Interactive Grid Generation for Computational Fluid Dynamics in Flow Domains of Complex Geometry

Ahmed Sadek Mohamed Tawfik

April 1, 2015

ABSTRACT

The thesis presents, in detail, an interactive computer program developed for creating quadrilateral and hexahedral grids for Computational Fluid Dynamics (CFD) applications. The program relies heavily on interaction with the user in a sophisticated visual graphical environment.

The software package is based on the block structured concept where the flow domain is divided into sub-domains. A structured grid is created in each sub-domain and the sub-domains are all connected in the final grid. The program allows the user to have full control on the grid parameters necessary for obtaining numerically stable, bounded, and accurate solution of the governing equations, namely the grid spacing, intensity, orthogonality, and smoothness.

The package also allows the user to easily define a multitude of boundary surfaces and conditions to the final three-dimensional grid generated, these include inlets, outlets, planes of symmetry, baffles, etc. The package also featured the ability to export the grid data to files compatible with some commercial CFD packages formats.

The program was applied to generate grids in geometries of increasing complexity from the straight pipe, to the poppet valve duct of an internal combustion engine, a composite valve assembly and the rotor of a centrifugal pump among other cases. The sequences of grid generation, presented in chapter seven, for these cases demonstrated the power, flexibility, and speed with which the three-dimensional grids were created and the ease with which the boundary surfaces were defined.

ACKNOWLEDGEMENTS

A very special gratitude goes to Professor Ashraf M. Y. Ahmed for his work as my advisor and mentor in understanding the CFD Grids and his extensive support in teaching me the fundmentals of geometry. Second gratitude goes to Professor Ayman El Baz for his continuous encouragement during finishing this work.

Last, but not least, I thank my parents for the sacrifices they borne to ensure the fulfillment of my dreams.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

$[ij,\ k]$ Second Kind Christoffel Symbols

δ_i^j Kronecker delta

η Natural Coordinate

Γ_{ij}^k Christoffel Symbols

$\mathbb{T}$ Stress Tensor

$\mathbf{e}^i$ Contravariant base vectors

$\mathbf{e}_i$ Covariant base vectors

$\mathbf{J}$ Jacobian matrix

$\mathbf{r}$ Position vector

$\mathbf{V}$ Velocity vector

ρ Fluid Density

ξ Natural Coordinate

g^{ij} Contravariant metric tensor

g_{ij} Covariant metric tensor

h_i Scale factors

r^i Contravariant vector components

r_i Covariant vector components

ABBREVIATIONS

CFD	Computational Fluid Dynamics
CAE	Computer Aided Engineering
CAD	Computer Aided Design
OpenGL	Open Graphics Library
FDM	Finite Difference Method
FEM	Finite Element Method
FVM	Finite Volume Method
PDE	Partial Differential Equation

CHAPTER I

INTRODUCTION

1.1 BACKGROUND

Computational Fluid Dynamics (CFD) is a complex task that has always been considered as a workflow consisting of many processes. Each process has its unique algorithms, tools, and applications. The available techniques used by CFD applications cannot be adopted without the existence of a computaional grid that fills the flow domain inside and/or outside the flow domain under study. Fulfilling this important requirement led to many attempts in writing software packages to acheive just that [1, 2, 3]. The computational cells have to be connected with each other, covering all the empty space, and with definite boundaries that hold the initial fluid properties that will be carried through the solution process as shown in Figs. (1.1-1.2).

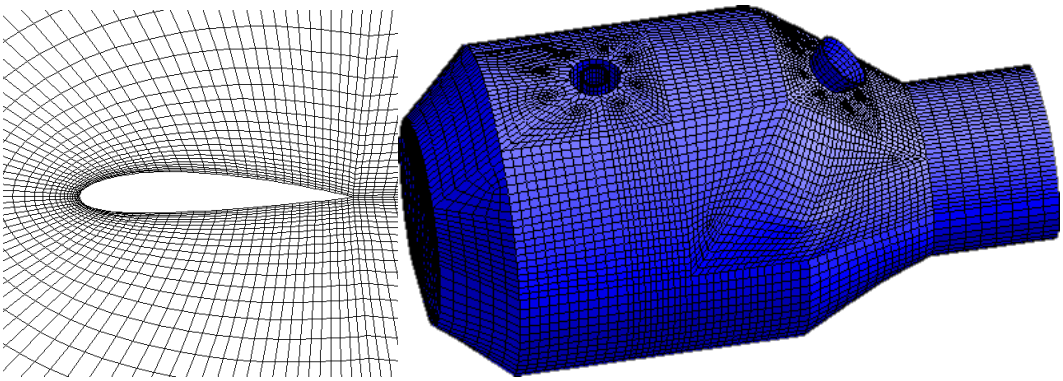


Figure 1.1: Multi Gridded Samples I[4]

