```

local i,j;

if ( p[3] <> 0 ) then
  print('Procedure basepoint was passed wrong point');
  RETURN(0);
fi;

i := p[1] * k;          # computing indices from coordinates
j := p[2] * k;

if ( i>k or j>k ) then
  print('Wrong combination of point and k was given to basePoint');
  RETURN(0);
fi;

if ( (i=0 or i=k) and (j=0 or j=k) ) then      # point on a vertex
  RETURN(pointVertex(p,k));
elif ( i=0 or i=k or j=0 or j=k ) then        # point on an edge
  RETURN(pointEdge(p,k));
else                                           # interior point
  RETURN(pointInterior(p,k));
fi;

end;

# *****
#
# Returns the basis function associated with point p on the k-th order
# pyramid. Point p is assumed to be positioned on a vertex.
#
# *****

pointVertex := proc(p,k)
  local temp, temp1, c;

  temp1 :=0;

```



```

# Calculating the basis function for point (0,0,0).
c := (-1)^k * (k^(k-1)/(k-1!))^-2;
temp := c * xiL(k,k) * etaL(k,k) / (2*zeta-1);
temp := temp * product(quadric(j,k), j=1..k-1);

if ( p[1] = 0 and p[2] = 0 and p[3] = 0 ) then
    RETURN(temp);
elif ( p[1] = 1 and p[2] = 0 and p[3] = 0 ) then      # point 1: rotation 1
    temp1 := simplify(subs([xi=eta, eta=1-xi], temp));
    RETURN(temp1);
elif ( p[1] = 1 and p[2] = 1 and p[3] = 0 ) then      # point 2: rotation 2
    temp1 := simplify(subs([xi=1-xi, eta=1-eta], temp));
    RETURN(temp1);
elif ( p[1] = 0 and p[2] = 1 and p[3] = 0 ) then      # point 3: rotation 3
    temp1 := simplify(subs([xi=1-eta, eta=xi], temp));
    RETURN(temp1);
else
    print('pointVertex was passed wrong point');
    RETURN(0);
fi;
end;

# *****
#
# Returns the basis function associated with point p on the order-k pyramid.
# Point p is assumed to be located in the interior of one of the edges of
# the base.
#
# *****

pointEdge := proc(p,k)
    local i, c, temp, temp1;

    if ( p[3]<> 0 ) then      # check that it is located on the base
        print('pointEdge was passed wrong point');
        RETURN(0);
    
```

```

fi;

if ( p[2]= 0 ) then                                # located on edge 0
  i := p[1] * k;                                   # compute index from coordinates
  if ( i=k or i=0 ) then                           # the point is located on a vertex
    print('pointEdge was passed wrong point');
    RETURN(0);
  fi;
  temp := findEdgeFunction(i,k);                   # computes actual function
  RETURN(temp);
elif ( p[1] = 1) then                              # located on edge 1
  i := p[2] * k;                                   # compute index from coordinates
  if ( i=k or i=0 ) then                           # the point is located on a vertex
    print('pointEdge was passed wrong point');
    RETURN(0);
  fi;
  temp := simplify(subs([xi =eta, eta=1-xi], findEdgeFunction(i,k)));
  RETURN(temp);                                    # Use first rotation
elif ( p[2] = 1) then                              # located on edge 2
  i := p[1] * k;                                   # compute index from coordinates
  if ( i=k or i=0 ) then                           # the point is located on a vertex
    print('pointEdge was passed wrong point');
    RETURN(0);
  fi;
  temp := simplify(subs([xi=1-xi, eta=1-eta], findEdgeFunction(k-i,k)));
  RETURN(temp);                                    # Use second rotation
elif ( p[1] = 0 ) then                             # located on edge 3
  i := p[2] * k;                                   # compute index from coordinates
  if ( i=k or i=0 ) then                           # the point is located on a vertex
    print('pointEdge was passed wrong point');
    RETURN(0);
  fi;
  temp := simplify(subs([xi=1-eta, eta=xi], findEdgeFunction(k-i,k)));
  RETURN(temp);                                    # Use third rotation
else
  print('pointEdge was passed wrong point');
  RETURN(0);
fi;

```

```

end;

# *****
#
# Returns the basis function associated with point $p$ in the interior of
# base of the order-k pyramid.
#
# *****

pointInterior := proc(p,k)
  local i,j,c,temp;

  if ( p[3] <> 0 ) then          # check that it is on the base
    print('pointInterior was passed wrong point');
    RETURN(0);
  fi;

  i := p[1] * k;
  j := p[2] * k;

  if ( i=0 or j=0 or i=k or j=k ) then    # check that it is in the interior
    print('pointInterior was passed wrong point');
    RETURN(0);
  fi;

  c := (-1)^(i+j) * k^(2*k) / (i! * j! * (k-i)! * (k-j)!);

  temp := c * xiR(0,k) * xiL(k,k) * etaR(0,k) * etaL(k,k) *
    XIR(k)/xiR(i,k) * ETAR(k)/etaR(j,k);

  RETURN(temp);
end;

# *****

