DEVELOPMENT OF A MULTIGRID ACCELERATED EULER SOLVER ON ADAPTIVELY REFINED TWO- AND THREE-DIMENSIONAL CARTESIAN GRIDS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCE
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MEHTAP ÇAKMAK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
MECHANICAL ENGINEERING

JULY 2009

Approval of the thesis:

DEVELOPMENT OF A MULTIGRID ACCELERATED EULER SOLVER ON ADAPTIVELY REFINED TWO- AND THREE-DIMENSIONAL CARTESIAN GRIDS

submitted by **MEHTAP ÇAKMAK** in partial fulfillment of the requirements for the degree of **Master of Science in Mechanical Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _______________
Dean, Graduate School of **Natural and Applied Science**

Prof. Dr. Suha Oral _______________
Head of Department, **Mechanical Engineering**

Prof. Dr. M. Haluk Aksel _______________
Supervisor, **Mechanical Engineering Dept., METU**

Asst. Prof. Dr. Cüneyt Sert _______________
Co-Supervisor, **Mechanical Engineering Dept., METU**

Examining Committee Members:

Instructor Dr. Tahsin A. Çetinkaya _______________
Mechanical Engineering Dept., METU

Prof. Dr. M. Haluk Aksel _______________
Mechanical Engineering Dept., METU

Asst. Prof. Dr. Cüneyt Sert _______________
Mechanical Engineering Dept., METU

Asst. Prof. Dr. Almıla Güvenç Yazıcıoğlu _______________
Mechanical Engineering Dept., METU

Prof. Dr. İsmail Hakkı Tuncer _______________
Aerospace Engineering Dept., METU

Date:

ABSTRACT

DEVELOPMENT OF A MULTIGRID ACCELERATED EULER SOLVER ON ADAPTIVELY REFINED TWO- AND THREE-DIMENSIONAL CARTESIAN GRIDS

Çakmak, Mehtap
M.S., Department of Mechanical Engineering
Supervisor: Prof. Dr. M. Haluk Aksel
Co-Supervisor: Asst. Prof. Dr. Cüneyt Sert

July 2009, 166 pages

Cartesian grids offer a valuable option to simulate aerodynamic flows around complex geometries such as multi-element airfoils, aircrafts, and rockets. Therefore, an adaptively-refined Cartesian grid generator and Euler solver are developed. For the mesh generation part of the algorithm, dynamic data structures are used to determine connectivity information between cells and uniform mesh is created in the domain. Marching squares and cubes algorithms are used to form interfaces of cut and split cells. Geometry-based cell adaptation is applied in the mesh generation. After obtaining appropriate mesh around input geometry, the solution is obtained using either flux vector splitting method or Roe's approximate Riemann solver with cell-centered approach. Least squares reconstruction of flow variables within the cell is used to determine high gradient regions of flow. Solution based adaptation method is then applied to current mesh in order to refine these regions and also coarsened regions where unnecessary small cells exist. Multistage time stepping is used with local time steps to increase the convergence rate. Also FAS multigrid technique is used in order to increase the convergence rate. It is obvious that implementation of geometry and solution based adaptations are easier for Cartesian meshes than other types of meshes. Besides, presented numerical results show the accuracy and efficiency of the algorithm by especially using geometry and solution based adaptation. Finally, Euler solutions of Cartesian grids around airfoils, projectiles and wings are compared with the experimental and numerical data available in the literature and accuracy and efficiency of the solver are verified.

Keywords: Cartesian Grid Generation, Ray-Casting Method, Marching Squares and Cubes Algorithm, Euler Equations, Least Square Reconstruction Algorithm, Multigrid Method

ÖZ

İKİ VE ÜÇ BOYUTLU UYARLAMALI KARTEZYEN HESAPLAMA AĞLARI İÇİN ÇOKLU AĞ YÖNTEMİ İLE HIZLANDIRILMIŞ EULER ÇÖZÜCÜSÜ GELİŞTİRİLMESİ

Çakmak, Mehtap
Yüksek Lisans, Makina Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. M. Haluk Aksel
Ortak Tez Yöneticisi: Yard. Doç. Dr. Cüneyt Sert

Temmuz 2009, 166 sayfa

Kartezyen yöntemi, uçaklar, roketler ve helikopterler gibi karmaşık geometriler çevresindeki hava akışını modellemek için doğru yaklaşımı sundu. Bu doğru modellemeyi gerçekleştirebilmek için kartezyen ağ üreticisi ve üç boyutlu Euler çözücüsü geliştirildi. Çözücü kısmı için, zamana bağlı olmayan iki veya üç boyutlu Euler denklemleri kullanıldı, akı formülasyonları ise akı vektör ayrıştırması yöntemleri ve akı fark ayrıştırması yöntemi kullanılarak gerçekleştirildi. Hücre merkezli sonlu hacim yöntemi kullanıldı. Ağ üretme kısmında ise, hücreler arasındaki bağlantı bilgisini belirlemek için dinamik veri yapıları kullanıldı ve geometriye bağlı hücre adaptasyonu, ağ üretme işleminde uygulandı. Çözüm elde edildikten sonra da, çözüme bağlı gradyan bilgisi göz önüne alınarak çözüme bağlı adaptasyon güncel ağa uygulandı. Yakınsamanın hızlandırılabilmesi için yerel zaman adımlarıyla birlikte çok kademeli zaman uygulaması kullanıldı ve yine yakınsamanın hızlandırılması için çoklu ağ yöntemi de kullanıldı. Son olarak, bu çözücü kullanılarak elde edilen veriler literatürde mevcut deneysel sonuçlarla karşılaştırıldılar.

Anahtar Kelimeler: Kartezyen Ağ Yaratılması, Işın Atma Yöntemi, Kenar ve Yüzey Yeniden Yapılandırma Algoritması, Euler Denklemleri, Ufak Kareler Yeniden Yapılandırması, Çoklu Ağ Yöntemi

Dedicated to my family, Cemil, Tülay, Melih Çakmak
and Necmettin Cevheri…

ACKNOWLEDGEMENTS

I would like to thank my supervisor and co-supervisor, Prof. Dr. Haluk Aksel and Asst. Prof. Dr. Cüneyt Sert, for their guidance and supports throughout my research. I am also grateful to TUBITAK for their financial support. Besides, I would like to give my appreciations to Bercan Siyahhan and Alp Emre Öngüt. Finally, I would like to express my gratitude to my family and my friends (especially Pınar Kural and Necmettin Cevheri) who supported and encouraged me throughout my education life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

A, A()	coefficient matrix
c	speed of sound
c_∞	far-field speed of sound
dA	surface area element
e	error vector
e	specific internal energy
E	specific total energy
F	flux vector
H	specific total enthalpy
L	domain size
M	Mach number
n	level of a cell
n	normal vector
p	static pressure
p_∞	far-field pressure
Res, **r**	residual vector
R	right characteristic vector
S_x , S_y	projections of edges
t	time
u	velocity in the *x*-direction
v	velocity in the *y*-direction
w	velocity in the *z*-direction
v	velocity vector
V_n	normal velocity
$\Delta\mathbf{V}$	wave strengths
x	exact solution
y	approximate solution

Greek Letters

α	angle of attack
α_k	stage coefficients
γ	ratio of specific heats
λ	Eigen values
ρ	density
ρ_∞	far-field density
Φ	flux vector perpendicular to surface
Ψ_x , Ψ_y	convective spectral radii
$d\Omega$	volume element

CHAPTER 1

INTRODUCTION

Fluid-flow problems generally have complex governing equations. Therefore, most of the fluid-flow problems cannot be solved by analytical methods due to the nonlinear terms in their governing equations. However, analytical solutions are sometimes possible when nonlinear terms are negligibly small. But generally, nonlinear terms are not as small as to be neglected. If the nonlinearities are important for the fluid-flow problems, then numerical methods and algorithms are used to solve and analyze these problems.

Computational Fluid Dynamics (CFD) is an important field of fluid dynamics which enables one to obtain numerical solutions of complex fluid-flow problems including nonlinear terms and also simulate fluid-flows that cannot be observed in laboratory situations due to the some flight regimes that cannot be simulated in wind tunnels; such as higher Mach numbers and higher flow field temperatures. Numerical solutions of many complex problems; such as compressible or incompressible, laminar or turbulent, single or multiphase flows are possible with CFD.

CFD techniques today are very powerful due to the high speed and large memory computers; however, turbulence modeling, selection of the accurate numerical techniques, algorithmic efficiency, surface modeling and grid generation around complicated and multi-component geometries are still barriers to CFD maturation, especially in three dimensions. Therefore, various new approaches to deal with these problems are being developed. For example, in order to handle grid generation problems and reduce the user intervention to generate grid, the grid generation and adaptation processes are tried to be automated. In addition, more accurate and efficient solutions are tried to be obtained by means of the advances in numerical methodologies.

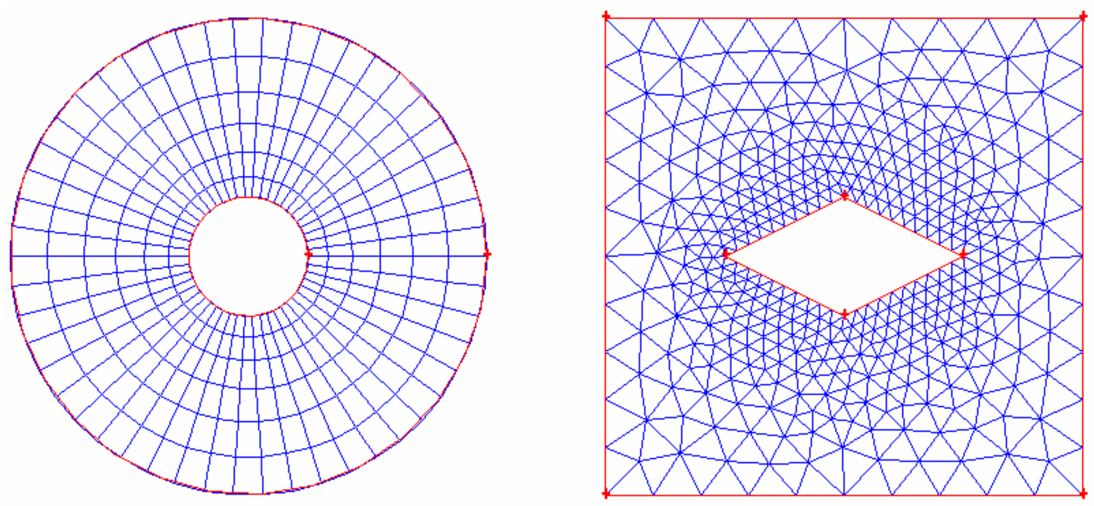
Steps of finding approximate solutions of complex flow problems can be summarized as follows. Firstly, fundamental physical principles of any fluid flow are expressed in the form of conservative or nonconservative governing equations. These can be either integral equations or partial differential equations. Then, flow domain must be split into sub-domains called elements or cells. The collection of all cells is called a mesh or grid. Then, the equations governing the motion of fluid are replaced with discretized algebraic forms and solved to obtain approximate solutions for the flow field values at each of the sub-domains [1]. Since one of the most difficult steps is grid generation, Cartesian Grid is an attractive approach to CFD. It enables to create grids around complex geometries easily. Furthermore, grid and solution adaptations are possible without user interventions, i.e. automatically with Cartesian Grids.

1.1 Grid Generation and Adaptation

As it was mentioned, grid generation is an important and time-consuming problem of CFD. It requires considerable expertise, since not only understanding of mathematical formulation and numerical algorithms is necessary, but also understanding of physical principles of flow problems is very important for obtaining satisfactory resolution in flow domain. Therefore, in order to solve discretized algebraic equations of any fluid flow, an efficient grid, which resolves the physical properties of flow, minimizes the errors and uses as fewer grid points as possible to save the memory usage must be generated [2]. This is a hard task. Therefore, fully automatic grid generation techniques are necessary in order to handle these difficulties.

Grid adaptation is to put more grid points in the regions where the large gradients in the flow field properties exist and remove grid points from the regions where these gradients are insignificant in order to decrease the local resolution of the grid. In other words, the aim of grid adaptation is to capture the physics of the flow effectively without using excessive grid points. There are two types of grid adaptation and their combination is also possible. The first one is r-refinement (grid point redistribution) and the other is h-refinement (grid point embedding). R-refinement is applied to the current grid by moving the grid points to the regions that need more resolution due to the high flow field gradients without changing the actual number of grid points. In this way, connectivity information does not change. H-refinement is the modification of current grid by changing connectivity information. This change is either adding extra grid points where the higher resolution of current grid is necessary or removing redundant grid points. For example, h-refinement is performed for Cartesian grid method by dividing parent cells in order to obtain child cells for refinement or removing children of a parent cell to obtain parent cell instead of its children for coarsening.

Grid generation methods can be classified into two groups: structured and unstructured. Both of these have advantageous and disadvantageous properties. Examples of these grids are given in Figure 1.1.



(a) (b)

Figure 1.1 Examples of (a) structured and (b) unstructured grids

1.1.1 Structured Grid

A structured grid is composed of quadrilaterals in two-dimensions and hexahedra in three-dimensions. It is one that grid points (vertex, node) are transferred from physical space (Cartesian coordinates for two-dimensional problems: *x, y*) to computational space (ζ, η) and they are represented by the indices *i, j*. Hence, connectivity information of the grid is implicitly known by the indices of grid points. For example, neighbors of a grid point are found by adding or subtracting an integer value to or from its indices [3]. This simplification in data structure has an important effect on creating efficient and simpler codes owing to the fact that the calculation of fluxes and gradients is simpler compared to unstructured grids. Besides, implementation of implicit scheme to structured grids is easier than that to unstructured grids. In addition, computer memory usage is less than unstructured grid. Since incrementing grid points near the boundary of the geometry is achieved easily by decreasing the spacing between them, viscous solutions of flow problems are obtained more effectively and accurately than that in unstructured grids. Finally, for a structured grid, each cell has only one neighboring cell on each of its faces. A smooth grid is obtained by means of this rule. These are the advantages of structured grid. However, structured grid generation around complex geometries is a very big problem. Therefore, although it has a lot of advantages, in general it is not a preferred grid generation technique. In order to generate structured grid around complex, multi-component geometries, some approaches such as multi-block and Chimera technique are used but these methods are very complicated and also decrease the advantages of structured grid. Furthermore, generating structured grid around complex geometry by using any one of these methods takes man-months. Detailed information can be found in reference [3]. In addition, transformation of governing equations from physical space to computational space is a very difficult task. The final disadvantage of structured grids is that an implementation of h-refinement causes the huge increase of grid points since adding a point to a structured grid requires adding a line on which the points lie [4]. Therefore, r-refinement is more suitable for structured grids than h-refinement.

1.1.2 Unstructured Grid

An unstructured grid is composed of mostly triangular and rarely quadrilateral cells in two-dimensions and of hexahedral, prismoidal, pyramidal and mostly tetrahedral cells in three-dimensions. There is no need for transformation between physical and computational space. In addition, for an unstructured grid, there is no ordering for grid points and neighboring cells. In other words, grid points in unstructured grids cannot be identified by their indices. As a result, complicated data structure is mandatory to construct the connectivity information between cells.

Memory requirement of unstructured grids is higher and computational efficiency of unstructured grids is lower than those of structured girds because of the necessity of complex data structure. But, in spite of these disadvantages, nowadays unstructured grid methods become increasingly popular since it is capable of handling geometrically complex problems. Furthermore, grid adaptation especially h-refinement technique is easier to accomplish on unstructured grids than on structured grids. The final and the most attractive advantage of unstructured grids is that an unstructured grid is very suitable to automatic grid generation and adaptation.

By the way, Advancing Front Method and Delaunay Triangulation Method are the most widely used techniques to generate two-dimensional unstructured grids. Detailed information can be found in references [5] and [6].

1.1.3 Cartesian Grid

Cartesian grids are a special type of unstructured grids. In fact, this method is one of the earliest and simplest methods used for mesh generation. However, in the past, it was almost impossible to deal with curved boundaries accurately due to limited memory and simplicity of data structures so it was not a popular method. Contrary to the past, Cartesian grids are now very attractive and popular method due to their inherent simplicity and the ability to generate automatic meshes especially around complex and multi-component geometries.

It consists of squares in two-dimensions and cubes in three-dimensions which are placed parallel to the coordinate axes. It requires complicated data structures such as quadtree and octree data structures for two-dimensional and three-dimensional problems, respectively. However, it has many advantages which make it popular. One of its advantages is that the generation of Cartesian grid is easy even for complex geometries. In addition, automatic mesh generation with a minor user intervention is possible and geometric and surface adaptations are easy to implement. For example, denser mesh around shock waves can be generated easily by means of solution adaptation applied to Cartesian grid method. Hence, effective results are obtained in a short time without huge number of grids.

Another important advantage is that implementation of higher order schemes and multigrid method can be accomplished easily due to the permission of data structure. Finally, since the edges of square and the faces of cubic elements are aligned with the coordinate axes, there is no need for any complex formulation of velocity vectors in order to get normal and tangential components of them with respect to edges and faces. Consequently, flux formulation is simpler than other grid generation methods. Non-adapted and geometrically adapted Cartesian grids about the geometry are given in Figure 1.2.

The most difficult aspect of Cartesian grid is the complexity associated with the computational cells that have intersections with boundaries. These cells are called cut or split cells according to their total number of separate computational volumes. Samples of cut and split cells are shown in Figures 1.3 and 1.4, respectively. These are irregular cells and violate all the simplicity of Cartesian grids. However, these cells are very important for the Cartesian grid method since they play a key role in dealing with curved boundaries and obtaining accurate computational results. But sometimes small cut or split cells can cause time stepping problems. They may put severe restrictions on convergence rate and lead to inaccuracies, i.e. damage the stability criteria. In this work, this problem is solved by coarsening of small cells which will be explained in the next chapters. Another difficulty of this grid is that traditional Cartesian grid is insufficient to model viscous flows [7].

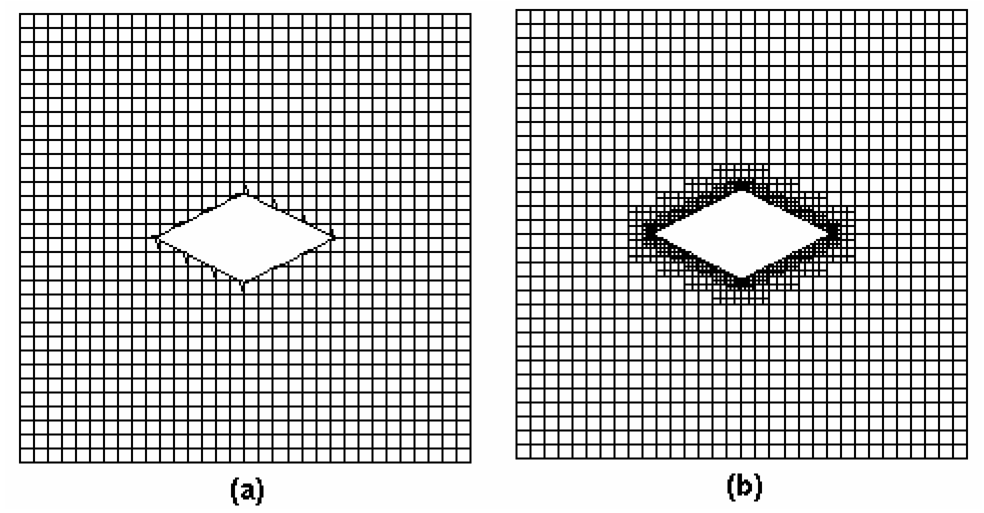


Figure 1.2 Examples of (a) non-adapted and (b) geometrically adapted Cartesian grids

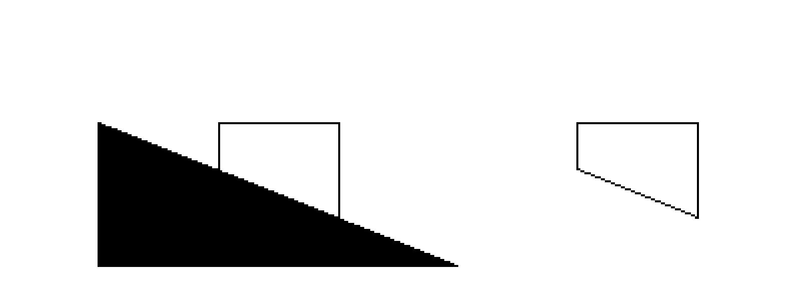


Figure 1.3 Example of a cut cell

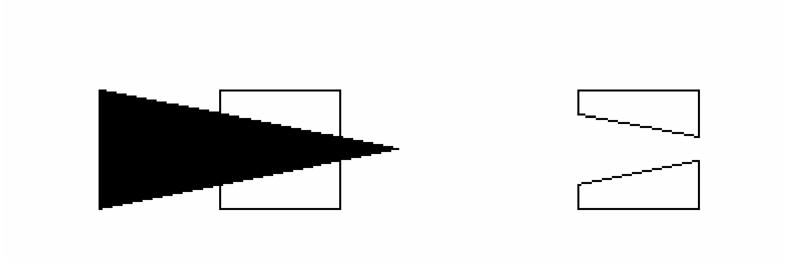


Figure 1.4 Example of a split cell

In 1993, De Zeeuw [4] wrote a computer code to solve two-dimensional Euler equations using a Cartesian grid. He used quadtree data structure and multigrid scheme to increase the convergence rate of the solution.

In 1994, Coirier [8] wrote a code to solve two-dimensional Euler and Navier Stokes equations using Cartesian grids. He used binary tree data structure and he refined and coarsened the cells according to the solution.

In 1995 and 1996 Melton, Aftosmis and Berger [9] developed techniques for handling complex surface geometries and their code CART3D solved three dimensional Euler equations using Cartesian grid accurately.

In 2004, Hunt [10] developed a code to solve the three-dimensional Euler equations by using parallel block adaptive Cartesian method. Data structure and handling the geometry were very similar to studies of Aftosmis. These references are the milestones of this study.

1.2 Scope of the Thesis

The purpose of this thesis is to develop an automatic, adaptive Cartesian grid for solving inviscid, compressible flows around simple and complex geometries. In this chapter, brief information regarding CFD and mesh generation techniques is given. Besides, past works about Cartesian method are summarized in the review of literature section. In Chapter 2, quadtree data structure and two dimensional grid generation are discussed. A number of topics like terminology for Cartesian grids, determination of neighbor cells, information about special computational cells, inside-outside testing methods, marching squares technique and adaptation types are explained as well. In Chapter 3, octree data structure and different aspects of three dimensional grid generation from two dimensional grid generation are discussed. Two and three dimensional flow solvers, including flux formulation, temporal discretization, reconstruction and multigrid method are discussed in Chapter 4.

Chapter 5 gives the results of various test cases to validate the code accuracy. Finally, in Chapter 6, summary of the present work and conclusions are presented.

CHAPTER 2

TWO DIMENSIONAL DATA STRUCTURE AND GRID GENERATION

2.1 Quadtree Data Structure

As it is mentioned in the introduction chapter, Cartesian grid is a special type of unstructured grid and for an unstructured grid, ordering information of grid points and neighboring cells is not apparent like structured grid. Therefore, connectivity (i.e. ordering) information has to be constructed since it is mandatory for flux calculations, reconstruction, multigrid method, refinement and coarsening etc. Namely, data structure is necessary to store connectivity and flow information for each cell.

For the Cartesian method, data structure is complicated since the number of cells cannot be predetermined. Hence, dynamic data structure is used. By this way, the number of cells can vary during the execution of the program.

In the literature, there are various methods used for two dimensional fluid flow problems to identify connectivity information such as two dimensional arrays, linked list, binary tree and quadtree data structures. In this work, the most appropriate method is chosen as quadtree data structure due to its advantages.

First and the foremost, the data structure conversion of the developed code from two dimensional to three dimensional grid generation is easy. In other words, the logic behind the quadtree and octree data structure is very similar; therefore, developed two dimensional grid generation code is easily converted to a three dimensional grid generation code. Furthermore, a local change in the grid such as cell refinement and coarsening and the implementation of multigrid method are very easy for the quadtree data structure due to its flexibility when compared to this change and implementation in two dimensional arrays or linked list. Since each element has a fixed index in the two dimensional arrays or has a fixed another element that follows it in the linked list, the implementation of multigrid and local changes in the grid require the generation of multiple grids [4].

Quadtree data structure can be thought as a family tree which demonstrates the relationships beginning from the oldest individual and then covering its children and grandchildren. The oldest individual of the family tree becomes the root of the quadtree data structure. Since each cell in the quadtree data structure has parent and four children, connectivity information is extracted from relationship between parent-children information. Figure 2.1 illustrates the quadtree data structure, root and children cells.

In the developed code, all cells are identified with nine pointers which are its parent, four children and four neighbors. These pointers and others stored for all cells can be seen below as:

- 1 word: Its parent
- 4 words: Its four children
- 4 words: Its four edge neighbors
- 2 words: Its *x* and *y* coordinates of the centroid
- 1 word: Its level
- 1 word: The definer for computational cells which is called “compcell” in the developed code and this will be explained in the multigrid section
- 1 word: The definer of parent cell which can be coarsened while the application of multigrid. This pointer is called “perform” in the developed code and this will also explained in the multigrid section.

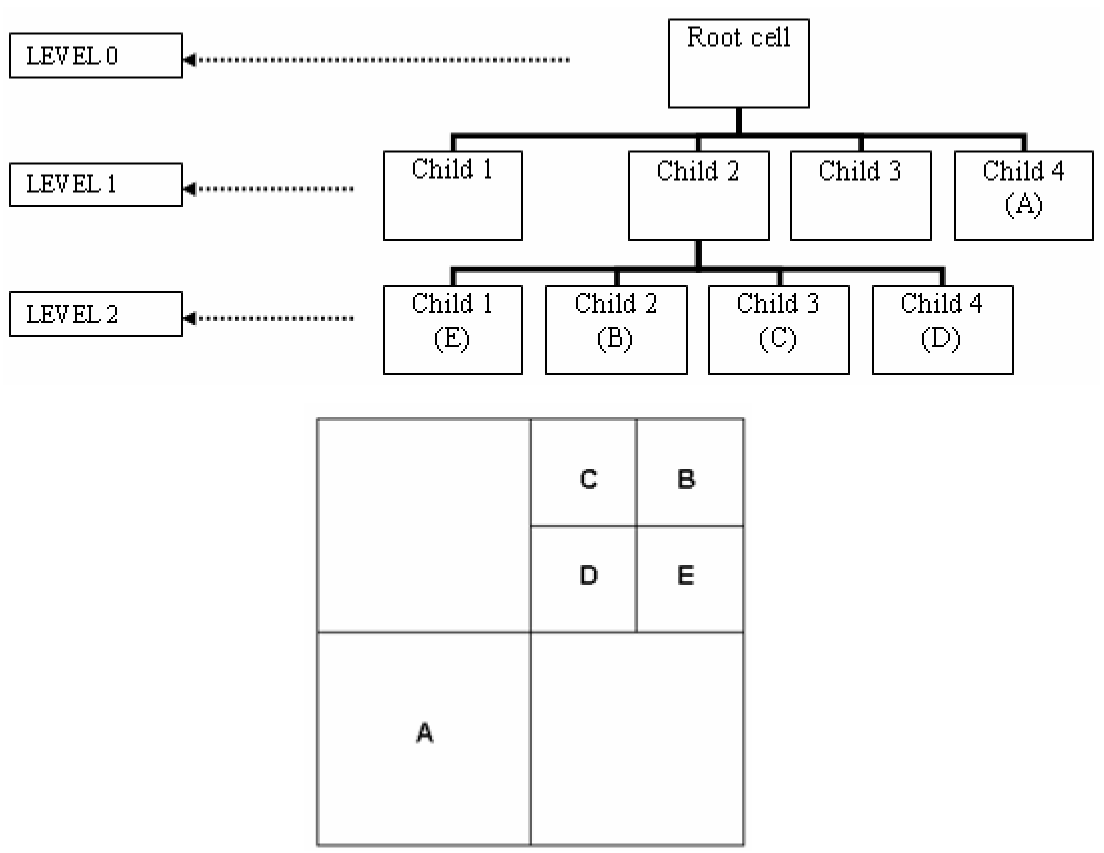


Figure 2.1 Illustration of the quadtree data structure

Pointer indicating the parent states that it is also a cell and this cell is the parent of its children. Each cell has a parent and four children whether it is assigned to another cell or to zero. The cell whose parent is assigned to zero is the root cell and the cells whose children are assigned to zero are the computational or leaf cells. Four pointers indicating children state that they are also cells and they are the children of their parent. Finally, instead of determining neighboring cells when they are needed, they are stored for each cell. If a cell has no neighbor, i.e. its neighbor is the far-field, it is set as zero. Determination of neighboring cells is given in the next chapter for three dimensional cells since the process for three-dimensions is more complicated than that for two-dimensions. Only an example figure (Figure 2.2) is given to illustrate the process for two-dimensions roughly and show the numbering of children cells.

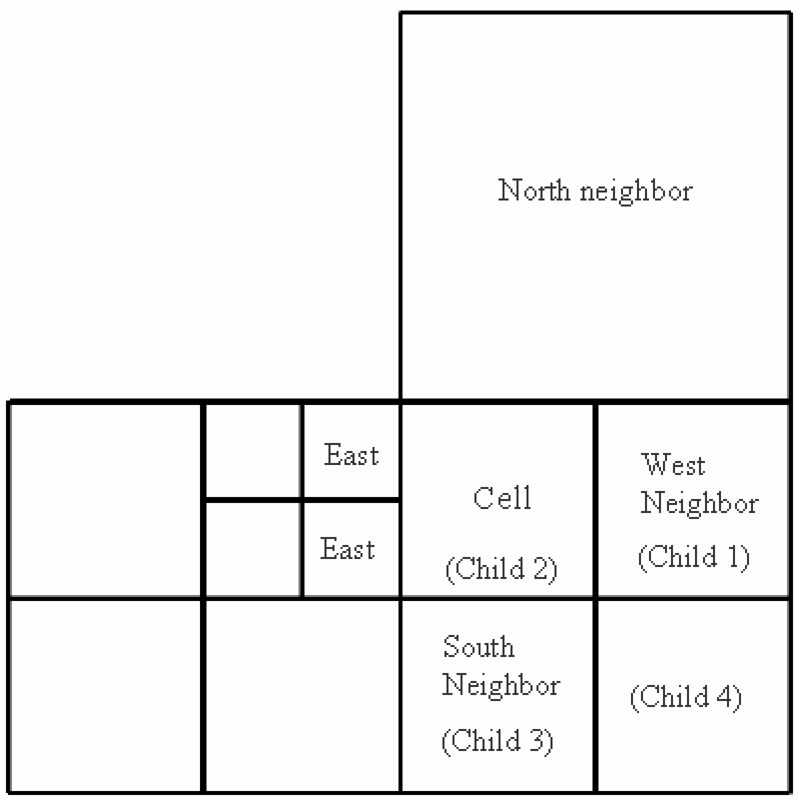


Figure 2.2 An example of neighboring cells and numbering of children cells

Moreover, coordinates of the centroid and level of each cell are important parameters for the developed code. Calculation of coordinates of the centroid of a cell is discussed in the next chapter for a three dimensional grid instead of a two dimensional one. The level of a cell is used for many reasons. Level of a cell is necessary for the calculation of coordinates of the centroid and length of an edge of a cell. Besides, coordinates of four corners are calculated with the use of level. There is a restriction called one level rule in the developed code. This rule enables grid smoothness and facilitates the flux calculation and application of reconstruction schemes. In addition, neighbor cells through the vertices of a cell can easily determined by means of this restriction. One level rule simply states that the level differences between two edge or vertex neighbors cannot exceed one. If this rule removed from the developed code, the solution accuracy would be harmed. Besides, one level rule prevents the data structure to become much more complicated.

By the way, as it is seen from Figure 2.1, the level of root cell is zero and level of its children is 1. In other words, level of a child cell is one level higher than its parent level.

There are two types of neighbors for two dimensional problems. One is edge neighbor which is stored for each cell and the other is vertex neighbor which is determined when it is required. Since vertex neighbors are not used as many times as edge neighbors and also they easily determined by the edge neighboring information, storage of them would be inefficient usage of memory.

Effective memory usage is very important for computer codes. Therefore, the programmer must balance the memory usage and computational time. In order to balance them, the necessity of stored information is explored. If the information is used repetitively in the developed code and calculation of it takes a long time, storage of this information is more logical. Coordinates of four cell corners can be given as an example to this case. On the other hand, if the information is used rarely and determination of it takes a short time, it is logical not to store this information. Determination of vertex neighbors can be given as an example to the second case.

As it is mentioned before, there are special cells which have no children cells. These are called computational or leaf cells and the all calculations are performed on these cells. They are divided into four groups according to their types. These are inside, outside, cut and split cells. In fact, inside cells cannot be thought as computational cells due to the fact that flux calculations are not performed on these cells. Therefore, in the developed code, the variables that are stored for computational cells except inside cells can be given below as:

- 4 words: Conservatives variables for continuity, *x*-momentum, *y*-momentum and energy equations
- 4 words: Its four corners
- 1 word: Its type

- 1 word: Its total square index. This will be discussed in the Marching Squares Algorithm section.
- 2 words: Its refinement or coarsening criteria
- 2 words: Divergence and curl of velocity vector
- 2 words: Lengths of edge projections in x and y directions
- 4 words: For forcing function which will be discussed in multigrid section

Each of the four pointers for corner structures defines one corner of a cell and this corner structure stores the x and y coordinates of the cell corner and a variable ϕ. This ϕ variable of one corner is used to determine whether this corner is inside or outside the given geometry. Detailed information regarding ϕ value and determination of cell type will be given in the next section. Numbering of corners can be seen in Figure 2.3.

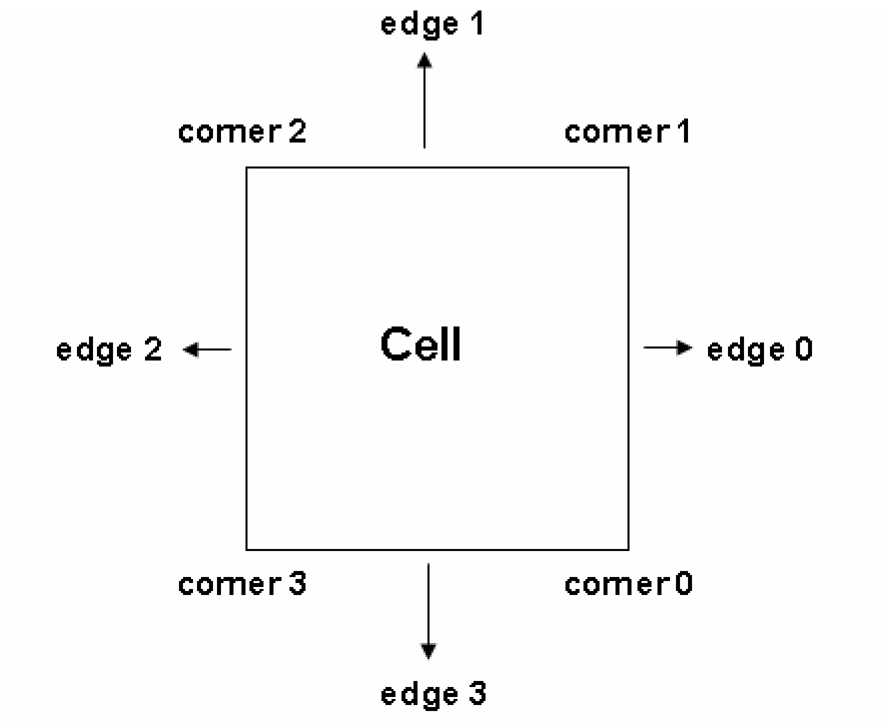


Figure 2.3 Identification of corners

Since the centroid and area of a cut or split cell is not directly calculated like outside cells, these types of computational cells require additional information to be stored. Calculation of centroid and area of these special cells are discussed later. In the developed code, the variables stored for cut cells can be seen below as:

- 3 words: *x* and *y* coordinates of centroid and area of a cell
- 4 words: *x* and *y* coordinates of cut locations (px[0], px[1], py[0] and py[1])

Some of the stored variables for split cells are twice as much as those for cut or outside cells because some of the split cells are composed of two separate computational control volumes. Flux, divergence and curl calculations are performed twice for a split cell since it has two separate control volumes. These doubled variables are conservative variables, divergence and curl of velocity vectors, edge projections, centroids, areas and cut point locations. On the other hand, split cells have single refinement criterion like outside and cut cells.

A demonstration of numbering of the intersection points of cell edges and the geometry for a cut cell is seen in Figure 2.4.

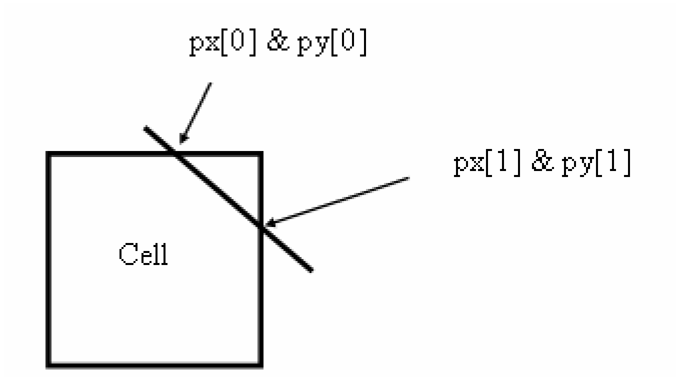


Figure 2.4 Demonstration of cut locations

Finally, calculation of centroid and area of cut cells can be summarized in this section because cell centroid and area are used in the formulation of least square reconstruction scheme and the calculation of the divergence and the curl of the velocity vector. It is clearly seen that the numbering of corners and also cut locations are given in the counter-clockwise sense. This facilitates the calculation procedure.

Firstly, outside part of cut cells are triangulated beginning from the first cut point (px[0] and py[0]), traversing the outside corners in the counter-clockwise sense and ending with the second cut point (px[1] and py[1]). Then the cross-product of vectors along faces of each of the triangles is evaluated. This product gives the area of each triangle. Summation of areas of all triangles which constitutes the outside area of cut cells is calculated. As a result, the area of the cut cell is obtained. The procedure is the same for split cells. The centroid of cut cell is calculated by using the following equation:

$$C = \frac{\sum_{i=1}^{nTriangles} (\mathrm{A}C)_i}{\sum_{i=1}^{nTriangles} \mathrm{A}_\mathrm{i}} \tag{2.1}$$

where C_i and A_i in the formula refer to the centroid and area of each triangle.

An example of triangulation process is given in Figure 2.5. Outside corners of any cut cell are known automatically by means of a formed table. This table is called corner-table and given in the next section.

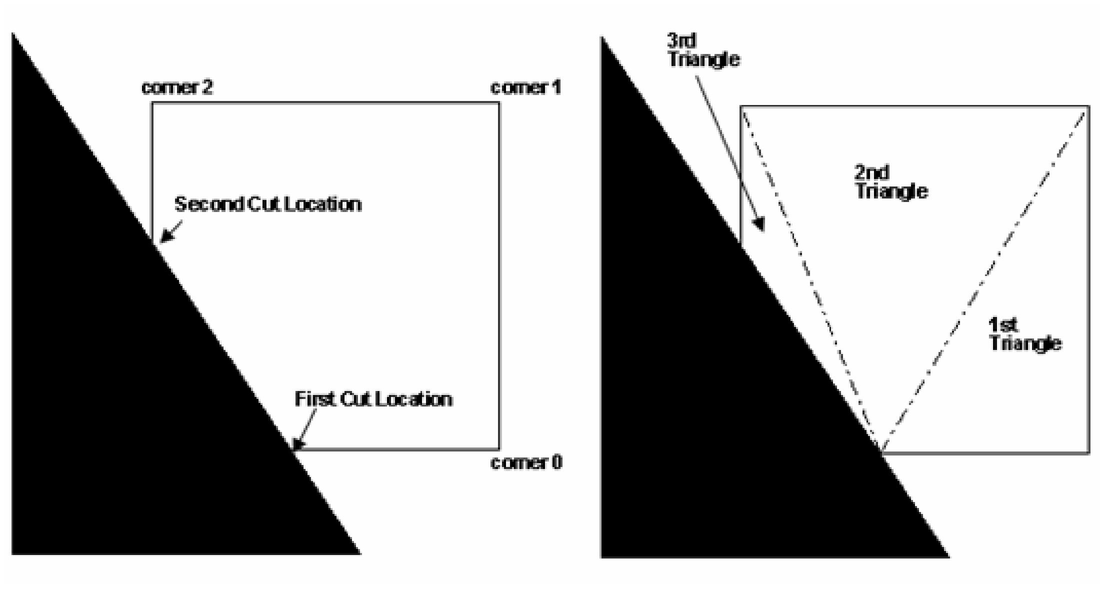


Figure 2.5 An example of triangulation process

2.2 Initial Grid Generation and Geometry Adaptation

Grid generation for two-dimensional Cartesian grids can be achieved in three steps. Initial step is the creation of the domain and uniform mesh generation. The determination of cell types by inside-outside test is the second step. Afterwards, geometric adaptation, which consists of three parts: box, cut & split cell and curvature adaptation, is applied to the uniform mesh. By the way, there are additional important intermediate steps between box adaptation and cut & split cell adaptation. These are marching squares method and the determination and classification of the split cells.

2.2.1 Creating the Domain and Uniform Mesh Generation

Initially, the geometry around which the external flow is solved is specified to the developed code as line segments. It is important that these line segments have to follow a sequence starting from a point and ending up with this point. In other words, line segments have to form a closed loop whose rotation direction is counter-clockwise.

Afterwards, since the geometry is not nondimensionalized to prevent computation errors due to the division such as machine zero effect, the maximum length of the given geometry is important to create the appropriate domain for solving the problem accurately and robustly. The multiplication of the maximum length by the size factor gives the domain size. This domain size is the length of the edges of the root cell. Then since the developed solver is external flow solver, the given geometry is placed to either the mid point of the created domain or the desired location of the user near the center of root.

Finally, uniform mesh for the two dimensional Cartesian method is obtained by dividing squares successively starting from the root until the level of computational cells reaches the desired level. This step is very important to obtain sufficiently small cells before geometry and solution adaptation steps. The given geometries (three-element airfoil and NACA0012) and uniform meshes around them are given in Figure 2.6 and 2.7.

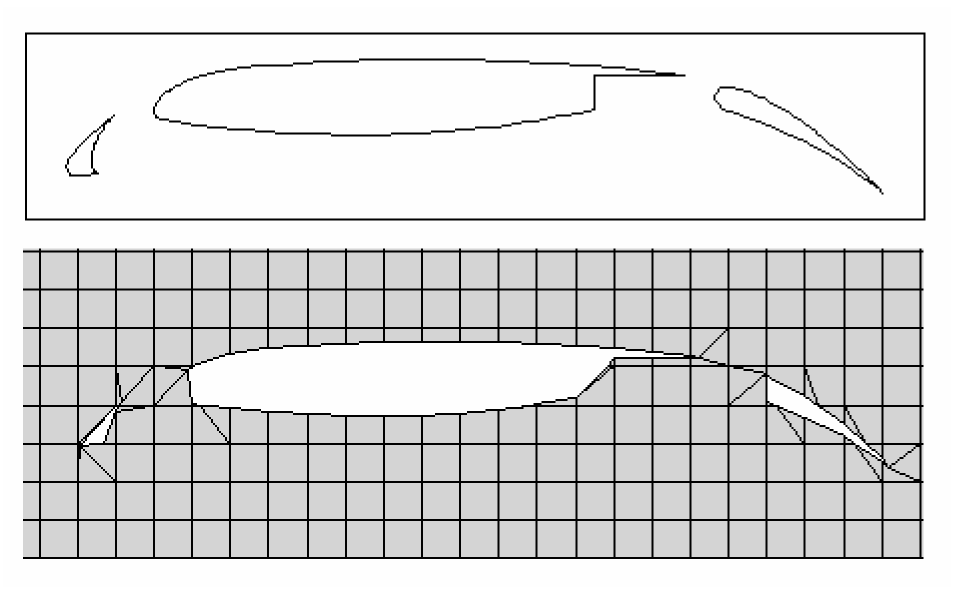


Figure 2.6 Three-element airfoil and uniform mesh around it

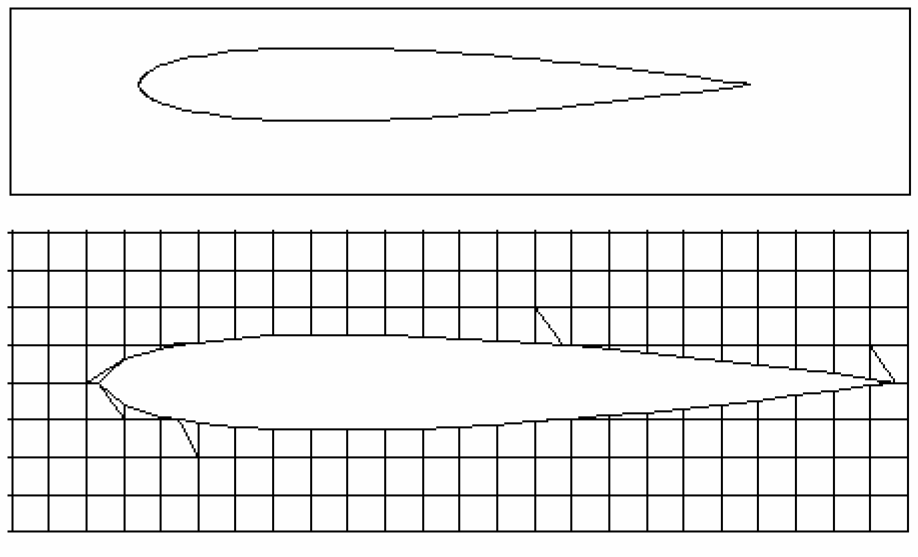


Figure 2.7 NACA0012 airfoil and uniform mesh around it

When sufficient resolution around the geometry is not obtained during the uniform mesh generation, sometimes small parts of multi-component geometries are not realized; therefore, adaptation steps do not notice these parts due to the insufficient resolution. As a result, incorrect mesh generations are obtained. In order to prevent this failure, some determination techniques are implemented into the developed code. But this does not mean that these implementations prevent all of these failures. In other words, the user of the program is always aware of the possibilities of these errors. For example, before these techniques, slat of the three-element airfoil in Figure 2.6 was not detected without sufficient resolution, i.e. with uniform mesh which has low desired level. But, it is possible now.

It is beneficial to indicate that there is no need to check the one level rule in the uniform mesh step. After creation of new cells with the division of their parent cells, centroidal coordinates and their neighbors have to be set. These procedures are discussed for three dimensional problems instead of two dimensional ones due to the complexity.

The increase of total number of cells during the uniform mesh is exponential. Therefore, it is stated in references [11] and [12] that desired level may be kept within the limit of two or three to avoid this increase. However, in the developed code, there is no restriction on the number of desired level for uniform mesh because coarsening of cells is applicable in this code. Therefore, although exponential increase occurs at the beginning of the mesh generation, this number decreases by means of coarsening process during the solution adaptation. Besides, exponential increase does not harm the convergence rate in a detectable manner on account of multigrid method. As it was mentioned before, higher desired level is necessary in some cases where small geometry components exist. Moreover, solution refinement is sometimes much more efficient with higher initial resolution. This can be exemplified in Figures 2.8 and 2.9 by comparison of pressure coefficients with AGARD data [13].

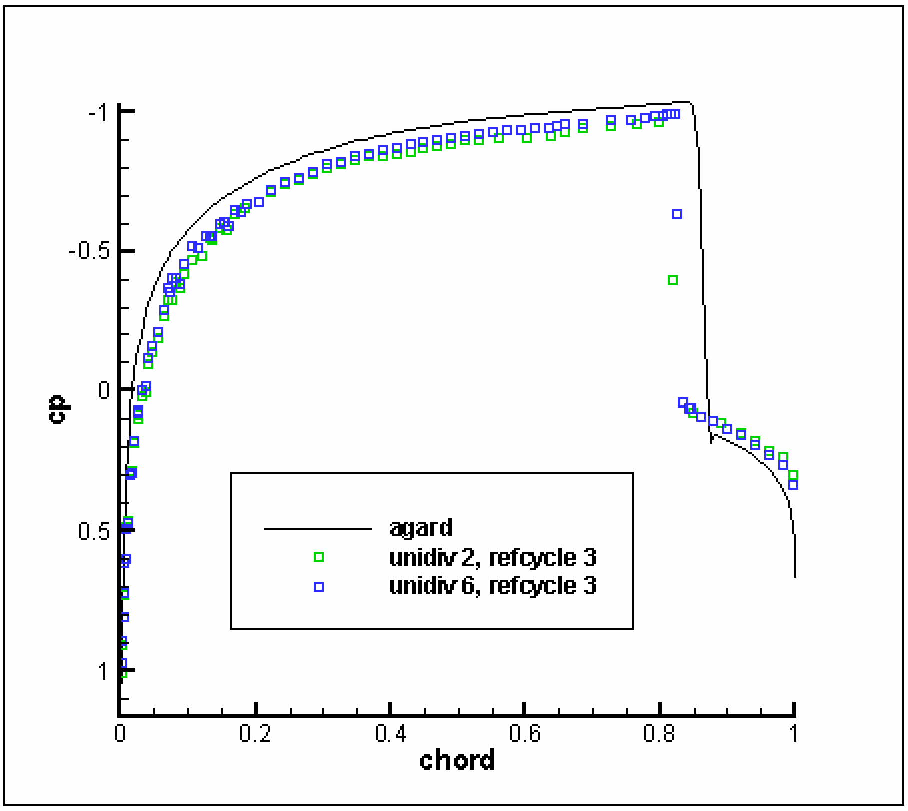


Figure 2.8 Pressure coefficients of upper part of NACA0012 airfoil

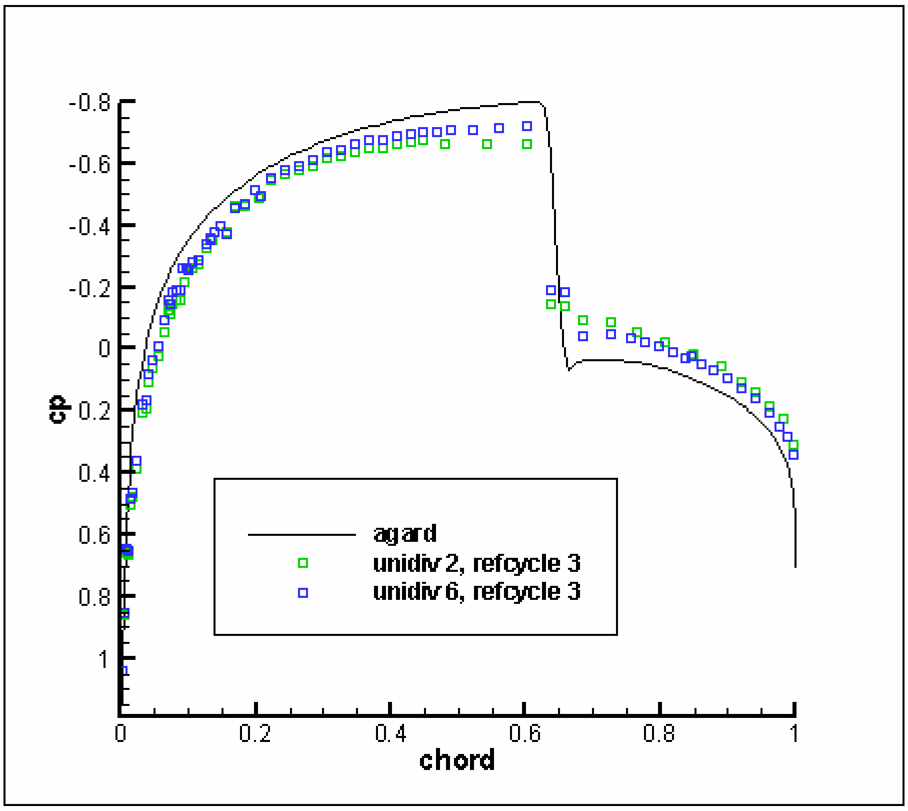


Figure 2.9 Pressure coefficients of lower part of NACA0012 airfoil

For example, green points indicate a mesh whose uniform division number (unidiv) is 2 and solution refinement cycle (refcycle) is 3 in Figures 2.8 and 2.9. When the pressure coefficients of two cases with uniform division 2 and 6, which have the same number of solution refinement cycles, are compared, it is clearly seen that locations of the shock on the upper and lower parts of NACA0012 airfoil are determined more accurately for the case with higher desired level. Besides, pressure coefficients of this case are nearer to the AGARD data. This validates the statement that initial resolution is important for accurate results. The reason of inaccurate results of the case whose uniform division is 2 is that the interaction between far-field boundary conditions and geometry is really effective due to the huge cells near the far-field. Although these huge cells are refined during the application of solution refinement, they harm the result because the solutions of initial iterations are without solution adaptation. Figure 2.10 shows two grids. First one is the mesh obtained after the geometric adaptation. The uniform division number of this case is 2. Second one is the mesh which is obtained after the solution refinement. This mesh is obtained after three refinement cycle.