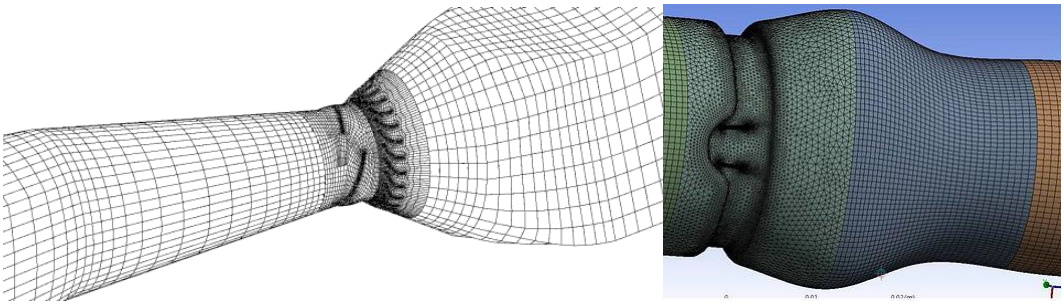


Figure 1.2: Multi Gridded Samples II[5]

Early CFD applications were mainly carried out on simple rectilinear grids. This had

the advantage of simpler governing equations and more robust solution algorithms. On the other hand, these grids failed to simulate flow domains of complex geometry with reasonable degree of accuracy despite the fact that such complex geometry flow conditions represent the majority of engineering applications. Figure (1.3) illustrates two such rectilinear grids with varying degrees of approximations.

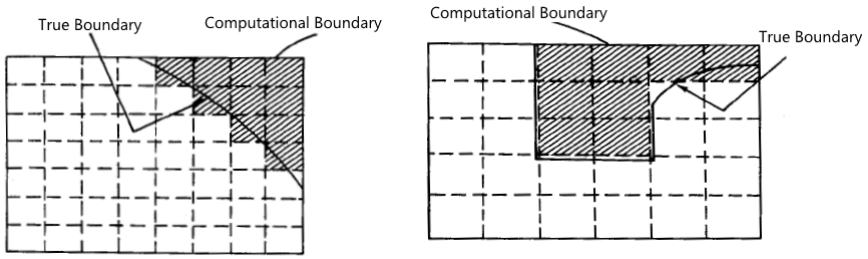


Figure 1.3: Rectilinear Grids in Irregular Geometry[6]

The first efforts concerned with the development of grid techniques were undertaken in the 1960s with significant number of advanced methods being created: algebraic, elliptic, hyperbolic, parabolic, variational, Delaunay, advancing-front, etc. These methods reached a stage where calculations in fairly complicated multi-dimensional domains became possible. Because of its successful development, the field of numerical grid generation has already formed a separate mathematical discipline with its own methodology, approaches, and technology.

In the mid 1980's serious attention was directed towards casting the governing equations in coordinate free form in preparation for discretization over non-orthogonal curvilinear grids covering complex geometry flow domains[7, 1]. The methodolgy required algorithms for generating the computational grid that were simple for testing accuracy, convergence, and stability of the proposed solution techniques. At the time, these algorithms were simple, cumbersome, and offered very little control over the generated grid from the computational view point.

At the end of the 1980s there started a new stage in the development of grid generation techniques. It is characterized by the creation of comprehensive, multipurpose, three-dimensional grid generation codes which aimed at providing a uniform environment for the construction of grids in arbitrary multidimensional geometries. Despite these efforts, grid generation remains a major obstacle in CFD workflow and the most demanding process in terms of effort and time. The review of NASA[8] on the future of CFD in aeroscience in the near future till 2030 states "Today, the generation of suitable meshes for CFD simulations about complex configurations constitutes a principal bottleneck in the simulation workflow process." . The review also states in conclusion "Given a suitable geometry representation and a desired level of solution accuracy, a fully automated meshing capability would construct a suitable mesh and adaptively refine this mesh throughout the solution process with minimal user intervention until the final accuracy levels are met."

This objective at the present time seems a long way to reach though efforts in this direction are already being made as further detailed in Chapter II.

1.1.1 Elements of CFD Packages

CFD codes are structured around the numerical algorithms that can tackle fluid flow problems. In order to provide easy access to their solving power, all commercial CFD packages include sophisticated user interfaces to input problem parameters and to examine the results. Hence all codes contain three main elements:

- Pre-processor
- Solver
- Post Processor

In the following, the function of each of these elements within the context of a CFD code is briefly discussed.