```

```

#
# Returns the basis function associated with point (i/k,0,0) on the
# order-k pyramid.
#
# *****

findEdgeFunction := proc(i,k)
    local c, temp;

    c := (-1)^(i+1) * (k^(2*k-1)) / ( i! * (k-i)! * (k-1)!);

    temp := c * xiR(0,k) * xiL(k,k) * etaL(k,k) / (2*zeta-1) * ETAR(k) *
        product(xiR(j,k),j=1..i-1) * product(xiL(j,k),j=i+1..k-1);
    RETURN(temp);
end;

# *****
#
# Highest level procedure. Takes the order k as input and generates
# all nodal points and basis functions associated with them.
#
# *****

basisFunctions := proc(k)
    local n,p,i,f, point, temp, h;

    n := 3*k^2 + 2;                # number of nodal points
    p := generatePoints(k,n);       # p now contains the coordinates of the points

    f := array(1..n);              # f will contain the functions

    if ( k = 1 ) then              # Base case of recursion -first order pyramid
        f[1] := - (xi+zeta-1)*(eta+zeta-1)/(2*zeta-1);
        f[2] := (xi-zeta)*(eta+zeta-1)/(2*zeta-1);
        f[3] := - (eta-zeta)*(xi-zeta)/(2*zeta-1);
        f[4] := (eta-zeta)*(xi+zeta-1)/(2*zeta-1);
        f[5] := 2*zeta;
    else

```

```

mappings := array(1..n);          # the mappings from order-k to order-(k-1)
computeMappings(k,n, mappings);
f_low_order := basisFunctions(k-1); # Basis functions for lower order pyramid

for i from 1 to n do
  f[i] := 0;
od;

for i from 1 to n do              # for all points
  point := p[i];
  if ( point[3] = 0 ) then        # point on the base
    f[i] := basePoint(point,k);
  elif ( mappings[i] > 0 ) then   # point not on the base
    temp := simplify(subs([xi=(2*k*xi-1)/(2*k-2), eta=(2*k*eta-1)/(2*k-2),
                          zeta=(2*k*zeta-1)/(2*k-2)],f_low_order[mappings[i]]));
    f[i] := 1/point[3] * zeta * temp; # use lower order function

    h := simplify(subs([xi=point[1], eta=point[2], zeta=point[3]], f[i]));
    if ( h <> 1 ) then            # check that basis function evaluates to 1 on point.
      print('Computed wrong basis function for ',i, 'in ', k);
    fi;
  else
    print('basisFunctions: A vertex with no mapping was found');
    fi;
  od;
fi;

RETURN(f);

end;

# *****
#
#   Computes the mappings. A nodal point in the order-k pyramid with
#   local index i corresponds to a nodal point on the order-(k-1) pyramid
#   with local index mappings[i].

```

```

#
# *****

computeMappings := proc(k, n, mappings)
  local i, j;

  for i from 1 to n do          # initialization
    mappings[i] := -1;
  od;

# top vertex

  mappings[5] := 5;

# edge 4

  mappings[4*k+2] := 1;

  j := 4*k-2;
  for i from 4*k+3 to 5*k do
    mappings[i] := j;
    j := j+1;
  od;

# edge 5

  mappings[5*k+1] := 2;

  j := 5*k-4;
  for i from 5*k+2 to 6*k-1 do
    mappings[i] := j;
    j := j+1;
  od;

# edge 6

  mappings[6*k] := 3;

  j := 6*k-6;

```

```

    for i from  $6*k+1$  to  $7*k-2$  do
      mappings[i] := j;
      j := j+1;
    od;

# edge 7

mappings[ $7*k-1$ ] := 4;

j :=  $7*k-8$ ;
for i from  $7*k$  to  $8*k-3$  do
  mappings[i] := j;
  j := j+1;
od;

# face 1

j := 6;
for i from  $k*k+6*k-1$  to  $k*k+7*k-4$  do
  mappings[i] := j;
  j := j+1;
od;

j :=  $k*k+4*k-6$ ;
for i from  $k*k+7*k-3$  to  $3/2*k*k+9/2*k-1$  do
  mappings[i] := j;
  j := j+1;
od;

# face 2

j :=  $k+4$ ;
for i from  $3/2*k*k+9/2*k$  to  $3/2*k*k+11/2*k-3$  do
  mappings[i] := j;
  j := j+1;
od;

j :=  $3/2*k*k+3/2*k-3$ ;
for i from  $3/2*k*k+11/2*k-2$  to  $2*k*k+3*k$  do