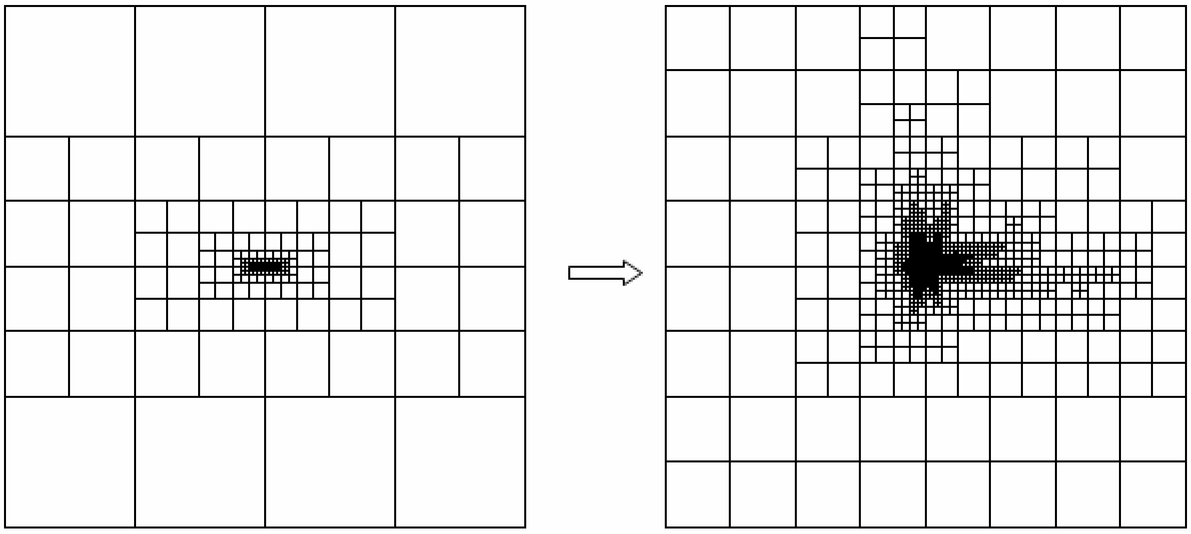


Figure 2.10 Geometric- adapted and solution-adapted meshes, respectively

2.2.2 Inside-Outside Determination

After uniform mesh generation, corners of sufficiently small cells are tested whether they are inside of the given geometry or not. This is called inside-outside testing. This step is obligatory for Cartesian grids because the cells that are cut by the given geometry are determined by this test.

There are various methods for the determination of a point whether it is located at the inside or outside part of a closed polygon. The most popular ones are Ray Casting and Winding Number methods.

In this thesis, Ray-Casting method is used due to its numerous advantages over Winding Number method. However, there is a restriction in order to apply both of these methods. This restriction is that the given geometry must be a closed loop although it consists of holes in it. Information regarding Winding Number method can be found in references [10] and [12].

2.2.2.1 Ray Casting Method

In this approach, a ray (line) which can be *x* = constant, *y* = constant or another line is cast from a point in two dimensional problems and the number of interactions between this ray and the line segments of the geometry is counted. If this number is odd then the point is inside the geometry, else it lies outside the geometry.

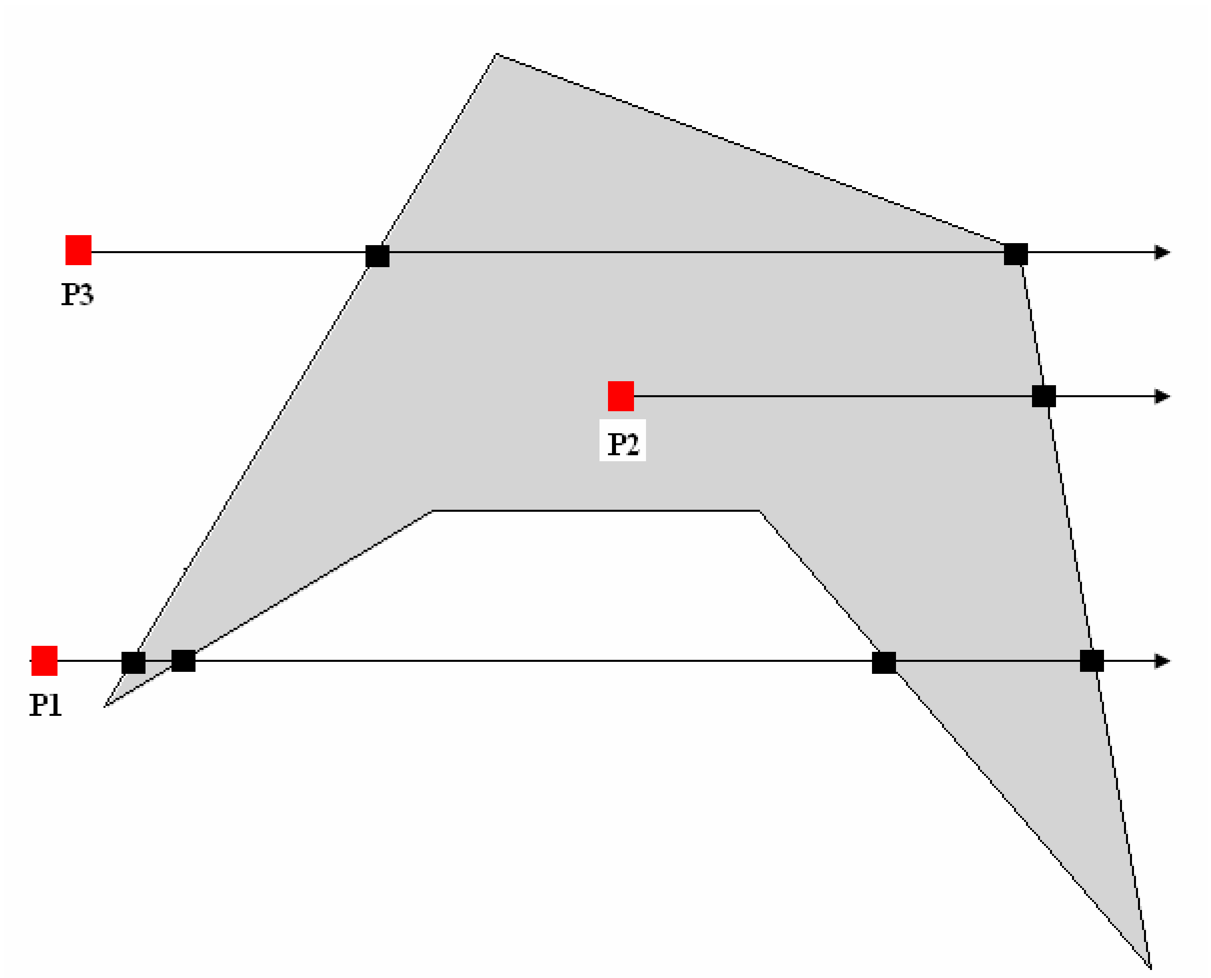


Figure 2.11 An example for Ray Casting Method

For example, three different cases are shown in Figure 2.11. In the first case, a ray is cast from point P1 in the positive *x*-direction (*y* = constant line) and it intersects four times with the simple-closed geometry. This means according to ray casting method that this point lies outside the geometry. In the second case, ray of point P2 in the positive *x*-direction intersects with the geometry once; therefore, it is inside the geometry. These two cases do not violate the rule of ray casting method. However, a ray in the positive *x*-direction which is cast from point P3 intersects with the geometry three times. In other words, three line segments of the geometry are cut by this ray. Therefore, ray casting method allocates point P3 as an inside point. The reason of this problem is that the end points of line segments have to be counted once. But instead of counting end points once, changing the direction of ray is much more logical when it intersects with an end point of any line segment. As a result, probable mistakes due to this intersection are eliminated. That is to say, when the ray, which is cast from point P3, changes to a *x* = constant line (ray in the positive *y* direction), the number of intersection becomes zero and this means point lies outside the geometry. Consequently, in the developed program, rays are cast along alternating directions until the ray does not intersect with endpoints of line segments. There is one more special case where the rule of ray casting method is violated. If the point is coincident with any line segment of the geometry, the number of intersection can be either odd or even although it must be odd. Therefore, firstly, the coincidence must be checked and if there is no coincidence, the testing method is applied to the point. If it is coincident with any one of the line segments, the point is on the geometry and it is immediately allocated as an inside point.

As it is mentioned before, ray casting method has various advantages when it is compared to winding number method. The first advantage is that unlike winding number method, ray casting method does not require to visit all of the line segments of the geometry. For example, if the ray is in the positive *x* direction, the line segments whose *y* coordinates of both end points are larger or smaller than the *y* coordinate of the point from which the ray is cast are not tested since their intersection is impossible. Another advantage is that ray casting method does not suffer from floating point round off errors [10]. The final advantage is that implementation of ray casting method into the three dimensional problems is easier and more accurate than that of other methods.

After the testing of each of four corners of a cell, type of this cell has to be determined according to the test results since this step is obligatory step for Cartesian grids. Then, ϕ value of each corner which was mentioned before is allocated. ϕ value of inside corner is set to -1, while ϕ value of outside corner is set to 1. This value is necessary in the marching squares method which is discussed later. As it is mentioned before, there are four types of computational cells. If all corners of a cell are outside of the geometry, the cell type is set as outside and if all corners are inside then the cell type is set as inside. Types of other remaining cells are set as cut cells. By this way, cut cells are determined. Figure 2.12 gives examples of these types of cells. However, there are some exceptions which violate rules of this determination. Two cases given in Figure 2.13 illustrate some of these exceptions. Although four corners of two sample cells in Figure 2.13 are outside, they are cut by the given geometry. These cells are set as split cells because classifying these cells as outside cells causes error during the flux calculations and they are discussed later in the next section.

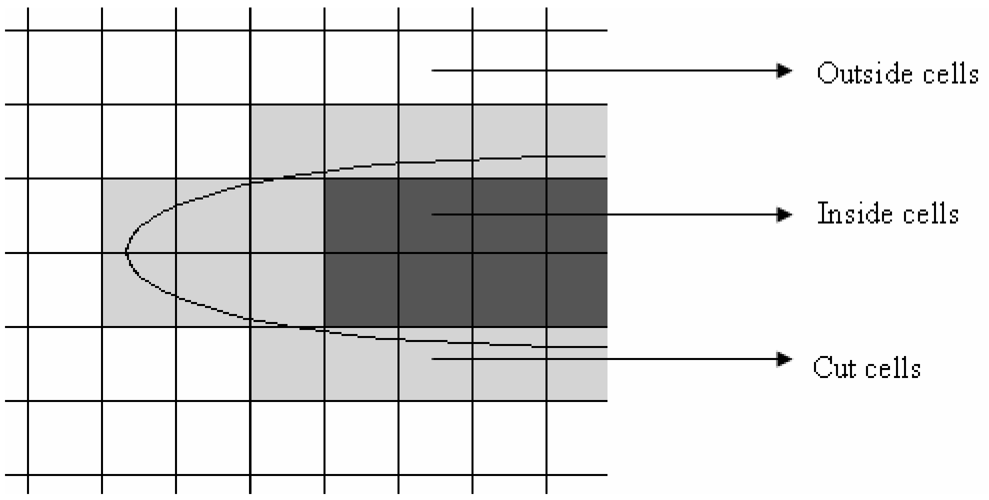


Figure 2.12 Examples of types of cells

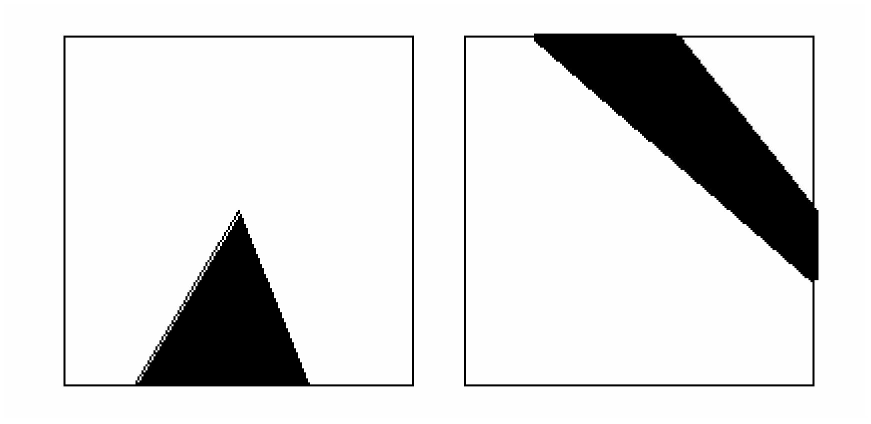


Figure 2.13 Sample two exceptions

After determining types of cells, cells around the body except inside cells are refined by three different ways until fine meshes are obtained around the body. The general name of these processes is geometric adaptation.

2.2.3 Geometric Adaptation

The adaptation process can easily be applied to a Cartesian grid; therefore, high resolution around the geometry as a result of adaptation enables more accurate results when compared to other types of grids. Geometric adaptation is applied to the uniform mesh after determination of cell types in three steps: box adaptation, cut & split cell adaptation and curvature adaptation. The amount of application of these adaptations is determined by the user. Suitable grid for solver part is obtained after all of these adaptations.

2.2.3.1 Box Adaptation

The first step of geometric adaptation is the box adaptation. By means of this adaptation, uniform mesh around the given geometry is refined and fine meshes are obtained in an imaginary rectangular box. The imaginary rectangular box includes cells which are inside the box or in contact with the box and these cells are flagged for refinement. After determining of these flagged cells, they are refined until a desired resolution is obtained inside the box. But it is important that one level rule is considered while the refinement of flagged cells. If the levels of the edge or vertex neighbors of flagged cell are one level lower than its level, firstly they are refined and then the cell is refined. After the application of box adaptation to the uniform mesh in Figure 2.6, Figure 2.14 is obtained.

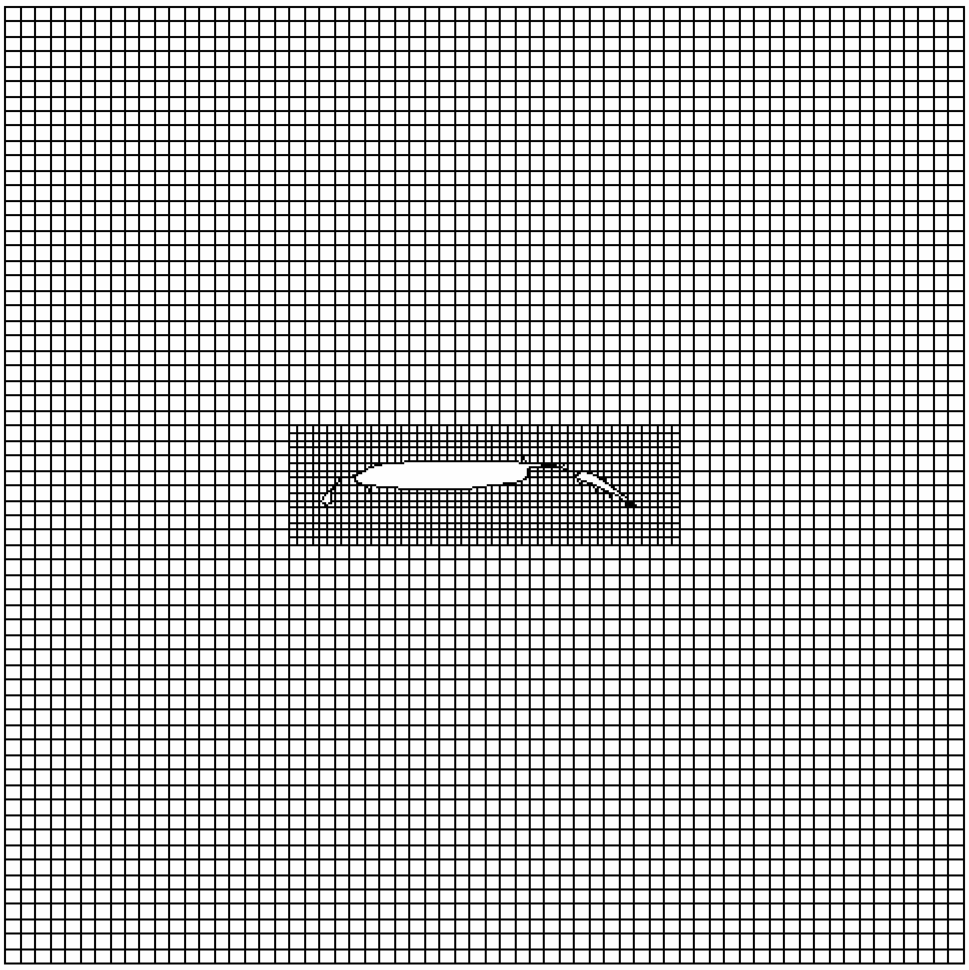


Figure 2.14 Application of box adaptation to three-element airfoil

As it was mentioned before, there are some intermediate steps between box adaptation and cut cell adaptation. These are discussed now.

2.2.3.1.1 Determination and Classification of Split Cells

After the types of cells are determined according to the rule given in the inside-outside testing section, it is seen that there are some exceptions and these exceptions necessitate some modifications in type determination. Therefore, additional type is added to the developed code called split cell. These cells have one or more separate control volumes. Flux calculation of split cells which have only one control volume is the same as cut cells. On the other hand, flux calculation of split cells which have two or more control volumes is different. Separate flux calculations are performed for each of separate control volumes of a split cell. Although split cells make the data structure more complicated, it is obligatory for the implementation of multigrid method. This will be exemplified with an example in the multigrid method section. Besides its complexity, split cells increase the computational time and decrease the usage of memory effectively. In the literature, some works assume split cells as one control volume in order to escape from more complexity in the data structure like references [4] and [12].

In this work, split cells are handled to ease the implementation of multigrid method. As it is mentioned before, classification of cells according to their types are performed after inside-outside testing. In this step, cells are divided into three groups: inside, outside and cut cells. Afterwards, all cells are tested again to determine how many cut points each of cells has. This test is called the determination of split cells. In fact, outside and inside cells should not have cut points and cut cells should have only two cut points. However, some cells violate this rule. For example, although types of two cells in Figure 2.13 are set as outside after inside-outside testing, they have two and four cut points, respectively. In addition, first two cells in Figure 2.15 are set as cut cell but they have four cut points instead of two. The last cell in Figure 2.15 is set as inside, but it is cut by the geometry.

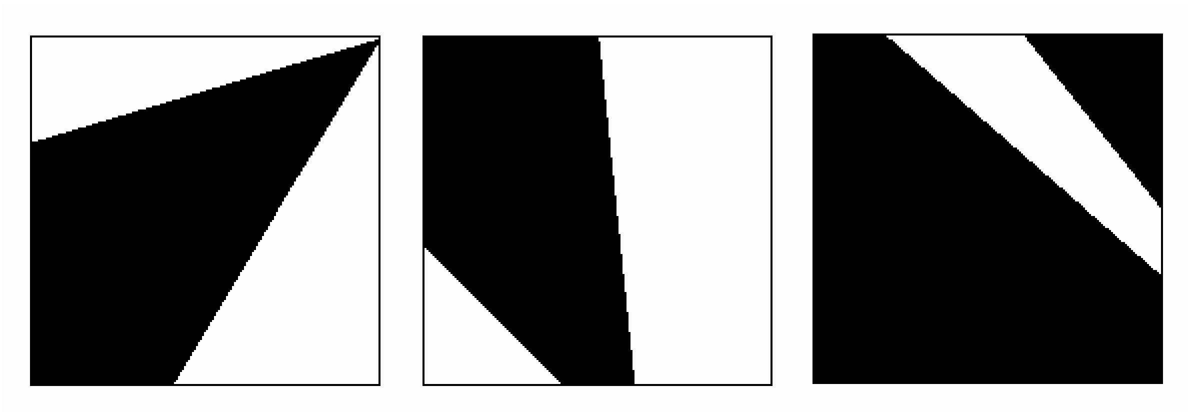


Figure 2.15 Three examples of split cells

As it is mentioned before, there is a pointer called total square index for all computational cells. This pointer makes easier the flux calculation process. Finding the value of this pointer for outside, inside and cut cells is discussed in the next section.

After all of computational cells are tested to find exceptions and set them as split cells, classification of these split cells are done. Split cells are classified in respect of their total square index values. Therefore, finding these values for split cells can be summarized here. If a cell is cut cell according to the inside-outside testing but it has four cut points, it is set as split cell and its total square index remains the same. If a cell is outside cell according to the test but it has two or four cut points, it is set as split cell and its total square index is assigned to -15 or -25, respectively. If a cell is inside cell according to the test but it has two or four cut points, it is set as a split cell and its total square index is assigned to -20 or -30, respectively. Finally, if a cell is cut cell according to the inside-outside test but it has more than four cut points, it is set as split cell and its total square index is assigned to -40. Split cells whose total square index is -40 are recursively refined until other types of split cells are obtained since the flux formulation for this case is very difficult to handle when compared to other types. This type of split cell is rarely found and after one or two refinement they are eliminated.

After classification of split cells, their cut point locations have to be determined. But it is important to note that numbering of cut points is not done randomly. Cut points for each cell follow a sequence. Hence, calculation of cut or split cell area, centroid and calculation of fluxes are very easy because of this sequence. Furthermore, each control volume of split cells can be considered as a cut cell and its flux calculations are performed like cut cells by means of total square indexes indicated in Appendix A. These indicated total square indexes are different from the actual classification total square index. For example, total square index of the cell given in Figure 2.16 is -25 in order to classify it. It is an outside cell as to inside-outside testing. After determination test, it is set as a split cell because it has four cut points. First control volume of split cells is the part which possesses cut locations p0 and p1. Second control volume is the part which has cut points p2 and p3. Flux calculation of the first control volume in Figure 2.16 is done assuming this part as a cut cell whose total square index is three. As seen in Figure 2.16, if the first control volume is assumed a cut cell, its total square index is directly calculated as three. Flux calculation of the second control volume is done assuming this part as a cut cell whose total square index is twelve. As seen in Figure 2.16, if the second control volume is assumed a cut cell, its total square index is directly calculated as twelve. In other words, total square index of split cell is -25 in order to classify it and total square indexes of each control volume of this split cell can be seen from the Figure 2.16 inside of the control volumes.

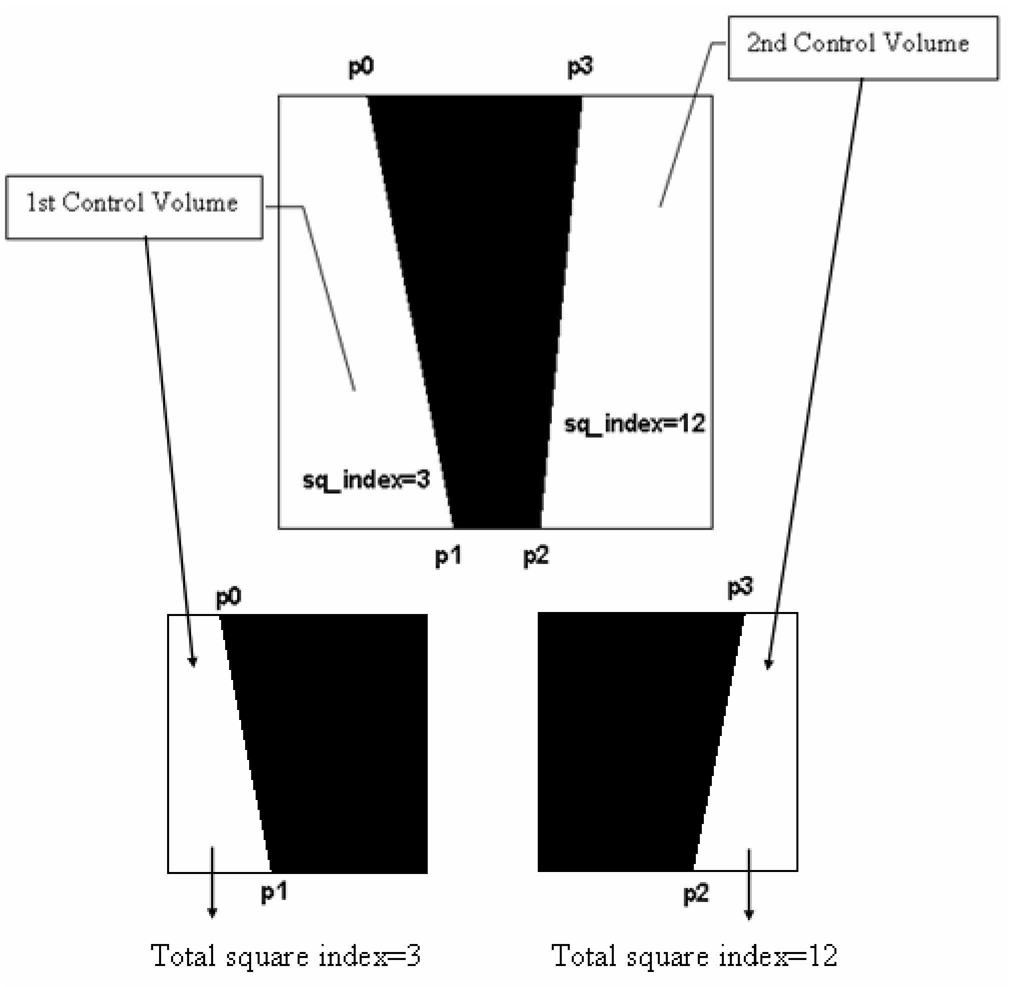


Figure 2.16 Numbering of cut points of a split cell and assuming it like a cut cell for flux calculations

2.2.3.1.2 Marching Squares Method

After box adaptation and determination of split cells, locations of cut points of cut cells are tried to be found by using marching squares algorithm for two-dimensional problems since its implementation is easy. Besides, memory usage is lower than other methods. Generally, in the literature, line or polygon clipping algorithms were used for two-dimensional flow problems to determine the cut locations and the part of cell that resides in the geometry. These algorithms are performed by testing each of four edges whether they are cut or uncut. Then as a result of clipping, cut and uncut edges and cut locations are stored separately and portion of cut edges that resides outside of the geometry is stored in the memory. This brings about excessive memory usage. For example, minimum additional eight pointers have to be stored for these values. On the other hand, in marching squares algorithm, cut edges are automatically known by a table; hence, cut locations are found and stored easily. In other words, there is no need to check all edges whether they are cut by the geometry. This is not a big problem for two dimensional problems but for three dimensional problems, there are twelve edges and testing of each of them will be time consuming. In this work, by means of one pointer, cut and uncut edges and portion of cut edges that resides outside of the geometry are found easily. Finally, marching squares algorithm is used in three-dimensional problems in order to find the area of cut surfaces and portions of those surfaces that are used for flux calculations.

Marching squares algorithm starts by indexing each corners and edges of a cell from 0 to 3. This can be seen from Figure 2.3. Then total square index value depending on number of corners which have negative ϕ value is determined. Square indexes of each corners according to their phi values and the loop used in the developed code in order to calculate total square index are given below.

- If ϕ of corner 0 = -1 then square index = 1
- If ϕ of corner 1 = -1 then square index = 2
- If ϕ of corner 2 = -1 then square index = 4
- If ϕ of corner 3 = -1 then square index = 8

```
if(cell->type==outside)
    cell->square_index=0;
else if(*neu->type==inside)
    cell->square_index=15;
else if(*neu->type==cutcell)
{   for(int i=0; i<4; i++)
    {   if(cell->corner[i]->phi==-1)
            cell->square_index=cell->square_index+pow(2,i);
    }
}
```

Figure 2.17 Loop for calculating total square index value of a computational cell

For example in Figure 2.18, corner 0 and corner 3 of the cell are inside the body. Hence, if total square index of this cell is calculated with the given algorithm, the result becomes 9. This value is obtained by adding 1 and 8, since ϕ values of corner 0 and 3 are both -1, respectively. Then by means of a table called line table given in Figure 2.19, cut edges are determined. For example, cut edges of the cell given in Figure 2.18 are edge 0 and edge 2 respectively according to the line table in Figure 2.19 since its square index is nine. By the way, green numbers in Figure 2.19 are comments and they indicate the total square index. Since the cut edges are known automatically by the table, intersection points of cut edges and line segments of the geometry are calculated. For this case, x coordinates of intersection points are known automatically since they are equal to x coordinates of corner 0 and corner 2, respectively. The only problem is to determine y coordinates of intersections. They can be found easily by using the line segment which cuts the cell. By the way, calculating the locations of cut points of split cells is very similar to cut cells.

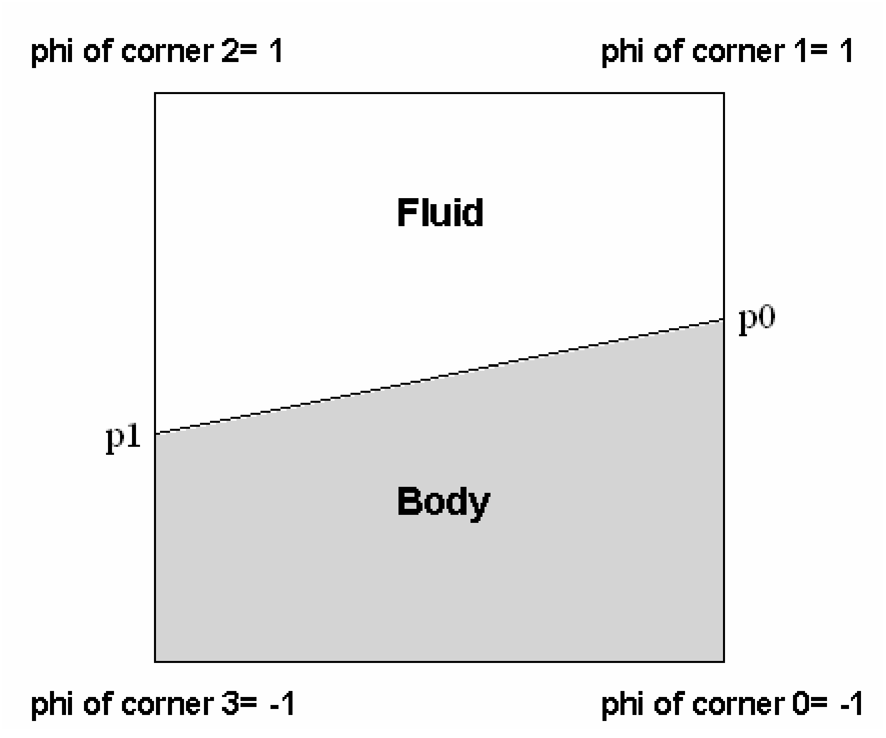
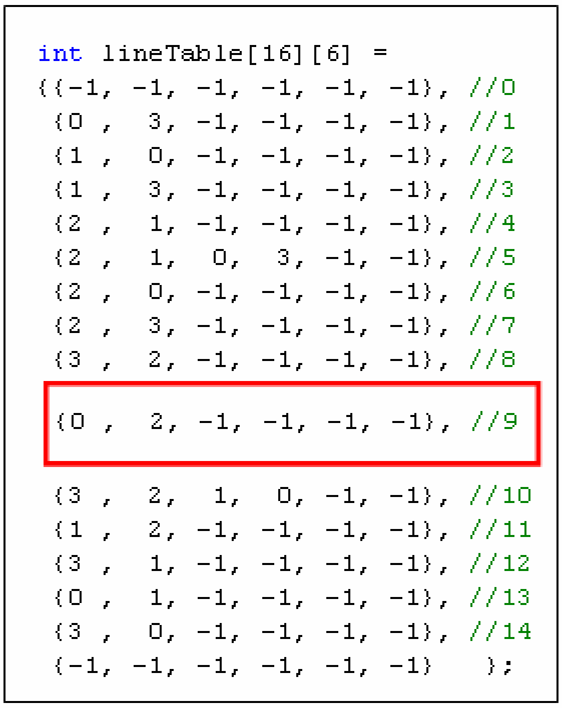
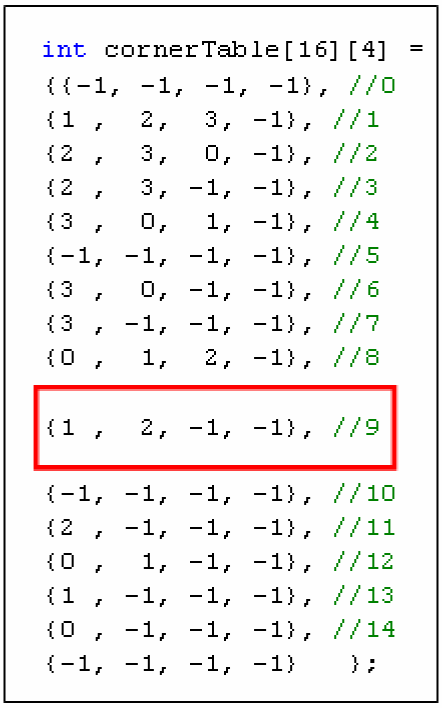


Figure 2.18 An example for explaining the calculation of total square index of a cell

Moreover, flux calculations of cut cells with the aid of this algorithm can be summarized here because it is mentioned that cut, uncut edges and outer portions of cut cells are not stored in the developed code. Instead, the table called corner table in Figure 2.19 includes almost the whole data that is necessary for flux calculations. Cut, uncut edges and outside portions of cut edges for the case in Figure 2.18 will be explained by means of the corner table in Figure 2.19. Since total square index of this case is nine, the corresponding row to this square index is 1, 2, -1, -1. These numbers (from 0 to 3) point out the number of outside corners of a cell. First and second outside corners for this case are corners 1 and 2, respectively. It is extremely important to note that rotation direction beginning from first cut point (p0) by passing the outside corners and ending with second cut point (p1) is always counter-clockwise. The first cut edge is the edge which possesses first cut point (p0) and first outside corner (corner 1) according to corner table. Namely, the first cut edge is edge 0 which is the edge located before corner 1 in the counter-clockwise sense. Instead of storing the neighbor for flux calculation, after the determination of first cut edge, it is automatically known that the first flux calculation for this case is done between the cell and east neighbor of this cell. Then it is time to determine other neighbors for flux calculations by means of this table. After the first cut edge, there are two possibilities. Second edge can be an uncut edge or cut edge. If the subsequent number is -1 in the corner table, this means that the next edge is a cut edge. If the subsequent number is a number from 0 to 3, this means that the next edge is an uncut edge. For this case, subsequent number is 2; therefore, next edge is an uncut edge which possesses corner 1 (the first index of row) and corner 2 (subsequent number). Namely, the second edge for flux calculation is edge 1 which is the edge located before corner 2 in the counter-clockwise sense. Neighbor of this cell on edge 1 is the north neighbor of the cell. After the determination of second neighbor for flux calculation, third neighbor is tried to find by means of corner 2. The number coming after corner 2 is -1; hence, the third edge is a cut edge which possesses corner 2 and the second cut point (p1) according to corner table. Namely, the third cut edge is edge 2, which is the edge located after corner 2 in the counter-clockwise sense and of course, this edge is the last edge for flux calculation since the next number is -1. The same procedure is followed for other cut cells whose square index is different from 9.



```
int lineTable[16][6] =
{{-1, -1, -1, -1, -1, -1}, //0
 {0 ,  3, -1, -1, -1, -1}, //1
 {1 ,  0, -1, -1, -1, -1}, //2
 {1 ,  3, -1, -1, -1, -1}, //3
 {2 ,  1, -1, -1, -1, -1}, //4
 {2 ,  1,  0,  3, -1, -1}, //5
 {2 ,  0, -1, -1, -1, -1}, //6
 {2 ,  3, -1, -1, -1, -1}, //7
 {3 ,  2, -1, -1, -1, -1}, //8
 {0 ,  2, -1, -1, -1, -1}, //9
 {3 ,  2,  1,  0, -1, -1}, //10
 {1 ,  2, -1, -1, -1, -1}, //11
 {3 ,  1, -1, -1, -1, -1}, //12
 {0 ,  1, -1, -1, -1, -1}, //13
 {3 ,  0, -1, -1, -1, -1}, //14
 {-1, -1, -1, -1, -1, -1}   };
```



```
int cornerTable[16][4] =
{{-1, -1, -1, -1}, //0
 {1 ,  2,  3, -1}, //1
 {2 ,  3,  0, -1}, //2
 {2 ,  3, -1, -1}, //3
 {3 ,  0,  1, -1}, //4
 {-1, -1, -1, -1}, //5
 {3 ,  0, -1, -1}, //6
 {3 , -1, -1, -1}, //7
 {0 ,  1,  2, -1}, //8
 {1 ,  2, -1, -1}, //9
 {-1, -1, -1, -1}, //10
 {2 , -1, -1, -1}, //11
 {0 ,  1, -1, -1}, //12
 {1 , -1, -1, -1}, //13
 {0 , -1, -1, -1}, //14
 {-1, -1, -1, -1}   };
```

Figure 2.19 The tables for marching squares algorithm

2.2.3.2 Cut and Split Cell Adaptation

In order to obtain higher resolution around the geometry, split and cut cells are flagged for refinement. The difference between box adaptation and cut and split cell adaptation is that neighbors of cells that are flagged for refinement are also flagged for refinement in order to obtain smooth grid around the geometry. As a result, transition between cells near the geometry and cells far from the geometry becomes smooth and degradation of solution due to level differences in the critical regions is prevented. After the application of cut and split cell adaptation to the box adapted-grid in Figure 2.14, Figure 2.20 is obtained.

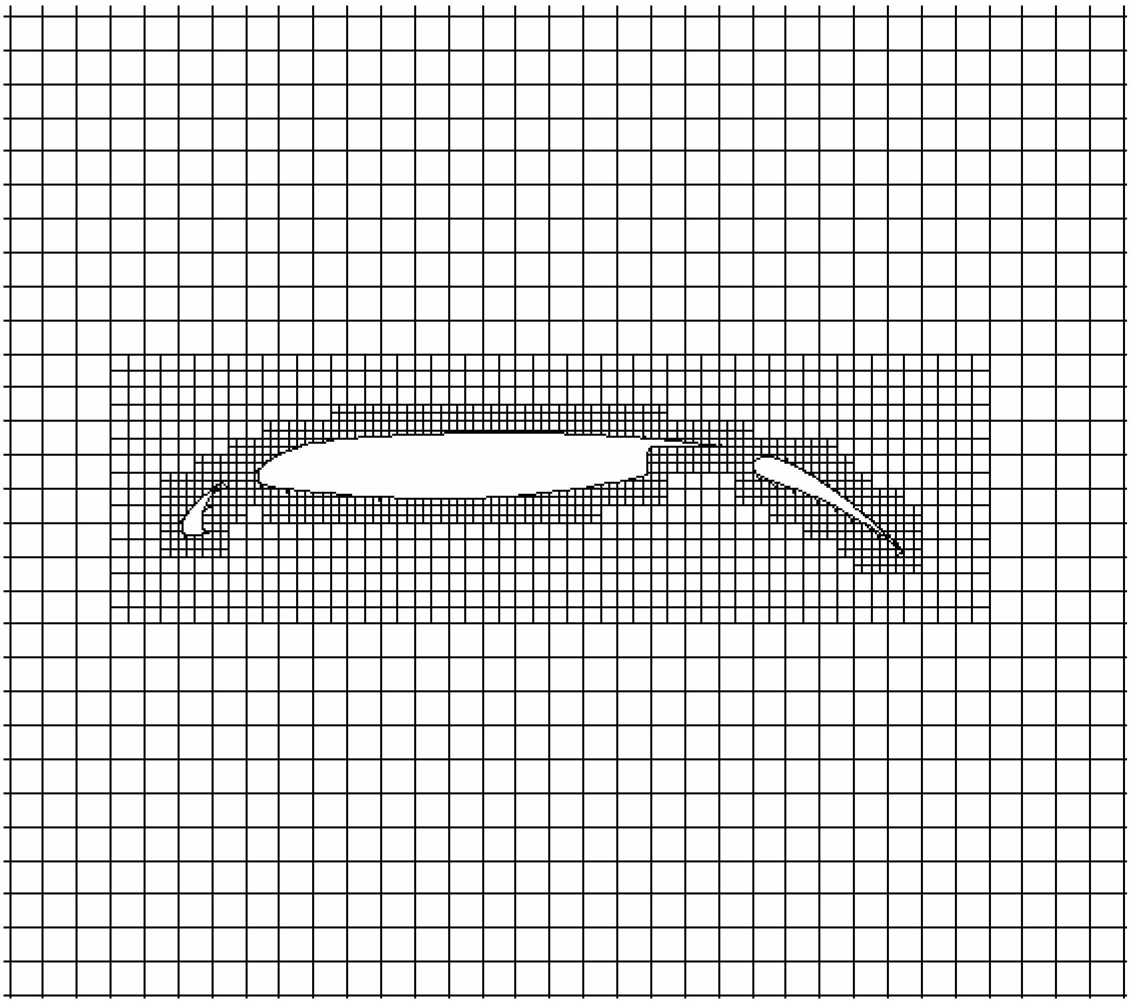


Figure 2.20 Application of cut and split cell adaptation to three-element airfoil

2.2.3.3 Curvature Adaptation

All cut and split cells have interfaces and if the curvature between two interfaces is high, this means that this region necessitates more resolution since regions like that cause high gradients. Therefore, cut or split cells which have neighbor interfaces (this can be edge or vertex neighbors only) are tested whether the curvature between their interfaces are higher. The curvature between two neighboring interfaces is the angle between the normal vectors of these interfaces. If the angle is higher than the threshold angle and then these cells are flagged for refinement and this process is called curvature adaptation. Detailed information can be found in reference [11]. After the application of curvature adaptation, Figure 2.21 is obtained.

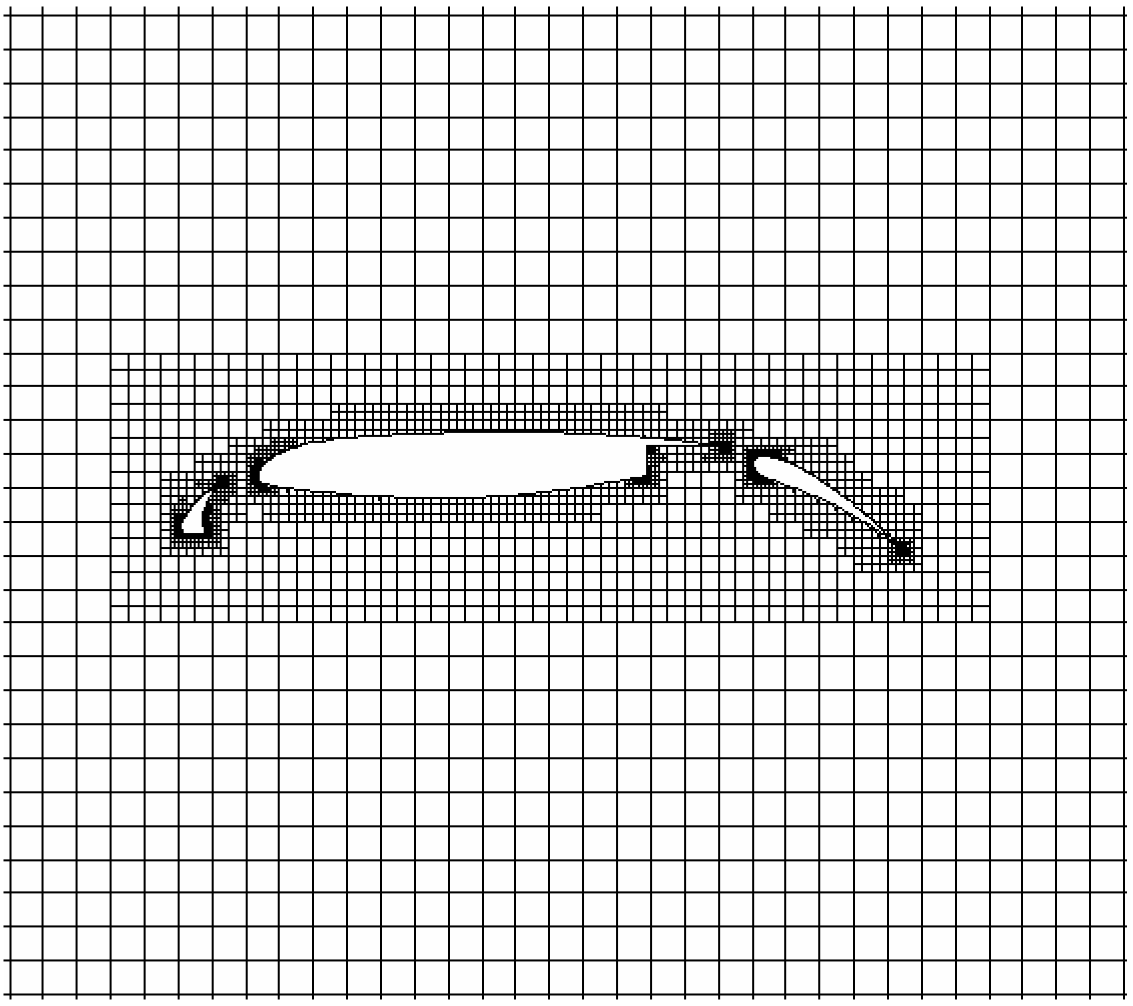


Figure 2.21 Application of curvature adaptation to three-element airfoil

CHAPTER 3

THREE DIMENSIONAL DATA STRUCTURE AND GRID GENERATION

3.1 Octree Data Structure

The octree data structure is chosen in order to build connectivity information for three-dimensional grid because of the simplicity of conversion of the code from two-dimensional grid generation to three-dimensional grid generation.

Like the quadtree data structure, the octree data structure starts with the root cell and other members of the data structure become its children, grandchildren and etc. Any cell in the data structure is identified with fifteen pointers. One is for its parent; eight of them are for its children and the rest of them (six of them) are for its surface neighbors. Although the maximum possible number of neighbors is twenty six, only eight of them are stored. Others are determined when they are necessary. As a result, memory is used effectively. The pointers stored for all cells are given below:

- 1 word: Its parent
- 8 words: Its children
- 6 words: Its surface neighbors
- 1 word: Its level
- 3 words: Its *x*, *y*, and *z* coordinates of the centroid
- 2 words: Parameters for multigrid method which are called perform and compcell in the developed code. These will be explained in the section of multigrid method.

Similar to the quadtree data structure, level of a cell is a very important parameter and it is always one higher than the level of its parent. Centroidal coordinates of a cell are the function of the level of the cell, domain size and centroidal coordinates of its parent and they are calculated by the equations given below. Equations (3.1) to (3.8) are for the calculation of centroidal coordinates of children of a parent starting from first child to eighth child, respectively. *L*, *n* and *c* indexes in these equations are the domain size, level of the cell and center of the cell, respectively.

$$x_c = (x_c)_{parent} + \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} + \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} + \frac{L}{2^{(n+1)}} \tag{3.1}$$

$$x_c = (x_c)_{parent} - \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} + \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} + \frac{L}{2^{(n+1)}} \tag{3.2}$$

$$x_c = (x_c)_{parent} - \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} - \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} + \frac{L}{2^{(n+1)}} \tag{3.3}$$

$$x_c = (x_c)_{parent} + \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} - \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} + \frac{L}{2^{(n+1)}} \tag{3.4}$$

$$x_c = (x_c)_{parent} + \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} + \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} - \frac{L}{2^{(n+1)}} \tag{3.5}$$

$$x_c = (x_c)_{parent} - \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} + \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} - \frac{L}{2^{(n+1)}} \tag{3.6}$$

$$x_c = (x_c)_{parent} - \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} - \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} - \frac{L}{2^{(n+1)}} \tag{3.7}$$

$$x_c = (x_c)_{parent} + \frac{L}{2^{(n+1)}} \quad y_c = (y_c)_{parent} - \frac{L}{2^{(n+1)}} \quad z_c = (z_c)_{parent} - \frac{L}{2^{(n+1)}} \tag{3.8}$$

One level rule is applied to the developed code like two-dimensional one. Hence, the data structure does not become much more complicated. A cell in three-dimensional grid has six surface neighbors which are called east, west, north, south, top and bottom neighbors. It is mentioned that since the determination of these neighbors for three-dimensional grids is more complicated than that for two-dimensional grids, this procedure for the first child is summarized in this section. Determination of neighbors of other children is similar to the first one. Therefore, others are not explained here. Numbering of children of a parent is shown in Figure 3.1.

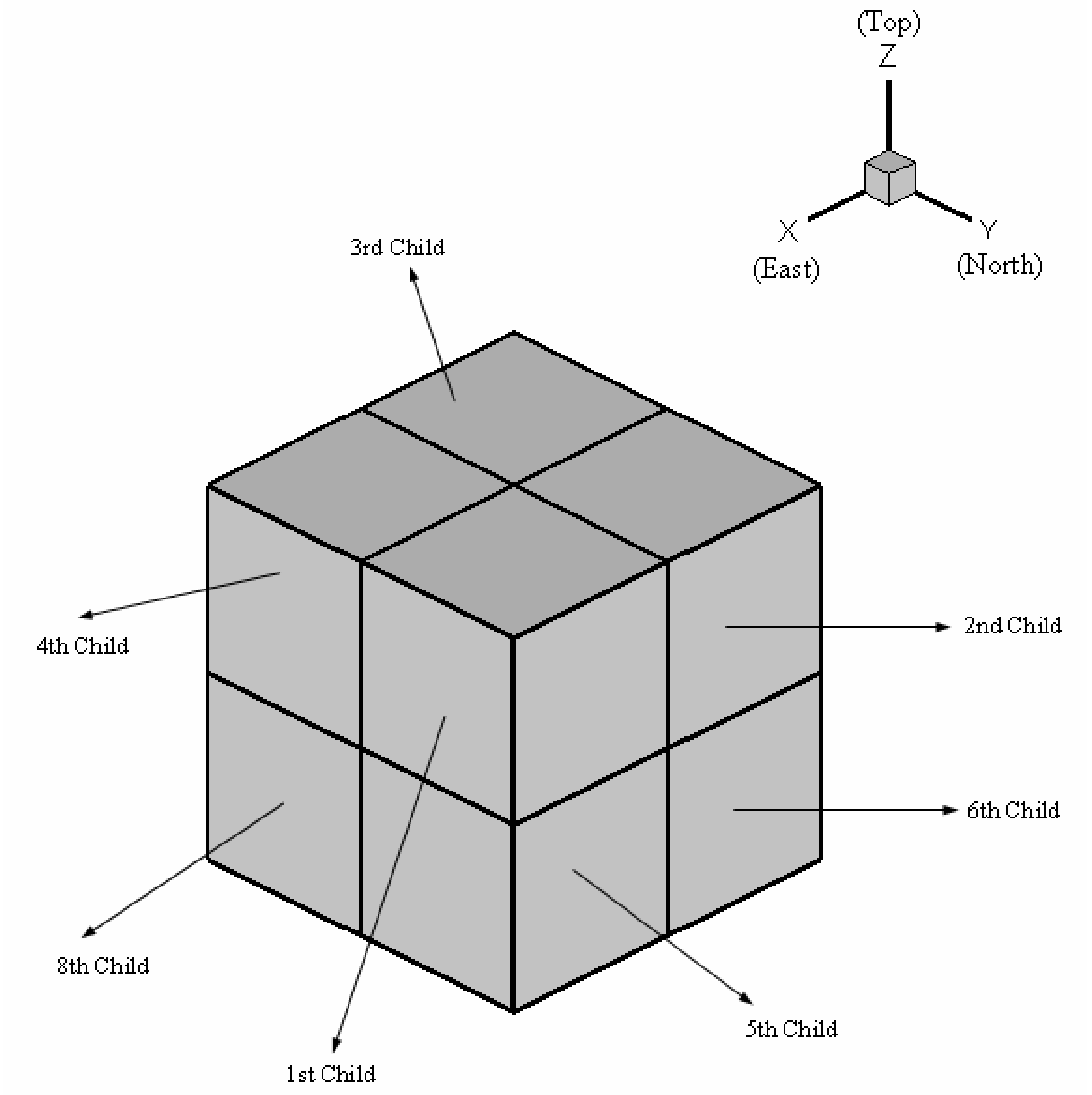


Figure 3.1 Numbering of children

Three surface neighbors of the first child in Figure 3.1 are automatically known because they are the children of its parent. Namely, west, south and bottom neighbors of the first child are second, fourth and fifth children of its parent, respectively. There are three possibilities for other neighbors according to the one level rule. The first possibility is that the neighbor of a cell is one level lower than the cell's level. Other possibility is that both the cell and its neighbor are at the same level and finally, neighbor's level is one higher than the level of the cell. There is only one neighbor if the level of neighbor is the same or one lower. However, if the level of neighbor is one higher than the level of the cell, this means that this cell has four neighbors along the same surface. Instead of storing all of these four neighbors, only the parent of these neighboring cells is stored in order to prevent the excessive usage of memory.

East, north and top neighbors of the first child are determined by means of the neighboring information of its parent. For example, if the east neighbor of its parent is assigned to null (i.e. it is not the far-field), the east neighbor of the first child is also assigned to null. Otherwise, if the east neighbor of its parent is not assigned to null, it is checked whether it has children or not. If it has no child, then east neighbor of the first child is set as the east neighbor of its parent. In this case, level difference between the first child and its east neighbor becomes one and neighbor's level is one level lower than the first child's level. If the east neighbor of first child's parent has children then the second child of the east neighbor of the parent is set as the east neighbor of the first child. This neighbor is at the same level as the cell. In fact, this cell might have children. Even this cell have children, it is set as the east neighbor to prevent complexity. But it is checked during the flux calculation and reconstruction schemes whether the neighboring cell, whose level is the same with the considered cell, has children. If yes, then its neighboring children are used in the calculations. Namely, the east neighbor of the first child is set as the second child of the east neighbor of the parent although this east neighbor has children. But in the calculations, four children of this neighbor are used. These children are the second, third, sixth and seventh children of the second child of the east neighbor of the first child's parent.

If the north neighbor of the parent of the first child is not set as null and it has no children, it is set as the north neighbor of the first child. However, if it has children, fourth child of the north neighbor of first child’s parent is set as the north neighbor of the first child. If this cell has also children, third, fourth, seventh and eighth children of this cell are used in flux calculations as the surface neighbors.

Finally, if the top neighbor of the parent of the first child is not set as null and it has no children, the top neighbor of the parent of the first child is set as the top neighbor of the first child. However, if it has children, fifth child of the top neighbor of first child’s parent is set as the top neighbor of the first child. If this cell has also children, fifth, sixth, seventh and eighth children of this cell are used in flux calculations.

As it is mentioned before, a cell has maximum twenty six neighbors. Six of them are surface neighbors and they are stored. Determination of these neighbors was explained here for the first child of a parent cell. Other twelve neighbors are edge neighbors. These are determined by means of surface neighbors of a cell. But sometimes a cell has no edge neighbor along one of its edges. The reason of this case is that the level of the surface neighbor is one lower than the level of the cell. The rest of eight neighbors are the corner neighbors. Like edge neighbors, they are also determined by means of surface neighbors of the cell.

Finally, in three-dimensional grid generation, there are three types of computational cell: inside, outside and cut cells. Since the three-dimensional grid generation is very difficult when compared to two-dimensional one, split cells are not handled. Instead, these irregular cells which are not inside, outside or cut cells are recursively refined until these three types are obtained. Consequently, the data structure is not as complicated as in two-dimensional grid generation code.