```

```

    mappings[i] := j;
    j := j+1;
  od;

# face 3

  j := 2*k+2;
  for i from 2*k*k+3*k+1 to 2*k*k+4*k-2 do
    mappings[i] := j;
    j := j+1;
  od;

  j := 2*k*k-k;
  for i from 2*k*k+4*k-1 to 5/2*k*k+3/2*k+1 do
    mappings[i] := j;
    j := j+1;
  od;

# face 4

  j := 3*k;
  for i from 5/2*k*k+3/2*k+2 to 5/2*k*k+5/2*k-1 do
    mappings[i] := j;
    j := j+1;
  od;

  j := 5/2*k*k-7/2*k+3;
  for i from 5/2*k*k+5/2*k to 3*k*k+2 do
    mappings[i] := j;
    j := j+1;
  od;

end;

# *****
# *****

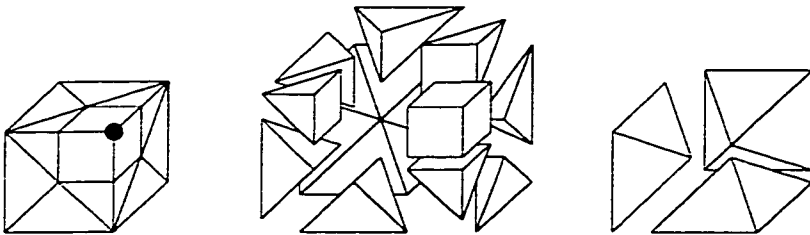
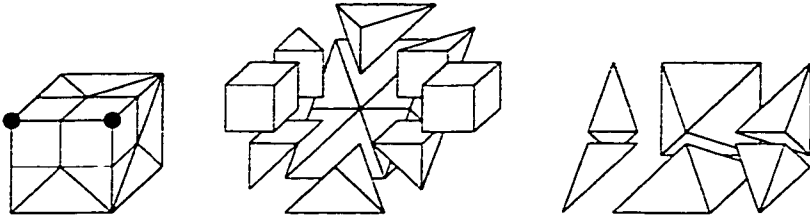
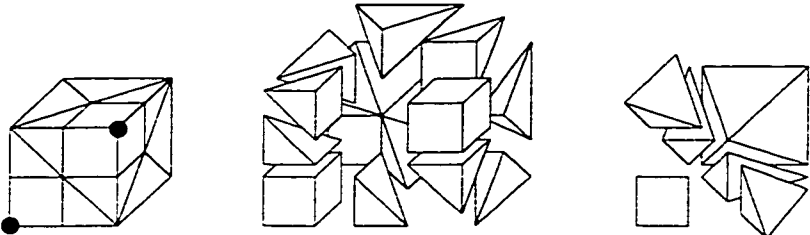
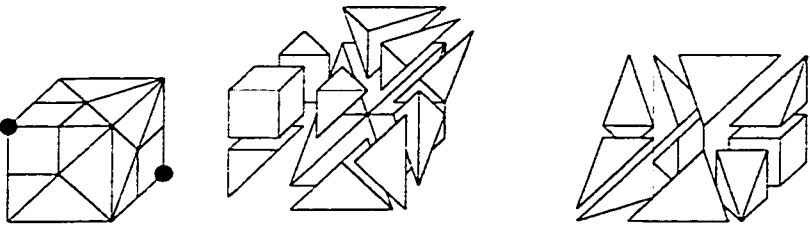
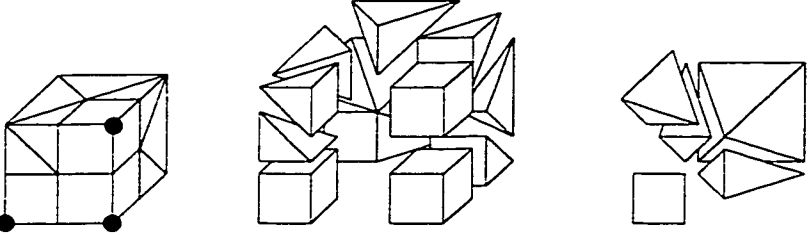
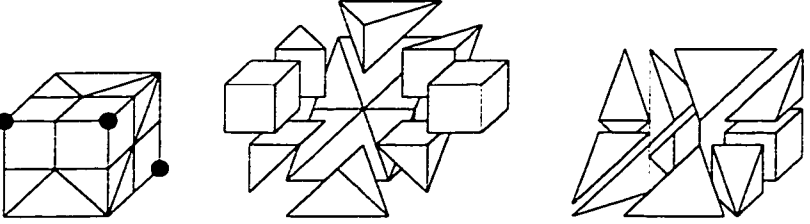
```