It is beneficial to state that there are additional stored variables for computational and cut cells. The pointers stored for computational cells are given below:

- 5 words: Conservatives variables for continuity, *x*-momentum, *y*-momentum, *z*-momentum and energy equations
- 8 words: Its eight corners
- 1 word: Its type
- 1 word: Its total cube index
- 2 words: Its refinement criteria
- 3 words: Divergence and curl of velocity vector and strength of the entropy wave
- 5 words: For forcing functions

The pointers stored for cut cells are given below:

- 3 words: *x*, *y* and *z* centroidal coordinates of a cut cell
- 9 words: *x*, *y* and *z* corner coordinates of a triangle which forms the cut surfaces

In two-dimensional grid generation, centroid and area of cut cells are calculated by triangulation of the outside part of cut cells. In three-dimensional grid, centroid and volume of cut cells are calculated by means of division of the outside part into tetrahedrons. As a result, volumes and centroids of each tetrahedron are calculated and summation of them gives the total volume. If four vertex coordinates of a tetrahedron are known as shown in Figure 3.2, calculation of its volume is the absolute value of scalar triple product. It is given in Equation 3.9.

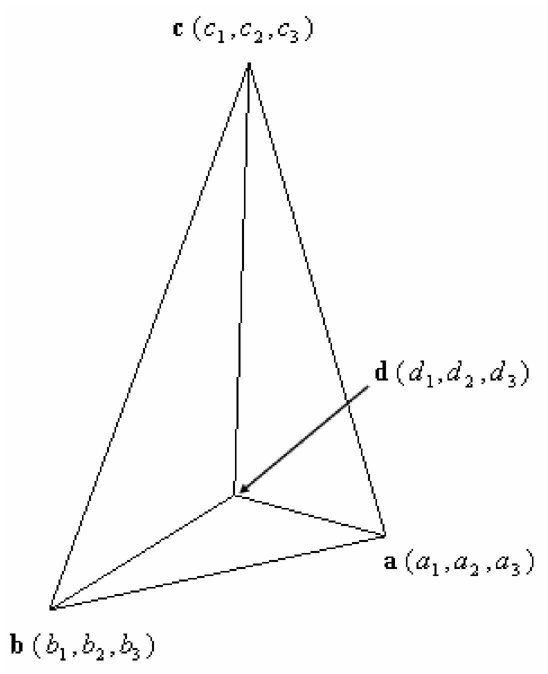


Figure 3.2 Coordinates of four vertex of a tetrahedron

$$Volume = \left|\vec{A} \bullet (\vec{B} \times \vec{C})\right| \tag{3.9}$$

where

$$\vec{A} = (a_1 - d_1)\vec{i} + (a_2 - d_2)\vec{j} + (a_3 - d_3)\vec{k}$$

$$\vec{B} = (b_1 - d_1)\vec{i} + (b_2 - d_2)\vec{j} + (b_3 - d_3)\vec{k}$$

$$\vec{C} = (c_1 - d_1)\vec{i} + (c_2 - d_2)\vec{j} + (c_3 - d_3)\vec{k}$$

3.2 Initial Grid Generation and Geometry Adaptation

Three-dimensional grid generation is very similar to two-dimensional grid generation. In order to generate three-dimensional grid, first of all, geometry is introduced to the code and then uniform mesh is generated around the geometry. The last step of grid generation is geometric adaptation and this is achieved in two steps: box and cut cell adaptation.

3.2.1 Creating Domain and Uniform Mesh Generation

As it is mentioned, introduction of the geometry around which the external flow will be analyzed is the initial step of three-dimensional grid generation. Surface or volume mesh of the geometry is used for the introduction according to the inside-outside testing method. There are two different methods used for inside-outside testing. One is ray casting which is explained in the previous chapter and the other is a method which is associated with cross product and will be explained later. In order to use both of these methods, both surface and volume meshes of the geometry are required. Surface and volume meshes are composed of triangles and tetrahedrons, respectively. After the generation of a mesh by means of GAMBIT software, it is exported as a file whose extension is ".neu". Examples of generated surface mesh around a sphere and its output file from GAMBIT are shown in Figure 3.3 and Appendix B, respectively.

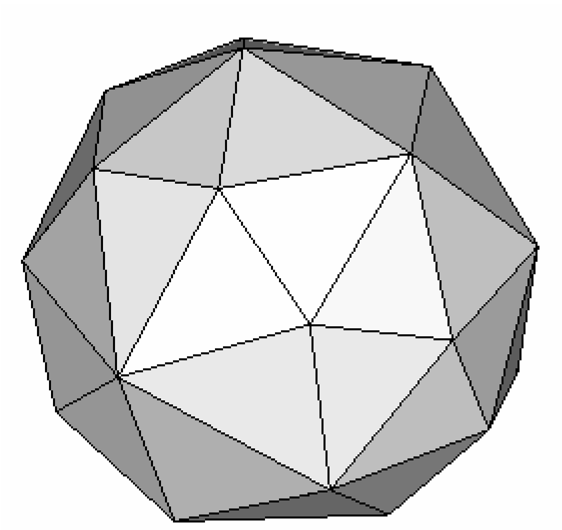


Figure 3.3 Example surface mesh on a sphere

After the introduction of the geometry, uniform mesh is generated around the given geometry. For example, if the uniform division number for the uniform mesh is set as three, the root cell is divided three times successively. As a result, Figure 3.4d is obtained.

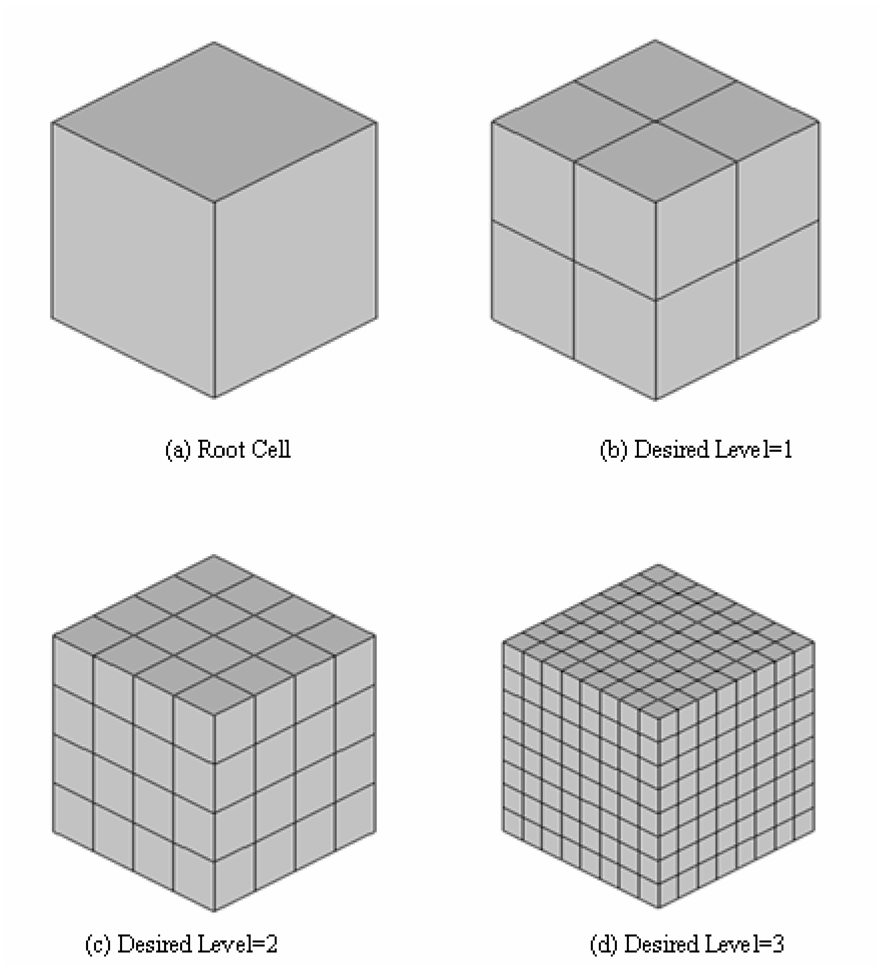


Figure 3.4 Root cell and different uniform meshes with different uniform division numbers

3.2.2 Inside-Outside Determination

As it is mentioned before, there are two different types of inside-outside testing method. Cross product of a corner with volume mesh is the first method which is used in the developed code. But the execution time of this testing method is very long when it is compared to the ray casting method and, therefore, it is not preferred. In spite of its ineffectiveness, this method is explained here because it is sometimes used due to its reliability.

In the first method, each of the tetrahedron meshes (volume meshes) of the given geometry is tested whether a corner of a cell is inside of this tetrahedron or not. Namely, if the volume mesh of the given geometry is composed of *n*-tetrahedrons, each corner of a cell is tested *n*-times. Therefore, it takes long time. In addition, for each tetrahedron, four terms are calculated. Figure 3.5 is an example of this method. Point P can be considered one of the corners of a cell and vertices of any tetrahedron are demonstrated as 1, 2, 3 and 4 in Figure 3.5. If all of the four terms, which are computed by using Equations 3.10, 3.11, 3.12 and 3.13, are positive or all are negative or all are zero, then the point P is inside of the tetrahedron. Otherwise, it is outside of the tetrahedron [14]. In other words, if the signs of all terms are the same then it means that this corner is inside of the tested tetrahedron.

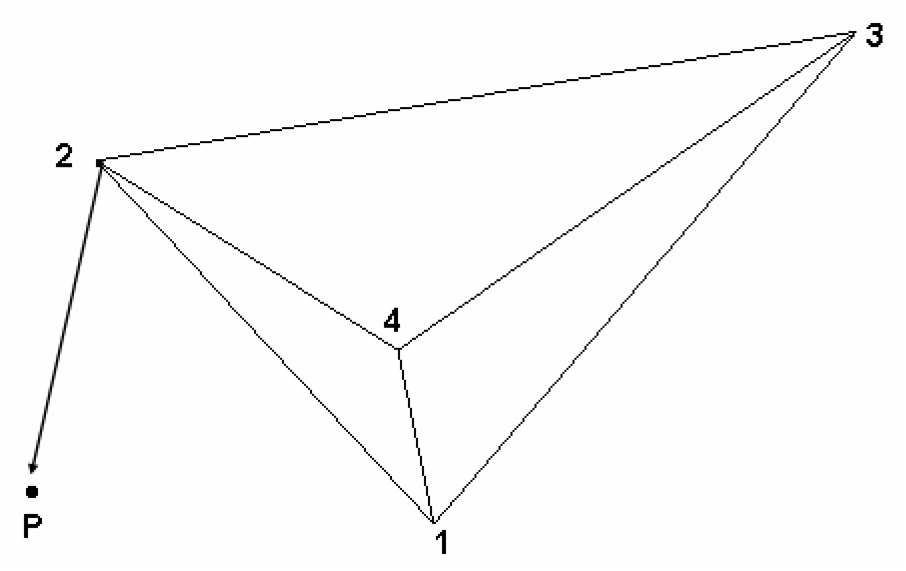


Figure 3.5 Inside-outside determination of point P in 3D

Term 1= [|21|x|32|]●|P2| (3.10)

Term 2= [|31|x|43|]●|P3| (3.11)

Term 3= [|34|x|23|]●|P3| (3.12)

Term 4= [|41|x|24|]●|P4| (3.13)

where

$$|ab| = (a_x - b_x)\cdot\vec{i} + (a_y - b_y)\cdot\vec{j} + (a_z - b_z)\cdot\vec{k}$$

Ray casting method was discussed in Chapter 2 for two-dimensional case. In this case, a ray intersects with the line segments of the given geometry (line-line intersection in two-dimensions). In three-dimensions, how many times a ray, which is cast from one of the eight corners of a cell, is intersected with the triangular surface meshes of a given geometry are counted. This type of intersection is line-triangle intersection in three-dimensions and the determination of this type of intersection is discussed in detail in reference [15].

After the inside-outside test of all corners of a cell, cell type is determined. If all of the eight corners are inside the geometry then it is an inside cell. Otherwise, if all of the eight corners are outside the geometry then it is an outside cell. Other cells which have both inside and outside corners are set as cut cell. After the cell type determination, geometric adaptation starts.

3.2.3 Geometric Adaptation

The purpose of the geometric adaptation in three-dimensions is the same as in the two-dimensional case. However, in this part, curvature adaptation is not applied to the geometry since the only application of box and cut cell adaptations is found to be satisfactory. In addition, it is more complicated than box and cut cell adaptations for three-dimensional case. This will be identified as a future work.

3.2.3.1 Box Adaptation

In order to obtain high resolution around the given geometry, box adaptation is an important step of geometric adaptation. After the box adaptation around a sphere, slice of Cartesian volume mesh in *xy* plane can be seen from Figure 3.6. Like two-dimensional grid generation, one level rule is applied to the surface, edge and corner neighbors during geometric adaptation. Therefore, smooth grid is obtained as seen Figure 3.6.

After the box adaptation, cut surfaces of cut cells are tried to be determined by marching cubes algorithm. Marching squares algorithm was discussed in Chapter 2 and marching cubes is similar to the marching squares. In two-dimensions, cut edges are tried to be found by means of total square index of a cell and a given table. In three-dimensions, cut surfaces are determined by means of total cube index of a cell and a table which is given in Appendix C.

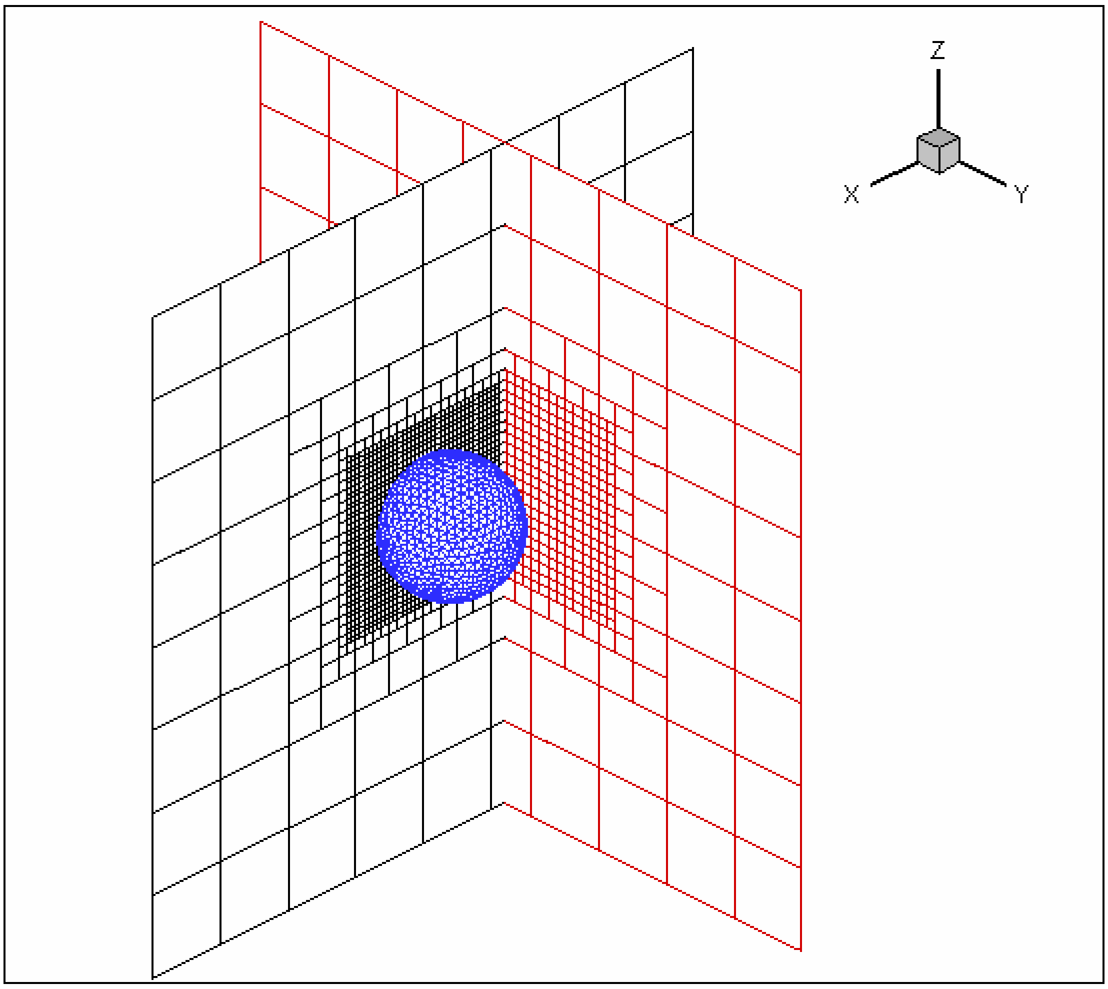


Figure 3.6 Application of box adaptation to sphere surface mesh exported from GAMBIT

Numbering of edges and corners of a cell is shown in Figure 3.7. Like marching squares algorithm, total cube index is calculated by examining the ϕ values of each corner of a cell. Total cube indexes of outside and inside cell are 0 and 255, respectively. Cube indexes of each corner according to their ϕ value used in the developed code are given below:

- If ϕ of corner 0 = -1 then cube index = 1
- If ϕ of corner 1 = -1 then cube index = 2
- If ϕ of corner 2 = -1 then cube index = 4
- If ϕ of corner 3 = -1 then cube index = 8
- If ϕ of corner 4 = -1 then cube index = 16
- If ϕ of corner 5 = -1 then cube index = 32
- If ϕ of corner 6 = -1 then cube index = 64
- If ϕ of corner 7 = -1 then cube index = 128

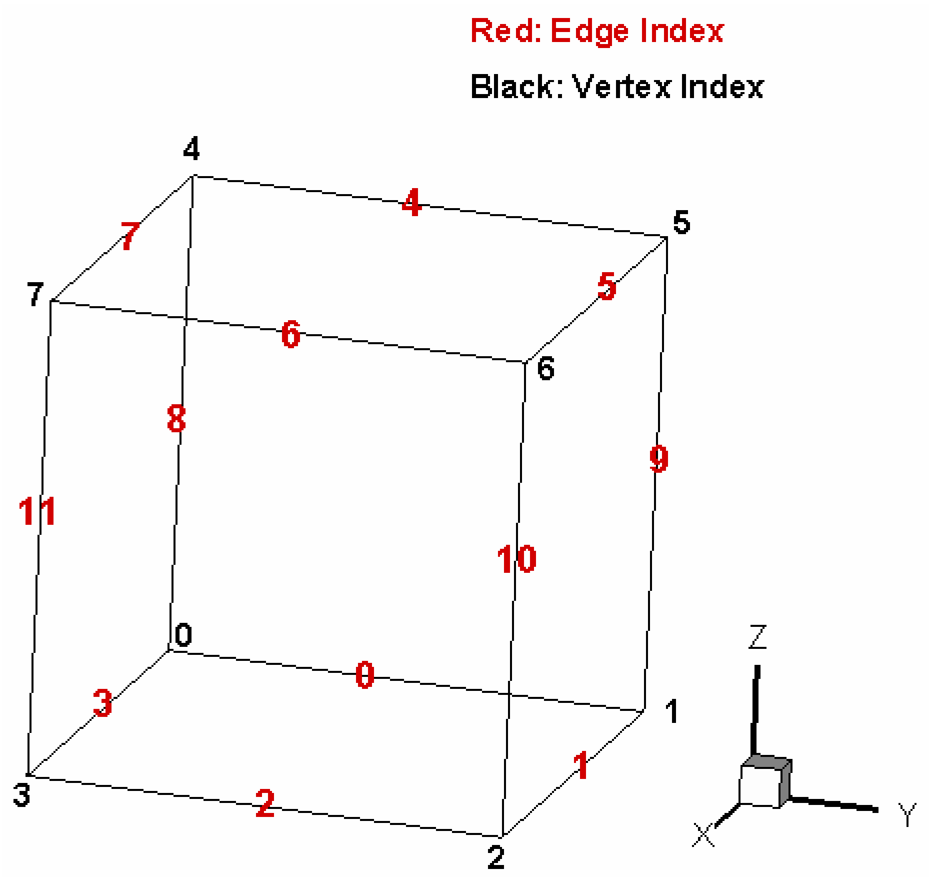
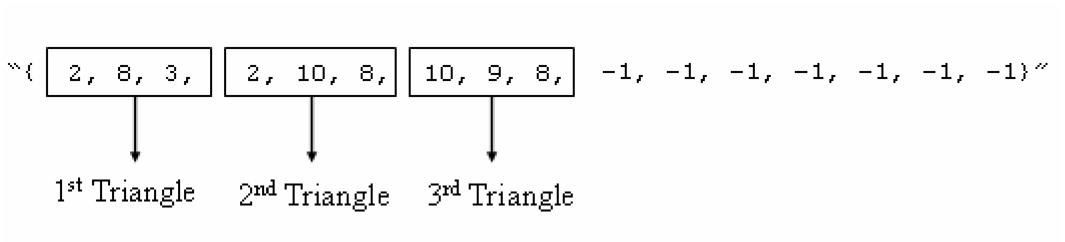


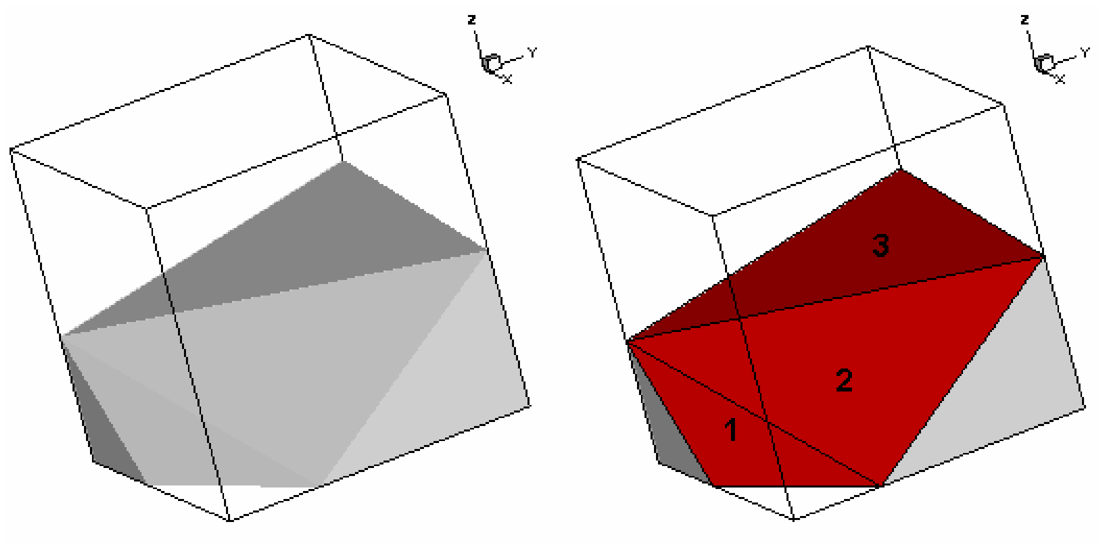
Figure 3.7 Numbering of edges and vertexes of a three dimensional cell

For example, total cube index of a cell whose zeroth, first and second corners are only inside of the geometry is 7. Cut edges of this cell and triangular cut surfaces are found by means of the table in Appendix C. The row corresponds to this total cube index in the triangle table is given below:


```
"{ 2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1}"
```

According to this information, cut edges are 2, 8, 3, 10 and 9. When cut edges are known, cut locations on cut edges are found the method given in reference [15] which is called line-triangle intersection. After finding cut locations on these edges, three triangles which pass along these cut locations can be drawn by means of following the given sequence in the triangle table in Appendix C. Inside and outside portions of the cell whose cube index is 7 are seen from the Figure 3.8a. According to the table, this cell has three triangular cut surfaces as seen in Figure 3.8b. Corners of the first triangle are on the second, eighth and third edges, respectively. Corners of the second triangle are on the second, tenth and eighth edges, respectively. Corners of the third triangle are on the tenth, ninth and eighth edges, respectively.

It is important to note that normal vectors of each triangular surface are pointing to the outside part of the cell. This feature of triangles facilitates the flux, volume and centroid calculations.



(a) (b)

Figure 3.8 Marching Cube Algorithm

3.2.3.2 Cut Cell Adaptation

Cut cell adaptation is also very important part of geometric adaptation since both the cut cells and their outside surface, edge and corner neighbors are refined. As a result, critical regions such as shocks, expansion waves are easily detected and solution refinement is applied to the grid in a more reliable manner. After the application of cut cell adaptation to the Figure 3.6, Figure 3.9 is obtained.

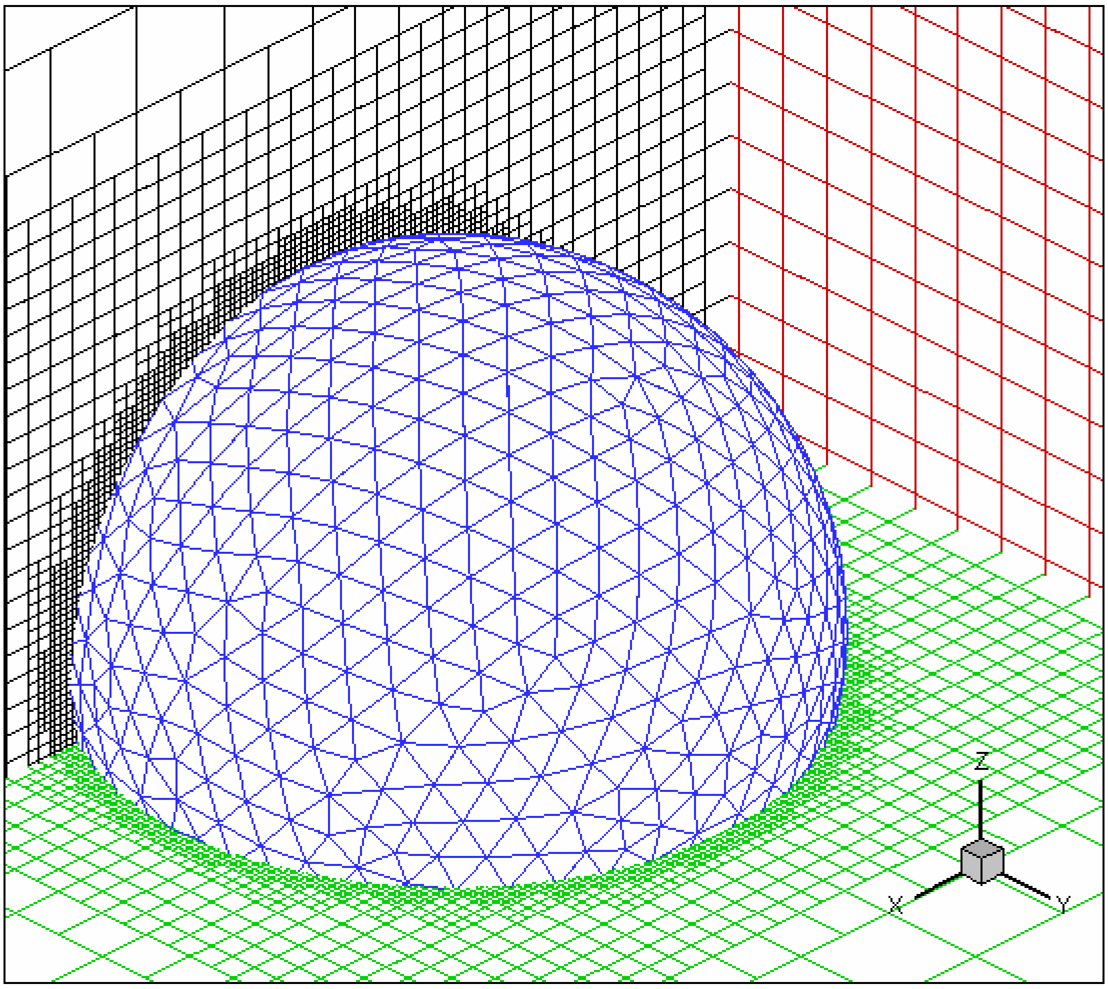


Figure 3.9 Cut cell adaptation

CHAPTER 4

FLOW SOLVER

A Cartesian grid generation method for the solution of the steady-state Euler equations was discussed in Chapters 2 and 3. In this chapter, first of all, integral form of three-dimensional inviscid and compressible governing equations (Euler Equations) is introduced. Then, spatial and temporal discretizations of these integral forms of equations are discussed. Numerical flux construction schemes are explained for three dimensional cases in order to calculate fluxes through the surfaces of the cells. These schemes are the Approximate Riemann Solver of Roe and Liou's Advection Upstream Splitting method (AUSM). For most of the test cases, first-order spatial accuracy is used for the conserved variables in the flux calculations in order to use multigrid application. The developed solver couldn't perform second order flux calculations with multigrid applications.

The finite volume formulation of the three-dimensional conservative Euler equations is achieved by using a cell-centered approach. Solution adaptation is used for resolving more critical regions in the solution domain because the Cartesian grid is very suitable for automatic grid generation. As a result of solution adaptation, sufficient resolution around critical regions is obtained without increasing the total grid number considerably. Primitive variables are reconstructed using the least squares methods to achieve solution adaptation. In order to ensure accurate and bounded values, limiters are employed in the reconstruction process. Divergence and curl of velocity vector and the strength of the entropy wave are used for resolving the critical regions. The combination of these three criteria is expected to give better results than a single one.

Multigrid convergence acceleration technique (Full Approximation scheme) is used in order to increase the convergence rate. Firstly, the problem under consideration is solved on a fine mesh. Then the grid are coarsened and refined successively in order to obtain the improved solution in a short time.

4.1 Three Dimensional Euler Equations

In the developed code, discretized forms of the integral equations are used; therefore, firstly, it is beneficial to introduce integral conservative form of Euler Equations.

$$\frac{\partial}{\partial t}\int_{\Omega}\mathbf{q}\,\mathrm{d}\Omega + \oint_{A}(\mathbf{F}\bullet\mathbf{n})\,\mathrm{d}A = 0 \tag{4.1}$$

Here, $\mathbf{q}$ is a vector of conserved variables and $(\mathbf{F}\bullet\mathbf{n})$ is a vector of fluxes perpendicular to the surface of a cell where the flux through this surface is calculated. *A* and *Ω* are the area of this surface and the volume of this cell, respectively. $\mathbf{n} = (n_x, n_y, n_z)^T$ is the normal vector which is pointing the outside of the cell and $\mathbf{F}$ is defined as $\mathbf{F} = \mathbf{F}_x\vec{i} + \mathbf{F}_y\vec{j} + \mathbf{F}_z\vec{k}$.

$$\mathbf{q} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{pmatrix} \tag{4.2}$$

$$\mathbf{F}_x = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ \rho uH \end{pmatrix}, \quad \mathbf{F}_y = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vw \\ \rho vH \end{pmatrix}, \quad \mathbf{F}_z = \begin{pmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + p \\ \rho wH \end{pmatrix} \tag{4.3}$$

Dot product of the flux and normal vectors gives the fluxes perpendicular to the surface and it is defined as Φ and given in Equation (4.4).

$$\Phi = \begin{pmatrix} \rho V_n \\ \rho u V_n + p n_x \\ \rho v V_n + p n_y \\ \rho w V_n + p n_z \\ \rho V_n H \end{pmatrix} \tag{4.4}$$

where $V_n = un_x + vn_y + wn_z$ is the normal velocity to the surface pointing the outside of the cell, ρ is the density of the fluid, $\mathbf{v} = u\vec{i} + v\vec{j} + w\vec{k}$ is the fluid velocity vector. E is the specific total energy, H is the specific total enthalpy and p is the fluid static pressure. Thermodynamic relations regarding E, H and p are given in the following equations.

$$E = e + \frac{u^2 + v^2 + w^2}{2} \tag{4.5}$$

$$H = E + \frac{p}{\rho} \tag{4.6}$$

$$p = \rho e(\gamma - 1) = \rho(\gamma - 1)(E - \frac{u^2 + v^2 + w^2}{2}) \tag{4.7}$$

where e is the specific internal energy and $\gamma = c_p/c_v$ is the ratio of specific heats. In the developed code, initial values of some variables are chosen close to unity in order to decrease the computational load. For example, far-field density and static pressure are chosen as 1 and $1/\gamma$, respectively. Far-field speed of sound is calculated as 1 by using the following equation.

$$c_\infty = \sqrt{\frac{p_\infty \gamma}{\rho_\infty}} \tag{4.8}$$

As a result, magnitude of the far-field velocity vector, $\|\mathbf{v}\|$, is obtained directly as the input Mach number.

4.1.1 Spatial Discretization of Euler Equations

Integral form of Euler Equations is solved easily by using finite volume method. Firstly, physical domain is divided into cells whose control volumes do not change in time. Therefore, flow variables can be stored at the centroids of cells and it can be assumed that they do not vary inside of the control volume. As a result, it is possible to write

$$\frac{\partial}{\partial t}\int_{\Omega} \mathbf{q}\, d\Omega = \Omega \frac{\partial \mathbf{q}}{\partial t} \tag{4.9}$$

Moreover, the surface integral in Equation (4.1) can be approximated by the sum of the fluxes through each face of a control volume. Consequently, spatial discretized form of the Equation (4.1) becomes

$$\frac{\partial \mathbf{q}}{\partial t} = -\frac{1}{\Omega}\sum_{i=1}^{nFaces} \Phi_i A_i \tag{4.10}$$

Residuals of a cell are nonlinear functions of the conservative variables and they may be defined as

$$Res(\mathbf{q}) = \sum_{i=1}^{nFaces} \Phi_i A_i \tag{4.11}$$

As a result, Equation (4.10) may be written as

$$\frac{\partial \mathbf{q}}{\partial t} = -\frac{Res(\mathbf{q})}{\Omega} \tag{4.12}$$

Finally, it is important to note that choosing the appropriate numerical flux construction scheme is very important to obtain the accurate results from spatial discretization.

4.1.2 Temporal Discretization of Euler Equations

A separate discretization in time is required for the solver part of the developed code. Although the steady-state Euler Equations are solved, temporal discretization is necessary for obtaining zero residuals quickly by means of multistage time stepping. Left hand side of Equation (4.12) is discretized in time as the equation given below.

$$\frac{\partial \mathbf{q}}{\partial t} = \frac{\mathbf{q}^{n+1} - \mathbf{q}^{n}}{\Delta t} \tag{4.13}$$

It is important to note that residuals of a cell is a function of flow variables as seen in Equation 4.11; therefore, time step at which the residuals are calculated determines the temporal discretization scheme. In other words, if the residuals are calculated by using flow variables obtained at time steps *n* and *(n + 1)*, this scheme is called implicit scheme and can be given by

$$\frac{\mathbf{q}^{n+1} - \mathbf{q}^{n}}{\Delta t} = -\frac{1}{\Omega}\left[Res(\mathbf{q}^{n+1})\right] \tag{4.14}$$

Otherwise, if the residuals are calculated by using flow variables obtained at the *n* time step, this scheme is called explicit scheme and can be given by

$$\frac{\mathbf{q}^{n+1} - \mathbf{q}^{n}}{\Delta t} = -\frac{1}{\Omega}\left[Res(\mathbf{q}^{n})\right] \tag{4.15}$$

where $Res(\mathbf{q}^{n+1})$ is equal to $Res(\mathbf{q}^{n+1}) = Res(\mathbf{q}^{n}) + \frac{\partial(Res)}{\partial \mathbf{q}}(\mathbf{q}^{n+1} - \mathbf{q}^{n})$ according to the Taylor Series when the higher orders are neglected.

4.1.2.1 Multistage Time Stepping (Runge-Kutta Method)

The discretized Euler equations are solved by starting from a known initial solution in the explicit multistage time stepping method. Three-stage time stepping scheme is given by

$$\begin{aligned}
\mathbf{q}^{0} &= \mathbf{q}^{n} \\
\mathbf{q}^{1} &= \mathbf{q}^{0} - \frac{\upsilon\alpha_1 \Delta t Res(\mathbf{q}^{0})}{\Omega} \\
\mathbf{q}^{2} &= \mathbf{q}^{0} - \frac{\upsilon\alpha_2 \Delta t Res(\mathbf{q}^{1})}{\Omega} \\
\mathbf{q}^{3} &= \mathbf{q}^{0} - \frac{\upsilon\alpha_3 \Delta t Res(\mathbf{q}^{2})}{\Omega} \\
\mathbf{q}^{n+1} &= \mathbf{q}^{3}
\end{aligned} \tag{4.16}$$

Residuals are found by using this initial solution. Then the improved solutions are obtained by means of some iteration. In Equation (4.16), α_k denotes the stage coefficients and Δt is the time step. CFL numbers and stage coefficients for 3, 4 and 5 stages time stepping are presented in Table 4.1 which is taken from reference [3]. For the results in the developed code, three-stage time stepping scheme is used with the first order accuracy.

Table 4.1 Stage coefficients and CFL numbers for the first order multistage scheme for two dimensional problems

	3	4	5
υ	1.5	2.0	2.5
α_1	0.1481	0.0833	0.0533
α_2	0.4	0.2069	0.1263
α_3	1.0	0.4265	0.2375
α_4		1.0	0.4414
α_5			1.0

Table 4.2 Stage coefficients and CFL number for the first order two-stage scheme for three dimensional problems

	2
υ	1.0
α_1	0.4361
α_2	1.0

4.1.2.2 Local Time Step

The main disadvantage of explicit multistage scheme is the limitation on the time step. It cannot be chosen arbitrarily because of the stability problems. Therefore, computation of local time step of each cell is an important issue. It depends on the cell size and the flow properties. It is important to note that there are large size differences between outside and cut cells in Cartesian method.

For an unsteady flow, the minimum local time step is chosen among the all calculated local time steps for each cell and it is used for all cells and this is a big disadvantage for convergence rate. On the other hand, for steady problems, every cell has its own local time step and these values are used for each cell during the calculations of discretized governing equations. Hence, local time stepping for steady problems is a valuable option to increase the convergence rate. Namely, steady problems are not restricted to use only the minimum local time step during the calculation.

In two-dimensional problems, local time step of each cell is calculated by using the following equation

$$\frac{\Delta t}{A_{cell}} = \frac{1}{\Psi_x + \Psi_y} \tag{4.17}$$

where Ψ_x and Ψ_y are the convective spectral radii and they are calculated by using the absolute values of the projections of edges (S_x and S_y) in x and y directions as follows:

$$\Psi_x = \frac{1}{2}(|u| + c_{cell}) \sum_{i=1}^{nEdges} |S_x|_i \tag{4.18}$$

$$\Psi_y = \frac{1}{2}(|v| + c_{cell}) \sum_{i=1}^{nEdges} |S_y|_i \tag{4.19}$$

where c_{cell} is the local speed of sound which is calculated by using the flow variables stored at the cell centroid.

In three-dimensional problems, local time step of each cell is calculated by means of the following equation

$$\frac{\Delta t}{\Omega_{cell}} = \frac{1}{\sum_{i=1}^{nFaces} (c_{cell} + |V_n|)_i A_i} \tag{4.20}$$

where V_n is the normal velocity to a face and A_i is the area of this face. As it is seen from Equation (4.20), local time step of a cell have to be calculated at every iteration since normal velocity and local speed of sound are changing continuously.

4.2 Flux Computation

One of the most important part of three-dimensional flow solver is the calculation of fluxes, Φ_i, through each face. In this study, two different methods are used for the calculation of fluxes. These are approximate Riemann solver of Roe [10], [16] and [17] and Liou's Advection Upstream Splitting Method (AUSM) [16], [18] and [11]. In the two-dimensional code, the former one is used for most of the cases, while the latter one is used for the most of the cases in the three-dimensional code.

4.2.1 Approximate Riemann Solver of Roe

Fluxes for each face are calculated at the mid-point of the face by using the flow variables of two neighboring cells. The flow variables of the cell whose flux value will be calculated are denoted as the left side and the neighboring cell is represented as right side.

$$\Phi(\mathbf{q_L},\mathbf{q_R}) = \frac{1}{2}\left[\Phi(\mathbf{q_L}) + \Phi(\mathbf{q_R})\right] - \frac{1}{2}\sum_{k=1}^{5}\left|\lambda_k\right|\Delta\mathbf{V_k}\,\mathbf{R_k} \tag{4.21}$$

where

$$\Phi(\mathbf{q_L}) = \begin{pmatrix} \rho V_n \\ \rho u V_n + p n_x \\ \rho v V_n + p n_y \\ \rho w V_n + p n_z \\ \rho V_n H \end{pmatrix}_{\mathbf{L}}, \quad \Phi(\mathbf{q_R}) = \begin{pmatrix} \rho V_n \\ \rho u V_n + p n_x \\ \rho v V_n + p n_y \\ \rho w V_n + p n_z \\ \rho V_n H \end{pmatrix}_{\mathbf{R}} \tag{4.22}$$

It is important to note that third term at the right hand side of Equation (4.21) is calculated by using Roe-averaged quantities. These quantities are given below.

$$\rho_{\mathrm{RL}} = \sqrt{\rho_{\mathrm{L}}\rho_{\mathrm{R}}} \tag{4.23}$$

$$u_{\mathrm{RL}} = \frac{u_{\mathrm{R}}\sqrt{\rho_{\mathrm{R}}} + u_{\mathrm{L}}\sqrt{\rho_{\mathrm{L}}}}{\sqrt{\rho_{\mathrm{R}}} + \sqrt{\rho_{\mathrm{L}}}} \tag{4.24}$$

$$v_{\mathrm{RL}} = \frac{v_{\mathrm{R}}\sqrt{\rho_{\mathrm{R}}} + v_{\mathrm{L}}\sqrt{\rho_{\mathrm{L}}}}{\sqrt{\rho_{\mathrm{R}}} + \sqrt{\rho_{\mathrm{L}}}} \tag{4.25}$$

$$w_{\mathrm{RL}} = \frac{w_{\mathrm{R}}\sqrt{\rho_{\mathrm{R}}} + w_{\mathrm{L}}\sqrt{\rho_{\mathrm{L}}}}{\sqrt{\rho_{\mathrm{R}}} + \sqrt{\rho_{\mathrm{L}}}} \tag{4.26}$$

$$H_{\mathrm{RL}} = \frac{H_{\mathrm{R}}\sqrt{\rho_{\mathrm{R}}} + H_{\mathrm{L}}\sqrt{\rho_{\mathrm{L}}}}{\sqrt{\rho_{\mathrm{R}}} + \sqrt{\rho_{\mathrm{L}}}} \tag{4.27}$$

$$c_{\mathrm{RL}} = \sqrt{(\gamma-1)\left(H_{\mathrm{RL}} - \frac{u_{\mathrm{RL}}^{2} + v_{\mathrm{RL}}^{2} + w_{\mathrm{RL}}^{2}}{2}\right)} \tag{4.28}$$

The normal velocity calculated by using Roe-averaged values is presented in the following equation

$$(V_n)_{\mathrm{RL}} = u_{\mathrm{RL}}n_x + v_{\mathrm{RL}}n_y + w_{\mathrm{RL}}n_z \tag{4.29}$$

Eigen values can be calculated by using the calculated normal velocity as

$$\lambda = \begin{bmatrix} (V_n)_{\mathrm{RL}} - c_{\mathrm{RL}} \\ (V_n)_{\mathrm{RL}} \\ (V_n)_{\mathrm{RL}} \\ (V_n)_{\mathrm{RL}} \\ (V_n)_{\mathrm{RL}} + c_{\mathrm{RL}} \end{bmatrix} \tag{4.30}$$

while the wave strengths are computed by

$$\Delta \mathbf{V} = \begin{bmatrix} \dfrac{\Delta p - \rho_{\mathsf{RL}} c_{\mathsf{RL}} \Delta V_n}{2 c_{\mathsf{RL}}{}^2} \\ \dfrac{c_{\mathsf{RL}}{}^2 \Delta\rho - \Delta p}{c_{\mathsf{RL}}{}^2} \\ \dfrac{n_y \Delta w - n_z \Delta v}{n_y{}^2 + n_z{}^2} \\ \dfrac{n_x n_y \Delta v + n_x n_z \Delta w}{n_y{}^2 + n_z{}^2} - \Delta u \\ \dfrac{\Delta p + \rho_{\mathsf{RL}} c_{\mathsf{RL}} \Delta V_n}{2 c_{\mathsf{RL}}{}^2} \end{bmatrix} \tag{4.31}$$

where $\Delta\rho = \rho_{\mathsf{R}} - \rho_{\mathsf{L}}$, $\Delta p = p_{\mathsf{R}} - p_{\mathsf{L}}$, $\Delta u = u_{\mathsf{R}} - u_{\mathsf{L}}$, $\Delta v = v_{\mathsf{R}} - v_{\mathsf{L}}$, $\Delta w = w_{\mathsf{R}} - w_{\mathsf{L}}$ and $\Delta V_n = (V_n)_{\mathsf{R}} - (V_n)_{\mathsf{L}}$.

Finally, right characteristic vectors are given below:

$$\mathbf{R} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ u_{\mathsf{RL}} - c_{\mathsf{RL}} n_x & u_{\mathsf{RL}} & 0 & \rho_{\mathsf{RL}}(n_y^2 + n_z^2) & u_{\mathsf{RL}} + c_{\mathsf{RL}} n_x \\ v_{\mathsf{RL}} - c_{\mathsf{RL}} n_y & v_{\mathsf{RL}} & -\rho_{\mathsf{RL}} n_z & \rho_{\mathsf{RL}} n_x n_y & v_{\mathsf{RL}} + c_{\mathsf{RL}} n_y \\ w_{\mathsf{RL}} - c_{\mathsf{RL}} n_z & w_{\mathsf{RL}} & \rho_{\mathsf{RL}} n_y & \rho_{\mathsf{RL}} n_x n_z & w_{\mathsf{RL}} + c_{\mathsf{RL}} n_z \\ H_{\mathsf{RL}} - c_{\mathsf{RL}} (V_n)_{\mathsf{RL}} & \dfrac{\delta}{2} & \rho_{\mathsf{RL}}(w_{\mathsf{RL}} n_y - v_{\mathsf{RL}} n_z) & \rho_{\mathsf{RL}}\left[(V_n)_{\mathsf{RL}} n_x - u_{\mathsf{RL}}\right] & H_{\mathsf{RL}} + c_{\mathsf{RL}} (V_n)_{\mathsf{RL}} \end{bmatrix} \tag{4.32}$$

where $\delta = u_{\mathsf{RL}}^2 + v_{\mathsf{RL}}^2 + w_{\mathsf{RL}}^2$.

4.2.2 Liou’s Advection Upstream Splitting Method (AUSM)

This method is chosen for the developed code since it is less complicated and expensive than Van Leer and Steger-Warming flux splitting methods. In this method, Mach number and pressure appearing in the convection flux terms are split.

$$\Phi(\mathbf{q_L},\mathbf{q_R}) = \frac{1}{2}\left[M_{1/2}\left(\mathbf{F}'(\mathbf{q_L}) + \mathbf{F}'(\mathbf{q_R})\right) - \left| M_{1/2} \right| \left(\mathbf{F}'(\mathbf{q_R}) - \mathbf{F}'(\mathbf{q_L})\right)\right] + \mathbf{p}_{1/2} \qquad (4.33)$$

Flux values may be written as

$$\Phi = V_n \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho H \end{pmatrix} + p \begin{pmatrix} 0 \\ n_x \\ n_y \\ n_z \\ 0 \end{pmatrix} = M \begin{pmatrix} \rho c \\ \rho uc \\ \rho vc \\ \rho wc \\ \rho Hc \end{pmatrix} + p \begin{pmatrix} 0 \\ n_x \\ n_y \\ n_z \\ 0 \end{pmatrix} = M \cdot \mathbf{F}' + \mathbf{p} \qquad (4.34)$$

where

$$\mathbf{F}'(\mathbf{q_L}) = \begin{pmatrix} \rho c \\ \rho uc \\ \rho vc \\ \rho wc \\ \rho Hc \end{pmatrix}_{\mathbf{L}} \quad \text{and} \quad \mathbf{F}'(\mathbf{q_R}) = \begin{pmatrix} \rho c \\ \rho uc \\ \rho vc \\ \rho wc \\ \rho Hc \end{pmatrix}_{\mathbf{R}} \qquad (4.35)$$

Split Mach number $M_{1/2}$ and split pressure $\mathbf{p}_{1/2}$ are the average of left and right sides.

$$M_{1/2} = \frac{1}{2}\left(M_{\mathbf{L}}^{+} + M_{\mathbf{R}}^{-}\right) \qquad \mathbf{p}_{1/2} = \frac{1}{2}\left(\mathbf{p}_{\mathbf{L}}^{+} + \mathbf{p}_{\mathbf{R}}^{-}\right) \qquad (4.36)$$

where

$$M_{\mathsf{L}}^{+} = \begin{cases} \frac{1}{4}\left(M_{\mathsf{L}}+1\right)^2 & \left|M_{\mathsf{L}}\right| \le 1 \\ \frac{1}{2}\left(M_{\mathsf{L}}+\left|M_{\mathsf{L}}\right|\right) & \left|M_{\mathsf{L}}\right| > 1 \end{cases} \tag{4.37}$$

$$M_{\mathsf{R}}^{-} = \begin{cases} -\frac{1}{4}\left(M_{\mathsf{R}}-1\right)^2 & \left|M_{\mathsf{R}}\right| \le 1 \\ \frac{1}{2}\left(M_{\mathsf{R}}-\left|M_{\mathsf{R}}\right|\right) & \left|M_{\mathsf{R}}\right| > 1 \end{cases} \tag{4.38}$$

$$\mathbf{p}_{\mathsf{L}}^{+} = \mathbf{p}_{\mathsf{L}} M_{\mathsf{L}}^{+} \begin{cases} 2 - M_{\mathsf{L}} & \left|M_{\mathsf{L}}\right| \le 1 \\ \dfrac{1}{M_{\mathsf{L}}} & \left|M_{\mathsf{L}}\right| > 1 \end{cases} \tag{4.39}$$

$$\mathbf{p}_{\mathsf{R}}^{-} = \mathbf{p}_{\mathsf{R}} M_{\mathsf{R}}^{-} \begin{cases} -2 - M_{\mathsf{R}} & \left|M_{\mathsf{R}}\right| \le 1 \\ \dfrac{1}{M_{\mathsf{R}}} & \left|M_{\mathsf{R}}\right| > 1 \end{cases} \tag{4.40}$$

4.3 Initial Guess and Boundary Conditions

As it is mentioned before, the discretized Euler equations are solved by starting from a known initial solution in the explicit multistage time stepping method. Therefore, the far-field boundary conditions are set as the initial guess for all cells.

For external fluid flows, there are two types of boundary conditions. These are far-field and solid wall boundary conditions. The first boundary condition is necessary for a cell whose neighbor is the far-field. In this case, ghost cell whose size is the same as the size of the cell is used and flow variables for this ghost cell are equated to the far-field conditions. Flow variables of the ghost cell and the cell, which neighbors the far-field, are assigned to the right and left states, respectively, for the flux calculations.

The second boundary condition is necessary for cut or split cells. The flux through the interface between a cut cell and the given geometry is also calculated by means of a ghost cell whose size, density, pressure and specific total enthalpy are the same as this cut cell. The velocity components of the ghost cell in normal and tangential directions are the same as the magnitudes of the cell. The only difference is the direction of the normal component of the velocity vector. These properties are depicted for two-dimensions in Figure 4.1.

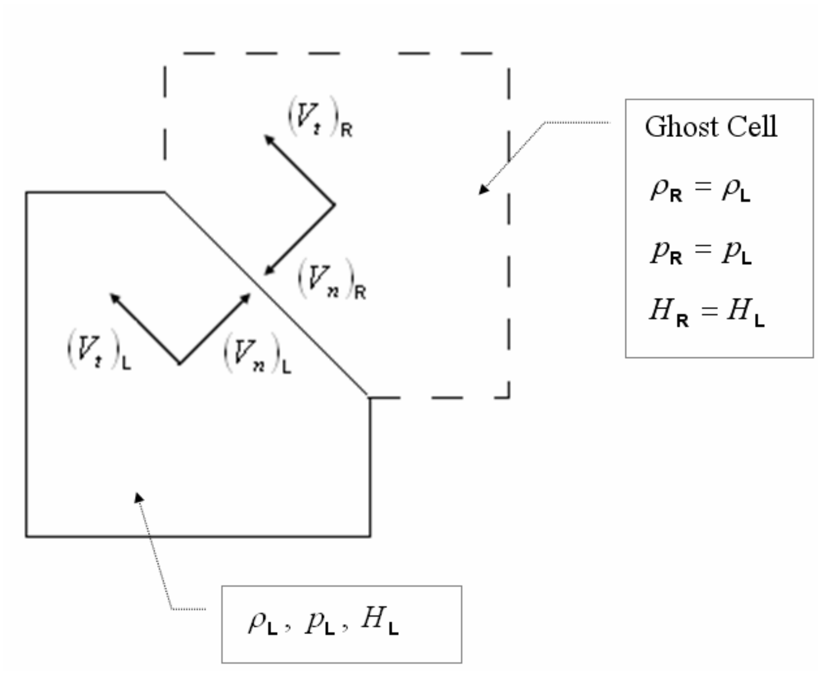


Figure 4.1 Example of a ghost cell in 2D

4.4 Multigrid Method

Fedorenko [19] and [20] developed the first multigrid scheme to solve Poisson equations. This scheme was then modified in order to use it for elliptic boundary value problems by mathematicians. But the effective multigrid method was developed by Brandt [21]. In addition, multigrid method for nonlinear problems was developed by Brandt and this method is called Full Approximation Storage (FAS) scheme. Another achievement regarding multigrid method is Full Multigrid (FMG) scheme. This scheme is based on the nested iteration and multigrid method [22]. The last improved multigrid method is Algebraic Multigrid (AMG) method. Multigrid method is mostly applied to linear and nonlinear boundary value problems. Other applications are hyperbolic, elliptic and Eigen value problems.

The purpose of the multigrid method is to accelerate the convergence rate of a problem. Multigrid method is based on two principles. The first one is error smoothing. High frequency errors are tried to be eliminated effectively in this principle by starting with an initial guess and using some iterative methods such as Jacobi or Gauss Seidel. Although high frequency errors are smoothed after some iterations in this step, low frequency errors improved slightly. The other principle is coarse grid principle and this principle tries to eliminate low frequency errors by using coarse level of grids. This is achieved by transforming the solutions from the present grid to the coarsened grids and performing iterations on these grids. Transforming the solutions from the present grid to the coarsened grids is called restriction. As a result, low frequency errors on the present grid act as high frequency errors on the coarsened grids. Hence, they can be eliminated iteratively on the coarsened grids. Finally, solutions which are obtained by using coarsened grids are interpolated to the fine grid and this process is called prolongation. In other words, low and high frequency errors are tried to be eliminated by using different levels of grids [23].

4.4.1 Multigrid Method for Linear Problems

The matrix notation in the following equation denotes the system of linear equations [24].

$$A\mathbf{x} = \mathbf{b} \tag{4.41}$$

where $\mathbf{x}$ is the exact solution of this system and $\mathbf{y}$ is the approximation to the exact solution. Bold symbols are used to indicate the vectors. $\mathbf{x}^h$ and $\mathbf{y}^h$ notations are used to indicate that vectors are belong to the Ω^h level of mesh. Error vector is found by using

$$\mathbf{x} - \mathbf{y} = \mathbf{e} \tag{4.42}$$

Residual vector is calculated easily for linear problems as follows

$$\mathbf{r} = \mathbf{b} - A\mathbf{y} \tag{4.43}$$

where $\mathbf{r}$ denotes the residual vector. Residual vector becomes zero if and only if the error vector becomes zero since the system is linear. Therefore

$$A\mathbf{e} = \mathbf{r} \tag{4.44}$$

Finally, the improved approximate solutions are obtained by using error vector.

$$\mathbf{y}^{new} = \mathbf{y} + \mathbf{e} \tag{4.45}$$

Multigrid method is generally composed of four steps: (i) fine grid iterations, (ii) restriction, (iii) prolongation and (iv) correction and final iterations [25].

(i) Fine Grid Iterations: Initially, some iteration is performed on the finest mesh with mesh spacing h in order to reduce high frequency errors. After these iterations, approximate solutions, which are denoted $\mathbf{y}^h$, are obtained. After the implementation of these solutions into Equation (4.43), residual vector $\boldsymbol{r}^h$ on this mesh level is found as

$$\mathbf{r}^h = \mathbf{b}^h - A^h\mathbf{y}^h \tag{4.46}$$

(ii) Restriction: In this step, mesh spacing is increased from h to ch. This new mesh is coarser than the finest mesh. Iterations are performed on this new mesh in order to eliminate low frequency errors. Generally, c index is chosen as 2. This facilitates the coarsening algorithm in Cartesian grids since transforming the mesh spacing from h to $2h$ is to delete children of a parent and to set this parent as a computational cell.

In order to perform iterations on this coarse mesh, residual vector, $\boldsymbol{r}^{2h}$ and coefficient matrix, A^{2h} on the coarse level mesh are required. Transfer of residual vector on the fine grid with mesh spacing h to mesh spacing $2h$ is called restriction process and the operator used for this transfer is called restriction operator and denoted by I_h^{2h}.

$$I_h^{2h}\mathbf{r}^h = \mathbf{r}^{2h} \tag{4.47}$$

This transfer is achieved by averaging the residual vectors of children in Cartesian grids. Figure 4.2 depicts the transfer of residual vectors of children to their parent for two dimensional heat problems.

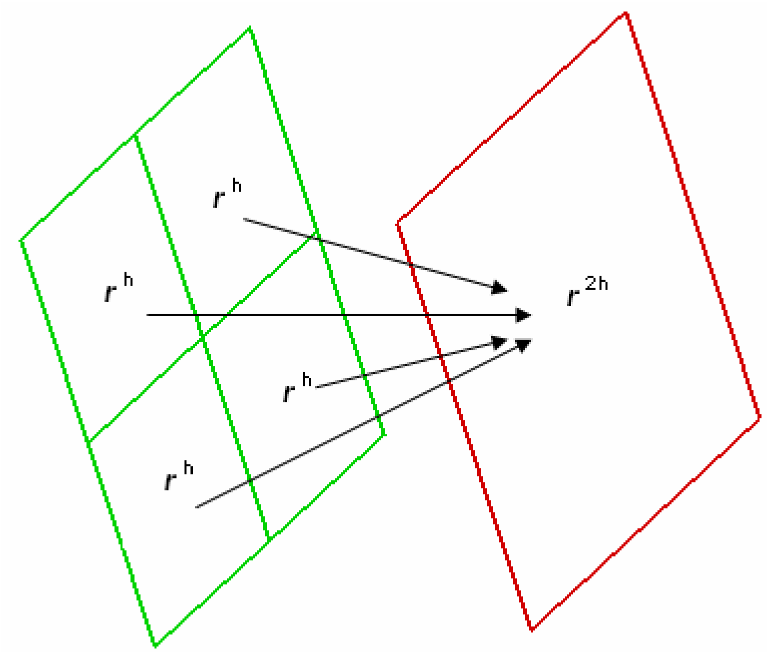


Figure 4.2 Restriction of residual vector in 2D

$$\mathbf{r}^{2h} = \sum_{i=1}^{nChildren} \left(\mathbf{r}^h\right)_i \tag{4.48}$$

After the transfer of residual vector, it is time to find coefficient matrix on the grid whose mesh spacing is $2h$. The error vector on $2h$ level mesh is set as zero and then iterations are performed to find the improved solution for error vector. Gauss Seidel method is chosen for the iterations in the developed code. This process is given in the following equation

$$A^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h} \tag{4.49}$$

The improved error vector, which is found by using Equation 4.49, can be thought to be the low frequency errors. Improved approximate solution $\mathbf{y}^{h(new)}$, whose low and high frequency errors are decreased effectively and rapidly, are obtained by interpolating and adding these errors to the approximate solution $\mathbf{y}^h$.

(iii) Prolongation: Error vectors which are found on the mesh with mesh spacing *2h* are interpolated to the mesh whose spacing is *h*. In a coarse grid, there are fewer points than the fine grid. Namely, interpolated information number on the coarse grid is lower. Therefore, interpolation operator is used. As a result, prolonged error vectors are obtained and these are denoted by $\mathbf{e}^{'h}$. In the code, which is developed for solution of two dimensional heat transfer problem, linear interpolation operator, I_{2h}^{h}, is used. For this code, linear interpolation process is given in the following equation for the third child of the parent whose error vector is denoted by $\mathbf{e}_1^{2h}$.

$$I_{2h}^{h}\mathbf{e}^{2h} = \mathbf{e}^{'h} \tag{4.50}$$

or

$$\mathbf{e}^{'h} = \frac{9\mathbf{e}_1^{2h} + 3\mathbf{e}_2^{2h} + 3\mathbf{e}_4^{2h} + \mathbf{e}_3^{2h}}{16} \tag{4.51}$$

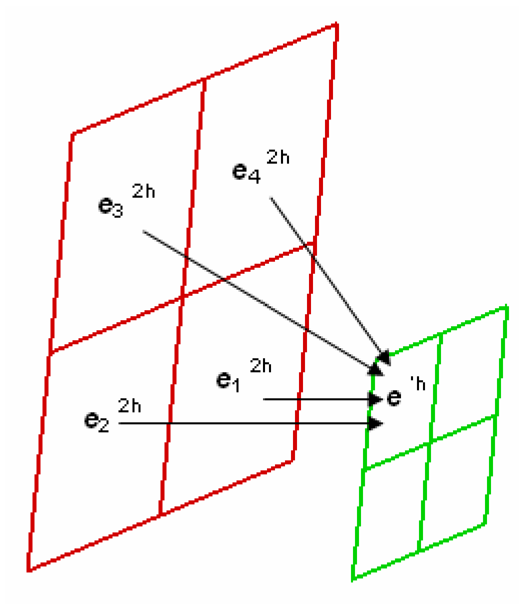


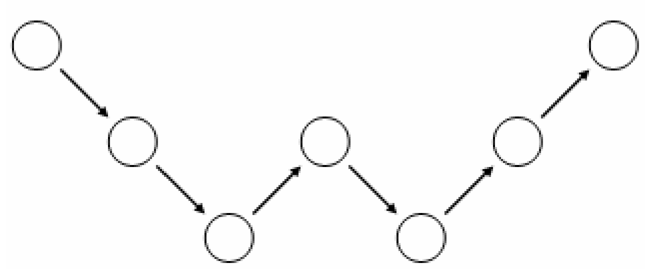
Figure 4.3 Prolongation of error vector in 2D

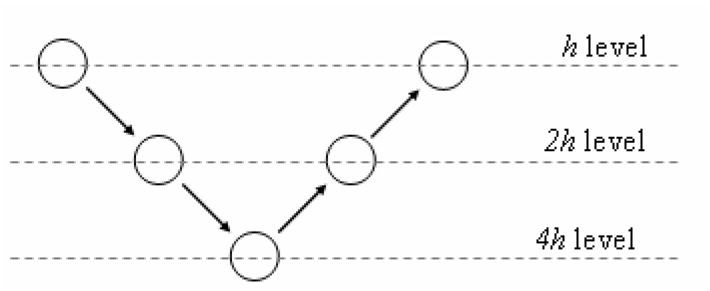
(iv) Correction and Final Iterations: After the calculation of error vectors for the fine grid in the third step, approximate solutions, which are calculated in the first step, are corrected by these error vectors and as a result, improved approximate solutions are obtained.

$$\mathbf{y}^{h(new)} = \mathbf{y}^{h} + \mathbf{e}'^{h} \tag{4.52}$$

Both low frequency errors and the difference between the approximate solution and the exact solution are decreased by this correction. Finally, since the approximations are used during the restriction and prolongation processes, a few iterations are required in order to decrease the effect of these approximations on the solution.

Initially, multigrid method was tested for two dimensional heat transfer problems since it is a linear problem. Afterwards, multigrid application is implemented to Euler solver. Examples of multigrid cycles are given in Figure 4.4 and an example of mesh levels which was used for heat transfer problem is given in Figure 4.5.


(a) W-cycle



(b) V-cycle

Figure 4.4 Multigrid cycles

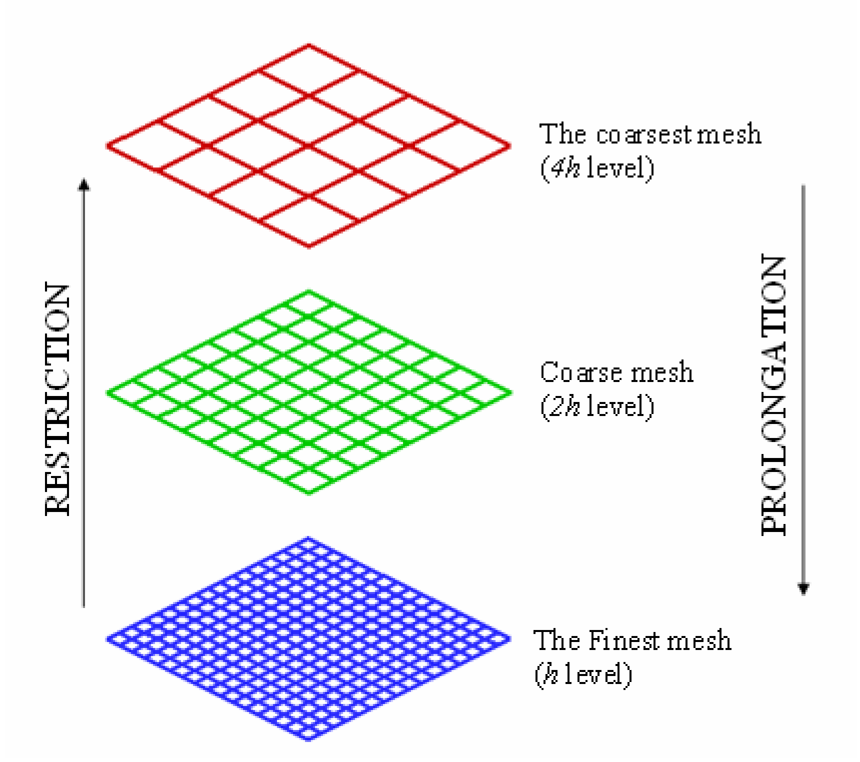


Figure 4.5 Different levels of meshes used in multigrid method

4.4.2 Multigrid Method for Nonlinear Problems

Studies of Jameson [26] and De Zeeuw [4] are very helpful for the implementation of multigrid method to Euler solvers. There are two possible methods for solving nonlinear problems with using multigrid method. These are Newton method and Full Approximation Storage (FAS) scheme. FAS method, which is developed for solving Euler Equations, is used in the code since the governing equations are nonlinear. Nonlinear systems of equations can be given as follows in matrix notation.

$$A(\mathbf{x}) = \mathbf{b} \tag{4.53}$$

where $\mathbf{x}$ and $\mathbf{y}$ are the exact and approximate solutions, respectively. The notation $A(.)$, rather than A, indicates the nonlinear coefficient matrix. Error and residual vector are given by in Equations 4.54 and 4.55, respectively.

$$\mathbf{e} = \mathbf{x} - \mathbf{y} \tag{4.54}$$

and

$$\mathbf{r} = \mathbf{b} - A(\mathbf{y}) \tag{4.55}$$

respectively. The following equation is obtained by subtracting Equation (4.55) from Equation (4.53).

$$A(\mathbf{x}) - A(\mathbf{y}) = \mathbf{r} \tag{4.56}$$

Although, error vector is the difference between the exact and approximate solution (Equation 4.54), it is not concluded that $\mathbf{e} = A(\mathbf{x}) - A(\mathbf{y})$. The reason of this condition is that matrix coefficient is nonlinear. The term ($\mathbf{y}$ + $\mathbf{e}$) can be written in Equation (4.56) instead of using the exact solution $\mathbf{x}$. As a result, the following equation is obtained.

$$A(\mathbf{y}+\mathbf{e})-A(\mathbf{y})=\mathbf{r} \tag{4.57}$$

After some iteration is performed on the finest grid, approximate solution $\mathbf{y}^h$ on the mesh with mesh spacing h is obtained and this solution is improved by using error vector on this level of mesh. This process is summarized below:

- Error vector on the grid with mesh spacing *2h* is found by using the following equation.

$$A^{2h}\left(\mathbf{y}^{2h}+\mathbf{e}^{2h}\right)-A^{2h}\left(\mathbf{y}^{2h}\right)=\mathbf{r}^{2h} \tag{4.58}$$

- The unknowns in Equation 4.58 are the approximate solution and residual vector. These are found by using

$$\mathbf{y}^{2h}=I_h^{2h}\mathbf{y}^h \tag{4.59}$$

and

$$\mathbf{r}^{2h}=I_h^{2h}\mathbf{r}^h=I_h^{2h}\left[\mathbf{b}^h-A^h\left(\mathbf{y}^h\right)\right] \tag{4.60}$$

respectively.

- As a result, improved approximate solution is obtained as

$$\mathbf{y}^{h(new)}=\mathbf{y}^h+I_{2h}^h\mathbf{e}^{2h} \tag{4.61}$$

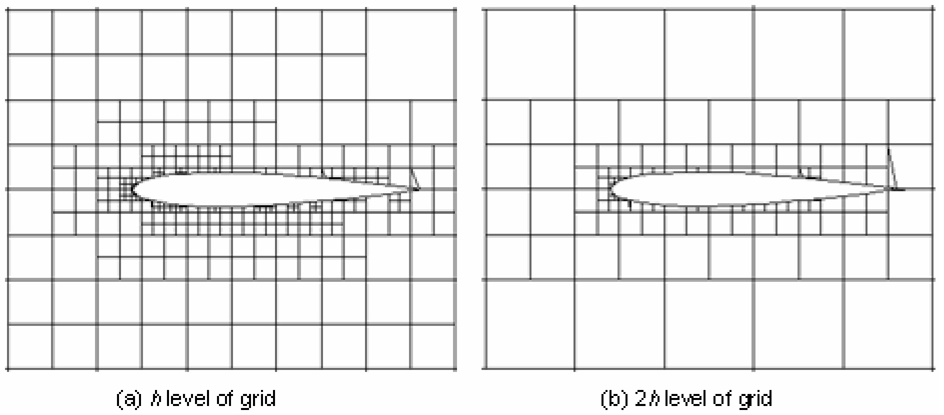
The effects of the multigrid method to the residuals and the convergence rate for the two and three dimensional Euler problems will be examined in Chapter 5 on discussion of results. Now, the implementation of multigrid method to the developed code is summarized. Like linear problems, implementation of multigrid method to nonlinear problems is achieved in four steps. These steps are (i) fine grid iterations, (ii) restriction, (iii) prolongation, and (iv) correction and final iterations.

(i) Fine grid iterations: Initially, some iterations are performed on the finest mesh with mesh spacing *h* by using explicit multistage time stepping scheme in order to solve discretized Euler equations as follows:

$$
\begin{aligned}
&\mathbf{q}_0^h = initial\_guess \\
&\mathbf{q}_1^h = \mathbf{q}_0^h - \upsilon \frac{\alpha_1 \Delta t}{\Omega}\left[Res(\mathbf{q}_0^h) + \mathbf{FF}^h\right] \\
&\ldots\ldots\ldots\ldots\ldots \qquad (4.62) \\
&\mathbf{q}_m^h = \mathbf{q}_0^h - \upsilon \frac{\alpha_m \Delta t}{\Omega}\left[Res(\mathbf{q}_{m-1}^h) + \mathbf{FF}^h\right]
\end{aligned}
$$

where $\mathbf{FF}^h$ term is the forcing function and this term is initially set as zero for the computational cells which form the finest grid. After the iterations on the finest mesh, high frequency errors are effectively reduced but low frequency errors are slightly reduced. Therefore, residual vectors on coarser grids are utilized to decrease these errors on the finest mesh. The last process in this step is the calculation of the residual vector on the finest mesh, which is denoted by $Res(\mathbf{q}_m^h)$, at the end of the iterations.

(ii) Restriction: In this step, the finest mesh with mesh spacing *h* is coarsened to 2*h*, 4*h* and 8*h* levels of meshes. The levels of meshes are depicted in Figure 4.6.



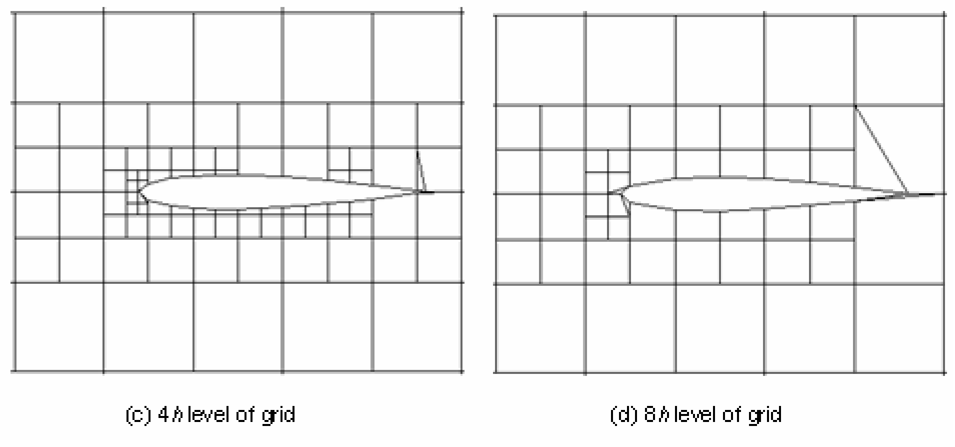
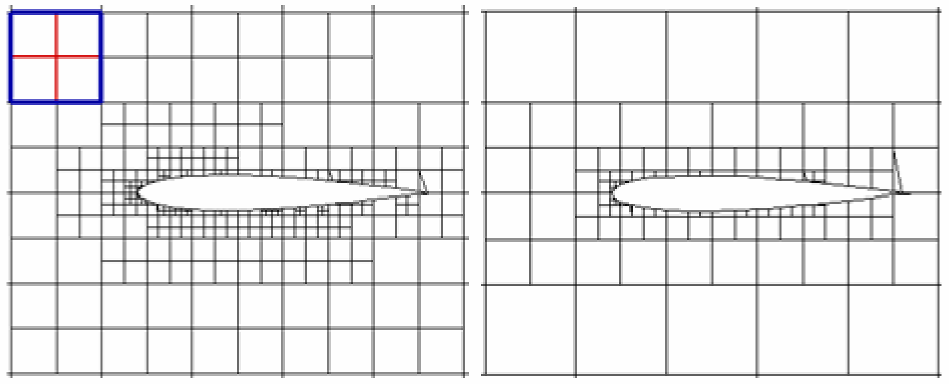
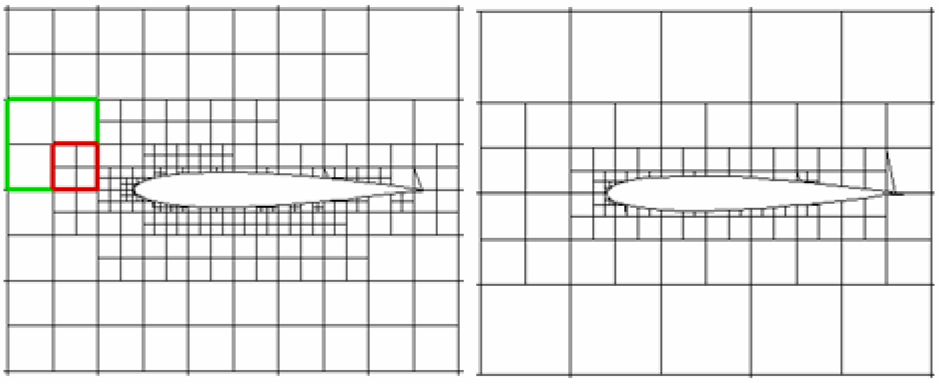


Figure 4.6 Multigrid level settings

Transfer of the grid from *h* level to 2*h* level is summarized in order to explain coarsening process. First of all, parents, whose children are all computational cells, are flagged [27]. For example, blue cell in the two-dimensional grid of Figure 4.7a, is a parent cell. All of its four children are computational cells. Namely, any one of its children has children. On the other hand, green cell in Figure 4.7b is also a parent cell but its first child is not a computational cell. Hence, the green cell is not flagged. Parents are flagged by using a pointer which is called *perform*.



(a) *h* and 2*h* levels of multigrids

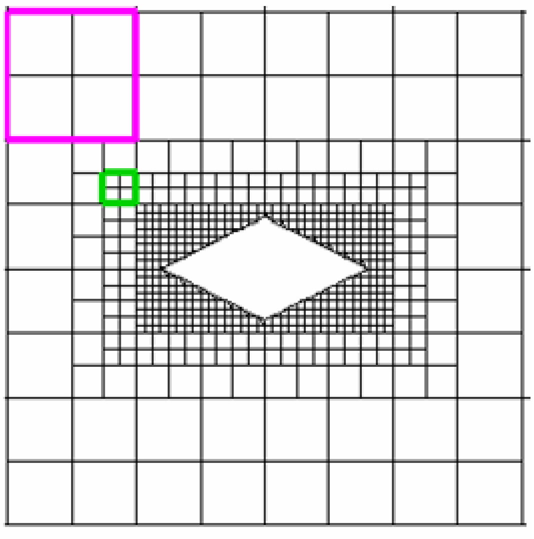


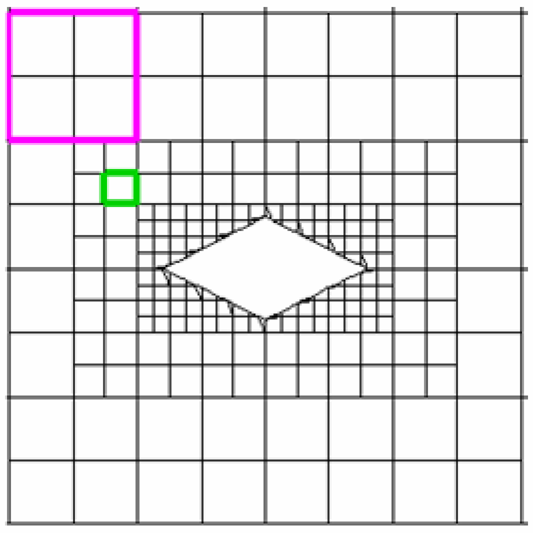
(b) *h* and 2*h* levels of multigrids

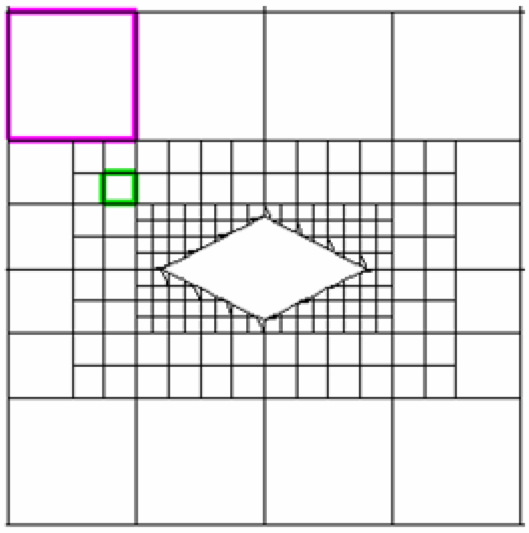
Figure 4.7 Examples of coarsening process

When the flagging process is completed, testing of one level rule is applied to the flagged parent. In other words, if the flagged parent cells do not violate the one level rule when they are coarsened, they are set as computational cells. This setting is performed by using another pointer which is called *compcell*. Instead of deleting children of a parent, which will be coarsened and set as a computational cell, it is assumed that it has no children by using this pointer.

There are two flagged parents in Figure 4.8a. These are pink and green cells. These cells are flagged since all of their children are computational cells. After the testing of one level rule, green cell is coarsened and it becomes a computational cell in the grid with mesh spacing 2*h* since it does not violate the one level rule when it is coarsened. This is seen from Figure 4.8b. On the other hand, if the pink cell was coarsened, it would violate the one level rule as seen from Figure 4.8c. Therefore, although it was flagged in the flagging process, it is not coarsened in the grid with 2*h* spacing not to violate one level rule as seen from Figure 4.8b.


(a) *h* level of multigrid


(b) 2*h* level of multigrid which is formed with testing one level rule



(c) $2h$ level of multigrid which is formed without testing one level rule

Figure 4.8 Examples of one level rule testing process

When a coarsened grid is obtained, restriction of approximate solutions and forcing functions of the computational cells, which form the grid with $2h$ mesh spacing, are required. These are calculated by using

$$\mathbf{q}_0^{2h} = I_h^{2h}\mathbf{q}_m^h = \begin{cases} \dfrac{\sum_{i=1}^{nChildren}(\mathbf{q}_m^h\Omega)_i}{\sum_{i=1}^{nChildren}(\Omega)_i} & \text{For parent cells in h level of multigrid} \\ \\ \mathbf{q}_m^h & \text{For leaf cells in h level of multigrid} \end{cases} \tag{4.63}$$

and

$$\mathbf{FF}^{2h} = \hat{I}_h^{2h}\left[Res(\mathbf{q}_m^h) + \mathbf{FF}^h\right] - Res(\mathbf{q}_0^{2h}) \tag{4.64}$$

respectively, where

$$\hat{I}{}_h^{2h}\left[\beta^h\right] = \begin{cases} \sum_{i=1}^{nChildren} (\beta^h)_i & \text{For parent cells in h level of multigrid} \\ \\ \beta^h & \text{For leaf cells in h level of multigrid} \end{cases} \tag{4.65}$$

I_h^{2h} and $\hat{I}{}_h^{2h}$ are volume weighted collection and residual collection operators, respectively.

After the determination of approximate solutions and forcing function for computational cells for the $2h$ level of multigrid, new approximate solutions are found as

$$\begin{aligned} \mathbf{q}_1^{2h} &= \mathbf{q}_0^{2h} - \upsilon \frac{\alpha_1 \Delta t}{\Omega}\left[Res(\mathbf{q}_0^{2h}) + \mathbf{FF}^{2h}\right] \\ &\ldots\ldots\ldots\ldots\ldots \\ \mathbf{q}_m^{2h} &= \mathbf{q}_0^{2h} - \upsilon \frac{\alpha_m \Delta t}{\Omega}\left[Res(\mathbf{q}_{m-1}^{2h}) + \mathbf{FF}^{2h}\right] \end{aligned} \tag{4.66}$$

Finally, new residual vectors, $\mathrm{Re}\,s(\mathbf{q}_m^{2h})$, are calculated by using the solutions of Equation (4.66) if higher levels of multigrid are used in the multigrid method such as $4h$ and $8h$.

(iii) Prolongation: The purpose of this step is to interpolate the approximate solutions, which are calculated in the restriction process. The following equation exemplifies this process with the interpolation of approximate solutions which are calculated for $4h$ level of multigrid to the finer multigrid with mesh spacing $2h$.

$$\mathbf{q}^{2h(new)} = \mathbf{q}_m^{2h} + I_{4h}^{2h}\left(\mathbf{q}^{4h(new)} - I_{2h}^{4h}\mathbf{q}_m^{2h}\right) \tag{4.67}$$

where I_{4h}^{2h} is the prolongation operator and there are two different prolongation operators. These are gradient and injection operators which are given as

$$I_{4h}^{2h}\left(\mathbf{q}^{4h}\right)=\nabla\left(\mathbf{q}^{4h}\right)\bullet \mathrm{d}\mathbf{r} \tag{4.68}$$

and

$$I_{4h}^{2h}\left(\mathbf{q}^{4h}\right)=\mathbf{q}^{4h} \tag{4.69}$$

, respectively. Injection operator is used in this work due to its simplicity. V-cycle and saw-tooth cycle are tested for the solution of Euler Equations in Chapter 5 during the discussion of results. The only difference between these cycles is that multistage time stepping scheme is also applied to the prolongation step in V-cycle. For this time stepping scheme, initial guess is taken as the improved approximate solutions, $\mathbf{q}_0^{2h(new)}$, and forcing functions are taken as the same values in the restriction step. In order to use the same forcing function values in restriction and prolongation steps, forcing function, which is calculated in the restriction step, must be stored. This causes storage of excessive number of variables. When the convergence rates of V-cycle and saw-tooth cycle are compared, a slight difference is observed. This will be verified in Chapter 5 during the discussion of results. Therefore, saw-tooth cycle is mostly used in this work due to its less memory requirement.

(iv) Correction and final iterations: In the prolongation step, improved approximation solutions of the finest mesh, $\mathrm{q}^{h(new)}$ are calculated. Then, these values are substituted into the following equations and some iteration is performed.

$$\begin{aligned}
\mathbf{q}_0^h &= \mathbf{q}^{h(new)} \\
\mathbf{q}_1^h &= \mathbf{q}_0^h - \upsilon\frac{\alpha_1\Delta t}{\Omega}\left[Res(\mathbf{q}_0^h)+\mathbf{FF}^h\right] \\
&\ldots\ldots\ldots\ldots\ldots \\
\mathbf{q}_m^h &= \mathbf{q}_0^h - \upsilon\frac{\alpha_m\Delta t}{\Omega}\left[Res(\mathbf{q}_{m-1}^h)+\mathbf{FF}^h\right]
\end{aligned} \tag{4.69}$$

As a result, approximate solutions, whose low and high frequency errors are reduced, are obtained.

4.4.3 Importance of Split Cells in Multigrid Application

In literature, previous suggestions for eliminating irregular cells in the mesh generation are the recursive refinement until these cells vanish with refinement application. However, the fundamental principle of multigrid method is to eliminate the low frequency errors by using coarser meshes. Therefore, split cells are mostly required for multigrid applications as mentioned before since irregular cells form at the coarser multigrid levels. In other words, recursive refinement is not a solution for irregular cell in multigrid application.

Especially, grids around multi-element airfoils require recursive refinements in order to eliminate irregular cells. For example, the grid in Figure 4.9 is generated around NLR7301 airfoil and a flap. Only five split cells remain in the grid after recursive refinements and they are located at the trailing edges of the main airfoil and the flap. Split cells are colored red in Figures 4.9 to 4.14. But they cannot be seen in Figure 4.9 since they are very small cells. Therefore, their types can be transferred from split to outside cells by modifying the geometry. Irregular cells are vanished by erasing the sharp trailing edges slightly where split cells are found. As a result of this erasing process, split cells become outside cells. Moreover, this modification affects the solution very slightly. By this method, complexities due to split cells are removed. However, if the grid in Figure 4.9 is taken as the first level of multigrid, *h* level, one coarser level becomes Figure 4.10. As it is seen in Figure 4.10, numbers of split cells increase and this time, the elimination of these split cells by modifying the geometry causes a little more solution errors. This can be also negligible. But as it is seen from Figures 4.11 to 4.14, numbers of split cells are continuously increasing at each coarser level. Finally, all the cells around the flap become split cells at the coarsest level, 32*h* level and modifying the geometry means erasing the flap part wholly. But this has a great effect on the solution and it can be said that the contributions of multigrid method in convergence rate go away. Six multigrid levels are used for the tests in Section 5.1.4 and the coarsest grid which has 32*h* mesh spacing is depicted in Figure 4.15. The grids in Figures 4.14 and 4.15 are the same. Generated grid around the geometry in Figure 4.15 is colored green and it is clearly seen that the leading edge part of the flap is only eliminated by using split cells instead of eliminating the whole flap.

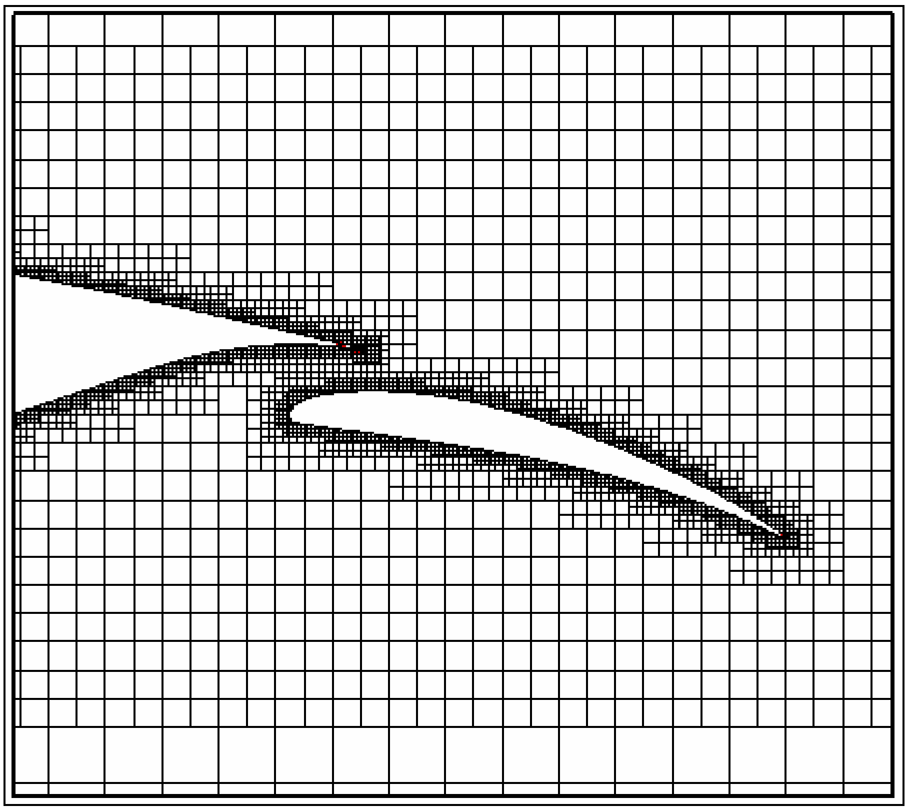


Figure 4.9 *h* multigrid level around NLR 7301 airfoil and flap

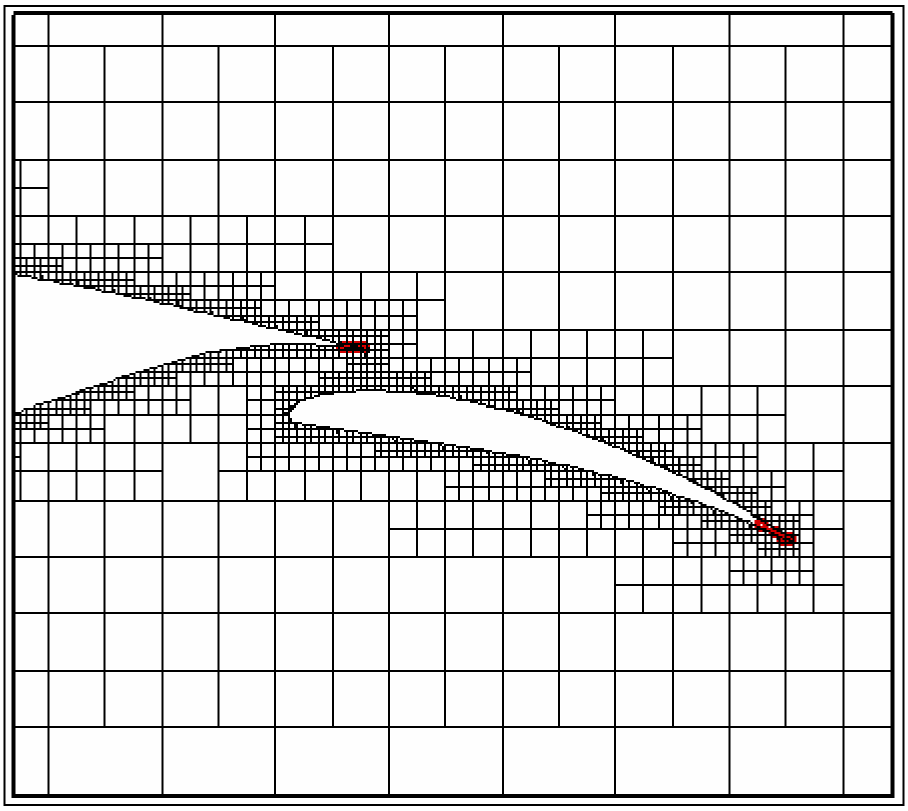


Figure 4.10 2*h* multigrid level around NLR 7301 airfoil and flap

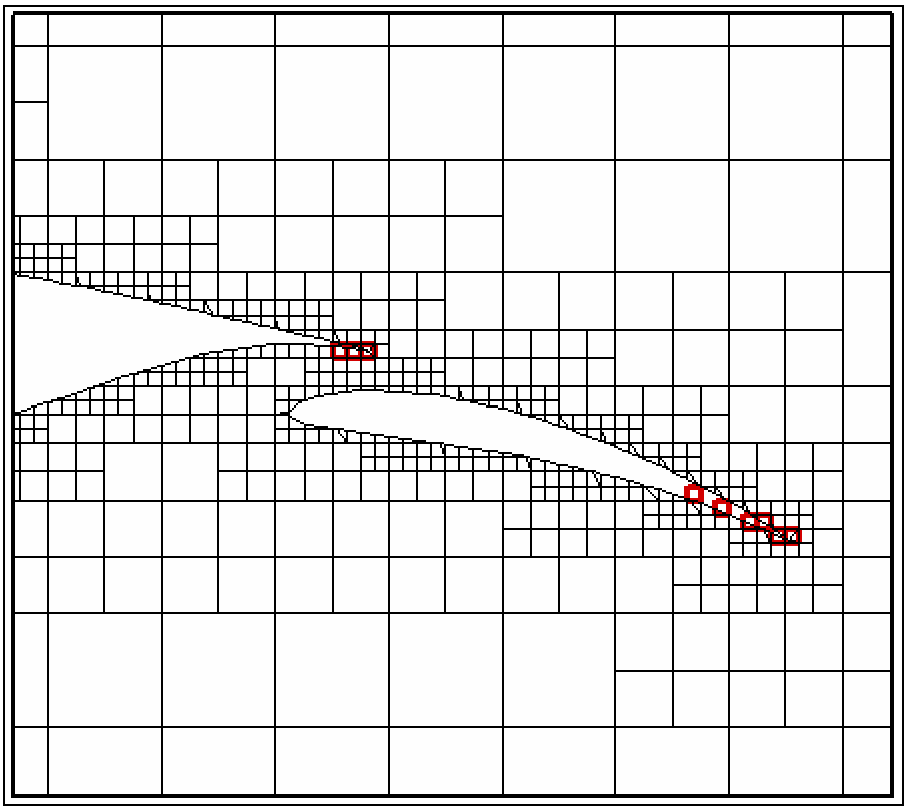


Figure 4.11 4*h* multigrid level around NLR 7301 airfoil and flap

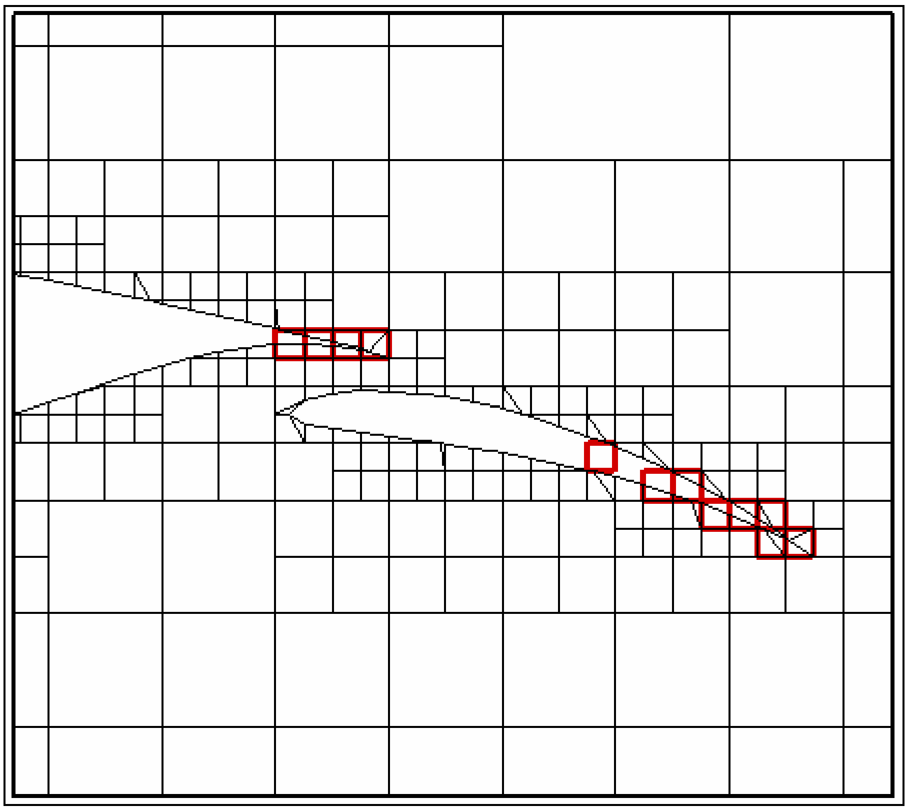


Figure 4.12 8*h* multigrid level around NLR 7301 airfoil and flap

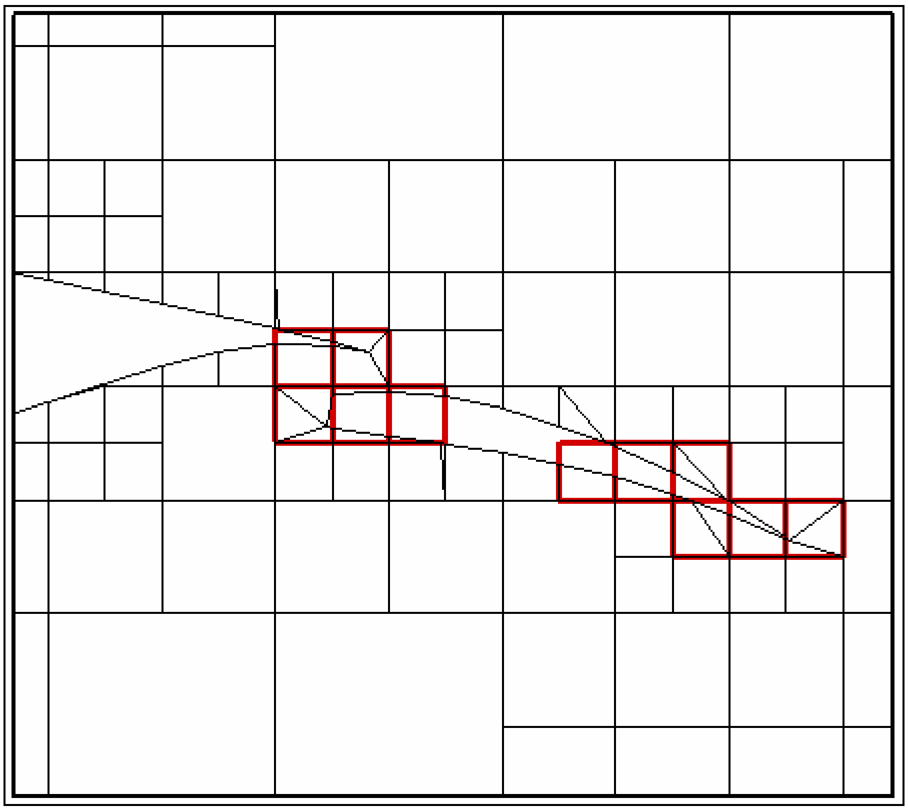


Figure 4.13 16*h* multigrid level around NLR 7301 airfoil and flap

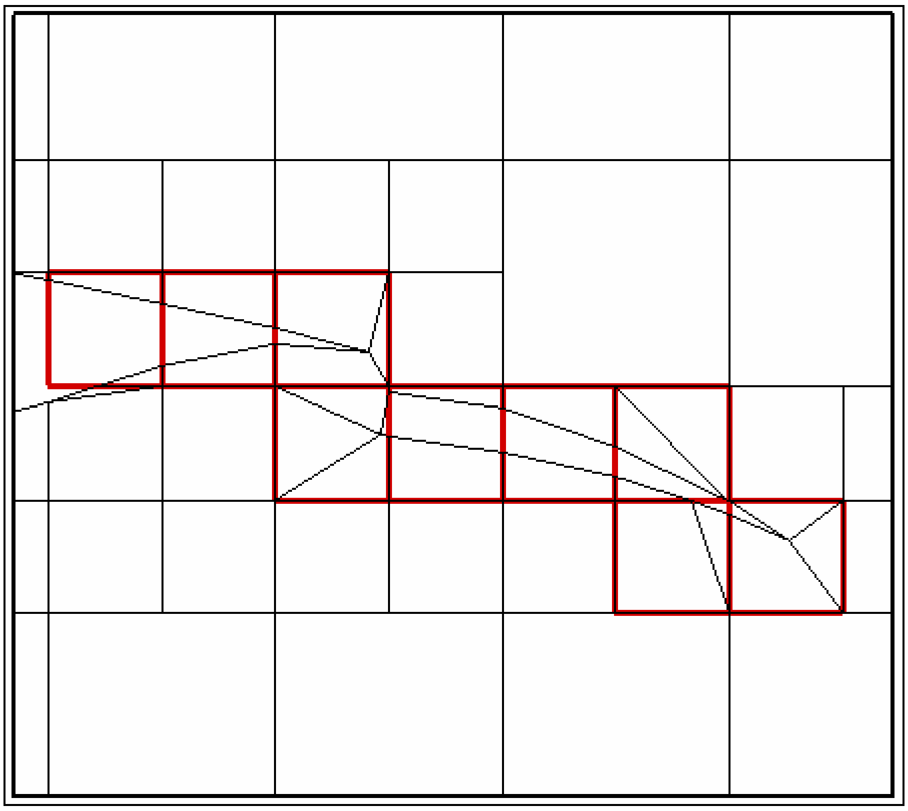


Figure 4.14 32*h* multigrid level around NLR 7301 airfoil and flap

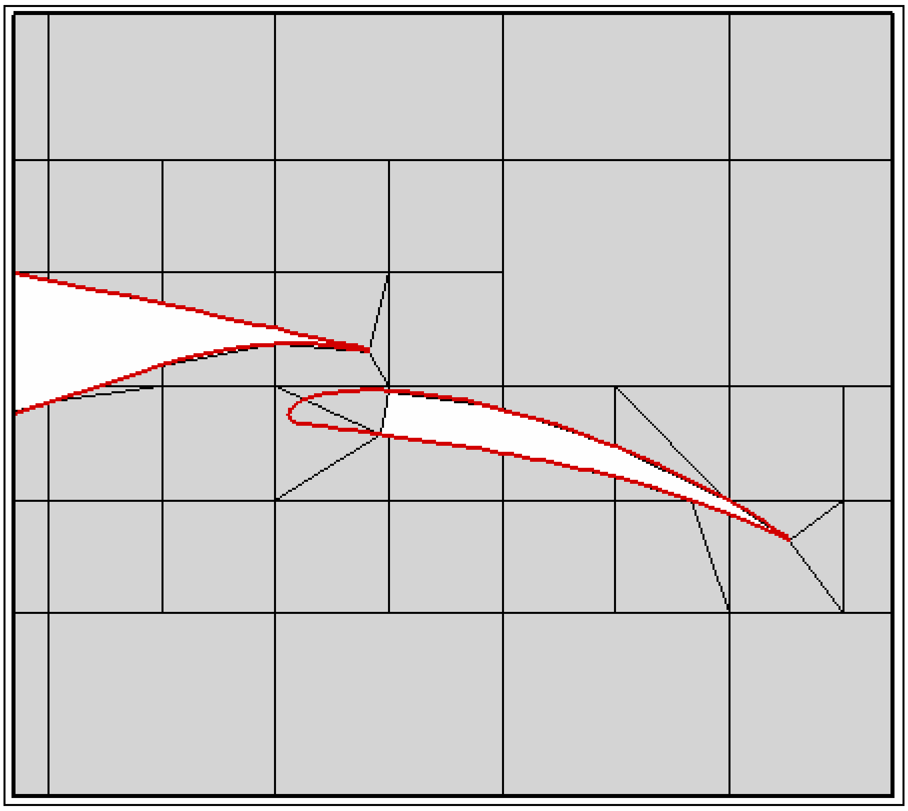


Figure 4.15 The coarsest mesh for six level multigrids in Section 5.1.4

4.5 Reconstruction

As it is mentioned before, flow variables are stored at the centroids of the cells and first order schemes are used to calculate flux calculations through the faces. Reconstruction is required for second order schemes and the determination of cells to be refined and coarsened. Therefore, least squares reconstruction method is used in the developed code to calculate gradients of flow variables in a cell and estimate the value of these variables at a certain point inside the cell.

4.5.1 Least Squares (Minimum Energy) Reconstruction

The following equation is presented the linear reconstruction.

$$\mathbf{w} = \mathbf{w}_c + \frac{d\mathbf{w}}{dx}(x - x_c) + \frac{d\mathbf{w}}{dy}(y - y_c) + \frac{d\mathbf{w}}{dz}(z - z_c) \tag{4.70}$$

where $\mathbf{w}$ is the vector of primitive variables at a certain point in a cell and $\mathbf{w}_c$ is the vector of primitive variables at the centroid of this cell. x_c, y_c and z_c are the coordinates of the centroid.

$$\mathbf{w} = \begin{pmatrix} \rho \\ u \\ v \\ w \\ p \end{pmatrix} \tag{4.71}$$

In Equation (4.70), the only unknown is the gradient of primitive flow variables. In order to find the unknown variables, the following equation is used. The derivation and detailed information regarding this equation can be found in references [28], [29], [11] and [8].

$$\mathbf{L}_{\mathbf{ij}}(\nabla \mathbf{w}_{\mathbf{j}}) = \mathbf{B}_{\mathbf{i}} \tag{4.72}$$

where $\nabla\mathbf{w}$, $\mathbf{L}$ and $\mathbf{B}$ are given below, respectively and this equation can be solved by using Cramer's rule. The subscript, *n,c*, together in the following equations denotes the variables of the neighboring cells.

$$\nabla\mathbf{w} = \begin{bmatrix} d\mathbf{w}/dx \\ d\mathbf{w}/dy \\ d\mathbf{w}/dz \end{bmatrix} \tag{4.73}$$

$$\mathbf{L} = \begin{bmatrix} \sum_{i=1}^{nNeighbors}(x_{n,c}-x_c)_i(x_{n,c}-x_c)_i & \sum_{i=1}^{nNeighbors}(x_{n,c}-x_c)_i(y_{n,c}-y_c)_i & \sum_{i=1}^{nNeighbors}(x_{n,c}-x_c)_i(z_{n,c}-z_c)_i \\ \sum_{i=1}^{nNeighbors}(x_{n,c}-x_c)_i(y_{n,c}-y_c)_i & \sum_{i=1}^{nNeighbors}(y_{n,c}-y_c)_i(y_{n,c}-y_c)_i & \sum_{i=1}^{nNeighbors}(y_{n,c}-y_c)_i(z_{n,c}-z_c)_i \\ \sum_{i=1}^{nNeighbors}(x_{n,c}-x_c)_i(z_{n,c}-z_c)_i & \sum_{i=1}^{nNeighbors}(z_{n,c}-z_c)_i(y_{n,c}-y_c)_i & \sum_{i=1}^{nNeighbors}(z_{n,c}-z_c)_i(z_{n,c}-z_c)_i \end{bmatrix} \tag{4.74}$$

$$\mathbf{B} = \begin{bmatrix} \sum_{i=1}^{nNeighbors}(\mathbf{w}_{n,c}-\mathbf{w}_c)_i(x_{n,c}-x_c)_i \\ \sum_{i=1}^{nNeighbors}(\mathbf{w}_{n,c}-\mathbf{w}_c)_i(y_{n,c}-y_c)_i \\ \sum_{i=1}^{nNeighbors}(\mathbf{w}_{n,c}-\mathbf{w}_c)_i(z_{n,c}-z_c)_i \end{bmatrix} \tag{4.75}$$

4.5.2 Gradient Limiting

The purpose of limiting procedure is to prevent obtaining variables at a certain point in a cell, which exceeds the flow variables of this cell and its neighbors. When a limiter is applied to Equation (4.70), it takes the following form

$$\mathbf{w} = \mathbf{w}_c + \varphi\left[\frac{d\mathbf{w}}{dx}(x - x_c) + \frac{d\mathbf{w}}{dy}(y - y_c) + \frac{d\mathbf{w}}{dz}(z - z_c)\right] \tag{4.76}$$

Limiter value, φ, must be between 0 and 1. The limiter value presented here is taken from the paper written by Barth and Jespersen [29].

$$\varphi = \min(\varphi_1, \varphi_2, \varphi_3, \ldots., \varphi_m, \ldots) \tag{4.77}$$

where

$$\varphi_j = \min\begin{cases} 1 \\ \dfrac{\left|\mathbf{w}_{j,c} - \mathbf{w}_j^{\max}\right|}{\left|\mathbf{w}_{j,c} - \mathbf{w}_j^{\max,cell}\right|} \\ \dfrac{\left|\mathbf{w}_{j,c} - \mathbf{w}_j^{\min}\right|}{\left|\mathbf{w}_{j,c} - \mathbf{w}_j^{\min,cell}\right|} \end{cases} \tag{4.78}$$

and

$$\mathbf{w}^{\max} = \max(\mathbf{w}_c, \mathbf{w}_{ic}) \tag{4.79}$$

$$\mathbf{w}^{\min} = \min(\mathbf{w}_c, \mathbf{w}_{ic}) \qquad \text{i=1,…,nNeighbors} \tag{4.80}$$

The values in Equation (4.78), $\mathbf{w}_j^{\max,cell}$ and $\mathbf{w}_j^{\min,cell}$, are the maximum and minimum values calculated in a cell by using Equation (4.70), respectively. Generally, maximum and minimum primitive variables are obtained at the vertices of the cell.