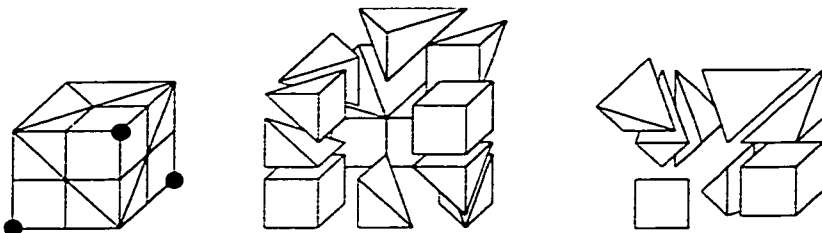
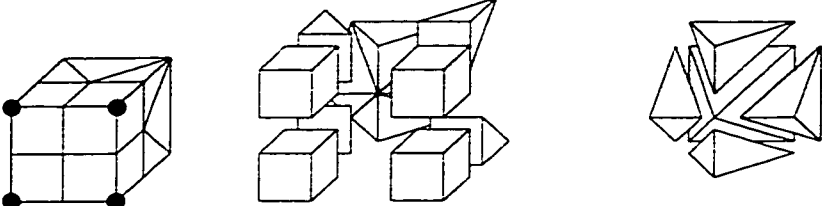
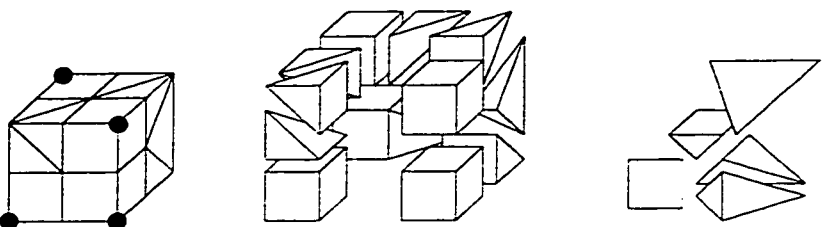
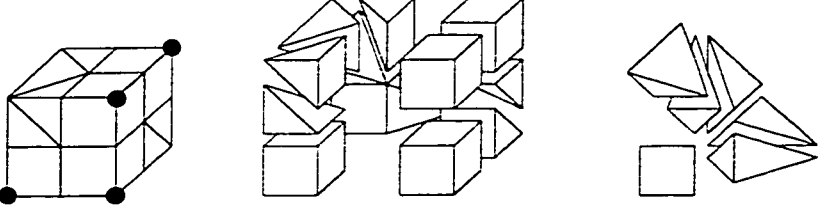
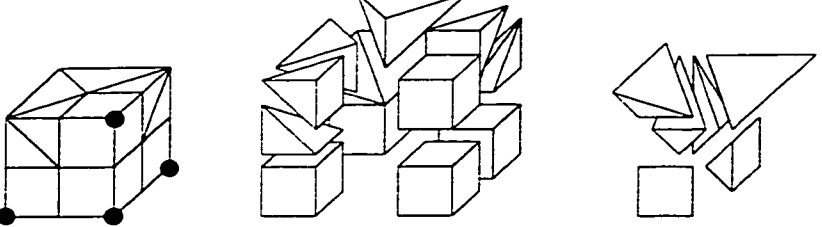
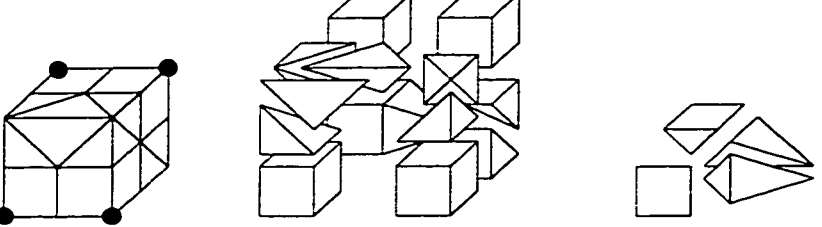

Appendix C

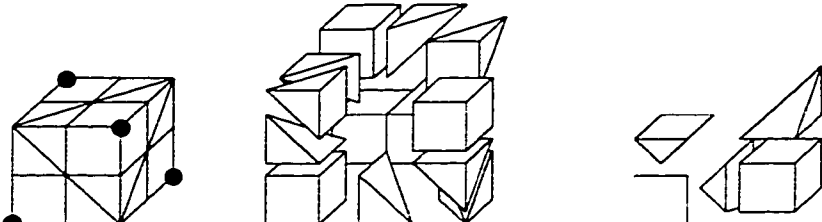
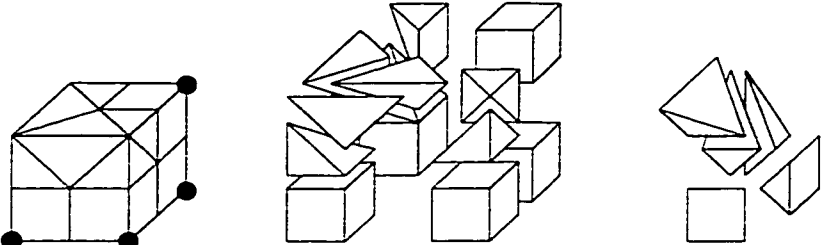
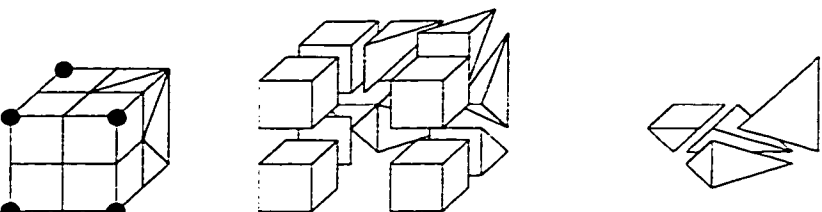
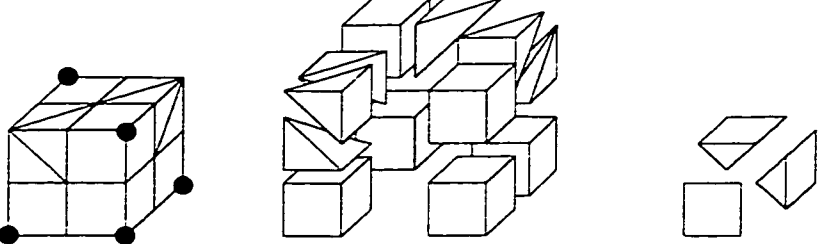
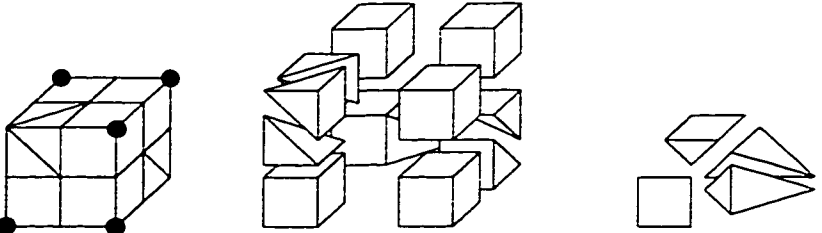
Non-Standard Refinement of Non-Conformal Cubes