4.5.3 Solution Refinement and Coarsening

As it is mentioned before, one of the most valuable properties of Cartesian grids is that it enables solution refinement and coarsening. Hence, satisfactory resolution is provided to critical regions such as shock locations and stagnation points by solution refinement. Large gradients at these locations minimizes by refining cells. Moreover, some regions where unnecessary high resolution exists are coarsened.

The criteria used in the developed code are the divergence and curl of the velocity vectors and the strength of the entropy wave [10]. These criteria for each cell are calculated by

$$\tau_{\mathbf{D}} = |\nabla \bullet \mathbf{V}|\Omega^{0.5} \tag{4.81}$$

$$\tau_{\mathbf{C}} = |\nabla \times \mathbf{V}|\Omega^{0.5} \tag{4.82}$$

$$\tau_{\mathbf{EW}} = |\nabla p - c^2 \nabla \rho|\Omega^{0.5} \tag{4.83}$$

Then, the standard deviations of these three criteria are calculated for the whole mesh by the following equation

$$\sigma_\alpha = \sqrt{\frac{\sum_{i=1}^{nCells} (\tau_\alpha)_i^2}{nCells}} \tag{4.81}$$

After the calculations of standard deviations of three criteria, it is time to determine cells to be refined and coarsened. A cell is selected for refinement if $(\tau_\alpha)_i > \sigma_\alpha$ for any α and selected for coarsening if $(\tau_\alpha)_i < 0.1\sigma_\alpha$ for all α.

CHAPTER 5

RESULTS AND DISCUSSIONS

Inviscid Euler flow around single & multi-element airfoils, wing and projectile will be analyzed in this chapter by using the codes which are developed for two and three dimensional problems. Test cases are divided into three groups. The first group is two-dimensional test cases, the second group is for the convergence history investigations with multigrid application and the final group is three-dimensional test cases. The results of these cases are compared with experimental and other numerical results, depending upon availability.

5.1 Two Dimensional Test Problems

In this section, both single and multi-element airfoils are examined. Test problems are tabulated below.

Table 5.2 Two dimensional test problems

Test Problem	Airfoil Profile	M_∞	α	Reference
5.1.1	NACA0012	0.85	1°	[13]
5.1.2	NACA0012	1.2	7°	[13]
5.1.3	RAE2822	0.75	3°	[13]
5.1.4	NLR7301+flap	0.185	6°	[30]
5.1.5	30p30n	0.2	8°	[31]

5.1.1 Problem 1: Transonic Flow about NACA0012 Airfoil

The first problem is the inviscid flow around a NACA0012 airfoil at a Mach number of 0.85 and an angle of attack of 1°. NACA0012 profile is widely used in order to validate developed solvers since the experimental data for many Mach numbers and angle of attacks are found easily in literature. Transonic flow is selected in order to demonstrate that shock locations and surface pressure coefficients can be obtained accurately and effectively by using Cartesian mesh. Importance of solution adaptation is depicted by comparing solutions with and without solution adaptation. As it is mentioned, solution adaptation enables to resolve high gradient regions by automatic meshing without increasing total number of cells extremely. The far-field boundary is approximately located 25 chords ahead of the airfoil similar to the reference case. The developed flow solver is iterated until the average density residual reaches 10^{-12}. For the computed solutions, six levels of meshes are used in the multigrid method. Table 5.2 gives the lift, drag coefficients, total number of cells and convergence histories for the results which are computed by the developed solver and extracted from reference [13]. The numerical solutions which are extracted from reference [13] are obtained by using Euler equations.

Table 5.2 Comparison of results for transonic flow around NACA0012 airfoil at $M_\infty = 0.85$ and $\alpha = 1$

	C_L	C_D	# of cells	Time (s)
Results from reference [13]	0.3584	0.058	20480	-
Results with solution adaptation	0.3219	0.0611	18641	1012
Results without solution adaptation	0.2361	0.0763	3538	68

As seen from the table 5.2, computed lift and drag coefficients of the solution-adapted case are in agreement with the results in reference [13]. Solution-adapted case underestimates the lift coefficient by 10 % and overestimates the drag coefficient by 5 %. In order to visualize the benefit of automatic solution adaptation in Cartesian grid, geometric adapted-grid and both geometric and solution-adapted grid are shown in Figures 5.1 and 5.2, respectively. Three refinement cycles are applied to the grid in Figure 5.1 to obtain the grid in Figure 5.2.

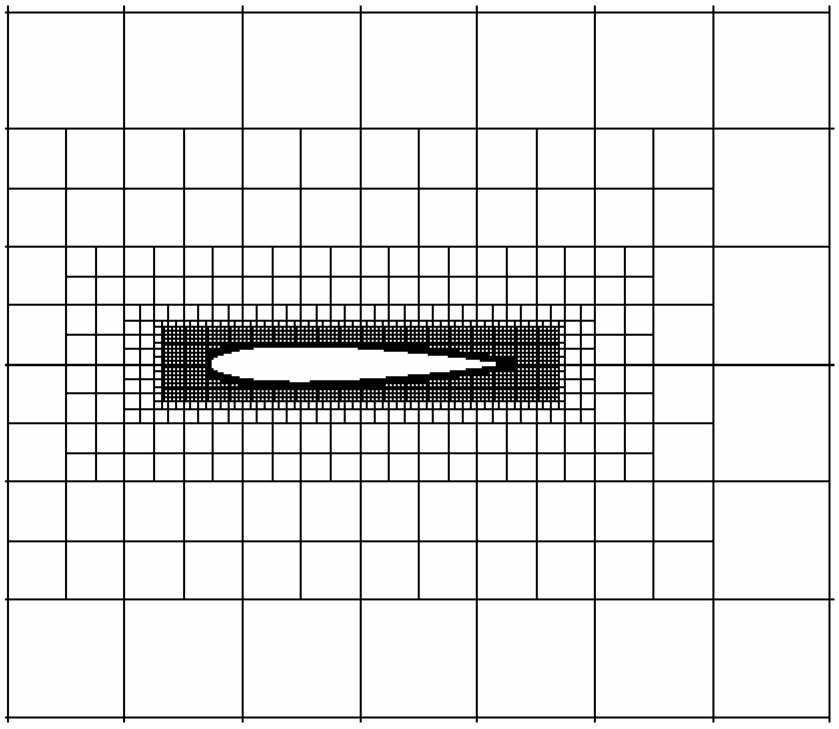


Figure 5.1 Geometric-adapted grids around NACA 0012

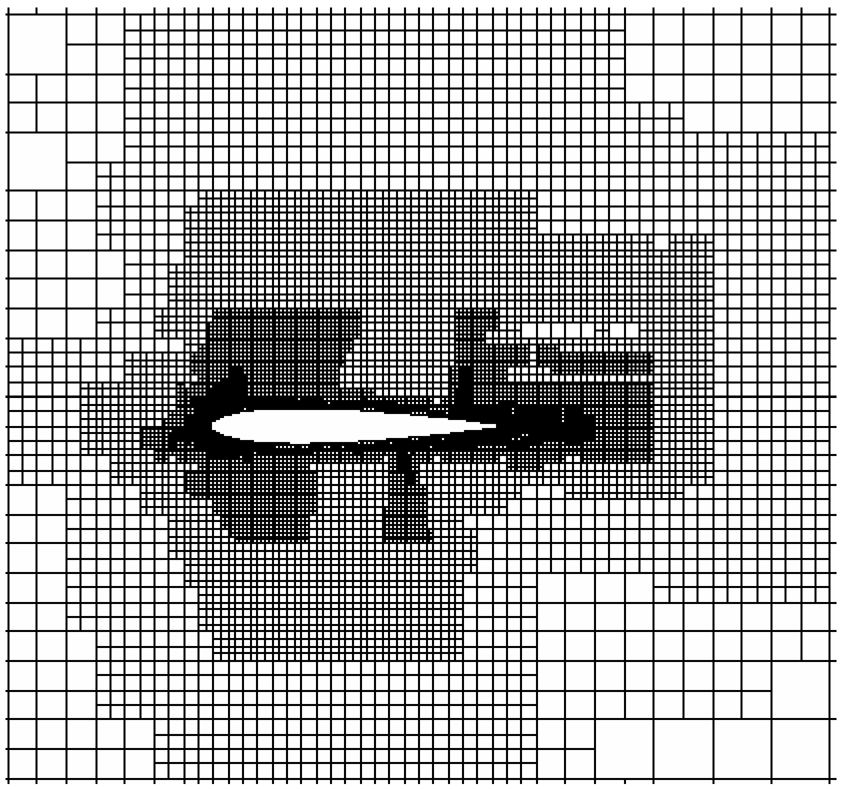


Figure 5.2 Both geometric and solution-adapted grids around NACA 0012

Pressure coefficient distribution is shown in Figure 5.3. Pressure contours of solutions with and without solution adaptation are depicted in Figures 5.4 and 5.5, respectively. Mach contours of solution-adapted case and extracted case from reference [13] are shown in Figures 5.6 and 5.7, respectively. It is clearly seen in Figure 5.3 that both upper and lower shock locations are captured well for the solution-refined case. Approximately 1 % chord length error is seen for both upper and lower shocks according to the reference solution. As seen in Figures 5.3 and 5.5, lower shock cannot be captured for the geometric-adapted grid. Maximum pressure loss occurs at the downstream of the shocks. As seen in Figures 5.6 and 5.7, Mach contours are alike. Besides, Mach number reaches to 1.35 just before the upper shock wave.

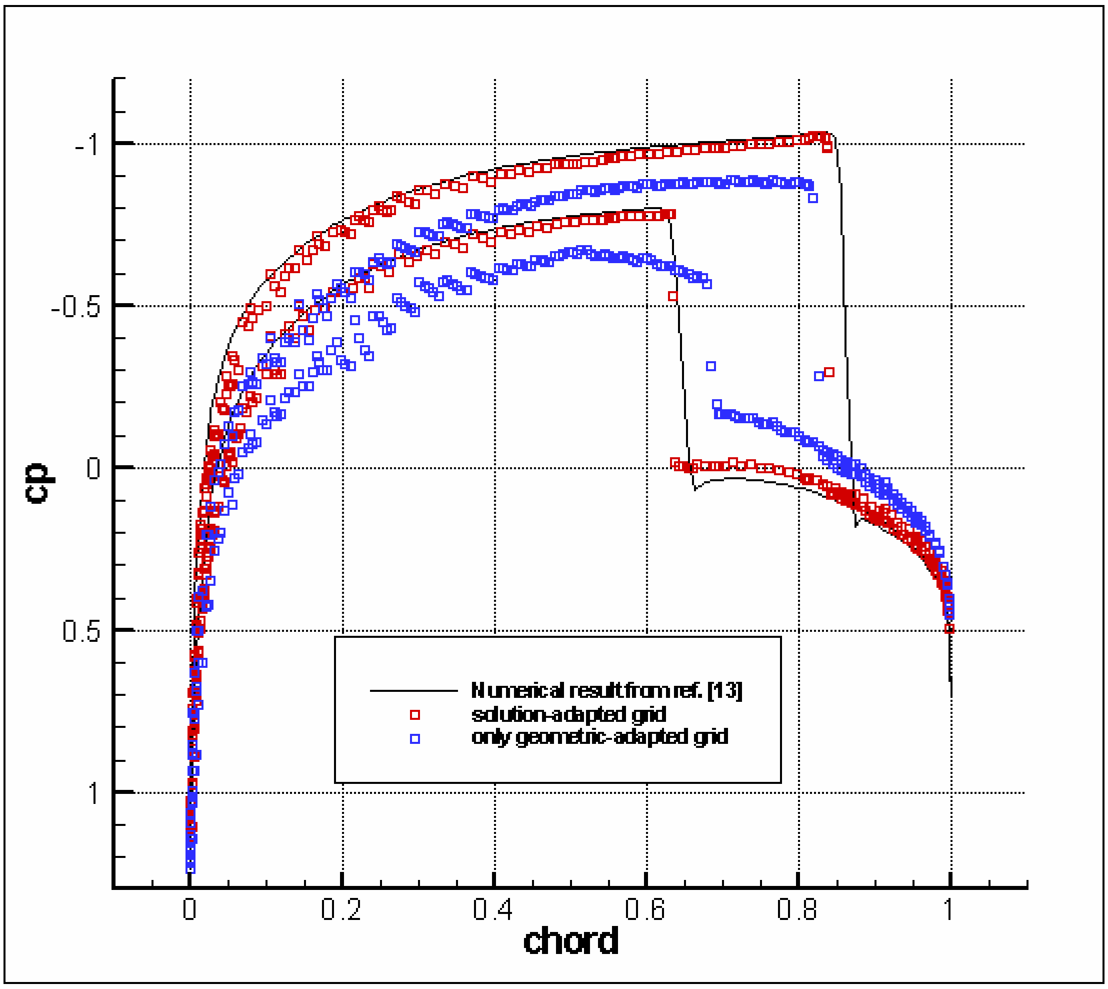


Figure 5.3 Pressure coefficient distribution on NACA0012 airfoil at $M_\infty = 0.85$ and $\alpha = 1$

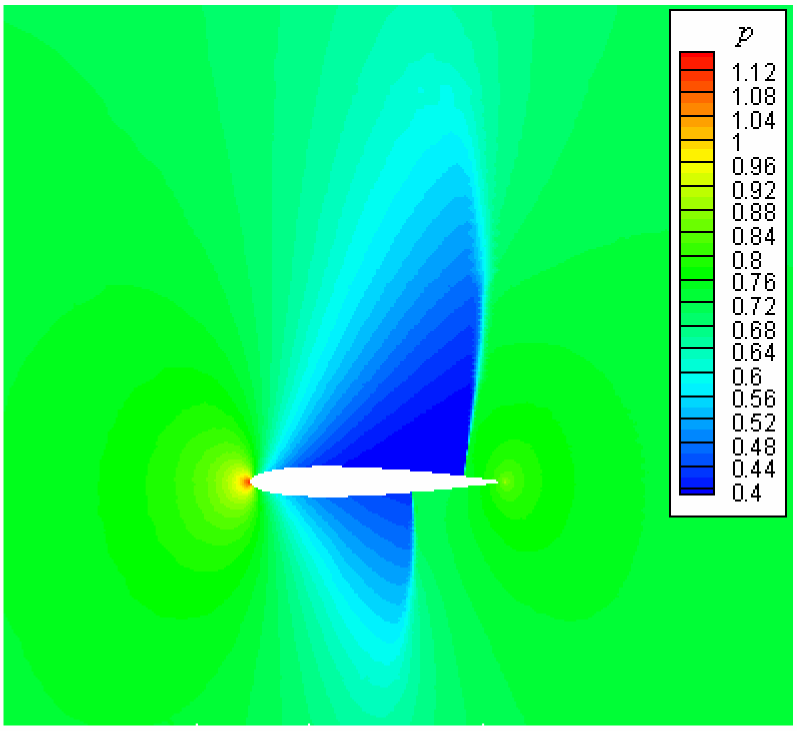


Figure 5.4 Pressure contours of the solution-adapted grid on NACA0012 airfoil at $M_\infty = 0.85$ and $\alpha = 1$

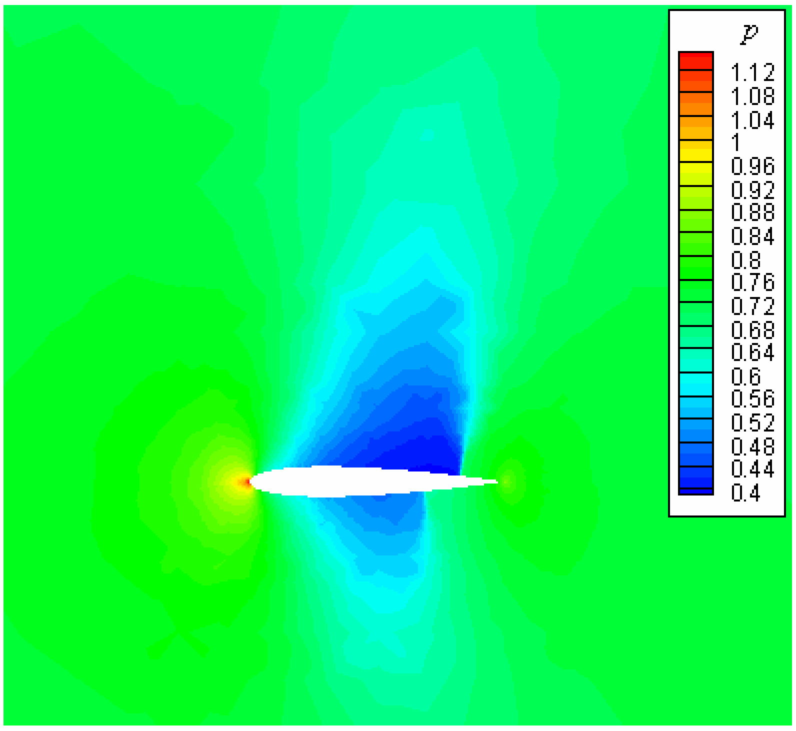


Figure 5.5 Pressure contours of the grid without solution adaptation on NACA0012 airfoil at $M_\infty = 0.85$ and $\alpha = 1$

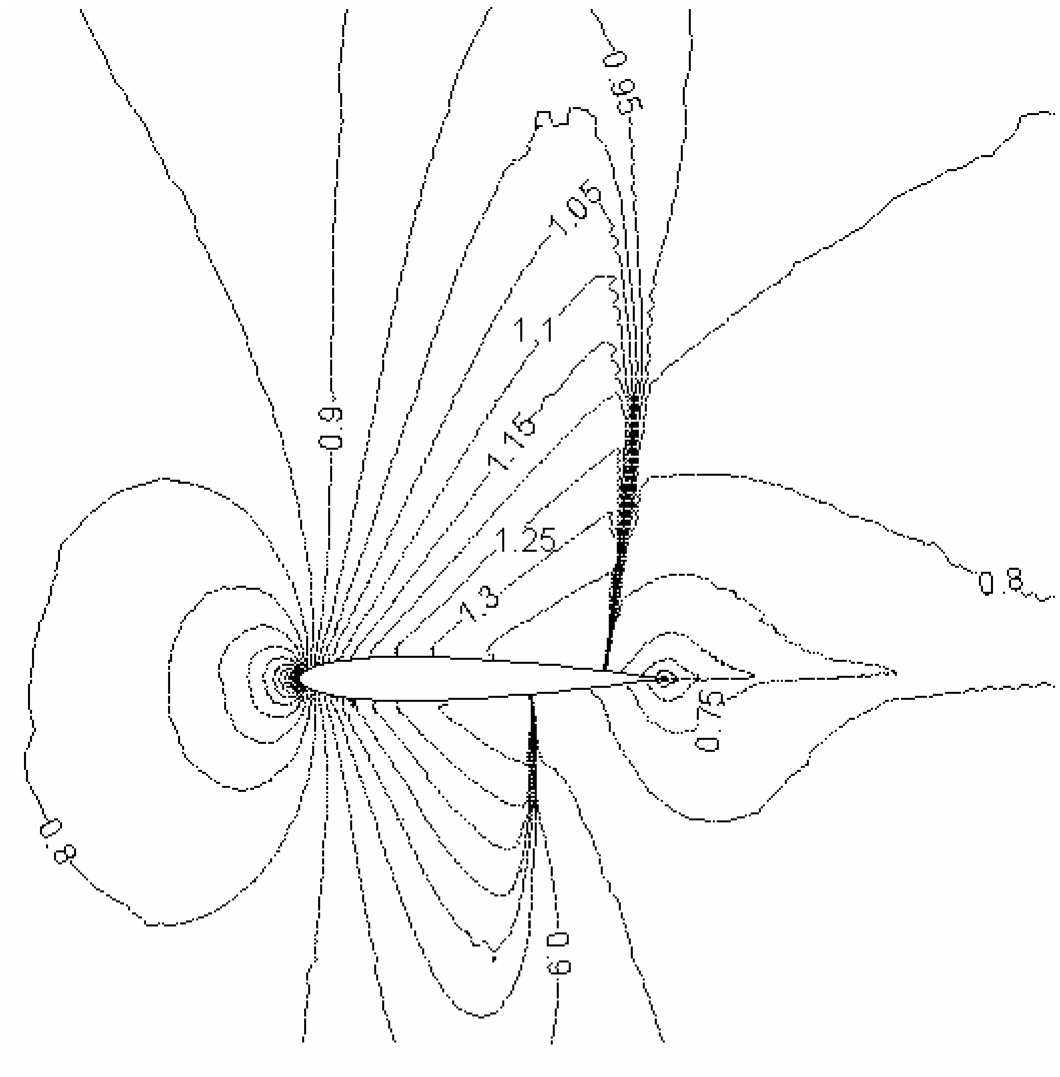


Figure 5.6 Mach contours of the solution-adapted grid on NACA0012 airfoil at M_∞ = 0.85 and α = 1

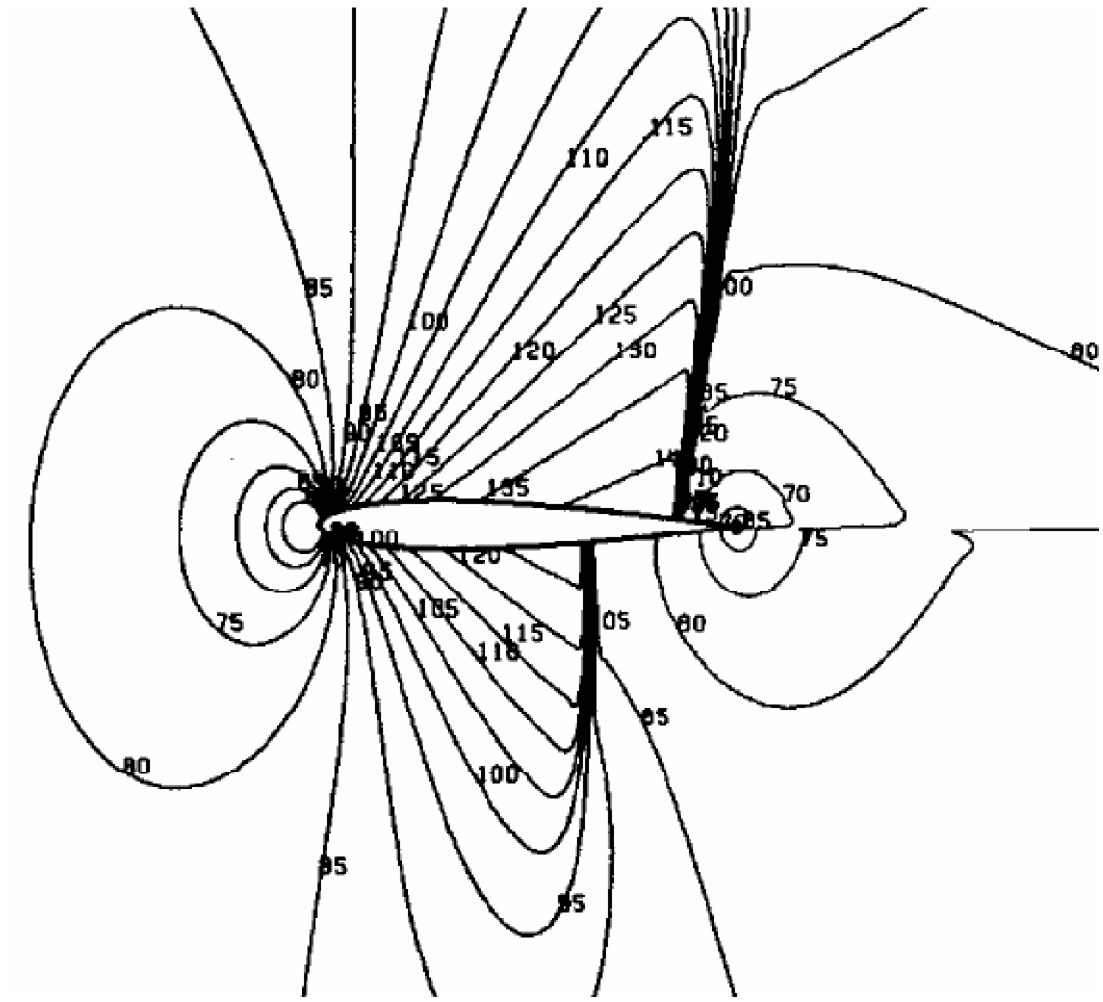


Figure 5.7 Mach contours in reference [13] on NACA0012 airfoil at $M_\infty = 0.85$ and $\alpha = 1$

5.1.2 Problem 2: Supersonic Flow about NACA0012 Airfoil

The second problem is the flow around NACA0012 airfoil at a Mach number of 1.2 and angle of attack of 7°. The purpose of this test case is to analyze whether the developed solver captures bow and oblique shocks accurately. First and second order schemes are used to calculate fluxes.

For all test cases in this section, the far-field boundary is approximately located 15 chords ahead of the airfoil. The developed flow solver is iterated until the average density residual reaches 10^{-12}. Multigrid method is not used due to some problems regarding second order scheme in the developed solver. Two refinement cycles are applied to the grids. Table 5.3 gives the lift, drag coefficients, total number of cells and convergence history for results which are computed by the developed solver and extracted from reference [13].

Table 5.3 Comparison of results for supersonic flow around NACA0012 $M_\infty = 1.2$ and $\alpha = 7°$

Case #	Descriptions of Test Cases	C_L	C_D	# of cells	Time (s)
Case-1	First Order Scheme with AUSM	0.5236	0.162	11761	887
Case-2	Second Order Scheme with AUSM without limiter	0.5216	0.156	13784	23241
Case-3	Second Order Scheme with AUSM with limiter	0.5217	0.1556	14081	16515
Case-4	First Order Scheme with Roe's flux differencing method	0.5201	0.1614	11717	1335
Case-5	Second Order Scheme with Roe's flux differencing method without limiter	0.5219	0.156	13561	32211
Case-6	Second Order Scheme with Roe's flux differencing method with limiter	0.5227	0.1555	13796	18733
Case-7	Results from reference [13]	0.5196	0.1543	10209	-

As seen from Table 5.3, computed lift and drag coefficients of the test cases are in agreement with the results in reference [13]. For example, the third case overestimates the lift coefficient by 0.4 % and overestimates the drag coefficient by 0.8 %. Pressure coefficient distributions of these cases are shown in Figures 5.8 and 5.9. Although pressure coefficient distributions are slightly different from each other, there are visible differences in pressure contours of the computed solutions, as seen in Figure 5.10.

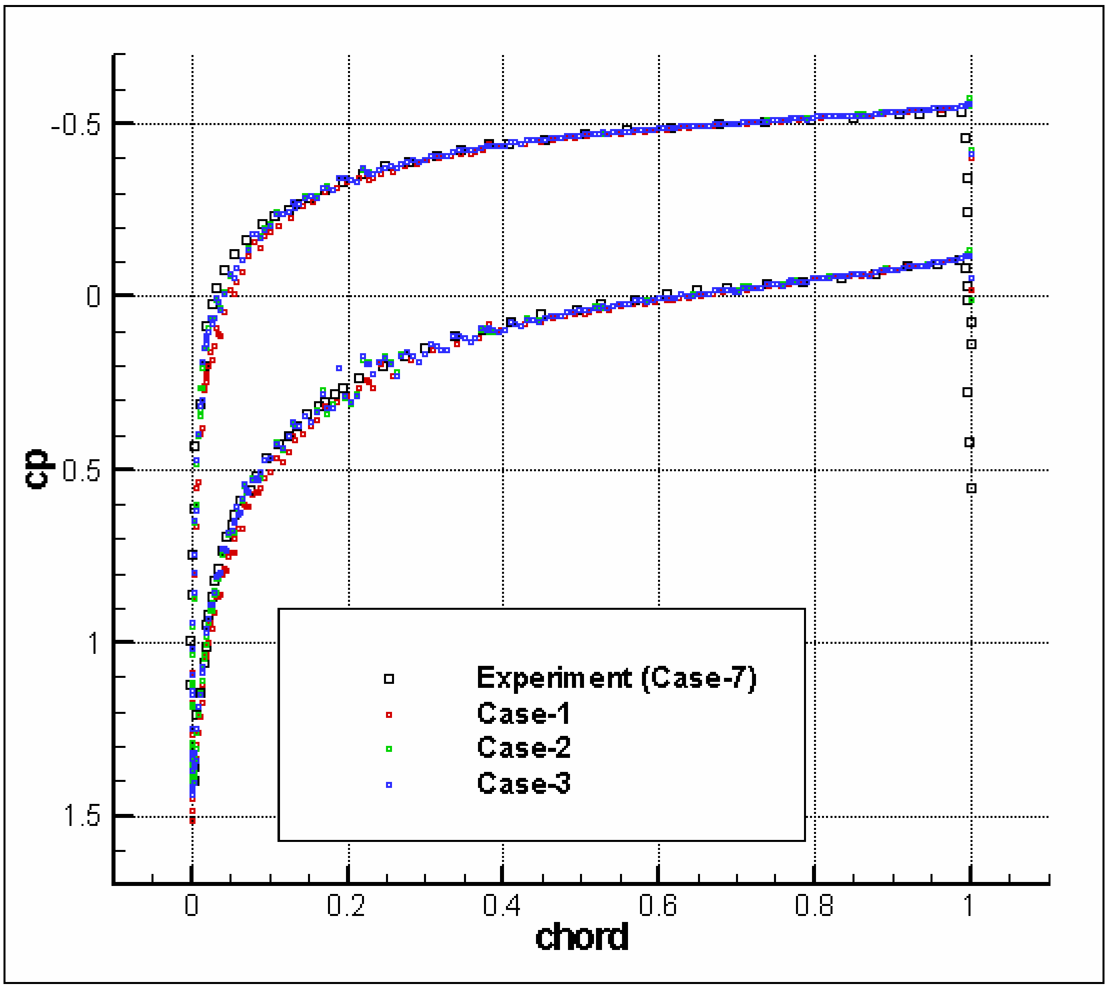


Figure 5.8 Pressure coefficient distributions for 1st, 2nd and 3rd test cases on NACA0012 airfoil at $M_\infty = 1.2$ and $\alpha = 7°$

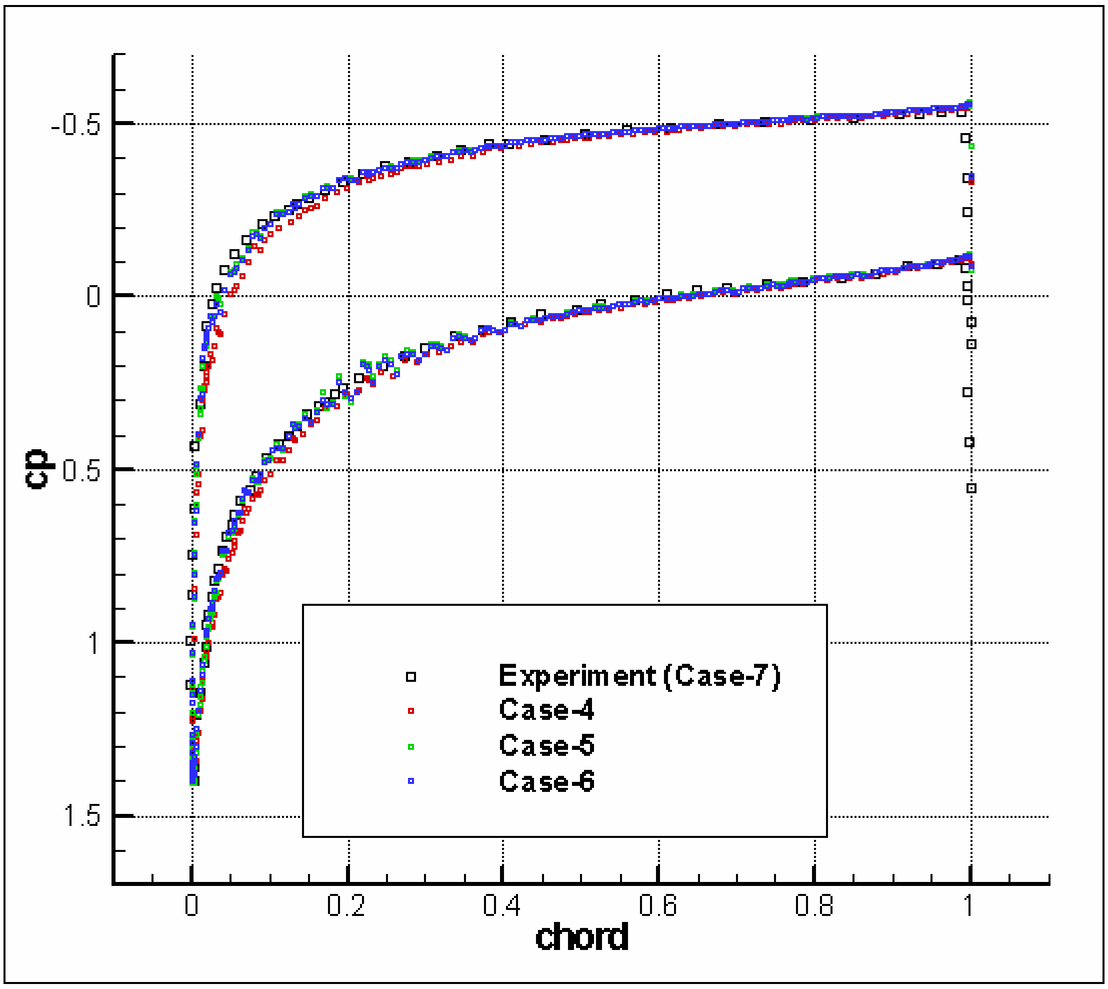


Figure 5.9 Pressure coefficient distributions for 4th, 5th and 6th test cases on NACA0012 airfoil at $M_\infty = 1.2$ and $\alpha = 7°$

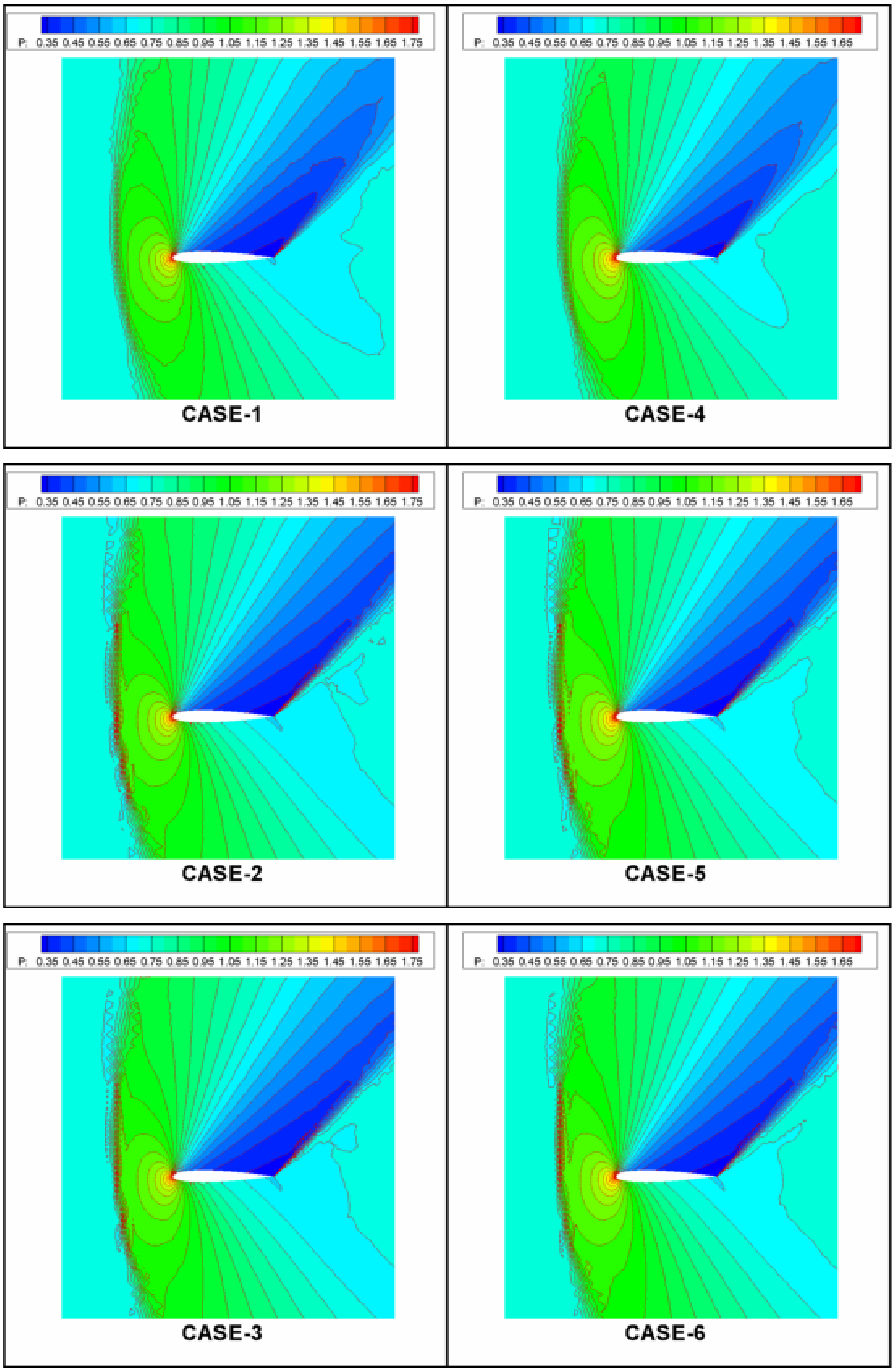


Figure 5.10 Pressure contours around NACA0012 at $M_\infty = 1.2$ and $\alpha = 7°$

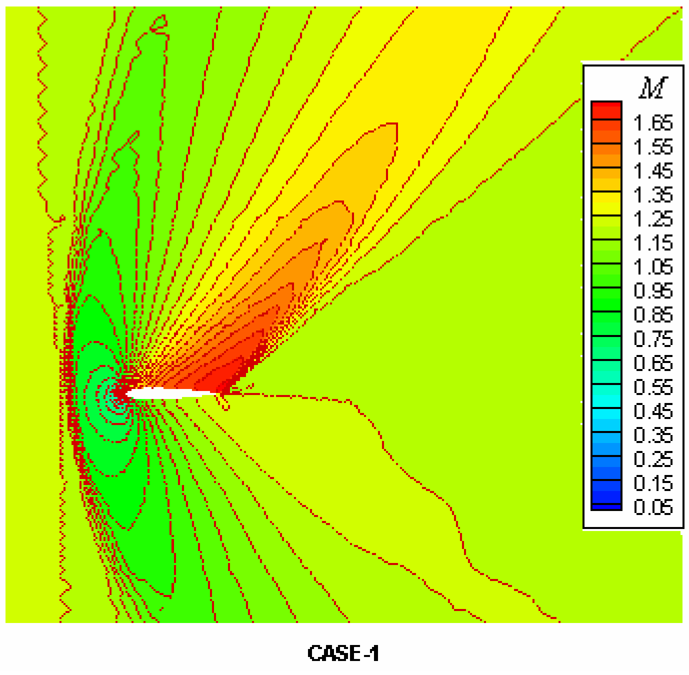


Figure 5.11 Mach contours of test case-1 around NACA0012 at $M_\infty = 1.2$ and $\alpha = 7°$

The reason of the differences in pressure contours is that there is pollution in second order schemes, especially ones without limiter. As seen in Figures 5.10 and 5.11, a strong bow shock exists before the leading edge of the airfoil and strength of this shock becomes weaker near the far-field region of the domain. Although bow shock is captured in second order schemes, pollution which is called carbuncle instability exists in the computed solutions [32]. This instability affects the region in front of the bow shock and the most visible pollution exists in the second order schemes without limiters (case-2 and case-5). Since the limiters damp the oscillations in the solution, absence of limiter causes more pollution. The bow shock is approximately located at a 55 % chord length distance away from the leading edge.

5.1.3 Problem 3: Transonic Flow about RAE2822 Airfoil

The third problem is the flow around RAE2822 airfoil at a Mach number of 0.75 and angle of attack of 3°. The purpose of this test case is to analyze flow around a non-symmetric airfoil and to stress the importance of number of refinement cycles. For all test cases in this section, the far-field boundary is approximately located 10 chords ahead of the airfoil. The developed flow solver is iterated until the average density residual reaches 10^{-12}. For the computed solutions, six levels of meshes are used in the multigrid method. Table 5.4 gives the lift, drag coefficients, total number of cells and convergence history for results which are computed by the developed solver and extracted from reference [13].

Table 5.4 Comparison of results for transonic flow around RAE 2822 airfoil at $M_\infty = 0.75$ and $\alpha = 3°$

Case #	Descriptions of Test Cases	C_L	C_D	# of cells	Time (s)
Case-1	No Solution Refinement	0.7446	0.0705	2308	36
Case-2	One Refinement Cycle	0.8573	0.0546	4848	52
Case-3	Two Refinement Cycles	0.9214	0.0476	10029	163
Case-4	Three Refinement Cycles	0.9538	0.046	18492	428
Case-5	Four Refinement Cycles	0.9731	0.0446	32268	1477
Case-6	Five Refinement Cycles	0.9881	0.0444	54254	3478
Case-7	Results from reference [13]	1.1044	0.0448	20480	-

As seen from Table 5.4, computed lift and drag coefficients of the test cases are directly proportional to the number of refinement cycles. For example, the sixth case underestimates the lift coefficient by 10.5 % and underestimates the drag coefficient by 0.9 %.As the number of refinement cycle increases, the differences between the computed and reference coefficients decreases. Pressure coefficient distributions of these cases are shown in Figure 5.12.

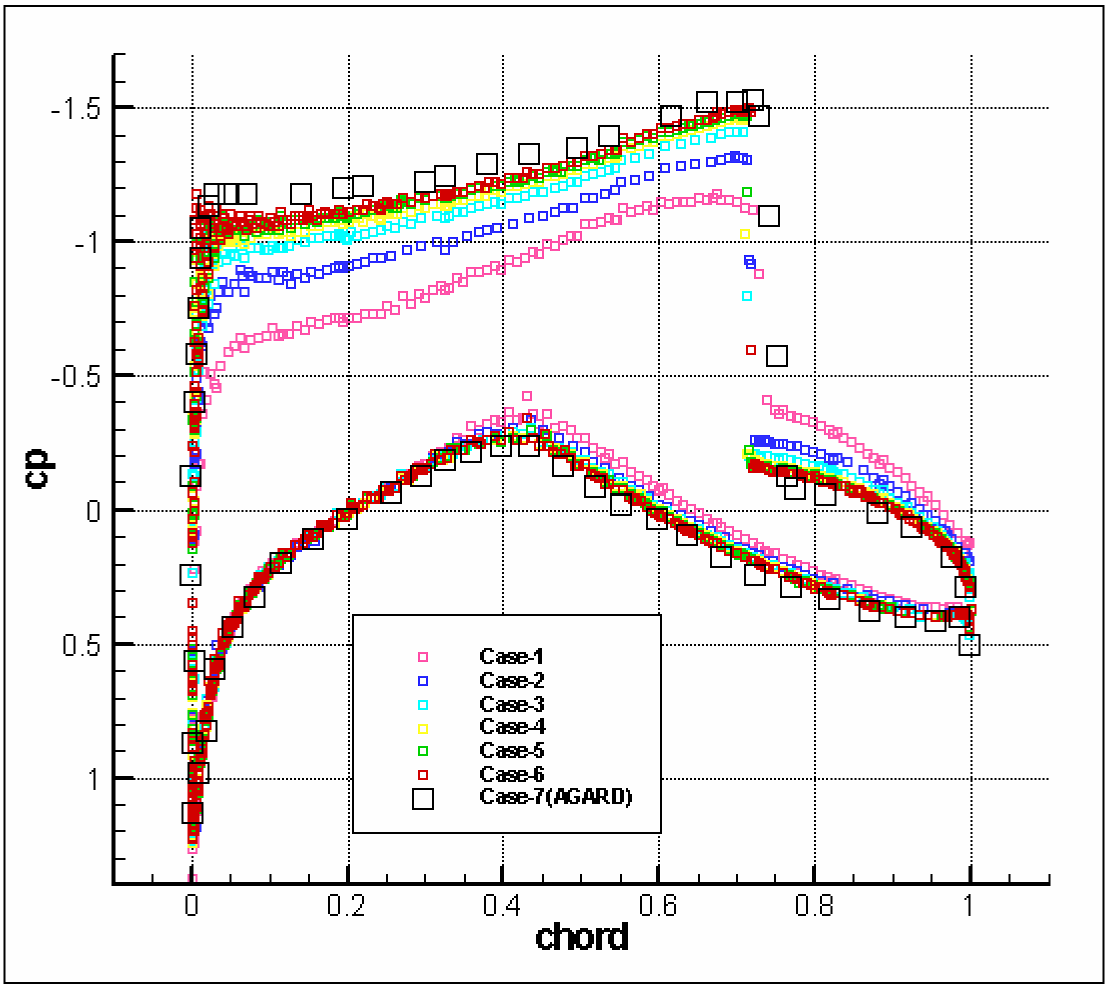


Figure 5.12 Pressure coefficient distributions on RAE 2822 airfoil at $M_\infty = 0.75$ and $\alpha = 3°$

As seen in Table 5.4 and Figure 5.12, the best results are the 5th and 6th cases where numbers of refinement cycles are four and five, respectively. It is important to note that when the number of refinement cycle exceeds five, very slight difference is observed for the drag and lift coefficients although the convergence time increases extremely. In other words, finer grids do not improve the solution accuracy but they result in longer computations. Therefore, the optimum refinement cycle number for this flow is five. Pressure and Mach contours of the 6th case are given in Figures 5.13 and 5.14, respectively.

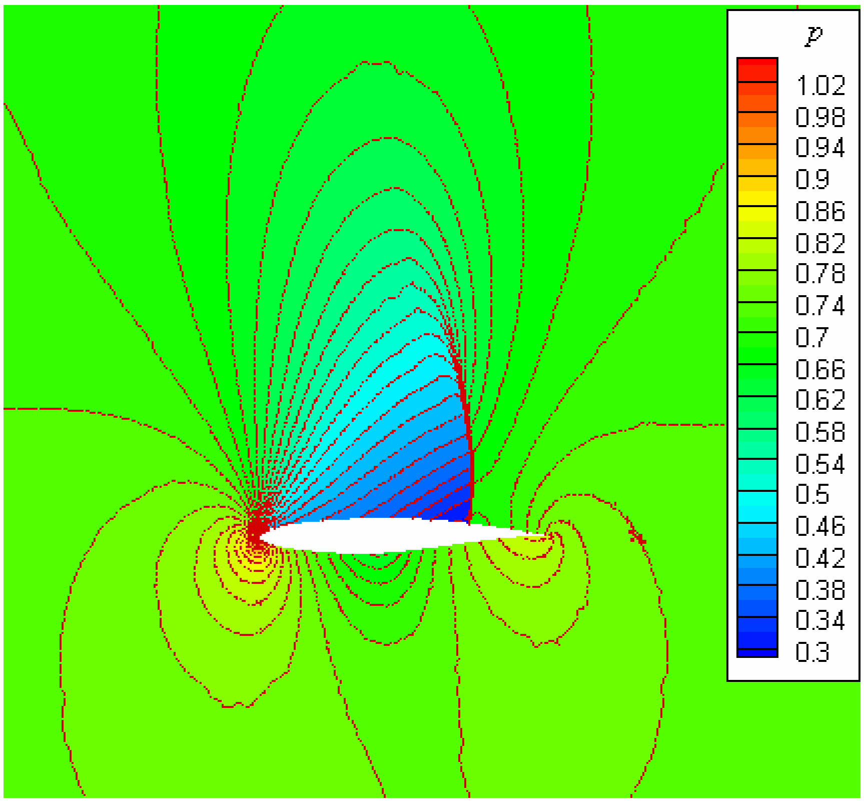


Figure 5.13 Pressure contours of case-6 on RAE 2822 airfoil at $M_\infty = 0.75$ and $\alpha = 3°$

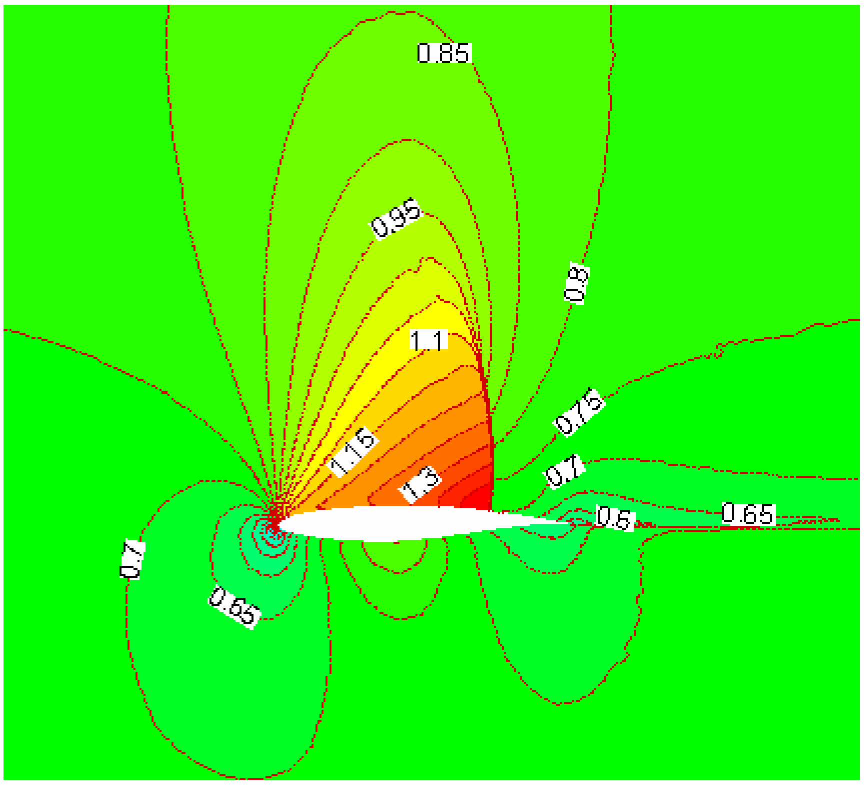


Figure 5.14 Mach contours of case-6 on RAE 2822 airfoil at $M_\infty = 0.75$ and $\alpha = 3°$

As seen in Figure 5.14, Mach number just before the upper shock reaches to 1.4. In addition, shock wave on the upper surface of the airfoil is accurately captured for the 6th case since capturing the shock accurately depends on using finer mesh around the shock. The grid, which is used for 6th case, is shown in Figure 5.15. Finer meshes around the shock are easily seen in this figure. As it is seen in Figure 5.12, results for the lower surface pressure coefficient distributions are quite successful. However, the upper surface pressure coefficient distributions around the leading edge region for the 6th case are slightly underestimated.

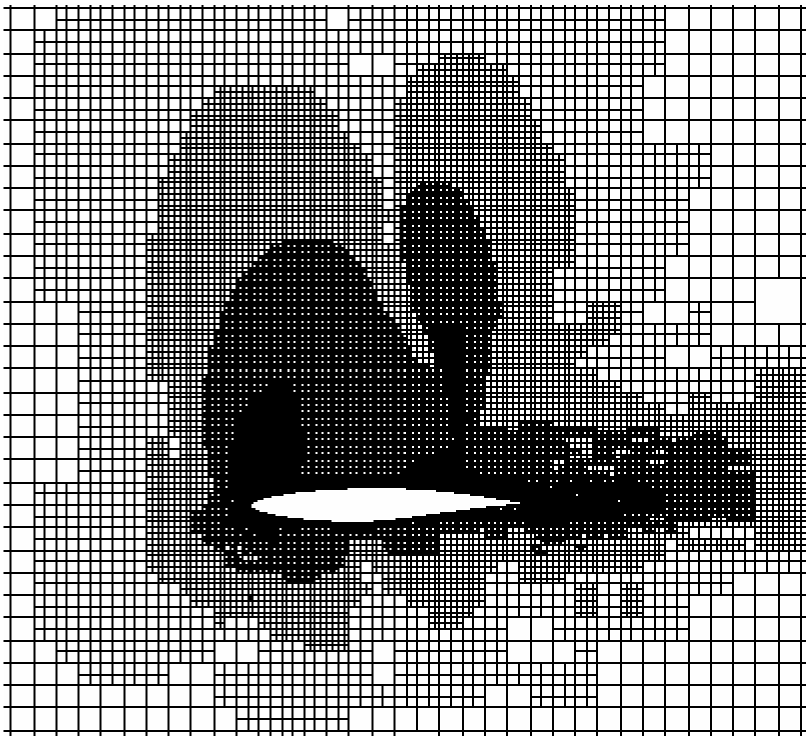


Figure 5.15 Grid used in the 6th case around RAE 2822 airfoil

5.1.4 Problem 4: Subsonic Flow about a Two-element Airfoil

The fourth problem is the flow around NLR7301 airfoil and flap at a Mach number of 0.185 and angle of attack of 6°. The purpose of this test case is to analyze flow around a multi-element airfoil. The far-field boundary is approximately located 15 chords ahead of the airfoil. The developed flow solver is iterated until the average density residual reaches 10^{-12}. For the computed solution, six levels of meshes are used in the multigrid method. The computed lift and drag coefficients are 1.49 and 0.1481, respectively. The comparison between the calculated and experimental [30] pressure coefficient distributions are given in Figure 5.16.

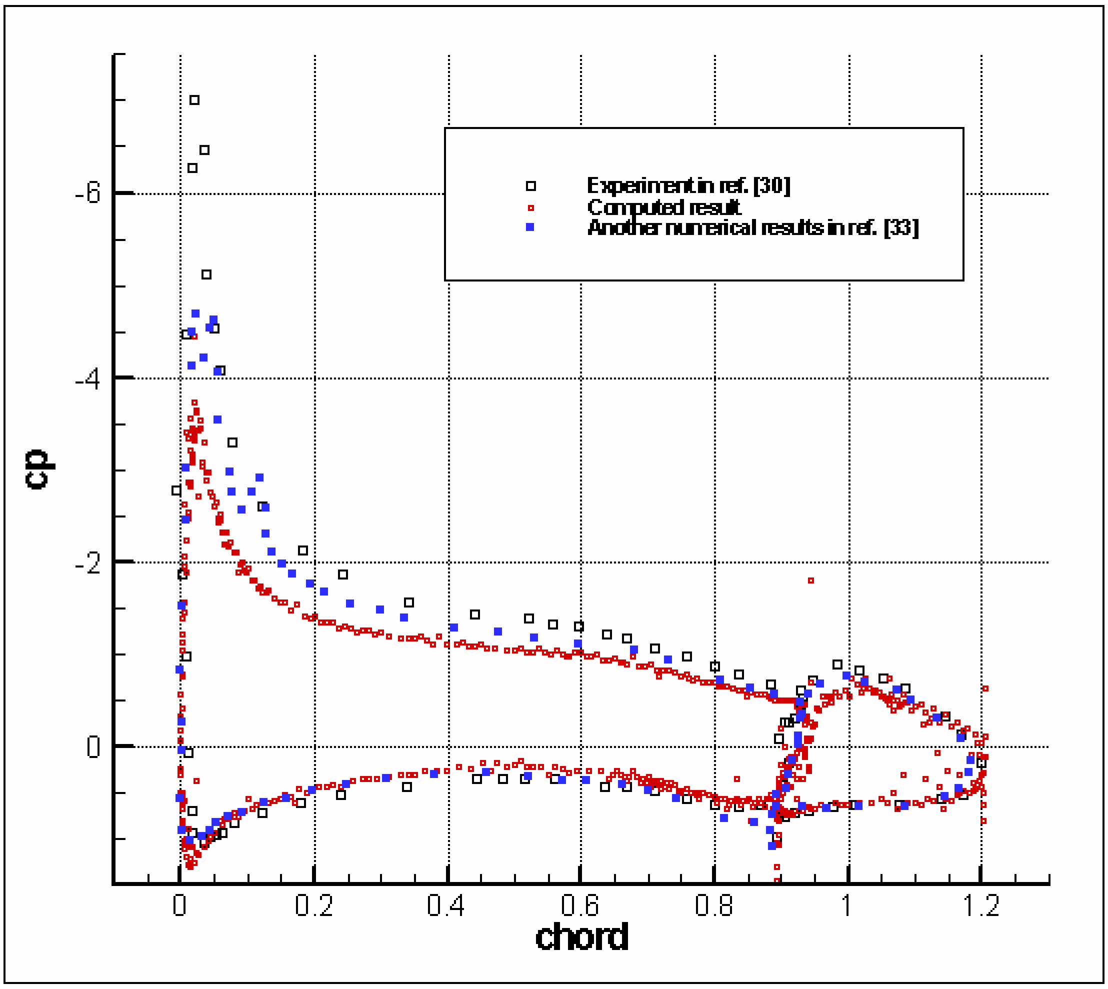


Figure 5.16 Pressure coefficient distribution on two-element airfoil at $M_\infty = 0.185$ and $\alpha = 6°$

It is clearly seen in Figure 5.16 that the peak on the upper surface is not captured in this case. Hence, the underestimation of pressure coefficient results in lower lift coefficient. One of the reason is the flow regime is not suitable for the code since this flow is not compressible, it is an incompressible flow. Both the computed solution and other numerical solution [33] cannot capture the peak although the other numerical results are obtained from the laminar flow solver. Finally, Mach contours are depicted in Figure 5.17.

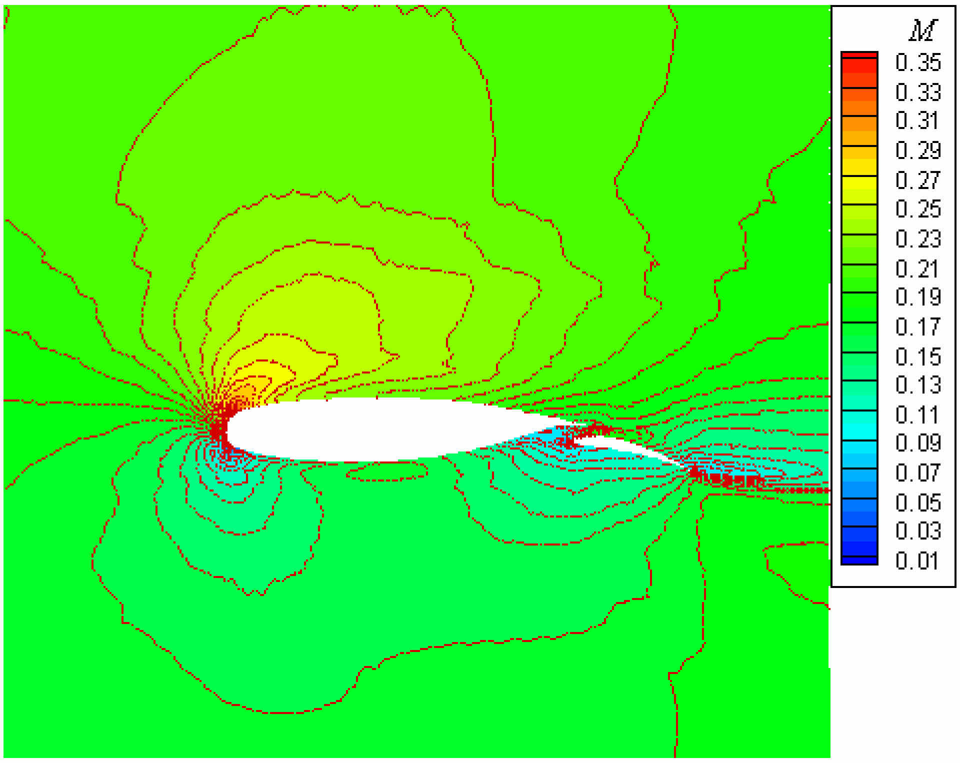


Figure 5.17 Mach contours of two-element airfoil at $M_\infty = 0.185$ and $\alpha = 6°$

5.1.5 Problem 5: Subsonic Flow about a Three-element Airfoil

The fifth problem is the flow around three-element airfoil at a Mach number of 0.2 and angle of attack of 8°. The far-field boundary is approximately located 15 chords ahead of the airfoil. The developed flow solver is iterated until the average density residual reaches 10^{-12}. For the computed solution, four levels of meshes are used in the multigrid method. The computed lift and drag coefficients are 1.28 and 0.1954, respectively. The comparison between the calculated and experimental pressure coefficient distributions are given in Figure 5.18. The Reynolds number for the experimental case is $9\text{x}10^{-6}$. As it is seen in this figure, pressure coefficient distribution of the first element of the geometry cannot be captured due to the very low far-field Mach number. In addition, upper surface pressures are underestimated.

Solution-adapted meshes around the geometry and the computed Mach contours are depicted in Figures 5.19 and 5.20, respectively.

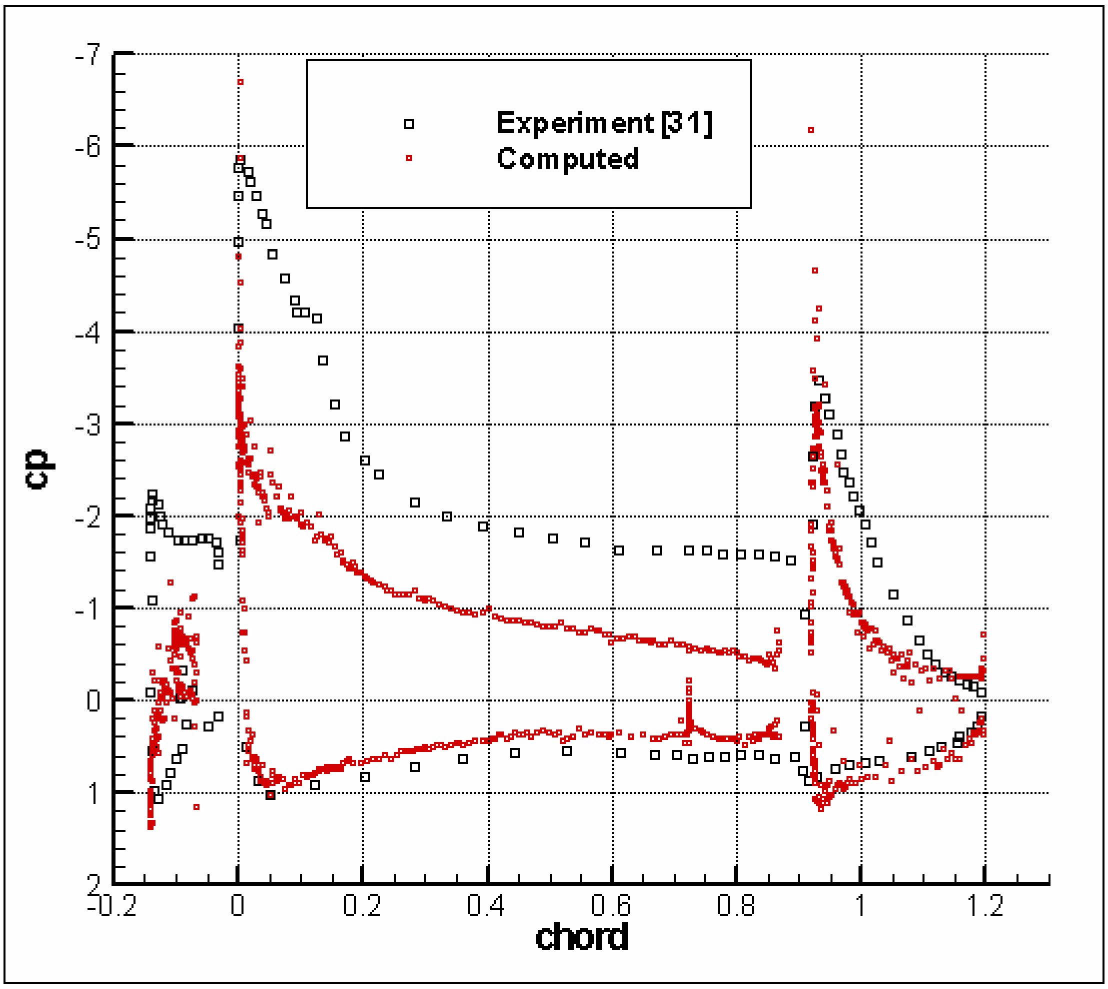


Figure 5.18 Pressure coefficient distributions on three-element airfoil at $M_\infty = 0.2$ and $\alpha = 8°$

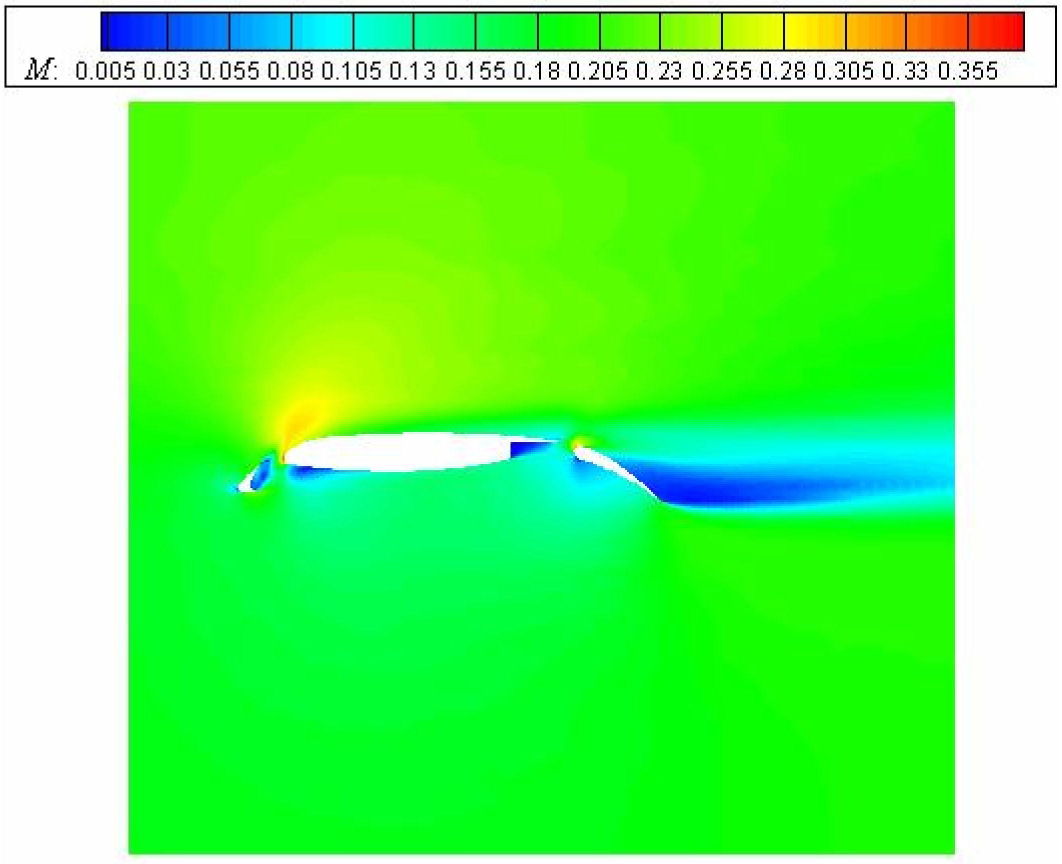
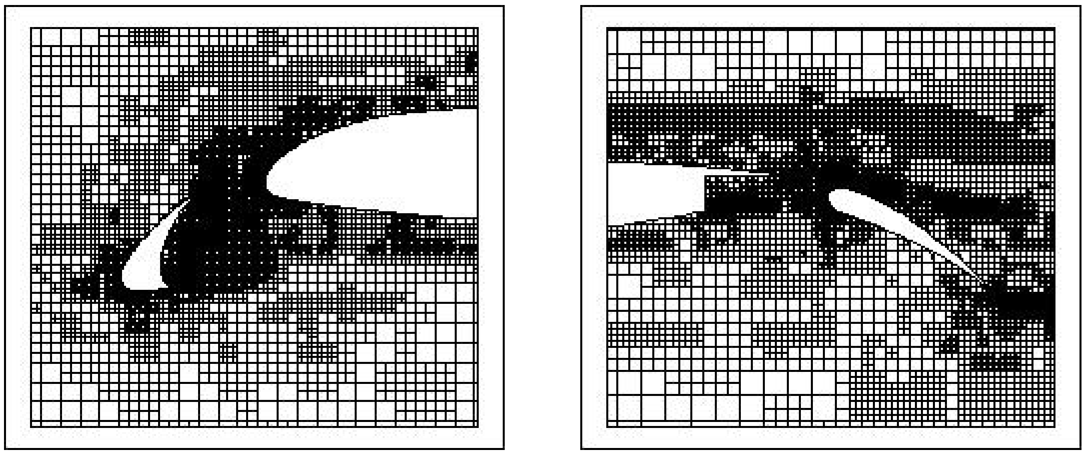


Figure 5.19 Mach contours of three-element airfoil at $M_\infty = 0.2$ and $\alpha = 8°$



(a) Meshes around the slat

(b) Meshes around the flap

Figure 5.20 Solution-adapted meshes around the three-element airfoil

5.2 Convergence History with Multigrid Applications

All test cases in this section are performed for a flow around NACA0012 airfoil at a Mach number of 0.85 and angle of attack of 1° which is examined in the previous section. Since the developed solver for this flow condition is already validated, it is time to examine convergence histories for different aspects of multigrid application such as using different multigrid cycles, levels of meshes or different prolongation operators. First of all, performance of saw-tooth and V-cycles are examined. Then, the performances of the injection and gradient prolongation operators are discussed. Afterwards, the optimum number of steps, i.e. total number of mesh levels, is determined. Finally, optimum iteration numbers on each step are found. Table 5.5 summarizes all the test cases.

Table 5.5 Test cases for multigrid application

Case-1	Determination of multigrid cycle for no solution refinement
Case-2	Determination of multigrid cycle for solution refinement
Case-3	Determination of prolongation operator for solution refinement
Case-4	Determination of the optimum number of multigrid levels
Case-5	Determination of iteration numbers for each multigrid level

Geometric-adapted and both geometric and solution-adapted grids, which are used in this section, are given in Figures 5.21 and 5.22, respectively. Effects of multigrid method on the 1st and 2nd test cases can be seen in Figures 5.23 and 5.24, respectively and also in Tables 5.6 and 5.7, respectively. It is clearly seen that multigrid method enables better convergence for the 1st test case, which is without solution refinement, than the 2nd test case. According to the values in Tables 5.6 and 5.7, the ratio between the convergence rates of the solutions with and without multigrid method is about 7 % for the 1st test case. On the other hand, the ratio between the convergence rates of the solutions with and without multigrid method is about 10 % for the 2nd test case. As a result of these ratios, the noticeable effect of multigrid method to the convergence acceleration is verified. In Figures 5.23 and 5.24, there is a slight convergence rate difference between saw-tooth and V-cycles. But it is important to note that forcing functions have to be stored for V-cycles and this causes higher memory usage than the saw-tooth cycle. Since the convergence difference between these cycles is low and saw-tooth cycle uses the memory more effectively, most of the test cases in this study are performed by using saw-tooth cycle.

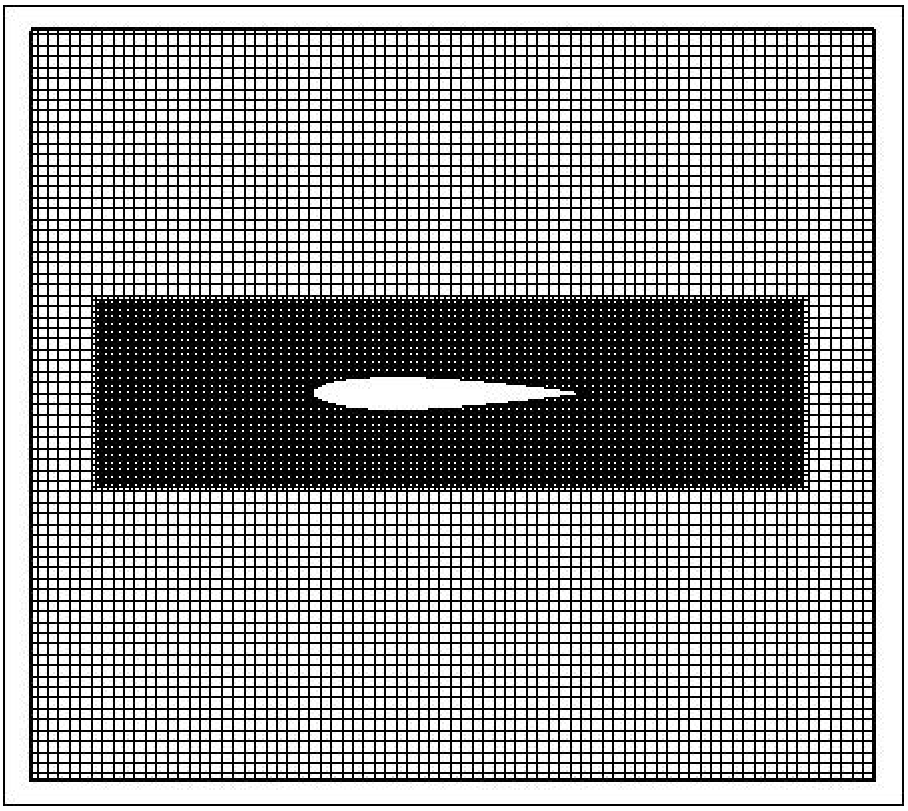


Figure 5.21 Geometric-adapted grid

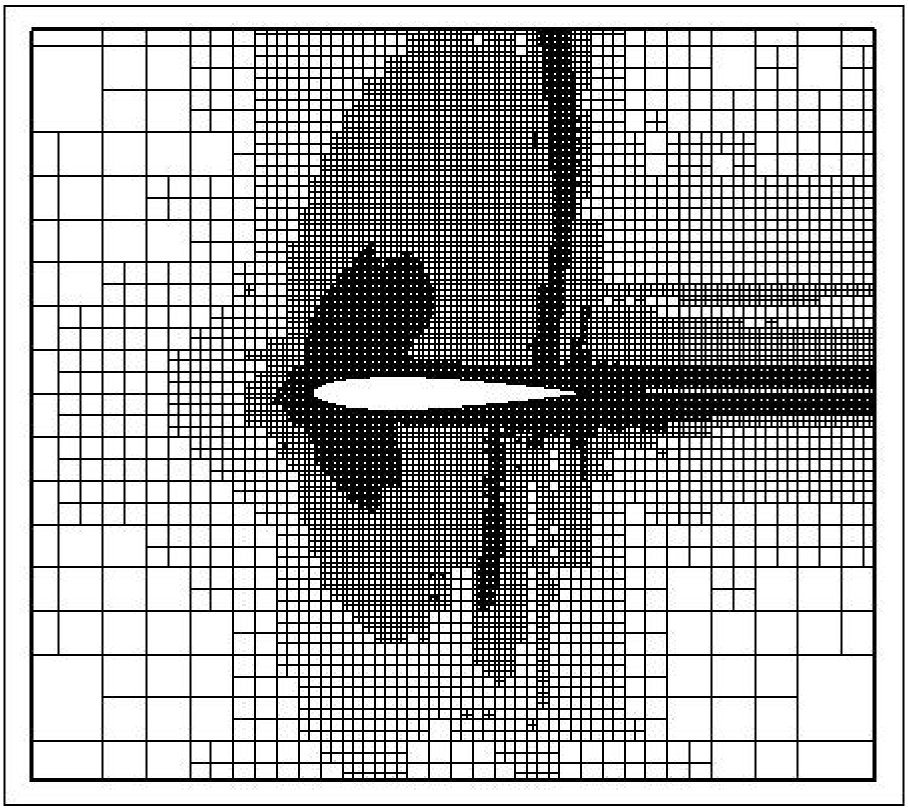


Figure 5.22 Geometric and solution-adapted grid

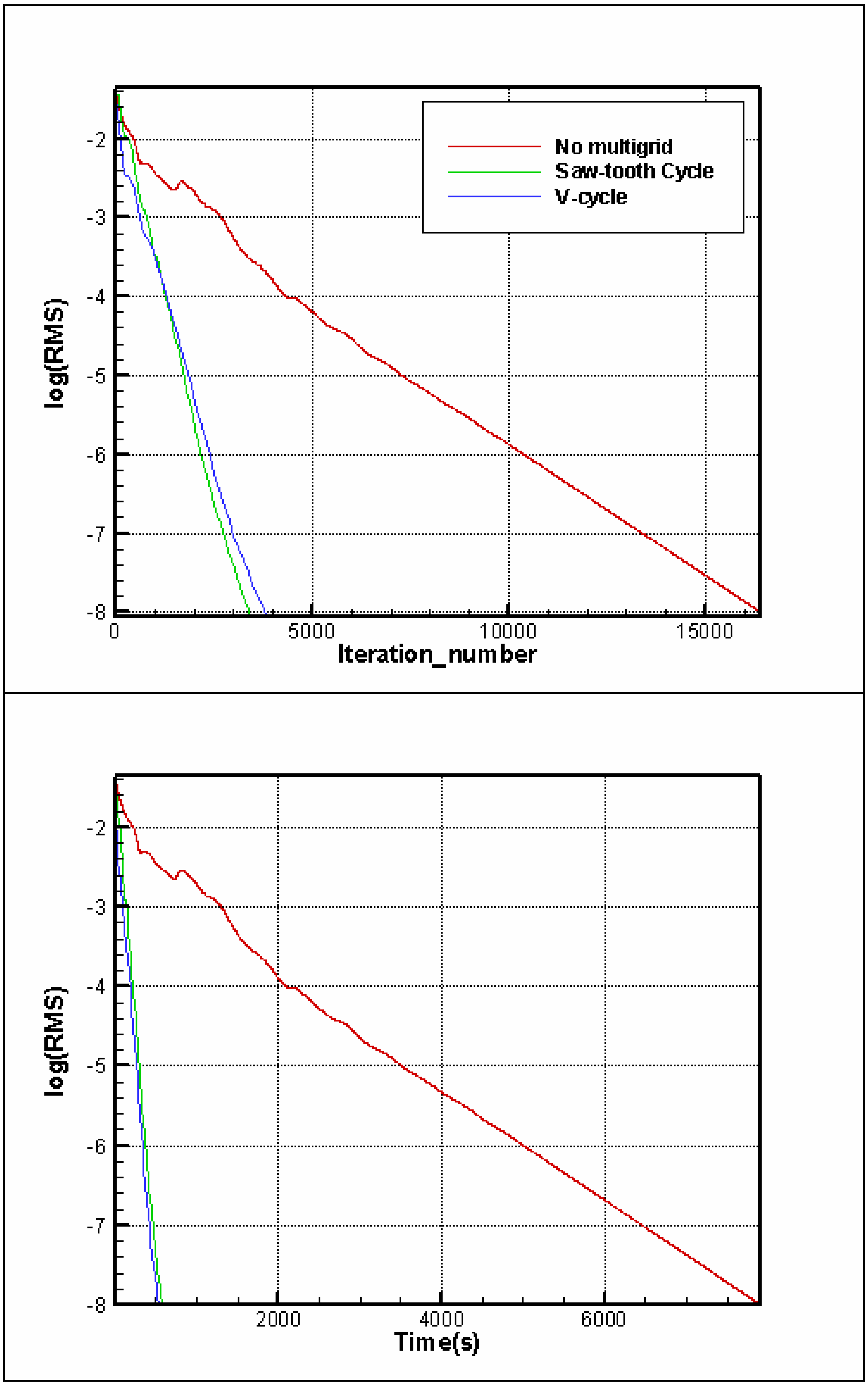


Figure 5.23 Convergence histories of the 1st test case

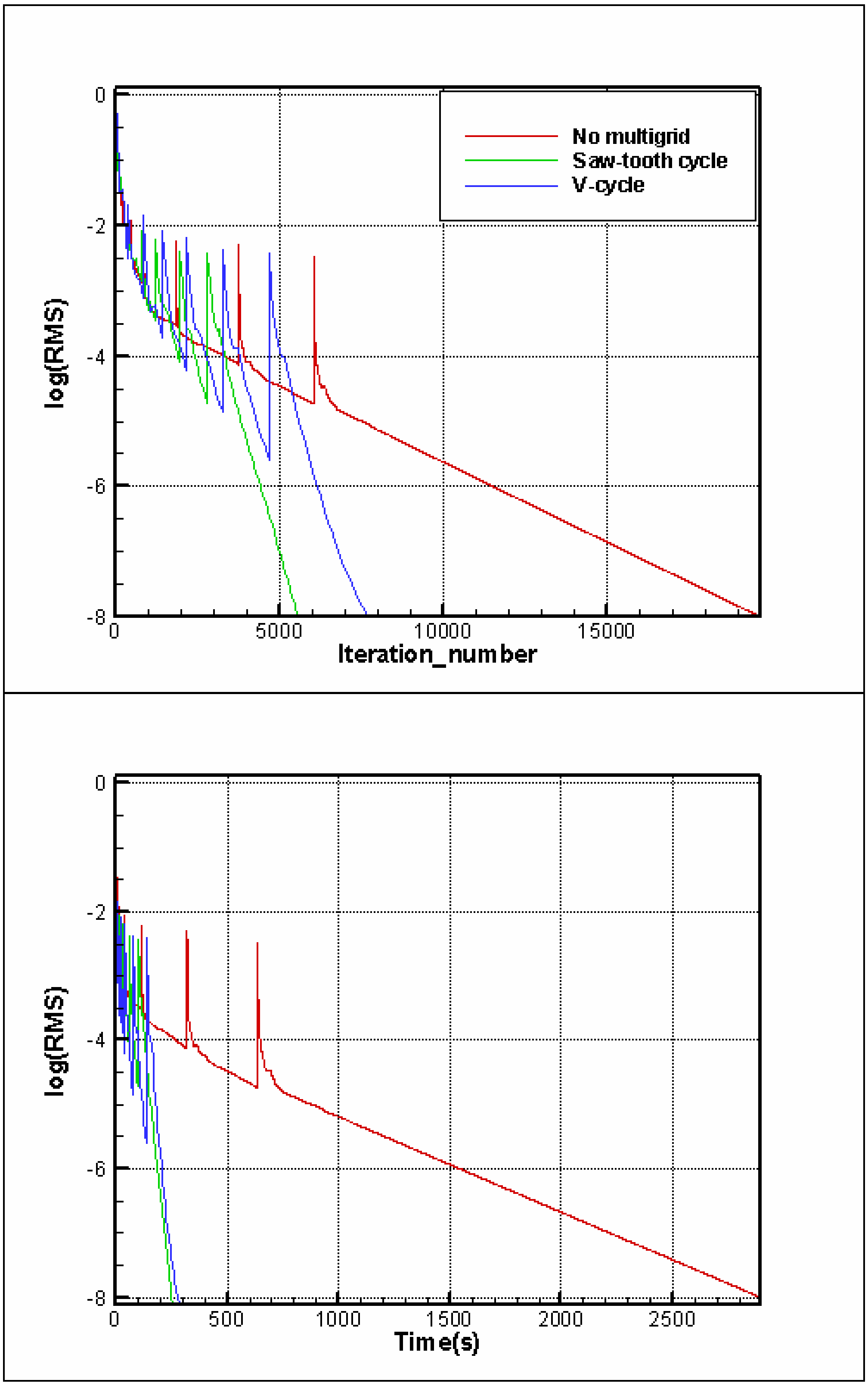


Figure 5.24 Convergence histories of the 2nd test case

Table 5.6 Convergence histories of the 1st test case

	Cycle	Iteration Number	Time (s)
No Multigrid	16405	16405	7907
Saw-tooth Cycle	49	3430	579
V-cycle	32	3840	535

Table 5.7 Convergence histories of the 2nd test case

	Cycle	Iteration Number	Time(s)
No Multigrid	19660	19660	2894
Saw-tooth Cycle	80	5600	253
V-cycle	65	7800	284

As it was mentioned in the previous chapter, there are two different prolongation operators, which are called injection and gradient operators. Results of the 3rd test case, which can be seen in Figure 5.25 and Table 5.8 demonstrates that the convergence rates of the solutions with injection and gradient operator are nearly identical. Therefore, the simplest operator, which is injection, is used for most of the test cases in this study.

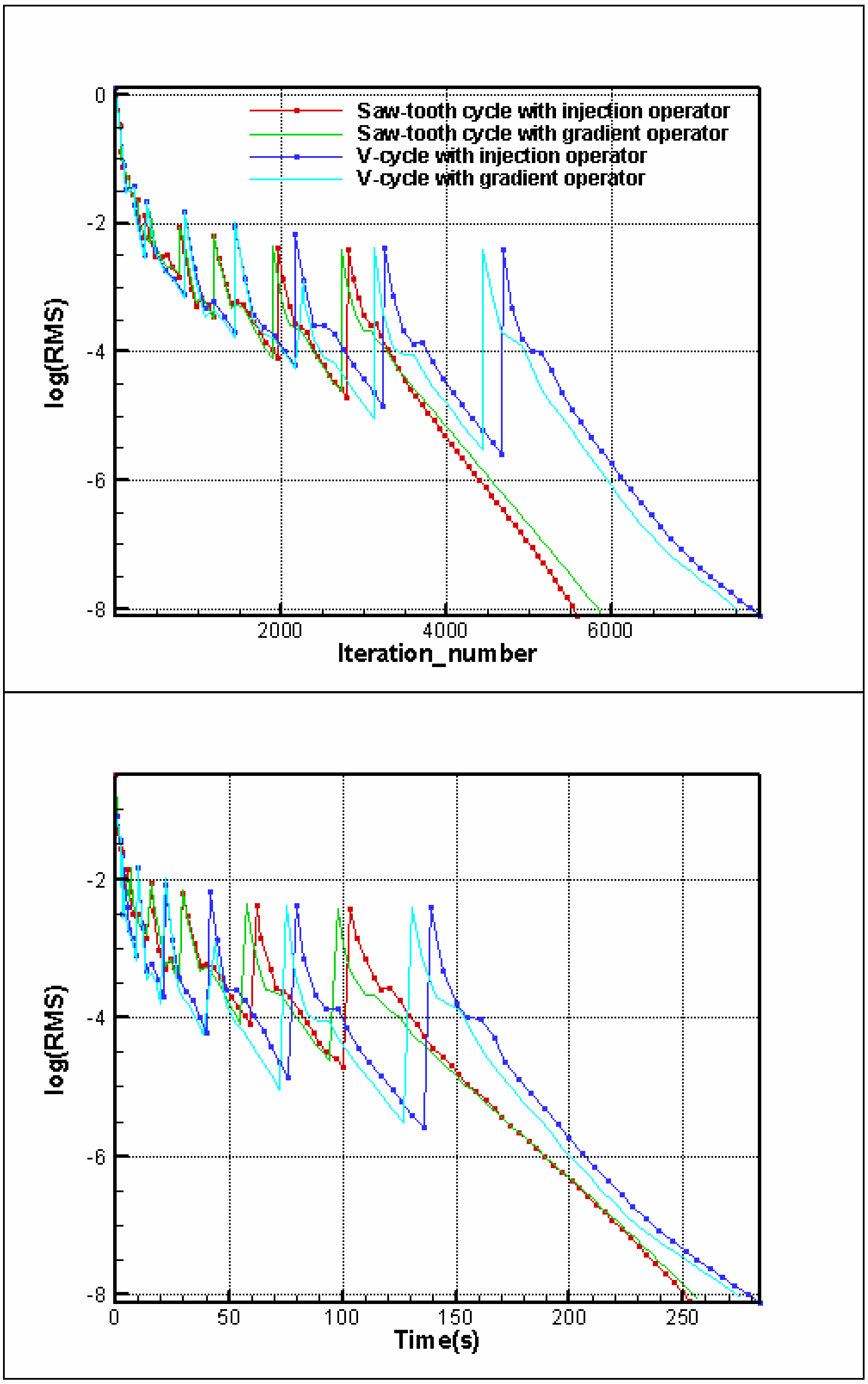
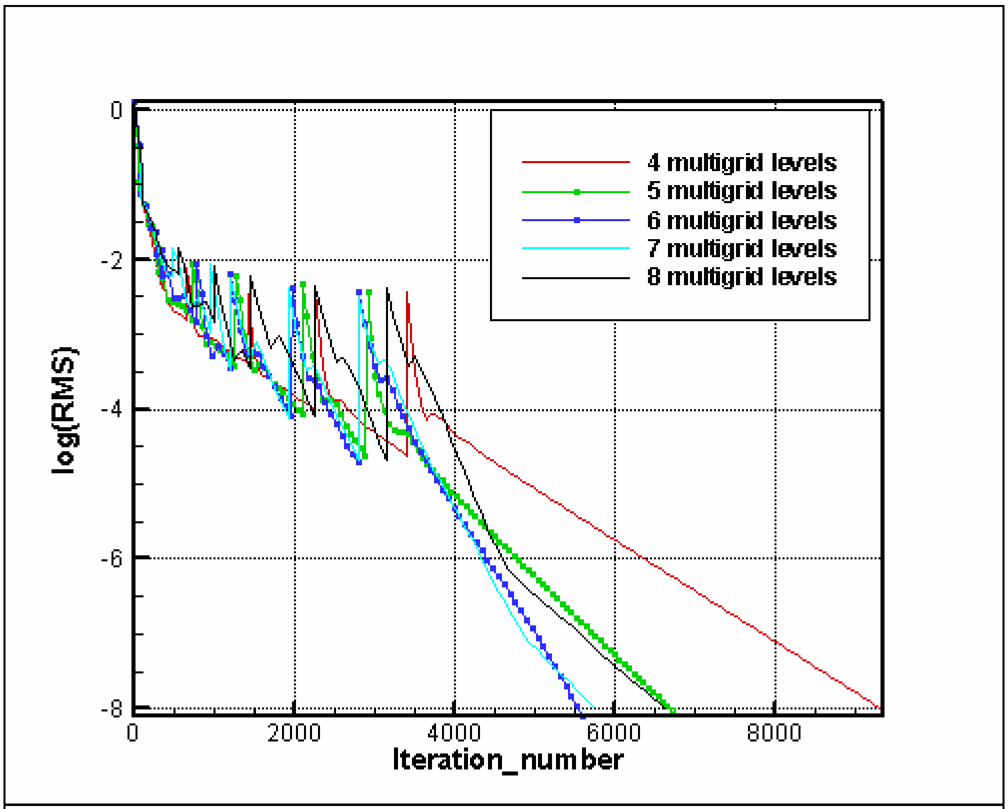


Figure 5.25 Convergence histories of the 3rd test case

Table 5.8 Convergence histories of the 3rd test case

	Cycle	Iteration #	Time(s)
Saw-tooth cycle with injection operator	80	5600	253
Saw-tooth cycle with gradient operator	84	5880	257
V-cycle with injection operator	65	7800	284
V-cycle with gradient operator	63	7560	276

One of the factors, which has an important effect on the convergence rate in multigrid method, is the number of multigrid levels. Since the performances of lower number of levels are very low, convergence accelerations of 4 and more multigrid levels are compared in the 4th test case. As seen in Figure 5.26 and Table 5.9, six and seven multigrid levels give the best results for denser meshes because denser meshes require more coarse levels to eliminate low frequency errors effectively. But it is important to note that six and seven multigrid levels may be too much for coarser meshes. Therefore, it can be suggested that six and seven multigrid levels are appropriate for denser meshes and three and four multigrid levels are suitable for coarser meshes.



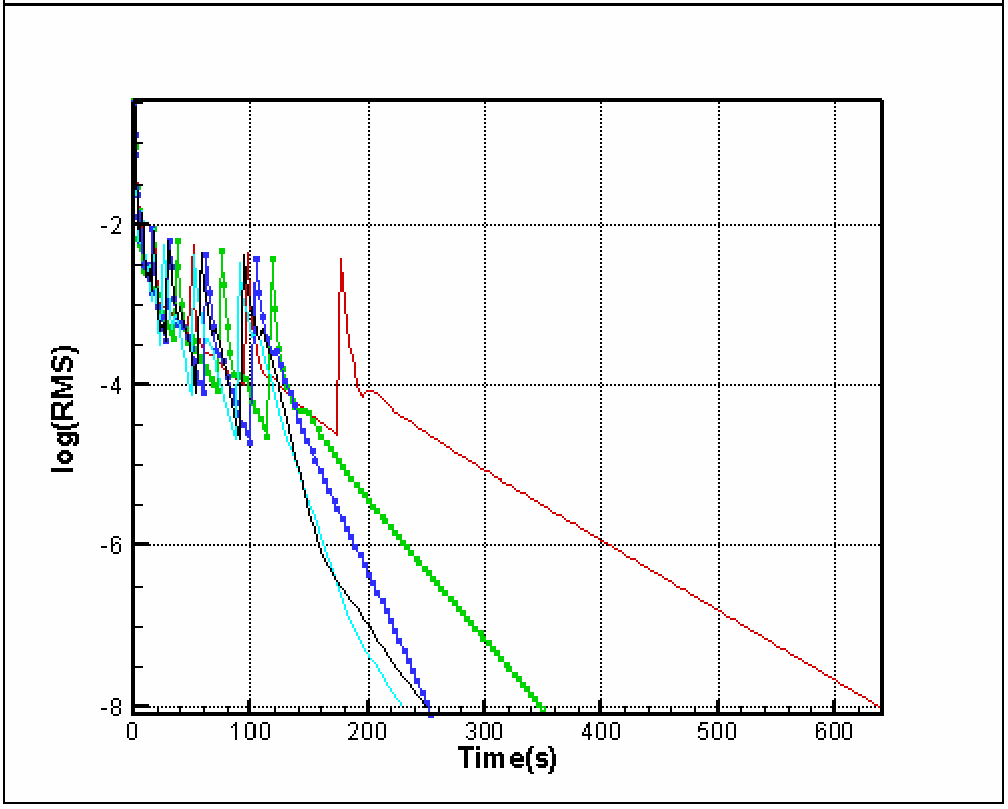
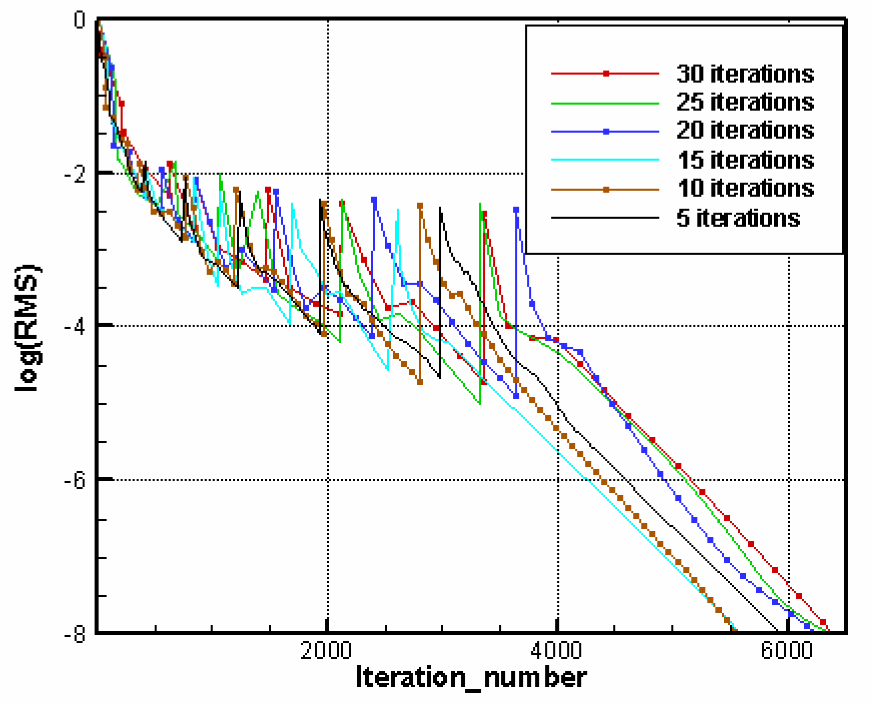


Figure 5.26 Convergence histories of the 4th test case

Table 5.9 Convergence histories of the 4th test case

	Cycle	Iteration #	Time(s)
4 multigrid levels	187	9350	641
5 multigrid levels	112	6720	350
6 multigrid levels	80	5600	253
7 multigrid levels	72	5760	230
8 multigrid levels	74	6660	250

Finally, numbers of iterations on each multigrid level have to be determined in order to get the maximum efficiency from the multigrid method. Convergence rate for different number of iterations using six multigrid levels are analyzed and the results of these tests are given in Figure 5.27 and Table 5.10. Iteration numbers between 10 and 20 give the best results for this condition. Therefore, the number of iterations is chosen as 10 for the most test cases in this study.



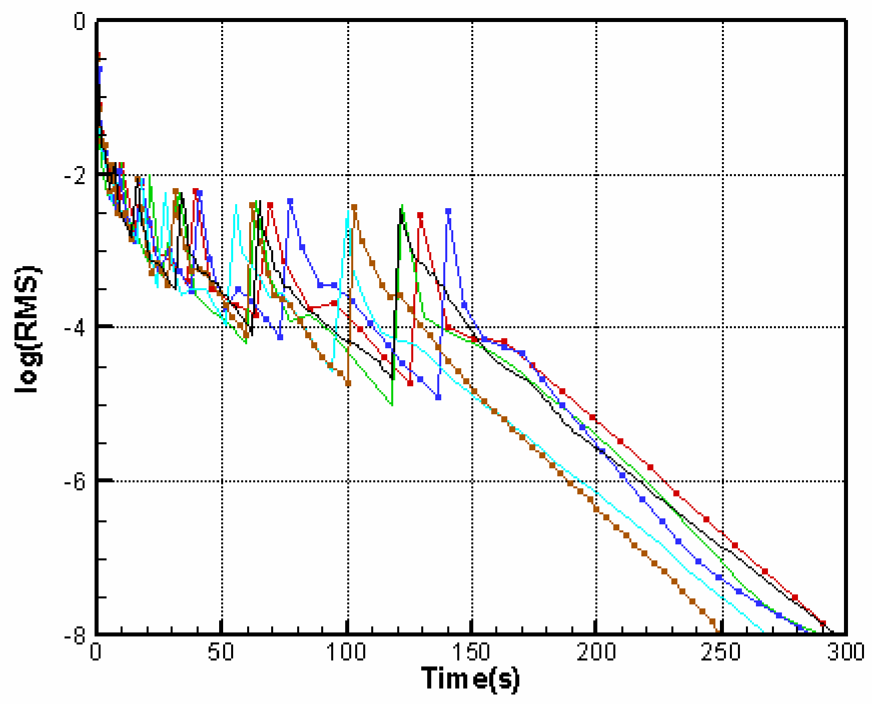


Figure 5.27 Convergence histories of the 5th test case

Table 5.10 Convergence histories of the 5th test case

	Cycle	Iteration #	Time(s)
30 iterations on each multigrid level	31	6510	302
25 iterations on each multigrid level	37	6475	296
20 iterations on each multigrid level	45	6300	289
15 iterations on each multigrid level	54	5670	272
10 iterations on each multigrid level	80	5600	253
5 iterations on each multigrid level	170	5950	297

5.3 Three Dimensional Test Cases

5.3.1 Transonic Flow about a Wing

The inviscid flow around a constant cross-section wing whose profile is NACA0012 airfoil is tested at a Mach number of 0.799 and an angle of attack of 2.26°. Aspect ratio of this wing is chosen as 20 in order to diminish vortex effects at the wing tips and obtain solutions as if the test is two dimensional. Hence, the solution can be compared with the results of two dimensional experiments in reference [34]. In addition, the computed results are compared with the inviscid results in reference [35].

The far-field boundary is approximately located 5 times the maximum length of the wing ahead of the wing. In other words, the domain size of the generated grid is 100 chords. For that reason, since the domain is too large when compared to the chord of the wing section, obtaining finer grid near the input geometry is really difficult. The developed flow solver is iterated until the average density residual reaches 10^{-7}. For the computed solutions, six levels of meshes used in the multigrid application. One refinement cycle is applied to the geometric-adapted grid and it is given in Figure 5.28. Pressure coefficient distributions are shown in Figure 5.29 and Mach contours of this case is shown in Figure 5.30. It is clearly seen in Figure 5.29 that both the shock wave location and its peak point cannot be captured accurately due to coarse grid although solution refinement is applied. But the number of refinement cycle is not satisfactory. Moreover, the surface mesh of the wing exported from GAMBIT is not very good due to large aspect ratio.

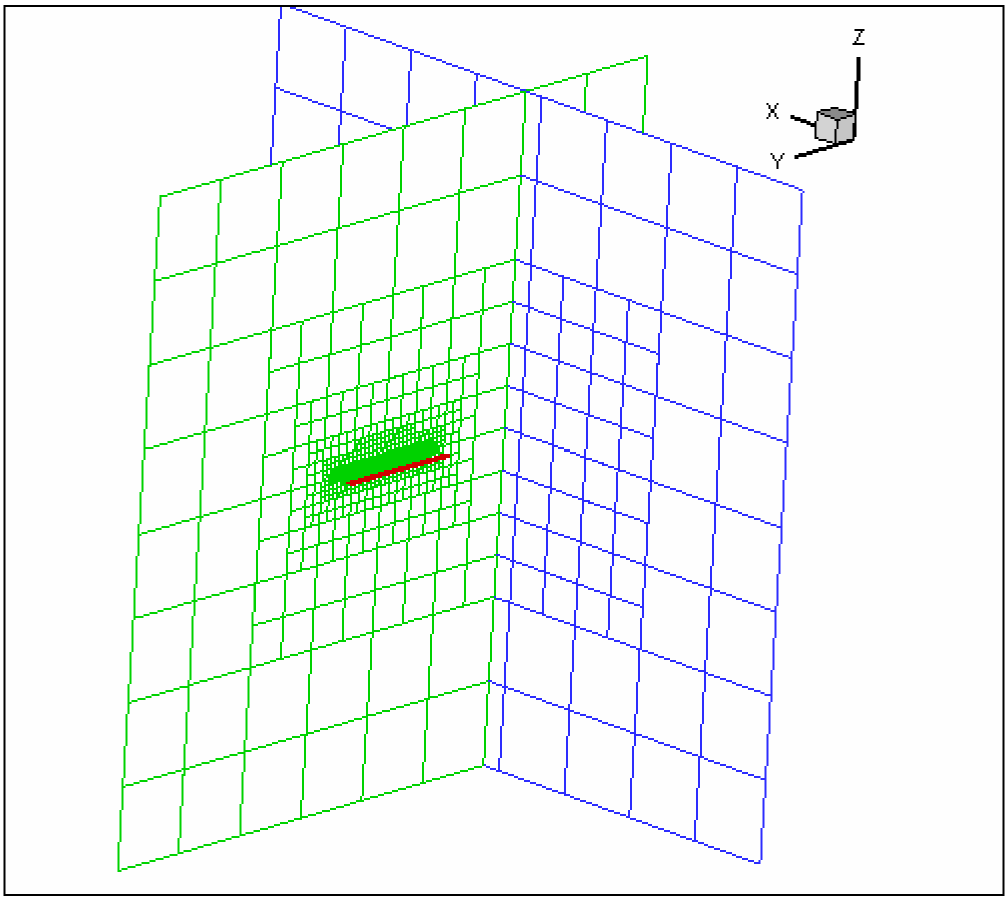


Figure 5.28 Slices of the solution-adapted grid

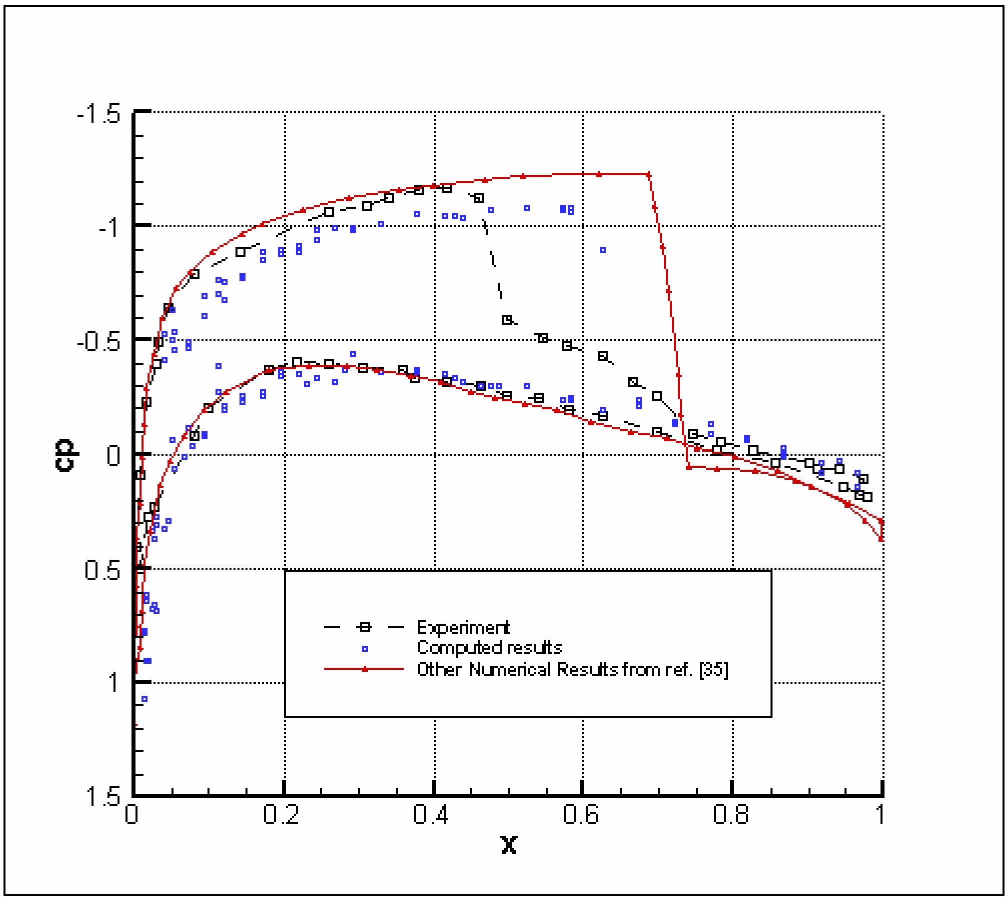


Figure 5.29 Pressure coefficient distributions around the wing at $M_\infty = 0.799$ and $\alpha = 2.26°$

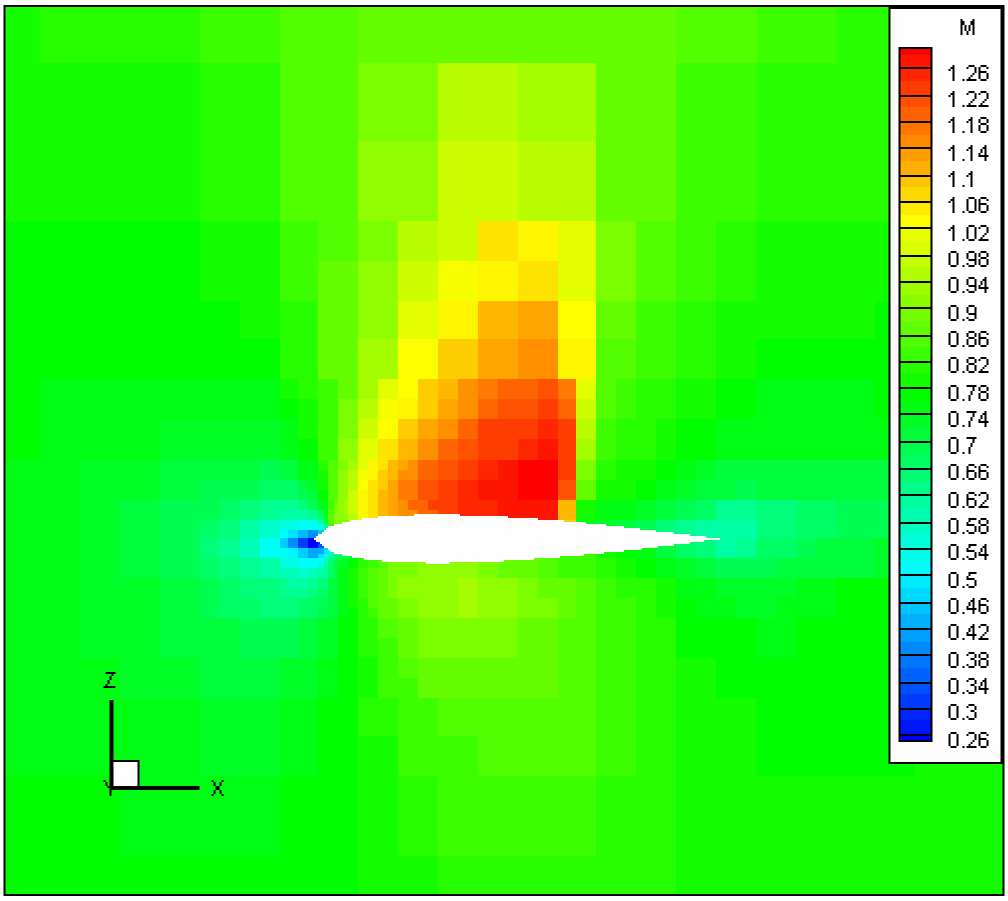


Figure 5.30 Mach contours in *xz* plane at y=0 at $M_\infty = 0.799$ and $\alpha = 2.26°$

5.3.1 Transonic Flow about a Projectile

The inviscid flow around a secant-ogive-cylinder-boat tail projectile (SOCBT) with a boat tail angle 7° is tested at a Mach number of 0.95 and an angle of attack of 0°. The pressure coefficient distributions are compared with the experimental results extracted from [36] and Mach contours are compared with the computed results in reference [36]. The configuration of SOCBT can be found in both [37] and [38]. The far-field boundary is located 10 times the maximum length of the projectile ahead of the tail. The developed flow solver is iterated until the average density residual reaches 10^{-7}. One refinement cycle is applied to the geometric-adapted grid and a slice, which is taken in *xz* plane at y=0, and the created surface mesh around the projectile are given in Figures 5.31 and 5.32, respectively.