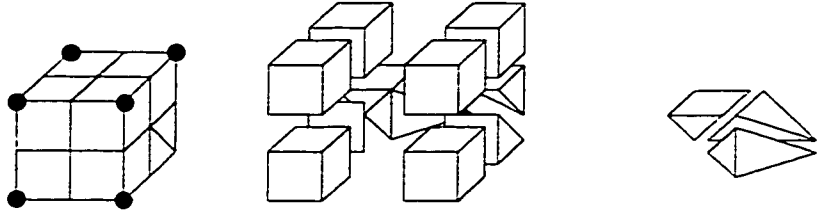
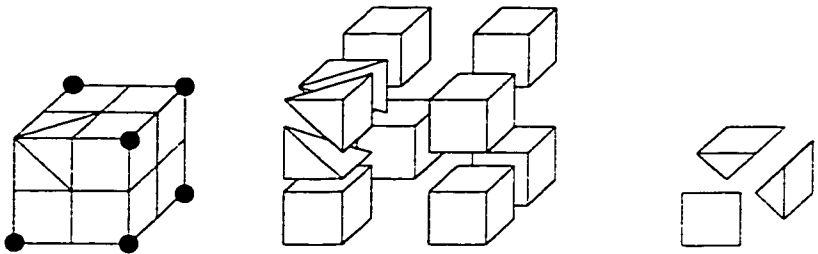
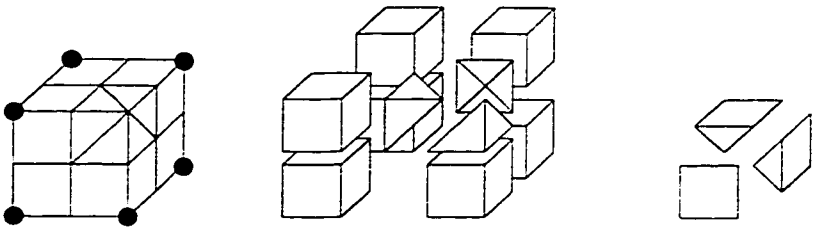
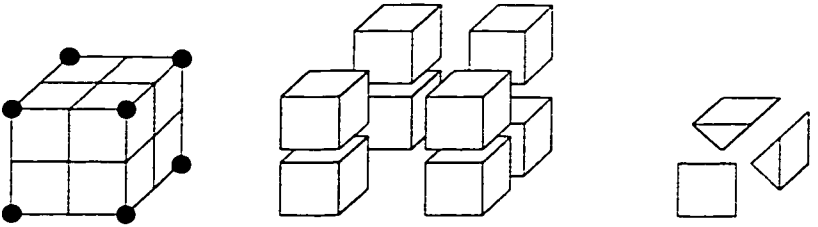
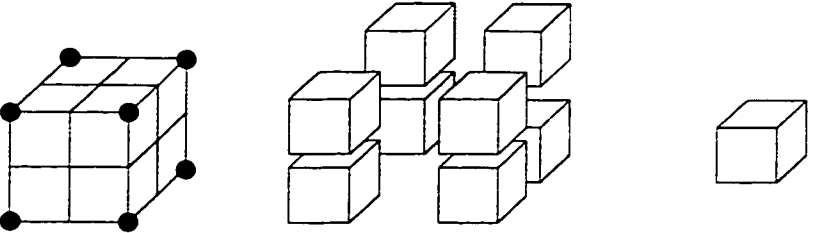
In Section 7.5, we saw that during the refinement of 3D crystalline meshes we need to identify cubes that share one or more vertices with a higher level cube and refine them (using non-standard refinement) in order to obtain a legal mesh from a conformable mesh. We also saw that there are 22 distinct neighboring configurations, that is, the same as the number of equivalence classes for 3-variable boolean functions (Harvard classes [44])

We proved that if each element is refined so that its faces are cut in a way consistent with Figure 7.14, the resulting mesh will be conformable. Now we will present the refinement used in all 22 configurations. The first column of the tables shown, gives the numbering of the specific configuration. The second column, shows which vertices on the cube are shared with cubes having higher level number (dark circles). The third column, shows how part of the cube will be partitioned. The last column shows how the remaining of the cube will be partitioned. Notice that, in all cases, only cubes and the shapes shown in Figure 4.7, are used.

Case 1:	
Case 2:	
Case 3:	
Case 4:	
Case 5:	
Case 6:	

<p>Case 7:</p>	
<p>Case 8:</p>	
<p>Case 9:</p>	
<p>Case 10:</p>	
<p>Case 11:</p>	
<p>Case 12:</p>	

Case 13:	
Case 14:	
Case 15:	
Case 16:	
Case 17:	

<p>Case 18:</p>	
<p>Case 19:</p>	
<p>Case 20:</p>	
<p>Case 21:</p>	
<p>Case 22:</p>	

Bibliography

- [1] Ansys modeling and meshing guide. Available at <http://www.ansys.com/services/documentation/manuals.html>.
- [2] John H. Argyris. Energy theorems and structural analysis, part I: General theory. *Aircraft Engineering*, 1954.
- [3] Alan J. Baker. *Finite element computational fluid mechanics*. Hemisphere Pub. Corp., 1983.
- [4] Steven E. Benzley, Ernest Perry, Karl Merkle, Brett Clark, and Greg Sjaardema. A comparison of all hexagonal and all tetrahedral finite element meshes for elastic and elastic-plastic analysis. *4th International Meshing Roundtable*, pages 179–192, 1995.
- [5] Marsha Berger and Philip P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82, 1989.
- [6] Marshall W. Bern, David Eppstein, and John R. Gilbert. Provably good mesh generation. *Journal of Computer and System Sciences*, 48(3):384–409, June 1994.
- [7] Marshall W. Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. In *Proceedings of the 3rd Workshop in Algorithms and Data Structures*, number 709 in Lecture Notes in Computer Science, pages 188–199. Springer-Verlag, August 1993.
- [8] Ted D. Blacker. Cubit mesh generation environment. Volume 1: Users manual. Technical Report SAND94-1100, Sandia National Laboratories, May 1994.
- [9] Frank J. Bossen and Paul S. Heckbert. A pliant method for anisotropic mesh generation. In *5th International Meshing Roundtable*, pages 63–74, October 1996. <http://www.cs.cmu.edu/~ph>.
- [10] David S. Burnett. *Finite elements analysis: from concepts to applications*. Addison-Wesley, 1987.
- [11] Graham F. Carey. *Computational grids*. Taylor & Francis, 1989.
- [12] José G. Castaños. *Parallel Adaptive Unstructured Computation*. PhD thesis, Department of Computer Science, Brown University, 2000.

- [13] José G. Castaños and John E. Savage. The dynamic adaptation of parallel mesh-based computation. Technical Report CS-96-31, Department of Computer Science, Brown University, October 1996.
- [14] José G. Castaños and John E. Savage. The dynamic adaptation of parallel mesh-based computation. In *Proceedings of the eighth SIAM conference on parallel processing for scientific computation*, March 1997.
- [15] José G. Castaños and John E. Savage. Parallel refinement of unstructured meshes. In *Procs. IASTED Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, November 1999.
- [16] José G. Castaños and John E. Savage. PARED: a framework for the adaptive solution of PDEs. In *IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, August 1999.
- [17] José G. Castaños and John E. Savage. Repartitioning unstructured adaptive meshes. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 823–832, Cancun, Mexico, May 2000.
- [18] M. J. Castro-Díaz, Frederic Hecht, and Bijan Mohammadi. New progress in anisotropic grid adaptation for inviscid and viscous flows simulations. In *4th Annual Intl. Meshing Roundtable*, October 1995. <http://www.ce.cmu.edu/sowen/Roundtable.agenda.html>.
- [19] Vasiliki Chatzi. Finding basis functions for pyramidal finite elements. *3rd CGC Workshop on Computational Geometry*, 1998.
- [20] Vasiliki Chatzi and Franco P. Preparata. Integer-coordinate crystalline meshes. In *Proceedings of the Swiss Conference of CAD/CAM 99*, pages 199–206, Neuchatel, Switzerland, February 1999.
- [21] Min-Ben Chen, Tyng-Ruey Chuang, and Jan-Jan Wu. Highly parallel mesh generation in high performance Fortran. In *International Parallel and Distributed Processing Symposium 2000*, pages 1–13, April 1999.
- [22] Nikos Chrisochoides and Demian Nave. On the parallelization of guaranteed-quality 3D Delaunay mesh generators. In *2nd Symposium on Trends in Unstructured Mesh Generation*, University of Colorado, Boulder, August 1999.
- [23] Nikos Chrisochoides and Demian Nave. Simultaneous mesh generation and partitioning for Delaunay meshes. In *Proceedings, 8th International Meshing Roundtable*, pages 55–66, South Lake Tahoe, CA, U.S.A., October 1999.
- [24] Ray W. Clough. The finite element method in plane stress analysis. In *Proceedings of the 2nd ASCE conference on Electronic Computation*, Pittsburg, Pennsylvania, 1960.

- [25] W. J. Coirier and K. G. Powel. An accuracy assessment of Cartesian-mesh approaches for the Euler equations. *Journal of Computational Physics*, 117:121–131, 1995.
- [26] Philip P. Colella and Peter Woodward. The piecewise parabolic method (ppm) for gasdynamical simulations. *Journal of Computational Physics*, 54:174–201, 1984.
- [27] Gianni Comini. *Finite element analysis in heat transfer: basic formulation and linear problems*. Taylor & Francis, 1994.
- [28] Boris Delaunay. Sur la sphère vide. *Bull. Acad. Science USSR VII: Class. Sci. Mat. Nat.*, pages 793–800, 1934.
- [29] Paul Fischer and David Gottlieb. On the optimal number of subdomains for hyperbolic problems on parallel computers. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):65–76, Spring 1997.
- [30] Paul F. Fischer, N.I. Miller, and Henry M. Tufo. An overlapping Schwarz method for spectral element simulation of three-dimensional incompressible flows. In *Proceedings of the IMA Workshop on Parallel Solution of Partial Differential Equations*. Minneapolis, MN, June 1997.
- [31] Michel Fortin, Marie-Gabrielle Vallet, Julien Dompierre, Yves Bourgault, and Wagdi G. Habashi. Anisotropic mesh adaptation: Theory, validation and applications. In *Third EC-COMAS Computational Fluid Dynamics Conference*, Paris, September 1996.
- [32] William H. Frey. Selective refinement: A new strategy for automatic node placement in graded triangular meshes. *International Journal for Numerical Methods in Engineering*, 24:2183–2200, 1987.
- [33] Adam Gaither, Dave Marcum, Donna Reese, and Nigel Weatherill. A paradigm for parallel unstructured grid generation. In *5th International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 731–740, Mississippi State University, April 1996.
- [34] Jerome Galtier and Paul-Luis George. Prepartitioning as a way to mesh subdomains in parallel. In *5th International Meshing Roundtable*, pages 107–122, Sandria National Laboratories, October 1996.
- [35] Erich Gamma, Richard Helm, Johnson Ralph, and Vlissides John. *Design patterns*. Addison-Wesley, 1994.
- [36] Geompack90 file formats for regions and meshes. Available at <http://tor-pw1.attcanada.ca/bjoe/index.htm>.
- [37] P. L. George. *Automatic mesh generation: Application to finite element methods*. John Wiley & Sons, 1991.