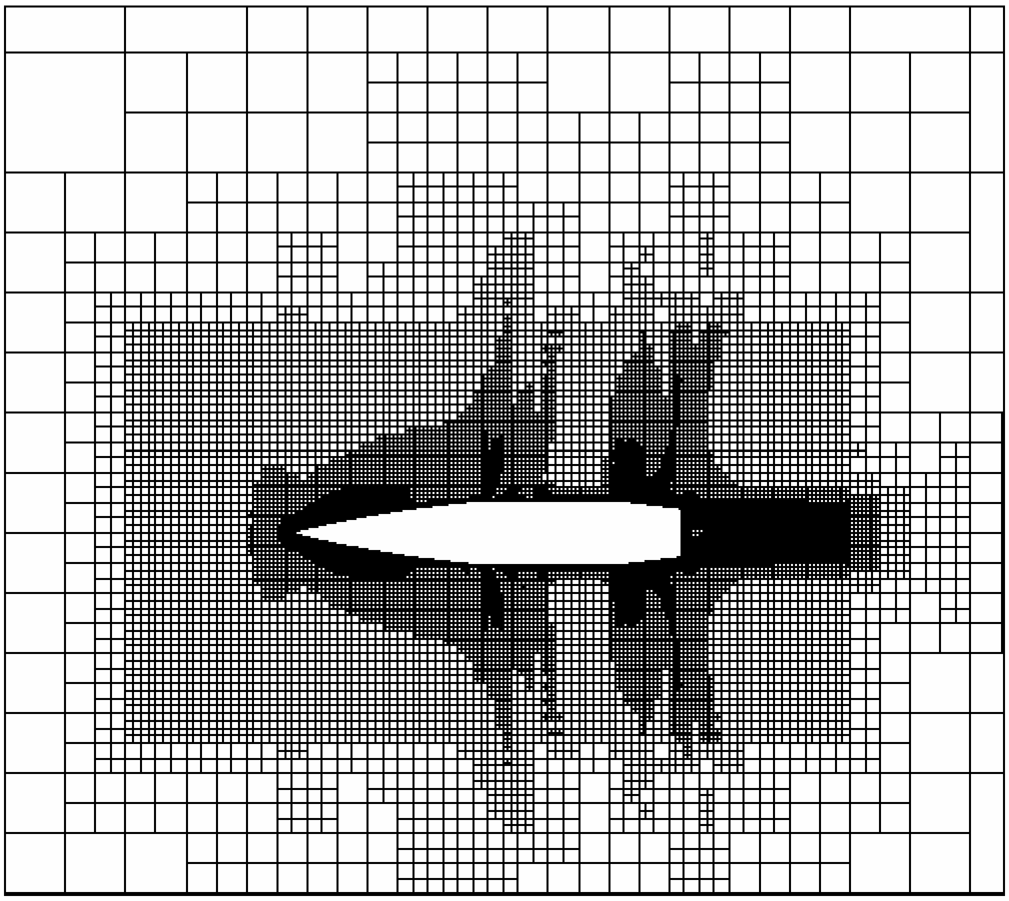


Figure 5.31 A slice of the mesh in *xz* plane at *y*=0

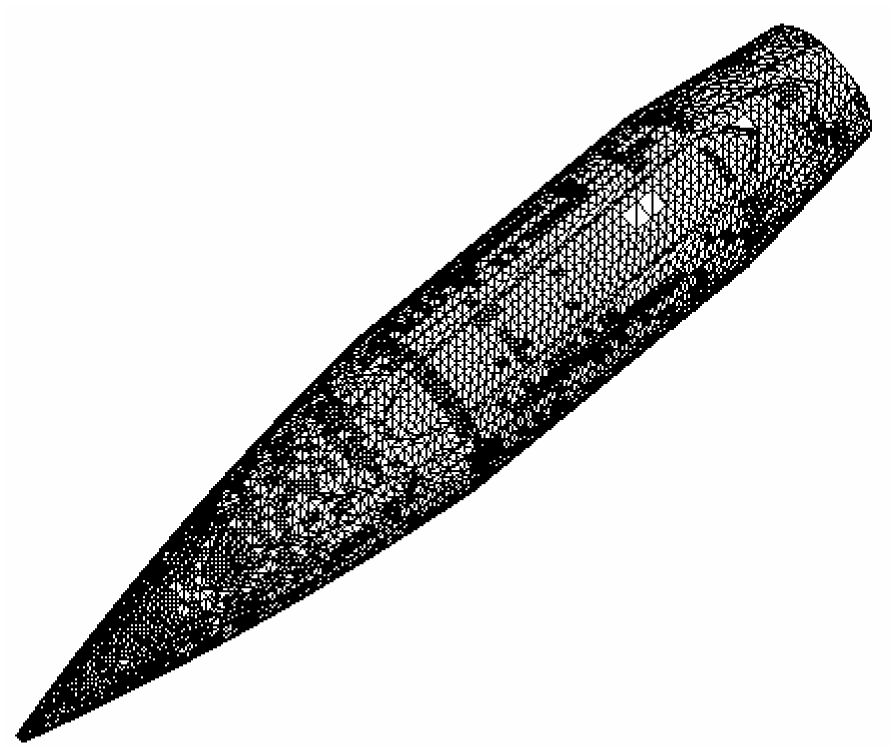


Figure 5.32 A created surface mesh around SOCBT

Pressure coefficient distributions are shown in Figure 5.33 and Mach contours of this case and the reference are given in Figures 5.34 and 5.35, respectively. As it is seen in Figures 5.33 and 5.34, there are two shock waves, one is at the midpoint of the chord and the other is at the boat tail. The computed and experimental results are in agreement. Finally, in order to eliminate oscillations in the solution, more solution refinements are required.

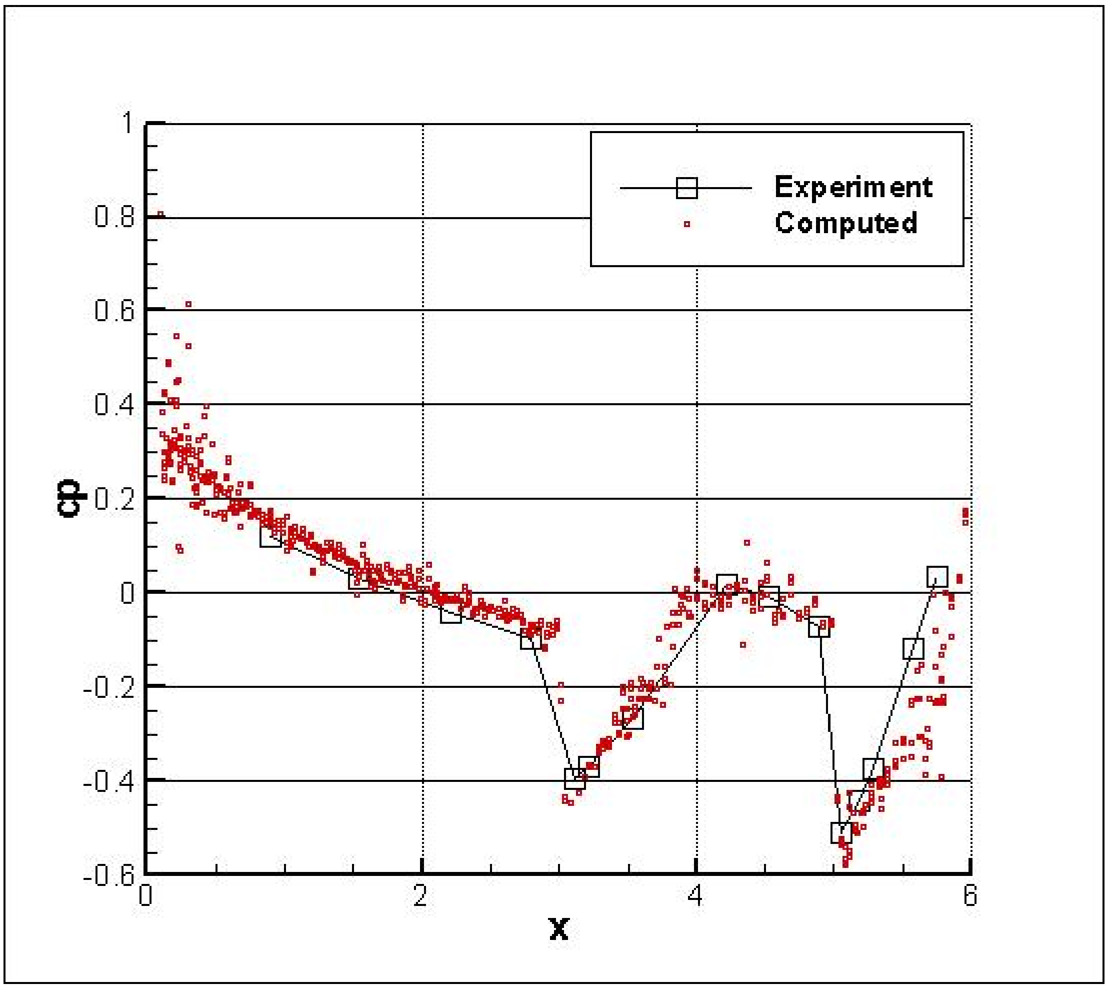


Figure 5.33 Pressure coefficient distributions at $M_\infty = 0.95$, $\alpha = 0°$ and $\beta = 7°$

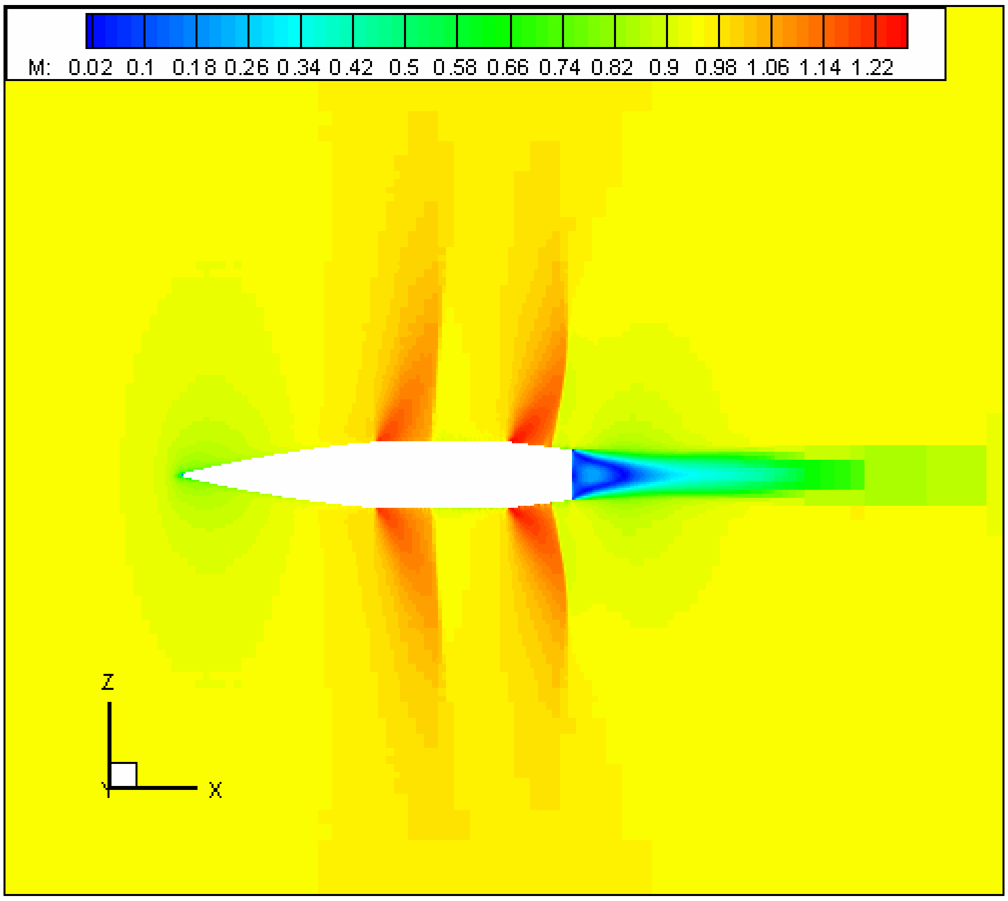


Figure 5.34 Mach contours in *xz* plane at *y*=0 at $M_\infty = 0.95$, $\alpha = 0°$ and β=7°

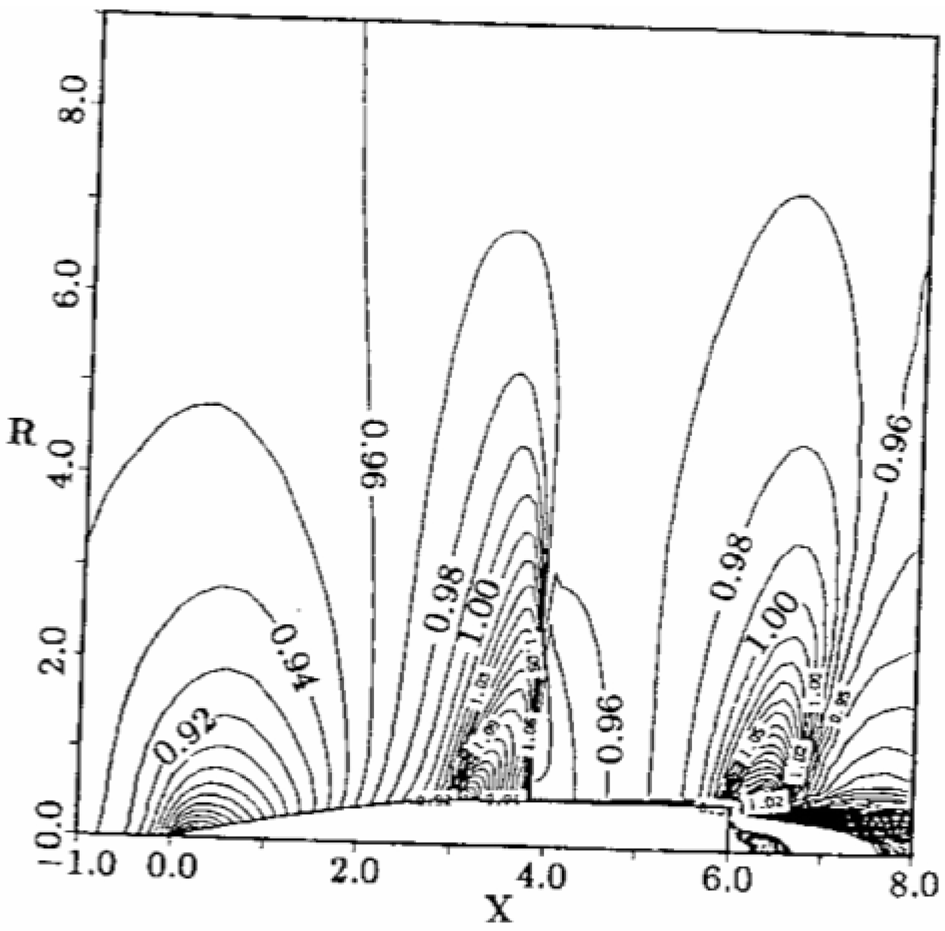


Figure 5.35 Mach contours in reference [36] at $M_\infty = 0.95$, $\alpha = 0°$ and $\beta=7°$

CHAPTER 6

CONCLUSION

Five different test cases have been analyzed in order to verify the two dimensional Euler solver. In the first case, the ability of capturing shock waves accurately is investigated for transonic flow over NACA0012 airfoil. The performance of the test case with solution refinement is really satisfactory. Both the locations of shocks and the peak point of pressure coefficients are captured very well. In other words, the importance of refinement to the solutions is proved by this test case.

Moreover, first and second order flux calculation schemes are applied to the second test case which is the supersonic flow with an angle of attack of 7° over NACA0012 airfoil. In second order schemes, instabilities have occurred just before the bow shock wave. Although limiters damp these instabilities by arranging gradients effectively, they are not eliminated completely. On the other hand, first order schemes are in agreement with the reference solution. In addition, there is a problem regarding second order schemes and multigrid methods. The code could not perform second order scheme and multigrid method together. One of them gives meaningful results apart from the other. The implementation of second order scheme and multigrid method to the developed solver together can be given as a future work.

Another test case for RAE2822 airfoil is performed to find the optimum number of refinement cycles. Increasing refinement cycles extremely does not give superior results since this means to slow down the convergence rate. When the number of refinement cycle is five or six for this test case, the most accurate results and the satisfactory convergence rate are obtained.

In the fourth and fifth test cases, the capability of Cartesian grids, which is automatic mesh generation, is demonstrated. There are detectable differences between reference and computed solutions for these test cases. Moreover, solutions on the airfoil surface are not smooth. There are oscillations especially near the slat and the leading edge of three-element airfoil. They can be easily seen in the figures of pressure coefficient distributions. One of the reasons of these oscillations can be the large variations in the cell size on the body. For example, it is possible to find cut cells whose size is 10^{-4} smaller than its neighboring outside cell. There is no agreement between the reference and computed pressure coefficient distributions of the slat in Figure 5.18. One of the reason can be the flow regime. In fact, the flow is an incompressible flow but the solver tries to solve this problem by using Euler equations.

Implementation of multigrid method has a valuable effect on the convergence rate. All test cases in Section 5.2 validate the increase of convergence rate. But solutions with second order schemes and multigrid application converge up to a value which is not enough then they oscillate around this value. Being frozen the limiter values after a certain point in the convergence is suggested in reference [9]. This method has been implemented the solver but this hasn't solved the convergence problem.

REFERENCES

[1] Anderson J. D. Jr., *Computational Fluid Dynamics: The Basics with Applications,* McGraw-Hill, 1995.

[2] Potter M. C., Wiggert D. C., Hondzo M., and Shih T. I-P. *Mechanics of Fluids,* Brooks/Cole, 2002.

[3] Blazek J., *Computational Fluid Dynamics: Principles and Applications,* Elsevier, 2005.

[4] De Zeeuw Darren L., *A Quad-Tree Based Adaptively-Refined Cartesian-Grid Algorithm for the Solution of the Euler Equations*, PhD Thesis in the University of Michigan, 1993.

[5] Carey G., *Computational Grids: Generation, Adaptation and Solution Strategies,* CRC Press, 1997.

[6] Thompson J. F., Soni B. K., and Weatherill N. P., *Handbook of Grid Generation,* CRC Press, 1998.

[7] Marshall David D., *Extending the Functionalities of Cartesian Grid Solvers: Viscous Effects Modeling and MPI Parallelization*, PhD Thesis in the Georgia Institute of Technology, 2002.

[8] Coirier William J., *An Adaptively Refined, Cartesian, Cell-Based Scheme for the Euler and Navier Stokes Equations*, PhD Thesis in the University of Michigan, 1994.

[9] Aftosmis M. J., *Solution Adaptive Cartesian Grid Methods for Aerodynamic Flows with Complex Geometries*, Von Karman Institute for Fluid Dynamics Lecture Series 28th Computational Fluid Dynamics, March 1997.

[10] Hunt J., *An Adaptive 3D Cartesian Approach for the Parallel Computation of Inviscid Flow about Static and Dynamic Configurations*, PhD Thesis in the University of Michigan, 2004.

[11] Siyahhan Bercan, *A Two Dimensional Euler Flow Solver on Adaptive Cartesian Grids*, MS Thesis in the Middle East Technical University, 2008.

[12] Bulgök Murat, *A Quadtree-based Adaptively-Refined Cartesian-Grid Algorithm for Solution of the Euler Equations*, MS Thesis in the Middle East Technical University, 2005.

[13] AGARD Subcommittee C., *Test Cases for Inviscid Flow Field Methods*, AGARD Advisory Report 211, 1986.

[14] Yıldırım Cengizhan, *Analysis of Grain Burnback and Internal Flow in Solid Propellant Rocket Motors in Three-Dimensions*, Ph.D. Thesis in the Middle East Technical University, 2007.

[15] Möller T. and Trumbore B., *Fast, Minimum Storage Ray/Triangle Intersection,* Journal of Graphics, gpu and Game Tools, Vol. 2, pp. 21-28, 1997.

[16] Laney Culbert B., *Computational Gas Dynamics*, Cambridge University Press, 1998.

[17] Hirsch Charles, *Numerical Computation of Internal and External Flows Volume 1 & 2,* John Wiley & Sons, 1990.

[18] Liou M. S., and Steffen C. J., *A New Flux Splitting Scheme*, Journal of Computational Physics, Vol. 107, pp. 23-39, 1993.

[19] Fedorenko R. P., *A Relaxation Method for Solving Elliptic Difference Equations,* USSR Computational Math. and Math. Phys., Vol. 1, 1962.

[20] Fedorenko R. P., *The Rate of Convergence of An Iterative Process,* USSR Computational Math. and Math. Phys., Vol. 4, 1964.

[21] Brandt Achi, *Multi-Level Adaptive Solutions to Boundary-Value Problems,* Mathematics for Computation, Vol. 31, pp. 333-390, 1977.

[22] Borzi A., *Introduction to Multigrid Methods,* http://www.ing.unisannio.it/borzi/mgintro.pdf, last access on; 28.07.2009

[23] Trottenberg U., Oosterlee C. W., Schüller A., *Multigrid,* Academic Press, 2001.

[24] Briggs William L., and McCormick, Steve F., *Multigrid Tutorial*, Siam 2000.

[25] Versteeg H. K., Malalasekera W., *An Introduction to Computational Fluid Dynamics: The Finite Volume Method,* Pearson/Prentice Hall, 2007.

[26] Jameson Antony, *Solution of the Euler Equations for Two-Dimensional Transonic Flow by a Multigrid Method*, Applied Mathematics and Computation, Vol. 13 Issues 3-4, pp. 327-355, 1983.

[27] Aftosmis M. J., Berger M. J., and Adomavicius G., *A Parallel Multilevel Method for Adaptively Refined Cartesian Grids with Embedded Boundaries*, AIAA Paper AIAA 2000-0808 38th Aerospace Sciences Meeting and Exhibit, Jan. 2000.

[28] Barth Timothy J., and Frederickson, Paul O., *Higher Order Solution of the Euler Equations*, AIAA Paper AIAA-90-0013, 1990.

[29] Barth Timothy J., and Jespersen Dennis C., *The design and Application of Upwind Schemes on Unstructured Meshes*, AIAA Paper AIAA-89-0366, 1989.

[30] Van den Berg B., and Gooden J. H. M., *Low-speed Pressure and Boundary Layer Measurement Data for the NLR 7301 Airfoil Section with Trailing Edge Flap*

[31] Sangho K., Alonso J. J., and Jameson A., *Design Optimization of High-Lift Configurations Using a Viscous Continuous Adjoint Method*, AIAA Paper AIAA-2002-0844, 2002.

[32] Eraslan Elvan, *Implementation of Different Flux Evaluation Schemes into a Two-Dimensional Euler Solver*, M.S. Thesis in the Middle East Technical University, 2006.

[33] Gönç Oktay L., *Computation of External Flow Around Rotating Bodies*, Ph.D. Thesis in the Middle East Technical University, 2005.

[34] Harris C. D., *Two-Dimensional Aerodynamic Characteristics of the NACA 0012 Airfoil in the Langley 8-Foot Pressure Tunnel*, NASA-TM-81927, 1981.

[35] Şahin Pınar, *Navier-Stokes Calculations over Swept Wings*, M.S. Thesis in the Middle East Technical University, 2006.

[36] Fu Jan-Kaung, and Liang Shen-Min, *Drag Reduction for Turbulent Flow over a Projectile:Part I*, Journal of Spacecraft and Rockets, Vol. 31, pp. 85-92, 1994.

[37] Sturek Walter B., Nietubicz Charles J., Sahu Jubaraj, and Weinacht Paul, *Applications of Computational Fluid Dynamics to the Aerodynamics of Army Projectiles*, Journal of Spacecraft and Rockets, Vol. 31, pp. 186-199, 1994.

[38] Sert Cüneyt, *Development of a Three Dimensional Object-Oriented Euler Solver using C++ Programming Language*, M.S. Thesis in the Middle East Technical University, 1998.

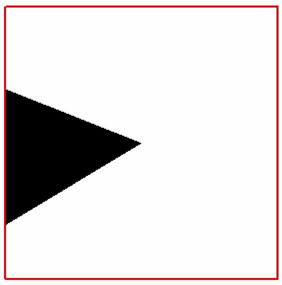
[39] Bourke Paul, *Polygonising a Scalar Field*, http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/, last access on; 28.07.2009

APPENDIX A

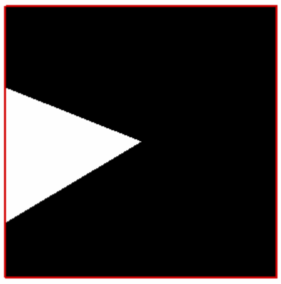
SPLIT CELLS

There are six types of split cells. First of all, examples are given to demonstrate each of these split cells. Classification of these cells depends on the former type and number of cut points as it will be explained below.

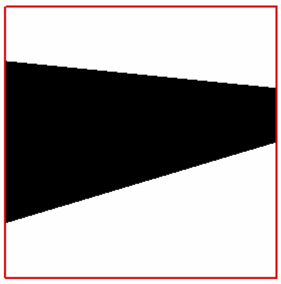
1-) Square index of this type of split cell is assigned to -15 since its former type is outside cell according to the inside-outside testing and it has two cut points on the edges of a cell.



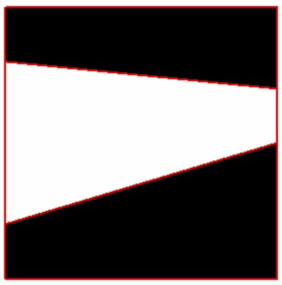
2-) Square index of this type of split cell is assigned to -20 since its former type is inside cell according to the inside-outside testing and it has two cut points on the edges of a cell.



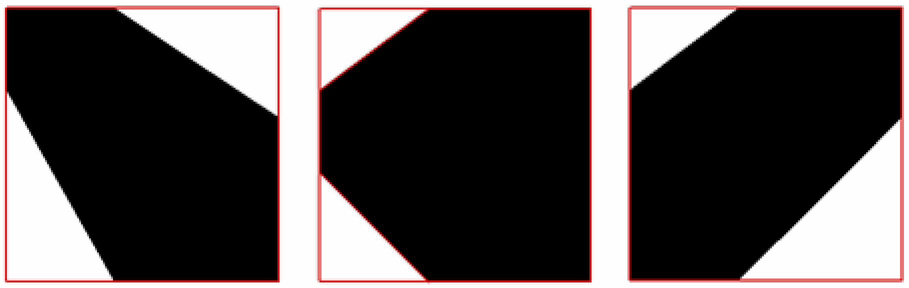
3-) Square index of this type of split cell is assigned to -25 since its former type is outside cell according to the inside-outside testing and it has four cut points on the edges of a cell.



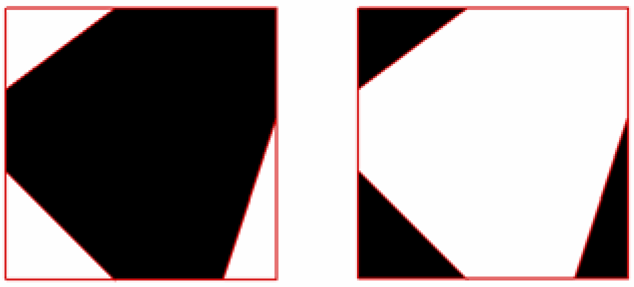
4-) Square index of this type of split cell is assigned to -30 since its former type is inside cell according to the inside-outside testing and it has four cut points on the edges of a cell.



5-) This type of split cells are converted from cut cells. According to the inside-outside testing, they are set as cut cells and their total square indexes are calculated. Regular cut cells must have only two cut points but these cells have more than two cut points. Therefore, these cells may be called as split-cut cells. Square indexes of these cells were calculated when they were cut cells. These cells are converted from cut cells to split cells. As a result, their types are changed but their square indexes remain the same.

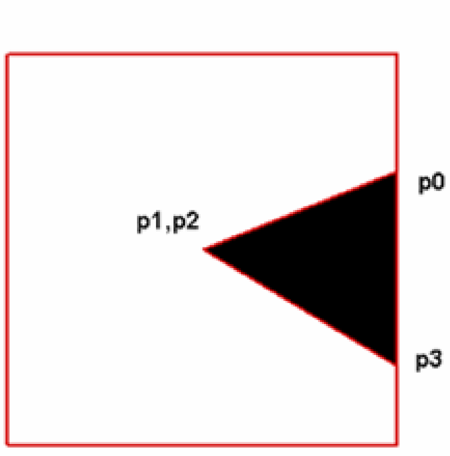


6-) Split cell whose square index is assigned to -40 since these cells have more than four cut points.

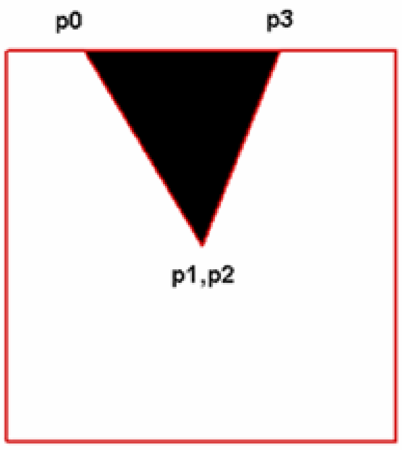


1-) Split cell whose square index is equal to -15

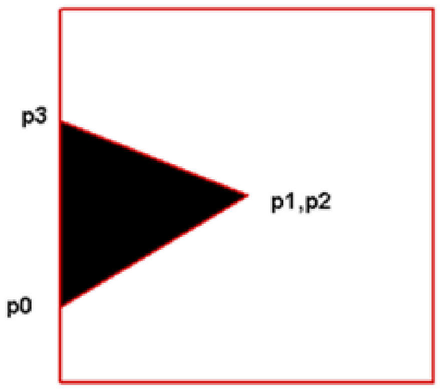
There are four sub-cases.



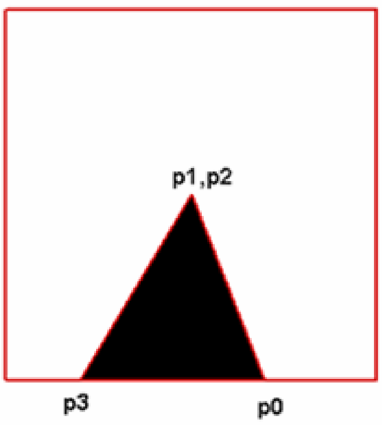
(1)



(2)



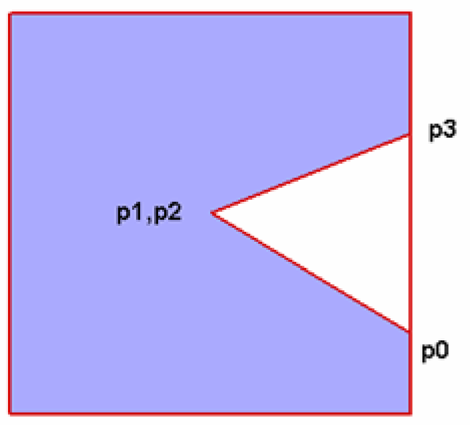
(3)



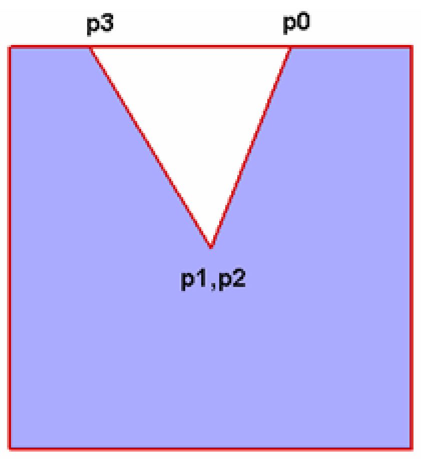
(4)

2-) Split cell whose square index is equal to -20

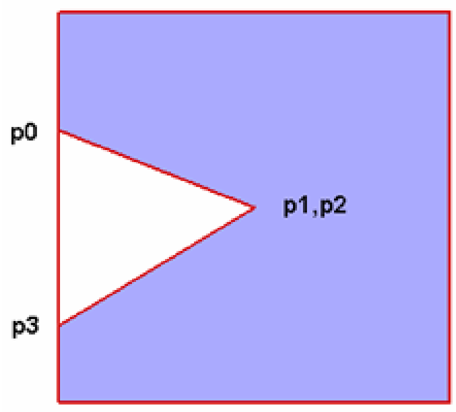
There are four sub-cases.



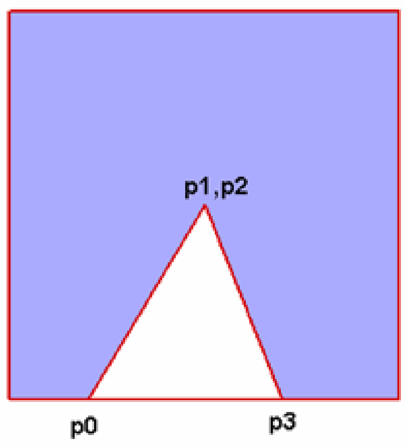
(1)



(2)



(3)

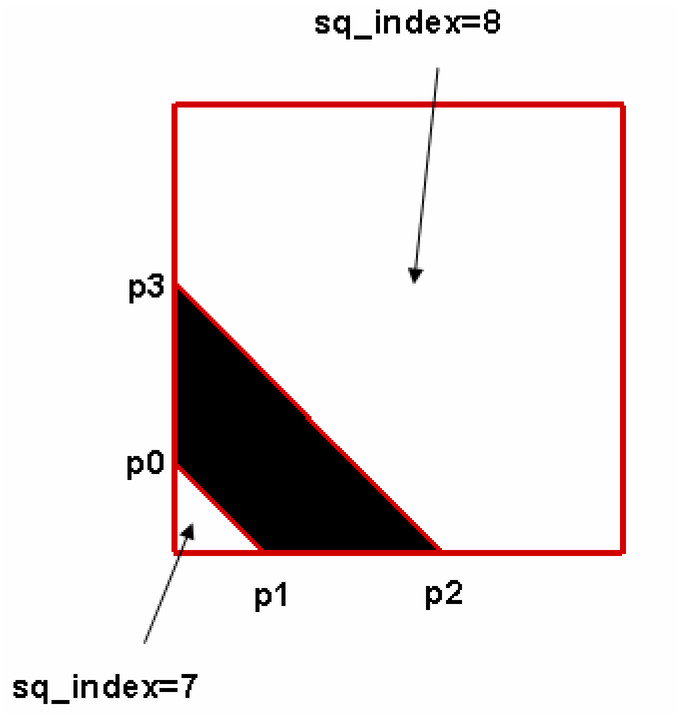


(4)

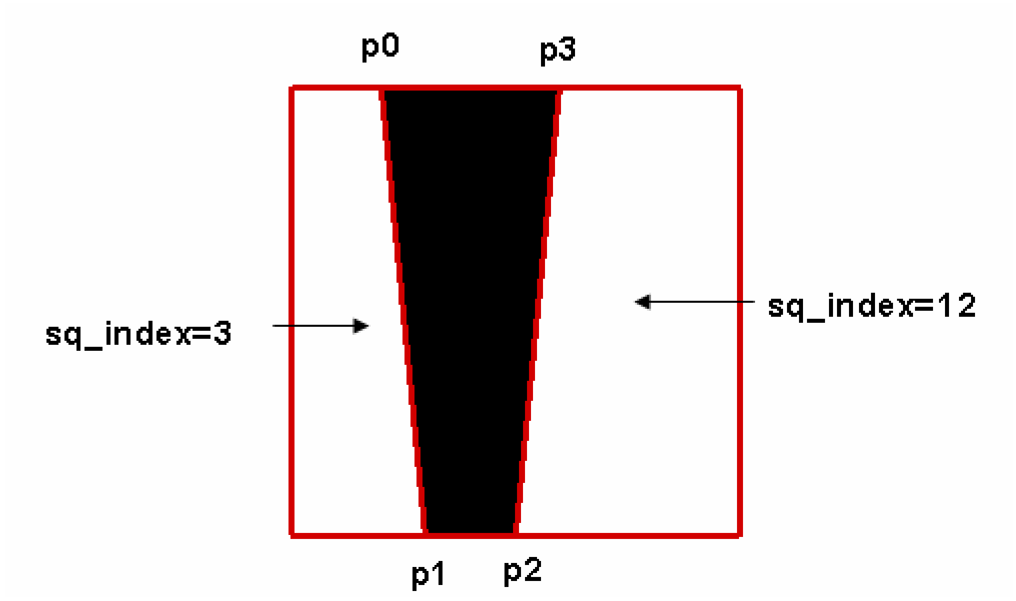
3-) Split cell whose square index is equal to -25

There are six sub-cases.

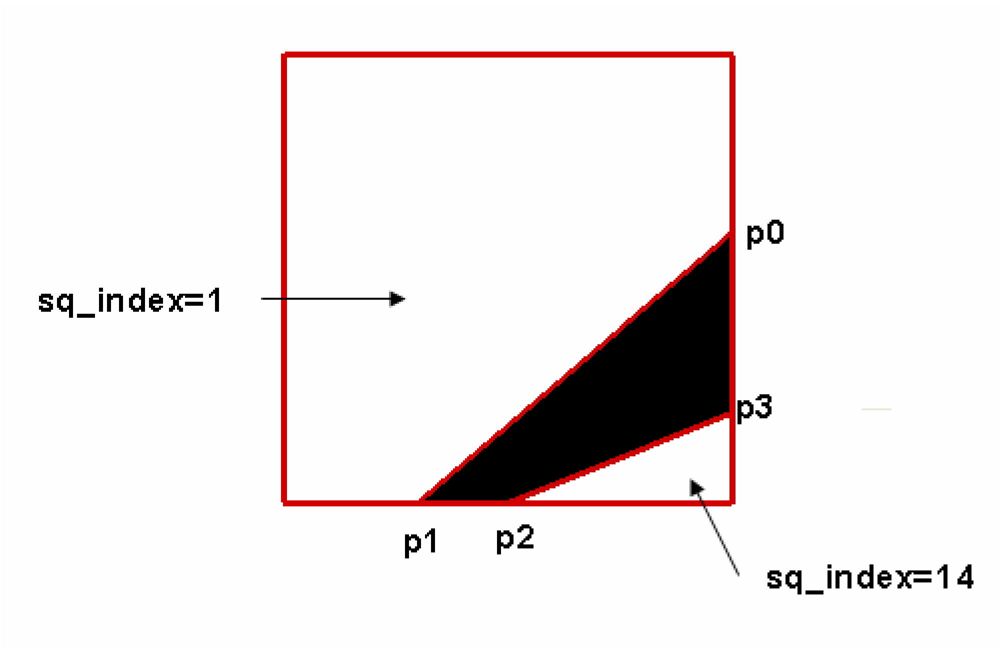
1-) Cut edges 3 & 2………………………..Split index=0



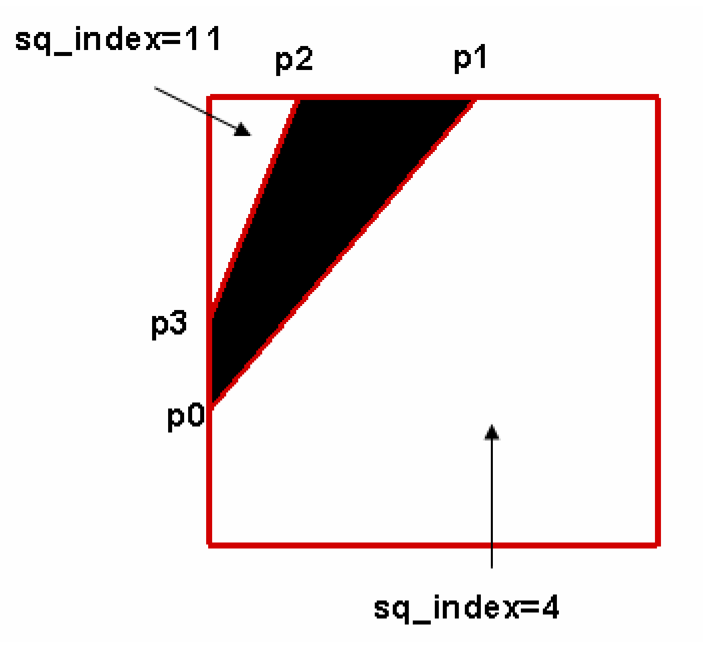
2-) Cut edges 3 & 1……………………….Split index=1



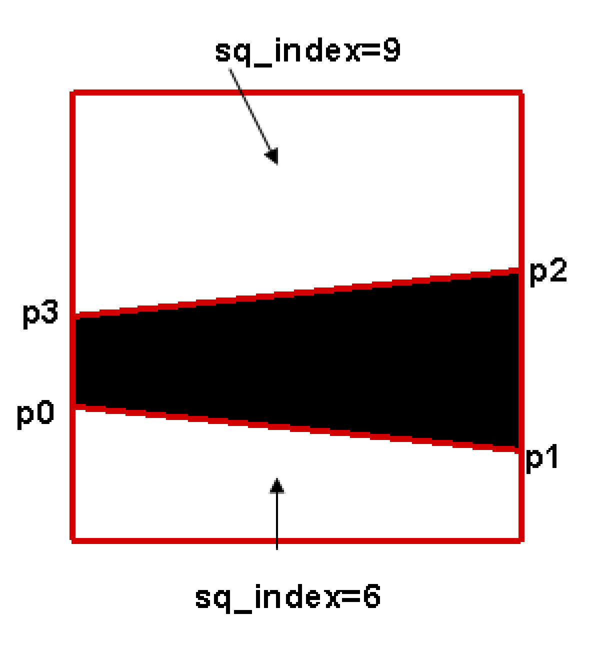
3-) Cut edges 3 & 0……………………….Split index=2



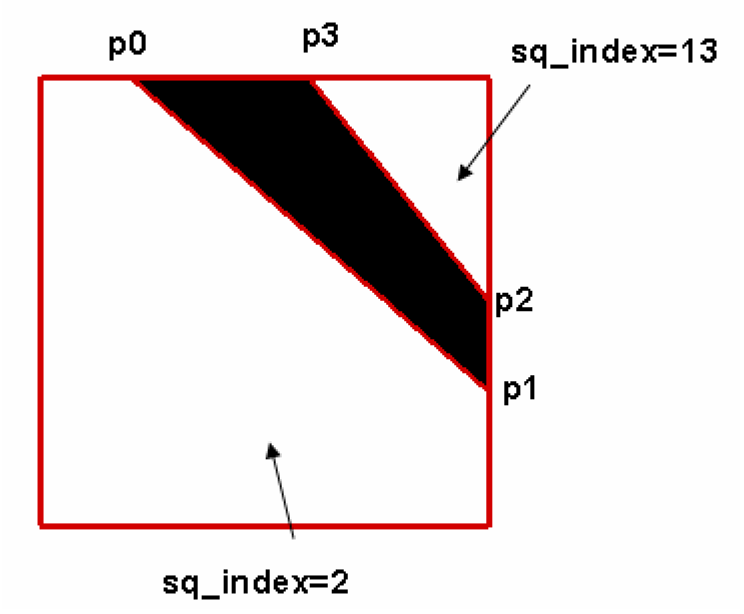
4-) Cut edges 2& 1………………………..Split index=3



5-) Cut edges 2& 0………………………..Split index=4



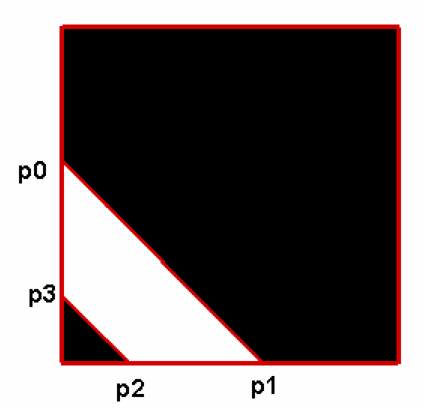
6-) Cut edges 1& 0………………………..Split index=5



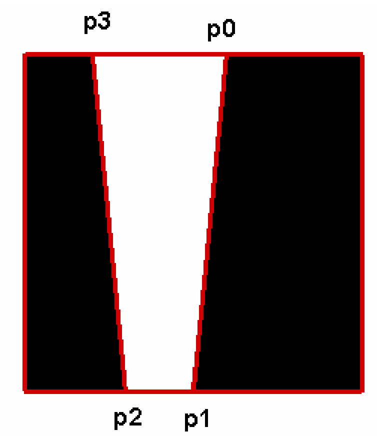
4-) Split cell whose square index is equal to -30

There are six sub-cases.

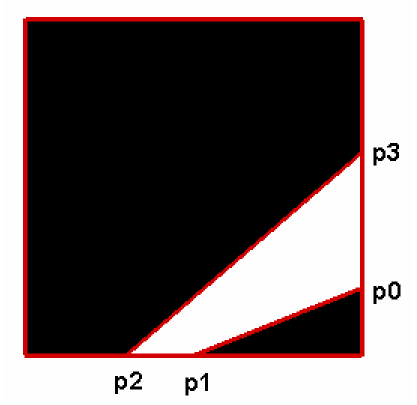
1-) Cut edges 3 & 2………………………..Split index=0



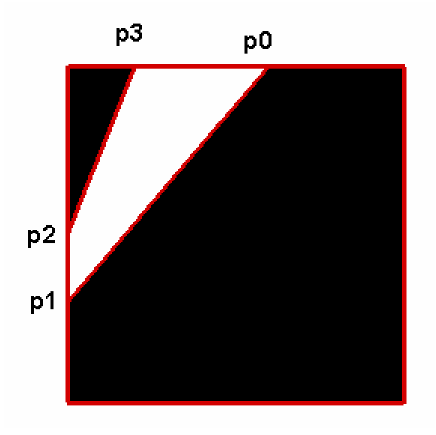
2-) Cut edges 3 & 1………………………..Split index=1



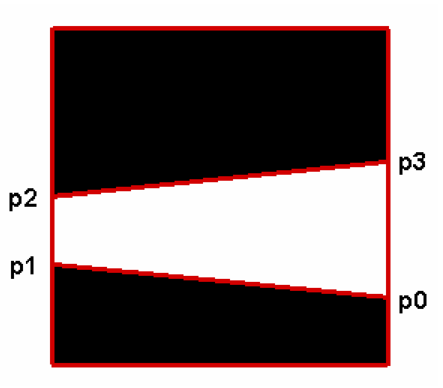
3-) Cut edges 3 & 0………………………..Split index=2



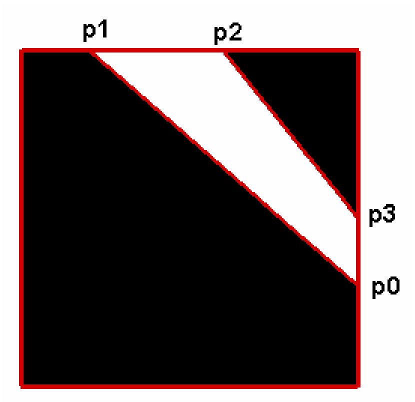
4-) Cut edges 2& 1………………………..Split index=3



5-) Cut edges 2& 0………………………..Split index=4



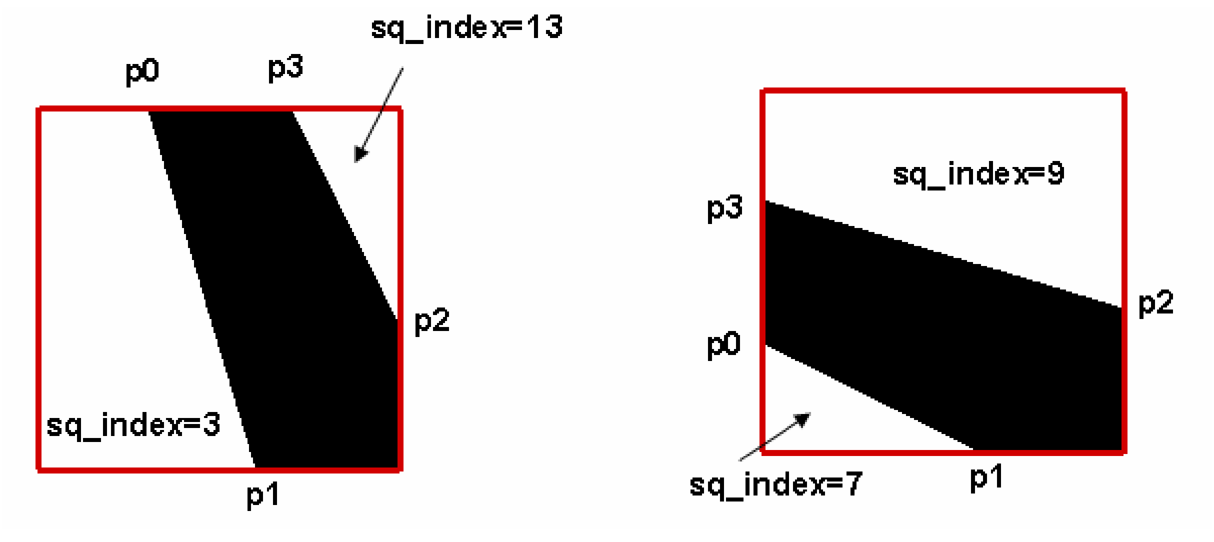
6-) Cut edges 1& 0………………………..Split index=5



5-) Split cell which is converted from cut cell

There are fourteen sub-cases.