- [38] N. A. Golias and Theodoros D. Tsiboukis. An approach to refining three-dimensional tetrahedral meshes based on Delaunay transformations. *International Journal for Numerical Methods in Engineering*, 37:793–812, 1994.
- [39] David Gottlieb and Steven Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*. Society of Industrial and Applied Mathematics, Philadelphia, PA USA, 1977.
- [40] Roberto Grosso. Adaptive refinement of hierarchical meshes. In *2nd Symposium on Trends in Unstructured Mesh Generation*, University of Colorado, Boulder, August 1999.
- [41] Ernest Hairer and Gerhard Wanner. *Solving ordinary differential equations II: stiff and differential-algebraic problems*. Springer-Verlag, 1996.
- [42] Matthias M. Hannemann and Karsten Weihe. On the discrete core of quadrilateral mesh refinement. *International Journal for Numerical Methods in Engineering*, 46:593–622, 1994.
- [43] Jonathan C. Hardwick. Implementation and evaluation of an efficient 2D parallel Delaunay triangulation algorithm. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.
- [44] Michael A. Harrison. *Combinatorial problems in Boolean algebras and applications to the theory of switching*. PhD thesis, University of Michigan, 1963.
- [45] Michelle Hribar and Valerie E. Taylor. Practical issues of 2-D parallel finite element analysis. In Jagdish Chandra, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 64–68, Boca Raton, FL, USA, August 1994. CRC Press.
- [46] Jianming Jin. *The finite element method in electromagnetics*. John Wiley & Sons, 1993.
- [47] Claes Johnson. *Numerical solutions of partial differential equations by the finite element method*. Cambridge University Press, 1995.
- [48] Shiro Kobayashi, Soo-Ik Oh, and Taylan Altan. *Metal forming and the finite-element method*. Oxford University Press, 1983.
- [49] William J. Layton, Joseph M. Maubach, and Patrick J. Rabier. Robustness of an elementwise parallel finite element method for convection-diffusion problems. *SIAM Journal on Scientific Computing*, 19(6):1870–1891, November 1998.
- [50] Lars-Erik Lindgren, Hans-Eke Haggblad, Moyra McDill, and A. S. Oddy. Automatic remeshing for three-dimensional finite element simulation of welding. *Computer Methods in Applied Mechanics and Engineering*, 147:401–409, 1997.
- [51] S. H. Lo. Generation of high quality gradation finite element mesh. *Engineering Fracture Mechanics*, 2(41):191–202, 1992.

- [52] S. H. Lo. 3D mesh refinement in compliance with a specified node spacing function. *Computational Mechanics*, 21(1):11–19, 1998.
- [53] Rainald Lohner. Mesh adaptation in fluid mechanics. *Engineering Fracture Mechanics*, 50(5):819–847, 1995.
- [54] Rainald Lohner. Progress in grid generation via the advancing front technique. *Engineering with Computers*. 12:186–210, December 1996.
- [55] Rainald Lohner and Juan R. Cebal. Parallel advancing front grid generation. In *Proceedings, 8th International Meshing Roundtable*, pages 67–74, South Lake Tahoe, CA, U.S.A., October 1999.
- [56] B. V. Manichev. New methods and algorithms of numeric solutions of differential-algebraic equation sets. In *Proceedings of 2nd International Conference on Science and Technology*, 1994.
- [57] B. V. Manichev, A. A. Mezentsev, and Y. B. Uvarov. Development and testing of the new numeric integration methods within the graph-theoretic simulation software. In *Proceedings of the Swiss Conference of CAD/CAM'99*, 1999.
- [58] A. Merrouche, A. Selmán, and C. Knopf-Lenoir. 3D adaptive mesh refinement. *Communications In Numerical Methods In Engineering*, 14:397–407, 1998.
- [59] G. L. Miller, D. Talmor, S. H. Teng, N. Walkington, and H. Wang. Control volume meshes using sphere packing: Generation, refinement and coarsening. In *5th International Meshing Roundtable*, pages 47–62, Sandria National Laboratories, October 1996.
- [60] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 8th Annual Symposium on Computational Geometry (SCG '92)*, pages 212–221, Berlin, FRG, June 1992. ACM Press.
- [61] Alfred L. Nelson, Karl W. Folley, and William M. Borgman. *Analytic geometry*. The Ronald press company, New York, 1949.
- [62] J. T. Oden, L. Demkowicz, T. Strouboulis, and P. Devloo. *Adaptive methods for problems in solid and fluid mechanics.*, chapter 14. John Wiley & Sons, 1986.
- [63] T. Okusanya and J. Peraire. Parallel unstructured mesh generation. In *5th International Conference on Numerical Grid Generation on Computational Field Simulations*, pages 719–729, Mississippi State University, April 1996.
- [64] T. Okusanya and J. Peraire. 3D parallel unstructured mesh generation. *Trends in Unstructured Mesh Generation*, 220:109–115, July 1997.
- [65] Kohnke P., editor. *ANSYS User's Manual for Revision 5.3*. ANSYS Inc., 1996.

- [66] Angel Plaza and Graham F. Carey. About local refinement of tetrahedral grids based on bisection. In *5th International Meshing Roundtable*, pages 123–136, Sandria National Laboratories, October 1996.
- [67] Angel Plaza, J. P. Suarez, and M. A. Padron. Mesh graph structure for longest-edge refinement algorithms. In *7th International Meshing Roundtable*, pages 335–344, Sandria National Laboratories, October 1998.
- [68] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.
- [69] Geometric objects in QMG. Available at <http://www.cs.cornell.edu/Info/People/vavasis/qmg2.0/geom.html>.
- [70] M. Rivara. Selective refinement/derifinement algorithms for sequences of nested triangulations. *International Journal for Numerical Methods in Engineering*, 28:2889–2906, 1989.
- [71] M. Rivara and C. Levin. A 3-D refinement algorithm suitable for adaptive and multigrid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.
- [72] U. Ruede. *Mathematical and computational techniques for multilevel adaptive methods*. SIAM, 1993.
- [73] Jim Ruppert. A Delaunay refinement alorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, pages 1–45, February 1994.
- [74] R. Said, N. P. Weatherill, K. Morgan, and N. A. Verhoeven. Distributed parallel Delaunay mesh generation. In *Computer methods in applied mechanics and engineering*, volume 177, pages 109–125, 1999.
- [75] R. Schneiders, R. Schindler, and F. Weiler. Octree-based generation of hexahedral element meshes. In *Proceedings 5th International Meshing Roundtable*, 1996.
- [76] H. R. Schwarz and J. R. Whiteman. *Finite element methods*. Academic press, 1988.
- [77] Mark S. Shephard and K. G. Marcel. Automatic three-dimensional mesh generation by the finite octree technique. *Int. J. Numer. Meth. Eng.*, 32:709–749, 1991.
- [78] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, Carnegie Mellon University, 1997.
- [79] Kenji Shimada. Automatic anisotropic mesh generation. In *Proc. 51st Ann. Conf., Inf. Proc. Soc. of Japan*, volume 1, pages 59–60, November 1995. 4C-7. In Japanese.
- [80] R. Sneiders. Refining quadrilateral and hexahedral element meshes. In *5th International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 679–688, Mississippi State University, April 1996.

- [81] Virgil Snyder and C. H. Sisam. *Computational Geometry of Space*. Henry Holt and company, New York, 1914.
- [82] M. L. Staten and N. L. Jones. Local refinement of three-dimensional finite element meshes. *Engineering with Computers*, 13:165–174, 1997.
- [83] R. Taghavi. Automatic, parallel and fault tolerant mesh generation from CAD. *Engineering with Computers*, 12:1778–185, December 1996.
- [84] Lazarus Teneketzis Tenek and John Argyris. *Finite element analysis for composite structures*. Kluwer Academic, 1998.
- [85] Harold E. Trease and Genevieve L. Barrett. Three-dimensional, parallel, hybrid grid generation for the ASCI program. In *2nd Symposium on Trends in Unstructured Mesh Generation*, University of Colorado, Boulder, August 1999.
- [86] Truegrid's truegeometry. Available at <http://www.truegrid.com/geometry.html>.
- [87] Henry M. Tufo and Paul F. Fischer. Fast parallel direct solvers for coarse grid problems. *Journal of Parallel and Distributed Computing*, 1997.
- [88] Marie-Gabrielle Vallet. *Génération de Maillages Éléments Finis Anisotropes et Adaptatifs*. PhD thesis, L'Université de Paris VI, September 1992. Abstract at <http://www.inria.fr/RRRT/TU-0197.html>.
- [89] Marie-Gabrielle Vallet, Frederic Hecht, and B. Mantel. Anisotropic control of mesh generation based upon a Voronoi type method. In *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, 1991.
- [90] C. Waishaw and M. Cross. Load-balancing for parallel adaptive unstructured grids. In *Numerical Grid Generation in Computational Field Simulations*, pages 781–790, University of Greenwich, July 1998.
- [91] N. P. Weatherill, R. Said, and K. Morgan. The construction of large unstructured grids by parallel Delaunay grid generation. In *Numerical Grid Generation in Computational Field Simulations*, pages 53–73, University of Greenwich, July 1998.
- [92] W. Weaver Jr. and P. Johnston. *Finite elements for structural analysis*. Prentice-Hall, 1984.
- [93] X. Xu, C. C. Pain, A. J. H. Goddard, and C. R. E. de Oliveira. An automatic adaptive meshing technique for Delaunay triangulations. *Computer methods in applied mechanics and engineering*, 161:297–303, 1998.
- [94] M. A. Yerry and Mark S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal of Numerical Methods in Engineering*, 20:1965–1990, 1984.

- [95] O. C. Zienkiewicz and R. L. Taylor. *The finite element method*. McGraw-Hill, 1994.