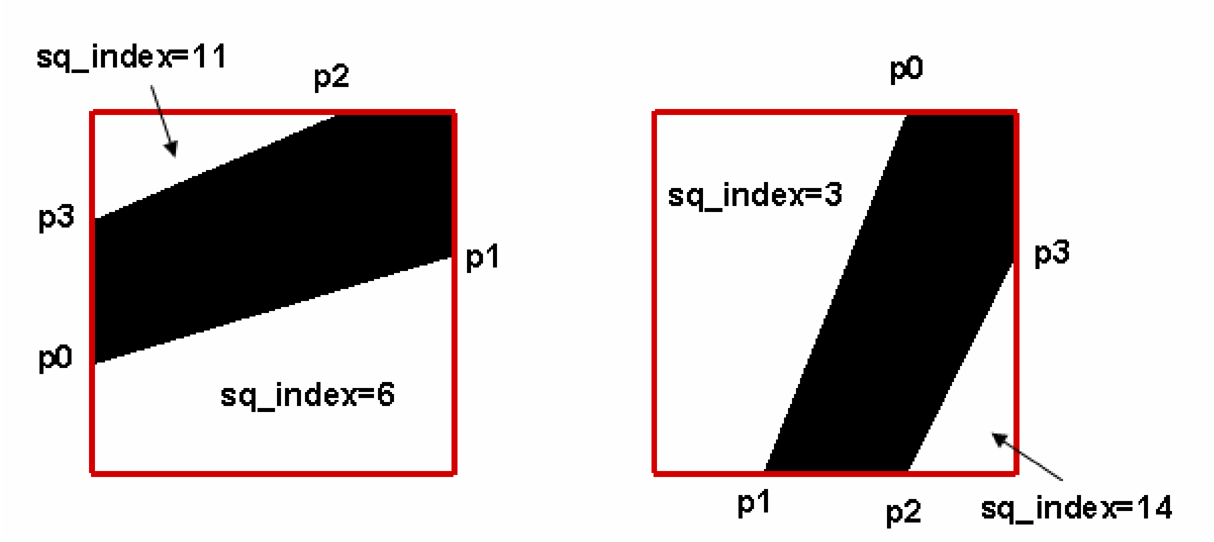
1-) Square index=1



a) Split index=0

b) Split index=1

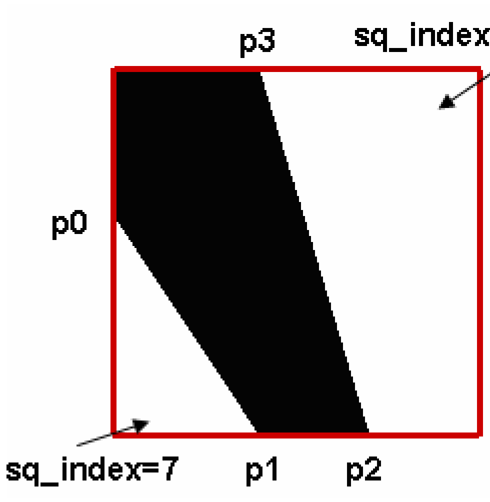
2-) Square index =2



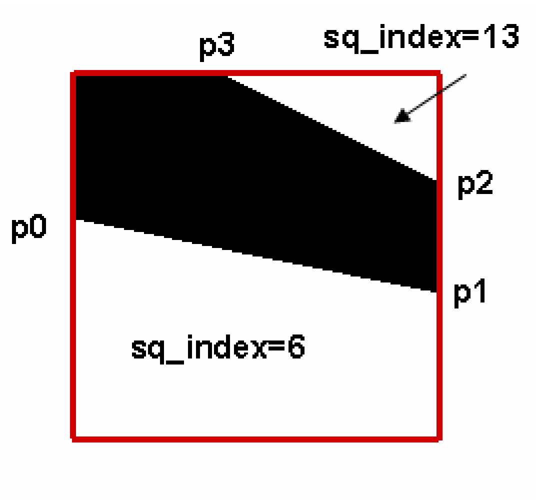
a) Split index=2

b) Split index=3

3-) Square index =4

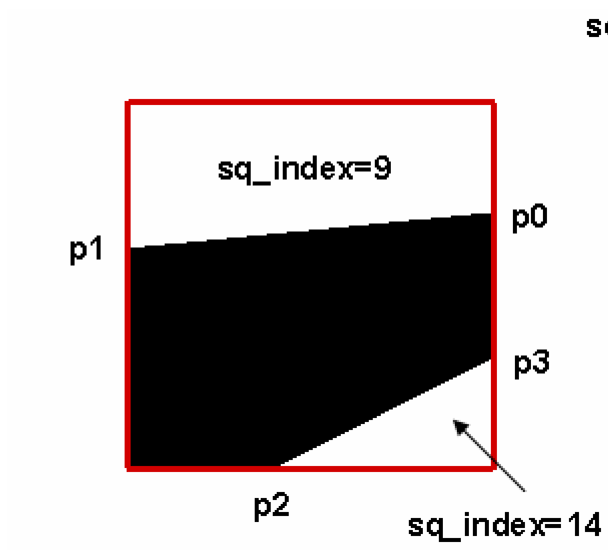


a) Split index=4

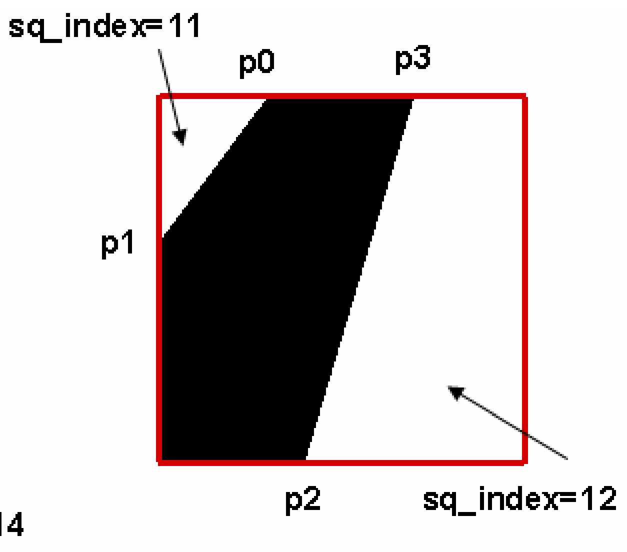


b) Split index=5

4-) Square index =8

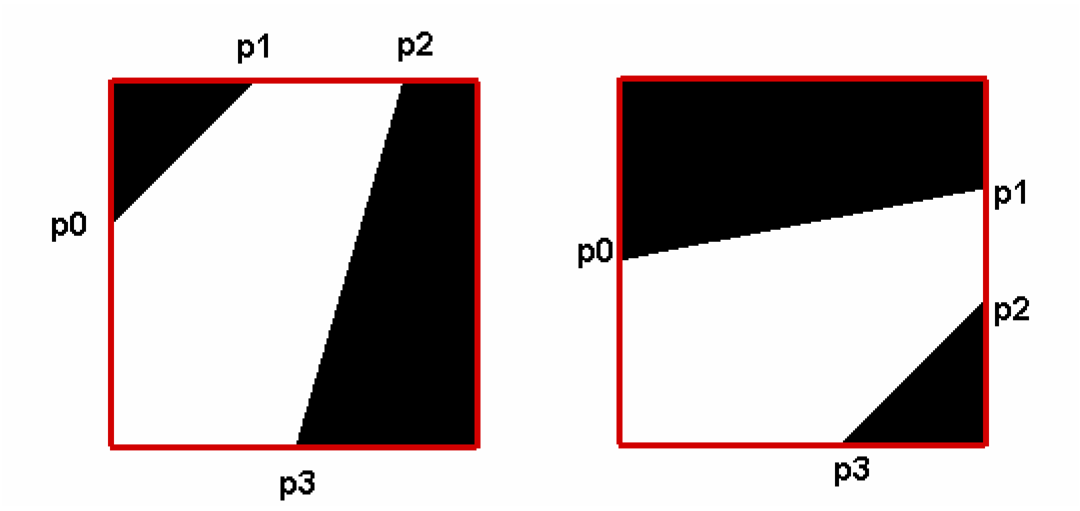


a) Split index=6



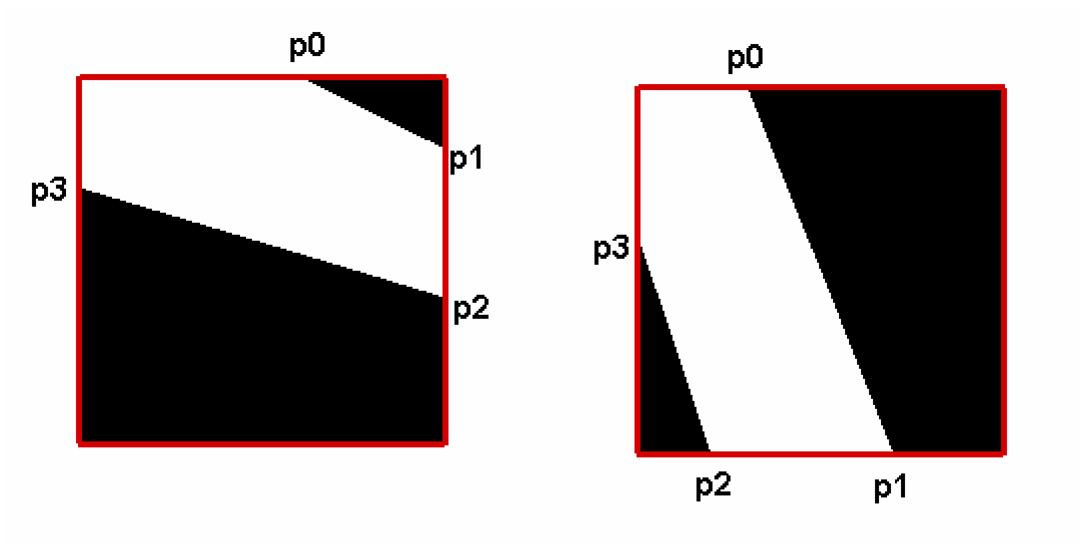
b) Split index=7

5-) Square index =7



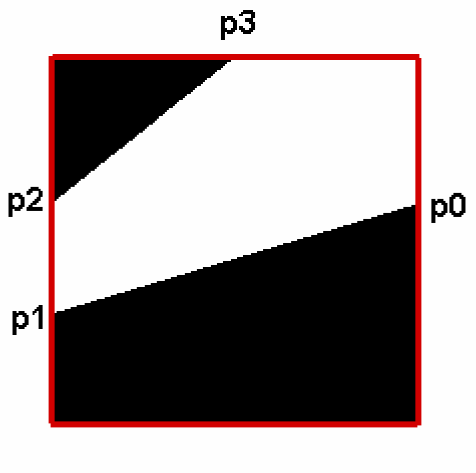
a) Split index=1 b) Split index=0

6-) Square index =11

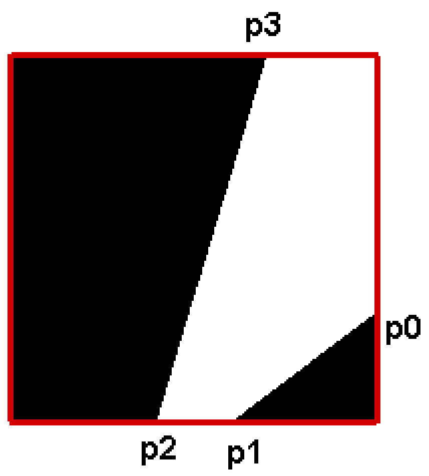


a) Split index=0 b) Split index=3

7-) Square index =13

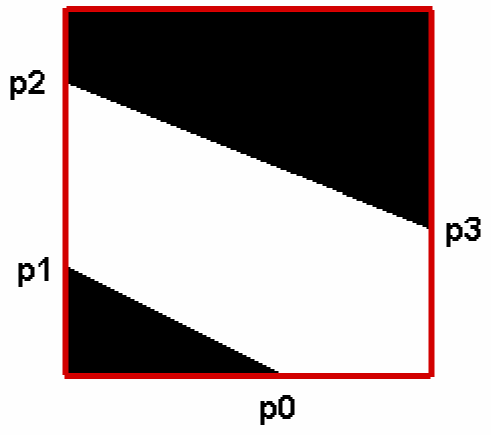


a) Split index=2

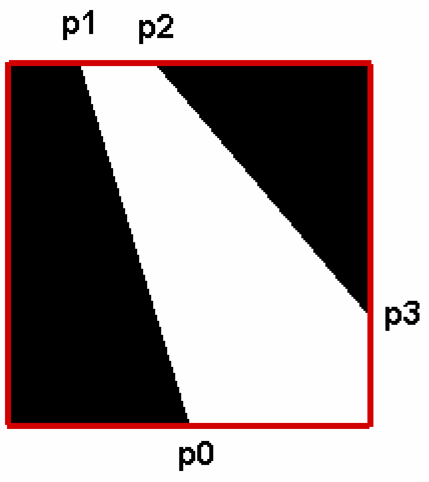


b) Split index=3

8-) Square index =14

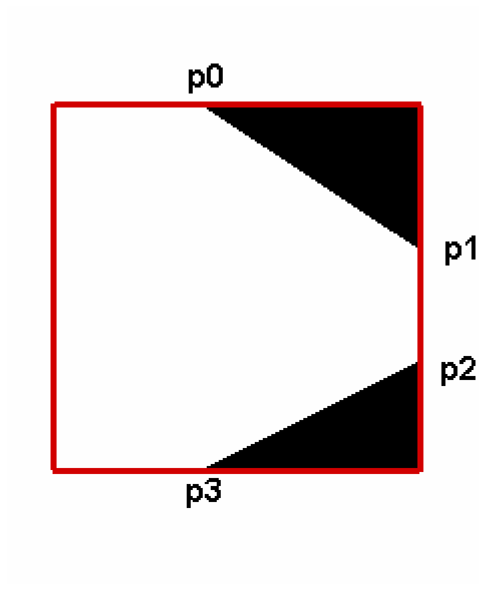


a) Split index=2

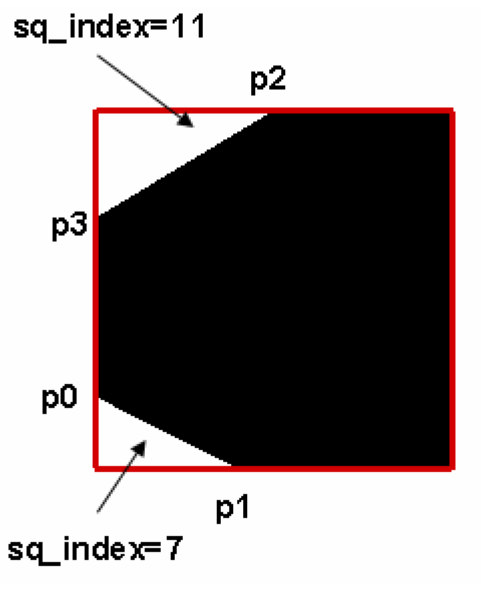


b) Split index=1

9-) Square index =3

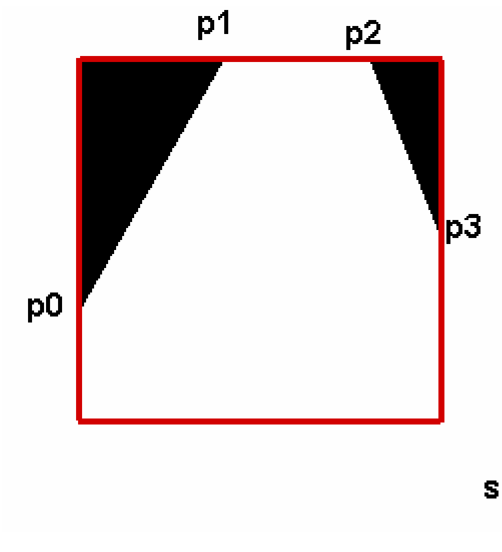


a) Split index= -1

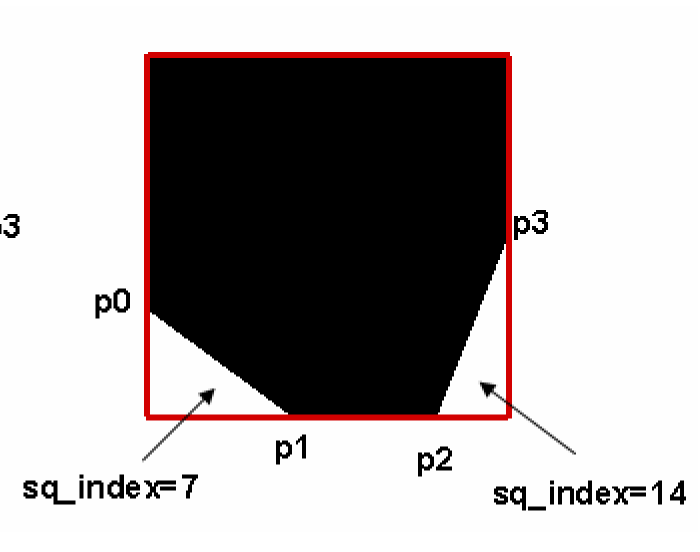


b) Split index=0

10-) Square index =6

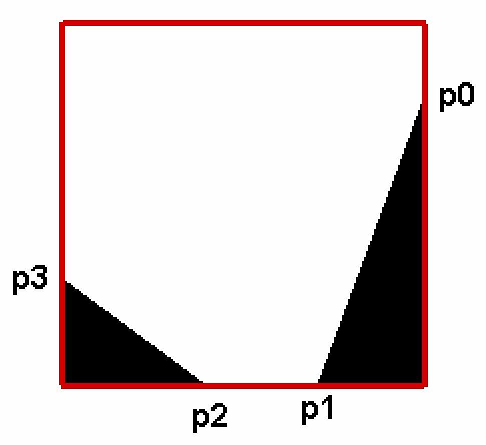


a) Split index= -1

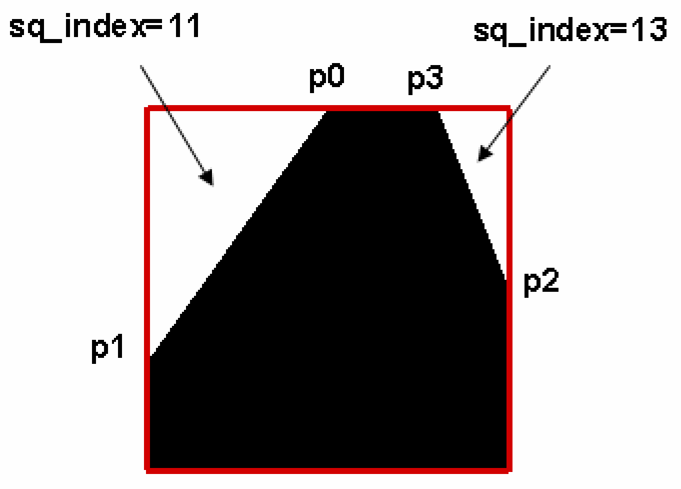


b) Split index=1

11-) Square index =9

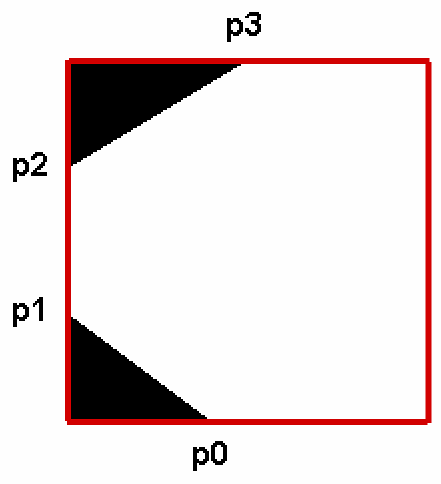


a) Split index= -1

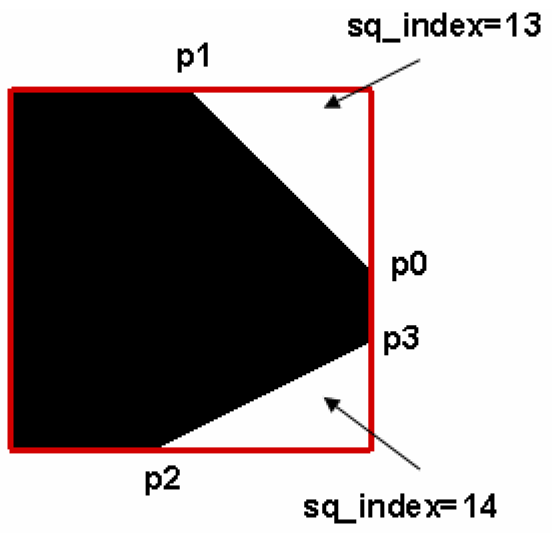


b) Split index=2

12-) Square index =12



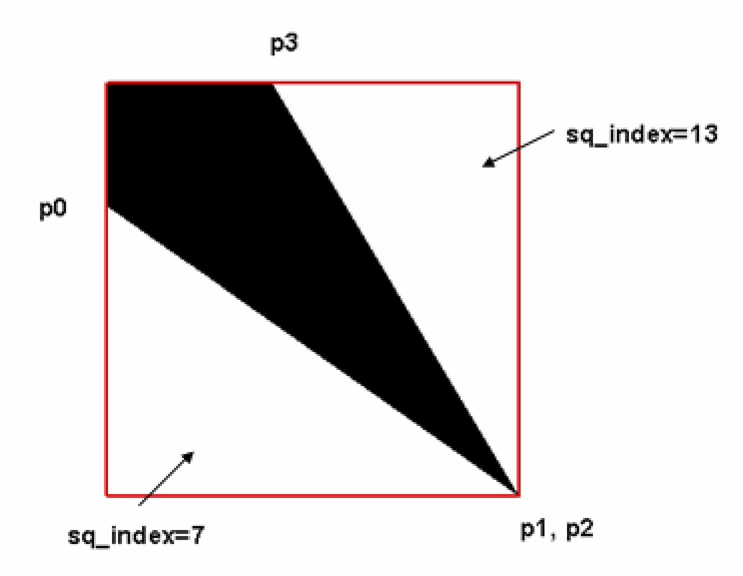
a) Split index= -1



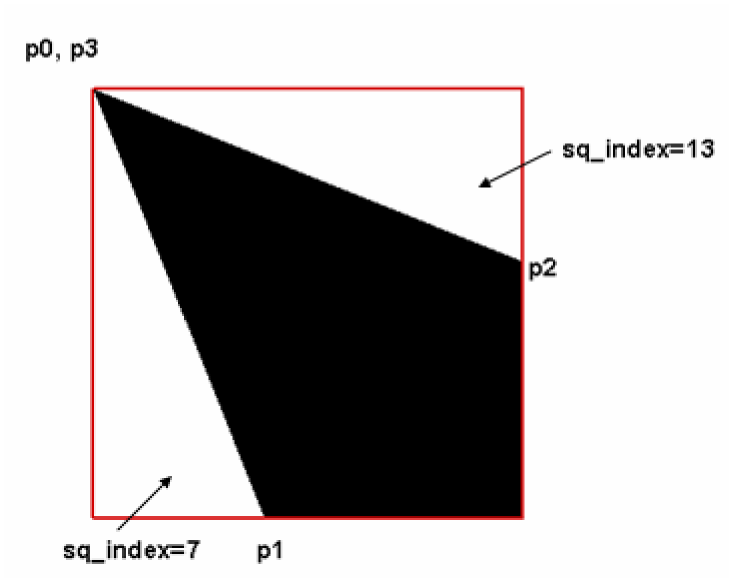
b) Split index=3

13-) Square index =5

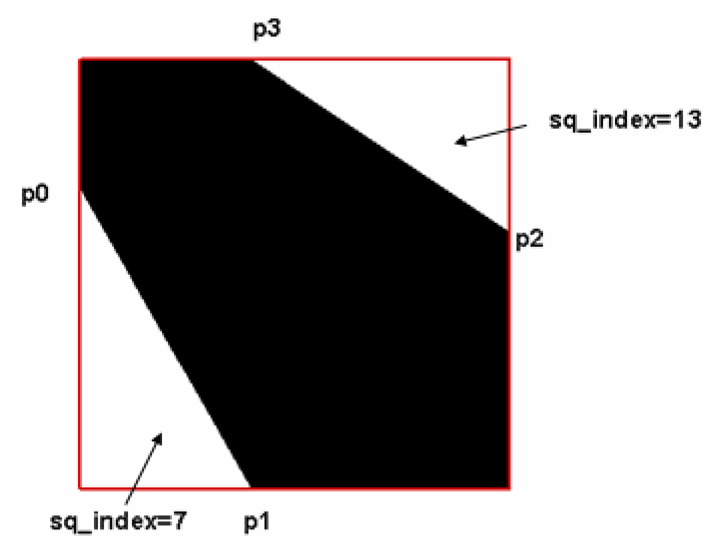
a) Split index=1


p3
sq_index=13
p0
sq_index=7
p1, p2

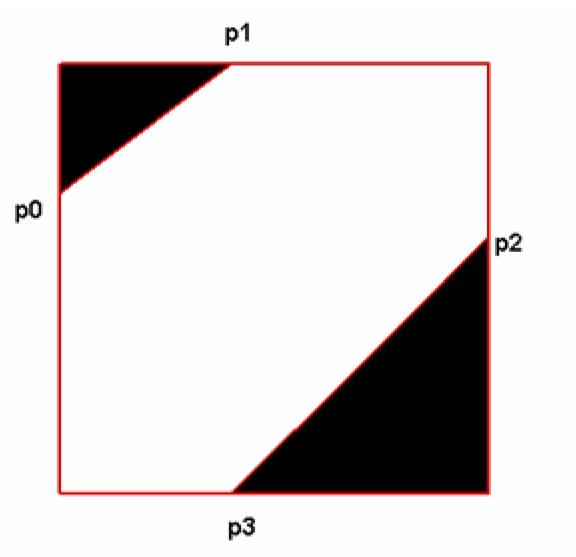
OR


p0, p3
sq_index=13
p2
sq_index=7
p1

OR

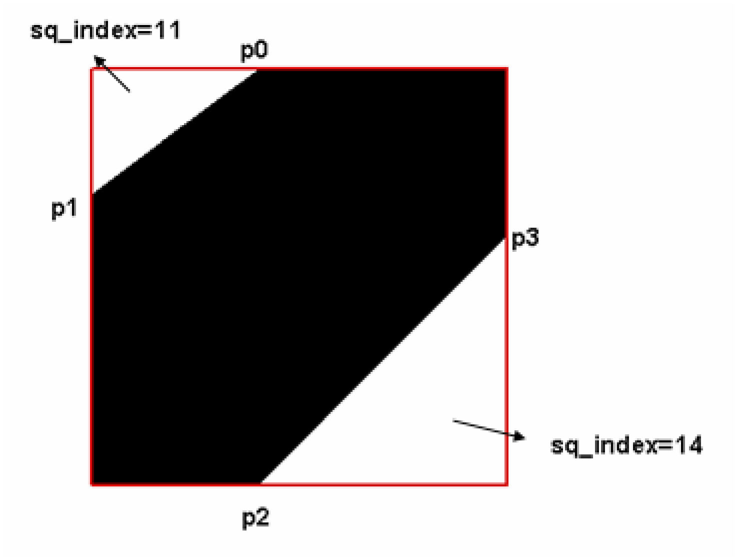


b) Split index=2

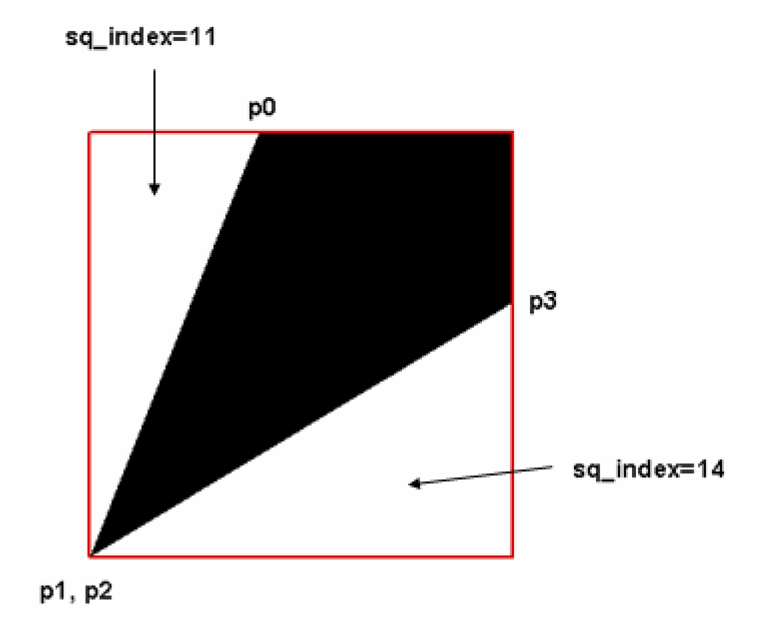


14-) Square index =10

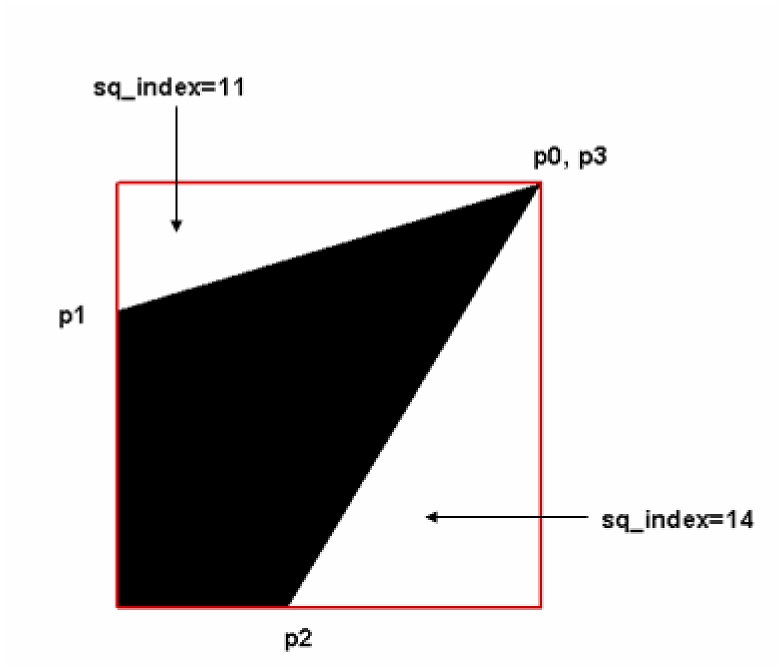
a) Split index=1



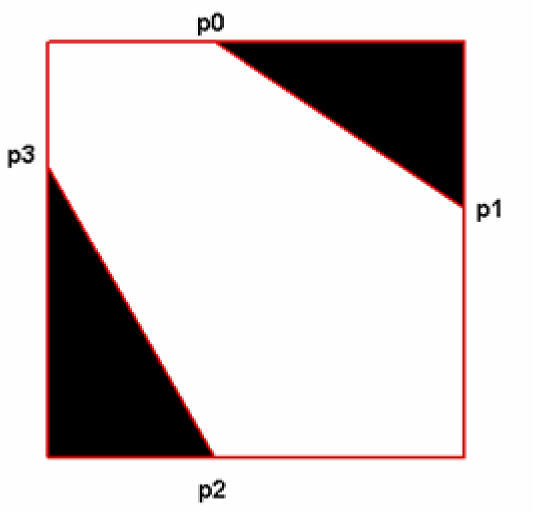
OR



OR



b) Split index=2



APPENDIX B

EXAMPLE OUTPUT FILE OF SURFACE MESH GENERATED BY GAMBIT

Total number of triangles formed surface mesh

Total node number around sphere

```
        CONTROL INFO 2.4.6
** GAMBIT NEUTRAL FILE
sphere
PROGRAM:                Gambit     VERSION:  2.4.6
 Jun 2009
 NUMNP      NELEM      NGRPS     NBSETS     NDFCD     NDFVL
    25         46          1          0         2         3
ENDOFSECTION
   NODAL COORDINATES 2.4.6
        1 -1.98288972275e+000  2.61052384440e-001  0.00000000000e+000
        2 -1.45577697798e+000  1.26273488645e-001  1.36556522965e+000
        3 -1.22755301316e+000  1.51559232680e+000  4.42824455988e-001
        4  1.66467887690e+000 -7.90424948610e-001 -7.77221099450e-001
        5 -1.51263359096e-001  1.22057996444e+000  1.57711887523e+000
        6  8.33244842460e-002 -3.52562841708e-001  1.96691547174e+000
        7 -8.31638007524e-001 -1.26804513721e+000  1.30400910826e+000
        8 -1.63860008609e+000 -1.14545623316e+000  5.40349496504e-002
        9 -1.39349606421e+000  1.12940622517e+000 -8.84652642326e-001
       10 -1.45570883917e+000 -2.15792594965e-001 -1.35438005431e+000
       11  1.39315797478e-001  1.96367326596e+000  3.52956673159e-001
       12 -3.82949059967e-001  1.83415696214e+000 -6.99441388313e-001
       13  1.19730956832e+000  4.97101249450e-001  1.52293799788e+000
       14  1.17281148967e+000  1.45628681471e+000  7.09747788310e-001
       15  4.09300059745e-001 -1.51163265674e+000  1.24396148340e+000
       16  1.36943001094e+000 -6.86750019647e-001  1.28570442001e+000
       17 -2.05031944964e-001 -1.98846808698e+000  6.29012609109e-002
       18 -7.44205012438e-001 -1.46075315461e+000 -1.14558243735e+000
       19  1.98870882714e+000  2.01972285618e-001  6.51490344266e-002
       20  1.02712841749e+000  1.52787645233e+000 -7.81409470374e-001
       21 -2.39687456491e-001  8.96476377967e-001 -1.77168846780e+000
       22  1.33810796503e+000 -1.48025166283e+000  1.35359109823e-001
       23  7.18107022266e-001 -1.44912995140e+000 -1.17658178148e+000
       24 -7.38694124748e-002 -5.51126718628e-001 -1.92114618130e+000
       25  1.23662915548e+000  6.40886642735e-002 -1.57055435275e+000
ENDOFSECTION
```

```
      ELEMENTS/CELLS 2.4.6
       1  3  3        1        2        3
       2  3  3        3        2        5
       3  3  3        5        2        6
       4  3  3        6        2        7
       5  3  3        7        2        8
       6  3  3        8        2        1
       7  3  3        1        3        9
       8  3  3        1        9       10
       9  3  3        1       10        8
      10  3  3        3        5       11
      11  3  3        3       11       12
      12  3  3        3       12        9
      13  3  3        5        6       13
      14  3  3        5       13       14
      15  3  3        5       14       11
      16  3  3        6        7       15
      17  3  3        6       15       16
      18  3  3        6       16       13
      19  3  3        7        8       17
      20  3  3        7       17       15
      21  3  3        8       10       18
      22  3  3        8       18       17
      23  3  3       13       16       19
      24  3  3       13       19       14
      25  3  3       12       11       20
      26  3  3       20       11       14
      27  3  3        9       12       21
      28  3  3        9       21       10
      29  3  3       16       15       22
      30  3  3       22       15       17
      31  3  3       14       19       20
      32  3  3       16       22       19
      33  3  3       21       12       20
      34  3  3       22       17       23
      35  3  3       23       17       18
      36  3  3       18       10       24
      37  3  3       24       10       21
      38  3  3       23       18       24
      39  3  3       24       21       25
      40  3  3       25       21       20
      41  3  3       20       19       25
      42  3  3       19       22        4
      43  3  3       19        4       25
      44  3  3       22       23        4
      45  3  3        4       23       25
      46  3  3       23       24       25
ENDOFSECTION
       ELEMENT GROUP 2.4.6
GROUP:          1 ELEMENTS:         46 MATERIAL:          2 NFLAGS:          1
                           fluid
       0
       1       2       3       4       5       6       7       8       9      10
      11      12      13      14      15      16      17      18      19      20
      21      22      23      24      25      26      27      28      29      30
      31      32      33      34      35      36      37      38      39      40
      41   42   43   44   45   46
ENDOFSECTION
```

Node numbers constitute each of triangles

APPENDIX C

TABLE FOR MARCHING CUBES ALGORITHM

A table (triangle table) is used to look up triangular facets (cut surfaces). There are at most five triangular facets in this table [39]. The table given here is a modified table in order to find centroids in the same manner for the developed method. Original table can be found in reference [39].

```
int triangleTable[256][16] =
{{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 9, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2,10, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1},
{3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 11, 2, 8, 11, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 9, 0, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 11, 2, 1, 9, 11, 9, 8, 11, -1, -1, -1, -1, -1, -1, -1},
{3, 10, 1, 11, 10, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 10, 1, 0, 8, 10, 8, 11, 10, -1, -1, -1, -1, -1, -1, -1},
{3, 9, 0, 3, 11, 9, 11, 10, 9, -1, -1, -1, -1, -1, -1, -1},
{9, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 8, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 3, 0, 7, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 1, 9, 4, 7, 1, 7, 3, 1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 4, 7, 3, 0, 4, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1},
{9, 2, 10, 9, 0, 2, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1},
{2, 10, 9, 2, 9, 7, 2, 7, 3, 7, 9, 4, -1, -1, -1, -1},
{8, 4, 7, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 4, 7, 11, 2, 4, 2, 0, 4, -1, -1, -1, -1, -1, -1, -1},
{9, 0, 1, 8, 4, 7, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1},
{4, 7, 11, 9, 4, 11, 9, 11, 2, 9, 2, 1, -1, -1, -1, -1},
{3, 10, 1, 3, 11, 10, 7, 8, 4, -1, -1, -1, -1, -1, -1, -1},
{1, 11, 10, 1, 4, 11, 1, 0, 4, 7, 11, 4, -1, -1, -1, -1},
{4, 7, 8, 9, 0, 11, 9, 11, 10, 11, 0, 3, -1, -1, -1, -1},
{4, 7, 11, 4, 11, 9, 9, 11, 10, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 4, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 5, 4, 1, 5, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 5, 4, 8, 3, 5, 3, 1, 5, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 8, 1, 2, 10, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1},
{5, 2, 10, 5, 4, 2, 4, 0, 2, -1, -1, -1, -1, -1, -1, -1},
{2, 10, 5, 3, 2, 5, 3, 5, 4, 3, 4, 8, -1, -1, -1, -1},
{9, 5, 4, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 11, 2, 0, 8, 11, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1},
{0, 5, 4, 0, 1, 5, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1},
{2, 1, 5, 2, 5, 8, 2, 8, 11, 4, 8, 5, -1, -1, -1, -1},
{10, 3, 11, 10, 1, 3, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 5, 0, 8, 1, 8, 10, 1, 8, 11, 10, -1, -1, -1, -1},
{5, 4, 0, 5, 0, 11, 5, 11, 10, 11, 0, 3, -1, -1, -1, -1},
{5, 4, 8, 5, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1},
{9, 7, 8, 5, 7, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 3, 0, 9, 5, 3, 5, 7, 3, -1, -1, -1, -1, -1, -1, -1},
{0, 7, 8, 0, 1, 7, 1, 5, 7, -1, -1, -1, -1, -1, -1, -1},
{1, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 7, 8, 9, 5, 7, 10, 1, 2, -1, -1, -1, -1, -1, -1, -1},
{10, 1, 2, 9, 5, 0, 5, 3, 0, 5, 7, 3, -1, -1, -1, -1},
{8, 0, 2, 8, 2, 5, 8, 5, 7, 10, 5, 2, -1, -1, -1, -1},
{2, 10, 5, 2, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1},
{7, 9, 5, 7, 8, 9, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 7, 9, 7, 2, 9, 2, 0, 2, 7, 11, -1, -1, -1, -1},
{2, 3, 11, 0, 1, 8, 1, 7, 8, 1, 5, 7, -1, -1, -1, -1},
{11, 2, 1, 11, 1, 7, 7, 1, 5, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 8, 8, 5, 7, 10, 1, 3, 10, 3, 11, -1, -1, -1, -1},
{5, 7, 0, 5, 0, 9, 7, 11, 0, 1, 0, 10, 11, 10, 0, -1},
{11, 10, 0, 11, 0, 3, 10, 5, 0, 8, 0, 7, 5, 7, 0, -1},
{11, 10, 5, 7, 11, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{ 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 0, 1, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 8, 3, 1, 9, 8, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1},
{1, 6, 5, 2, 6, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 6, 5, 1, 2, 6, 3, 0, 8, -1, -1, -1, -1, -1, -1, -1},
{9, 6, 5, 9, 0, 6, 0, 2, 6, -1, -1, -1, -1, -1, -1, -1},
{5, 9, 8, 5, 8, 2, 5, 2, 6, 3, 2, 8, -1, -1, -1, -1},
{2, 3, 11, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 0, 8, 11, 2, 0, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1},
{5, 10, 6, 1, 9, 2, 9, 11, 2, 9, 8, 11, -1, -1, -1, -1},
{6, 3, 11, 6, 5, 3, 5, 1, 3, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 11, 0, 11, 5, 0, 5, 1, 5, 11, 6, -1, -1, -1, -1},
{3, 11, 6, 0, 3, 6, 0, 6, 5, 0, 5, 9, -1, -1, -1, -1},
{6, 5, 9, 6, 9, 11, 11, 9, 8, -1, -1, -1, -1, -1, -1, -1},
{5, 10, 6, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 3, 0, 4, 7, 3, 6, 5, 10, -1, -1, -1, -1, -1, -1, -1},
{1, 9, 0, 5, 10, 6, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1},
{10, 6, 5, 1, 9, 7, 1, 7, 3, 7, 9, 4, -1, -1, -1, -1},
{6, 1, 2, 6, 5, 1, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 5, 5, 2, 6, 3, 0, 4, 3, 4, 7, -1, -1, -1, -1},
{8, 4, 7, 9, 0, 5, 0, 6, 5, 0, 2, 6, -1, -1, -1, -1},
{7, 3, 9, 7, 9, 4, 3, 2, 9, 5, 9, 6, 2, 6, 9, -1},
{3, 11, 2, 7, 8, 4, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1},
{5, 10, 6, 4, 7, 2, 4, 2, 0, 2, 7, 11, -1, -1, -1, -1},
{0, 1, 9, 4, 7, 8, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1},
{9, 2, 1, 9, 11, 2, 9, 4, 11, 7, 11, 4, 5, 10, 6, -1},
{8, 4, 7, 3, 11, 5, 3, 5, 1, 5, 11, 6, -1, -1, -1, -1},
{5, 1, 11, 5, 11, 6, 1, 0, 11, 7, 11, 4, 0, 4, 11, -1},
{0, 5, 9, 0, 6, 5, 0, 3, 6, 11, 6, 3, 8, 4, 7, -1},
{6, 5, 9, 6, 9, 11, 4, 7, 9, 7, 11, 9, -1, -1, -1, -1},
{10, 4, 9, 6, 4, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 10, 6, 4, 9, 10, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1},
{10, 0, 1, 10, 6, 0, 6, 4, 0, -1, -1, -1, -1, -1, -1, -1},
{8, 3, 1, 8, 1, 6, 8, 6, 4, 6, 1, 10, -1, -1, -1, -1},
{1, 4, 9, 1, 2, 4, 2, 6, 4, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 8, 1, 2, 9, 2, 4, 9, 2, 6, 4, -1, -1, -1, -1},
{0, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 3, 2, 8, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1},
{10, 4, 9, 10, 6, 4, 11, 2, 3, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 2, 2, 8, 11, 4, 9, 10, 4, 10, 6, -1, -1, -1, -1},
{3, 11, 2, 0, 1, 6, 0, 6, 4, 6, 1, 10, -1, -1, -1, -1},
{6, 4, 1, 6, 1, 10, 4, 8, 1, 2, 1, 11, 8, 11, 1, -1},
{9, 6, 4, 9, 3, 6, 9, 1, 3, 11, 6, 3, -1, -1, -1, -1},
{8, 11, 1, 8, 1, 0, 11, 6, 1, 9, 1, 4, 6, 4, 1, -1},
{3, 11, 6, 3, 6, 0, 0, 6, 4, -1, -1, -1, -1, -1, -1, -1},
{6, 4, 8, 11, 6, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 10, 6, 7, 8, 10, 8, 9, 10, -1, -1, -1, -1, -1, -1, -1},
{0, 7, 3, 0, 10, 7, 0, 9, 10, 6, 7, 10, -1, -1, -1, -1},
{10, 6, 7, 1, 10, 7, 1, 7, 8, 1, 8, 0, -1, -1, -1, -1},
{10, 6, 7, 10, 7, 1, 1, 7, 3, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 6, 1, 6, 8, 1, 8, 9, 8, 6, 7, -1, -1, -1, -1},
{2, 6, 9, 2, 9, 1, 6, 7, 9, 0, 9, 3, 7, 3, 9, -1},
{7, 8, 0, 7, 0, 6, 6, 0, 2, -1, -1, -1, -1, -1, -1, -1},
{7, 3, 2, 6, 7, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 3, 11, 10, 6, 8, 10, 8, 9, 8, 6, 7, -1, -1, -1, -1},
{2, 0, 7, 2, 7, 11, 0, 9, 7, 6, 7, 10, 9, 10, 7, -1},
{1, 8, 0, 1, 7, 8, 1, 10, 7, 6, 7, 10, 2, 3, 11, -1},
{11, 2, 1, 11, 1, 7, 10, 6, 1, 6, 7, 1, -1, -1, -1, -1},
{8, 9, 6, 8, 6, 7, 9, 1, 6, 11, 6, 3, 1, 3, 6, -1},
{0, 9, 1, 11, 6, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 8, 0, 7, 0, 6, 3, 11, 0, 11, 6, 0, -1, -1, -1, -1},
{7, 11, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{6, 11, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 8, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 1, 9, 8, 3, 1, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1},
{10, 1, 2, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 3, 0, 8, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1},
{2, 9, 0, 2, 10, 9, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1},
{6, 11, 7, 2, 10, 3, 10, 8, 3, 10, 9, 8, -1, -1, -1, -1},
{7, 2, 3, 6, 2, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{7, 0, 8, 7, 6, 0, 6, 2, 0, -1, -1, -1, -1, -1, -1, -1},
{2, 7, 6, 2, 3, 7, 0, 1, 9, -1, -1, -1, -1, -1, -1, -1},
{1, 6, 2, 1, 8, 6, 1, 9, 8, 8, 7, 6, -1, -1, -1, -1},
{10, 7, 6, 10, 1, 7, 1, 3, 7, -1, -1, -1, -1, -1, -1, -1},
{10, 7, 6, 1, 7, 10, 1, 8, 7, 1, 0, 8, -1, -1, -1, -1},
{0, 3, 7, 0, 7, 10, 0, 10, 9, 6, 10, 7, -1, -1, -1, -1},
{7, 6, 10, 7, 10, 8, 8, 10, 9, -1, -1, -1, -1, -1, -1, -1},
{6, 8, 4, 11, 8, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 6, 11, 3, 0, 6, 0, 4, 6, -1, -1, -1, -1, -1, -1, -1},
{8, 6, 11, 8, 4, 6, 9, 0, 1, -1, -1, -1, -1, -1, -1, -1},
{9, 4, 6, 9, 6, 3, 9, 3, 1, 11, 3, 6, -1, -1, -1, -1},
{6, 8, 4, 6, 11, 8, 2, 10, 1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 3, 0, 11, 0, 6, 11, 0, 4, 6, -1, -1, -1, -1},
{4, 11, 8, 4, 6, 11, 0, 2, 9, 2, 10, 9, -1, -1, -1, -1},
{10, 9, 3, 10, 3, 2, 9, 4, 3, 11, 3, 6, 4, 6, 3, -1},
{8, 2, 3, 8, 4, 2, 4, 6, 2, -1, -1, -1, -1, -1, -1, -1},
{0, 4, 2, 2, 4, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 9, 0, 2, 3, 4, 2, 4, 6, 4, 3, 8, -1, -1, -1, -1},
{1, 9, 4, 1, 4, 2, 2, 4, 6, -1, -1, -1, -1, -1, -1, -1},
{8, 1, 3, 8, 6, 1, 8, 4, 6, 6, 10, 1, -1, -1, -1, -1},
{10, 1, 0, 10, 0, 6, 6, 0, 4, -1, -1, -1, -1, -1, -1, -1},
{4, 6, 3, 4, 3, 8, 6, 10, 3, 0, 3, 9, 10, 9, 3, -1},
{10, 9, 4, 6, 10, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 5, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 4, 9, 5, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1},
{5, 0, 1, 5, 4, 0, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1},
{11, 7, 6, 8, 3, 4, 3, 5, 4, 3, 1, 5, -1, -1, -1, -1},
{9, 5, 4, 10, 1, 2, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1},
{6, 11, 7, 1, 2, 10, 0, 8, 3, 4, 9, 5, -1, -1, -1, -1},
{7, 6, 11, 5, 4, 10, 4, 2, 10, 4, 0, 2, -1, -1, -1, -1},
{3, 4, 8, 3, 5, 4, 3, 2, 5, 10, 5, 2, 11, 7, 6, -1},
{7, 2, 3, 7, 6, 2, 5, 4, 9, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 4, 0, 8, 6, 0, 6, 2, 6, 8, 7, -1, -1, -1, -1},
{3, 6, 2, 3, 7, 6, 1, 5, 0, 5, 4, 0, -1, -1, -1, -1},
{6, 2, 8, 6, 8, 7, 2, 1, 8, 4, 8, 5, 1, 5, 8, -1},
{9, 5, 4, 10, 1, 6, 1, 7, 6, 1, 3, 7, -1, -1, -1, -1},
{1, 6, 10, 1, 7, 6, 1, 0, 7, 8, 7, 0, 9, 5, 4, -1},
{4, 0, 10, 4, 10, 5, 0, 3, 10, 6, 10, 7, 3, 7, 10, -1},
{7, 6, 10, 7, 10, 8, 5, 4, 10, 4, 8, 10, -1, -1, -1, -1},
{6, 9, 5, 6, 11, 9, 11, 8, 9, -1, -1, -1, -1, -1, -1, -1},
{3, 6, 11, 0, 6, 3, 0, 5, 6, 0, 9, 5, -1, -1, -1, -1},
{0, 11, 8, 0, 5, 11, 0, 1, 5, 5, 6, 11, -1, -1, -1, -1},
{6, 11, 3, 6, 3, 5, 5, 3, 1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 9, 5, 11, 9, 11, 8, 11, 5, 6, -1, -1, -1, -1},
{0, 11, 3, 0, 6, 11, 0, 9, 6, 5, 6, 9, 1, 2, 10, -1},
{11, 8, 5, 11, 5, 6, 8, 0, 5, 10, 5, 2, 0, 2, 5, -1},
{6, 11, 3, 6, 3, 5, 2, 10, 3, 10, 5, 3, -1, -1, -1, -1},
{5, 8, 9, 5, 2, 8, 5, 6, 2, 3, 8, 2, -1, -1, -1, -1},
{9, 5, 6, 9, 6, 0, 0, 6, 2, -1, -1, -1, -1, -1, -1, -1},
{1, 5, 8, 1, 8, 0, 5, 6, 8, 3, 8, 2, 6, 2, 8, -1},
{1, 5, 6, 2, 1, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 3, 6, 1, 6, 10, 3, 8, 6, 5, 6, 9, 8, 9, 6, -1},
{10, 1, 0, 10, 0, 6, 9, 5, 0, 5, 6, 0, -1, -1, -1, -1},
{0, 3, 8, 5, 6, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{10, 5, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 5, 10, 7, 5, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 5, 10, 11, 7, 5, 8, 3, 0, -1, -1, -1, -1, -1, -1, -1},
{5, 11, 7, 5, 10, 11, 1, 9, 0, -1, -1, -1, -1, -1, -1, -1},
{10, 7, 5, 10, 11, 7, 9, 8, 1, 8, 3, 1, -1, -1, -1, -1},
{11, 1, 2, 11, 7, 1, 7, 5, 1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 1, 2, 7, 1, 7, 5, 7, 2, 11, -1, -1, -1, -1},
{9, 7, 5, 9, 2, 7, 9, 0, 2, 2, 11, 7, -1, -1, -1, -1},
{7, 5, 2, 7, 2, 11, 5, 9, 2, 3, 2, 8, 9, 8, 2, -1},
{2, 5, 10, 2, 3, 5, 3, 7, 5, -1, -1, -1, -1, -1, -1, -1},
{8, 2, 0, 8, 5, 2, 8, 7, 5, 10, 2, 5, -1, -1, -1, -1},
{9, 0, 1, 5, 10, 3, 5, 3, 7, 3, 10, 2, -1, -1, -1, -1},
{9, 8, 2, 9, 2, 1, 8, 7, 2, 10, 2, 5, 7, 5, 2, -1},
{1, 3, 5, 5, 3, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 7, 0, 7, 1, 1, 7, 5, -1, -1, -1, -1, -1, -1, -1},
{9, 0, 3, 9, 3, 5, 5, 3, 7, -1, -1, -1, -1, -1, -1, -1},
{9, 8, 7, 5, 9, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{5, 8, 4, 5, 10, 8, 10, 11, 8, -1, -1, -1, -1, -1, -1, -1},
{5, 0, 4, 5, 11, 0, 5, 10, 11, 11, 3, 0, -1, -1, -1, -1},
{0, 1, 9, 8, 4, 10, 8, 10, 11, 10, 4, 5, -1, -1, -1, -1},
{10, 11, 4, 10, 4, 5, 11, 3, 4, 9, 4, 1, 3, 1, 4, -1},
{2, 5, 1, 2, 8, 5, 2, 11, 8, 4, 5, 8, -1, -1, -1, -1},
{0, 4, 11, 0, 11, 3, 4, 5, 11, 2, 11, 1, 5, 1, 11, -1},
{0, 2, 5, 0, 5, 9, 2, 11, 5, 4, 5, 8, 11, 8, 5, -1},
{9, 4, 5, 2, 11, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 5, 10, 3, 5, 2, 3, 4, 5, 3, 8, 4, -1, -1, -1, -1},
{5, 10, 2, 5, 2, 4, 4, 2, 0, -1, -1, -1, -1, -1, -1, -1},
{3, 10, 2, 3, 5, 10, 3, 8, 5, 4, 5, 8, 0, 1, 9, -1},
{5, 10, 2, 5, 2, 4, 1, 9, 2, 9, 4, 2, -1, -1, -1, -1},
{8, 4, 5, 8, 5, 3, 3, 5, 1, -1, -1, -1, -1, -1, -1, -1},
{0, 4, 5, 1, 0, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 4, 5, 8, 5, 3, 9, 0, 5, 0, 3, 5, -1, -1, -1, -1},
{9, 4, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 11, 7, 4, 9, 11, 9, 10, 11, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 4, 9, 7, 9, 11, 7, 9, 10, 11, -1, -1, -1, -1},
{1, 10, 11, 1, 11, 4, 1, 4, 0, 7, 4, 11, -1, -1, -1, -1},
{3, 1, 4, 3, 4, 8, 1, 10, 4, 7, 4, 11, 10, 11, 4, -1},
{4, 11, 7, 9, 11, 4, 9, 2, 11, 9, 1, 2, -1, -1, -1, -1},
{9, 7, 4, 9, 11, 7, 9, 1, 11, 2, 11, 1, 0, 8, 3, -1},
{11, 7, 4, 11, 4, 2, 2, 4, 0, -1, -1, -1, -1, -1, -1, -1},
{11, 7, 4, 11, 4, 2, 8, 3, 4, 3, 2, 4, -1, -1, -1, -1},
{2, 9, 10, 2, 7, 9, 2, 3, 7, 7, 4, 9, -1, -1, -1, -1},
{9, 10, 7, 9, 7, 4, 10, 2, 7, 8, 7, 0, 2, 0, 7, -1},
{3, 7, 10, 3, 10, 2, 7, 4, 10, 1, 10, 0, 4, 0, 10, -1},
{1, 10, 2, 8, 7, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 1, 4, 1, 7, 7, 1, 3, -1, -1, -1, -1, -1, -1, -1},
{4, 9, 1, 4, 1, 7, 0, 8, 1, 8, 7, 1, -1, -1, -1, -1},
{4, 0, 3, 7, 4, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 8, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 10, 8, 8, 10, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 9, 3, 9, 11, 11, 9, 10, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 10, 0, 10, 8, 8, 10, 11, -1, -1, -1, -1, -1, -1, -1},
{3, 1, 10, 11, 3, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 11, 1, 11, 9, 9, 11, 8, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 9, 3, 9, 11, 1, 2, 9, 2, 11, 9, -1, -1, -1, -1},
{0, 2, 11, 8, 0, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 3, 8, 2, 8, 10, 10, 8, 9, -1, -1, -1, -1, -1, -1, -1},
{9, 10, 2, 0, 9, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 3, 8, 2, 8, 10, 0, 1, 8, 1, 10, 8, -1, -1, -1, -1},
{1, 10, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 3, 8, 9, 1, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 9, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 3, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}};
```