1.1.1.1 Pre-Processor

Pre-Processing consists of the input of a flow problem to a CFD program by means of an operator-friendly interface and subsequent transformation of this input into a form suitable for use by the solver. The user activities are the pre-processing stage involve:

- Definition of the geometry of the region of interest: the computational domain.
- Grid generation the sub-division of the domain into a number of smaller, non-overlapping sub-domains: a grid (or mesh) of cells (or control volumes or elements).
- Selection of the physical and chemical phenomena that need to be modeled.
- Definition of fluid properties.
- Specification of appropriate boundary conditions at cells which coincide with or touch the domain boundary.

The solution to a flow problem (velocity, pressure, temperature etc.) is defined at nodes inside each cell. The accuracy of a CFD solution is governed by the number of cells in the grid. In general, the larger the number of cells the better the solution accuracy. Both the accuracy of a solution and its cost in terms of necessary computer hardware and calculation time are dependent on the fineness of the grid. Optimal meshes are often non-uniform: finer in areas where large variations occur from point to point and coarser in regions with relatively little change. Efforts are under way to develop CFD codes with a (self) adaptive meshing capability. Ultimately such programs will automatically refine the grid in areas of rapid variations. A substantial amount of basic development work still needs to be done

before these techniques are robust enough to be incorporated into commercial CFD codes. At present it is still up to the skills of the CFD user to design a grid that is suitable compromise between desired accuracy and solution cost.

Over 70% of the time spent in industry on a CFD project is devoted to the definition of the domain geometry and grid generation. In order to maximize productivity of CFD personnel all the major codes now include their own CAD-style interface and/or facilities to import data from proprietary surface modelers and mesh generators.

1.1.1.2 Solver

The solver solves the governing equations to obtain the field values for the flow variables. To do so the equation must be cast in a form suitable for the solution methodology adopted.

There are three distinct methods of numerical solution techniques: Finite Difference Method, Finite Element Method and Spectral Methods. In outline, the numerical methods that form the basis of the solver perform the following steps:

- Approximation of the unknown flow variables by means of simple functions.
- Discretisation by substitution of the approximations into the governing flow equations and subsequent mathematical manipulations.
- Solution of the algebraic equations.

The main differences between the three separate methods are associated with manner in which the flow variables are approximated and with the discretisation processes.

1.1.1.3 Post-Processor

As in pre-processing, a huge amount of development work has taken place in the post-processing field. Due to the increased popularity of workstations computers, many of which have outstanding graphics capabilities. These include:

- Domain geometry and grid display
- Vector plots
- Line and shaded contour plots
- 2D and 3D surface plots
- Particle tracking
- View manipulation (translation, rotation, scaling etc.)

More recently these facilities have also include animation for dynamic result display and in addition to graphics all codes produce trusty alphanumeric output and have data export facilities for further manipulation external to the code. As many other branches of CAE (Computer Aided Engineer) packages, the graphics output capabilities of CFD codes have revolutionized the communication of ideas to the non-specialist.

1.2 MOTIVATION

As was noted above, the computational grid represents a corner stone and bottleneck in any CFD process. Grid generation proved to be the most demanding stage in terms of effort and time on the users side. There is a variety of techniques for grid generation, that range from algebraic to numerical solutions of a set of equations combined to yield the desired computational grid. Even though the founding concepts have been documented in research papers, periodicals, and text books, there is scarcely any publications on how these methods can be implemented into an integrated software package that actually runs and gives results.

Implementing and documenting the know-how of creating computational grids, permits CFD researchers to go through a more complex scenarios in the future. The latest NASA report on the future of CFD within the next fifteen years [8] clearly sets the goal for the desired development of grid generation techniques to overcome the current difficulties detailed earlier.

Although the current commercial codes and software packages allow many types of automatic grid generation, the process still needs a lot of interaction from the user to remove the unnecessary model parts and to ensure a grid quality that guarantees solver accuracy, stability, and economy. To reach such ultimate goal, the underling mathematical foundations and algortihms have to be fully understood and implemented using current programming tools and made ready for scalability to the next generations of computer hardware. The present study has, thus, been motivated by the urge for more comperhensive and practical implementation of theoretical concepts of grid generation into efficient, flexible, and easy-to-use grid generator, intended primarily for CFD developers.

1.3 THESIS OBJECTIVES

Most of grid generation packages available today are in-house projects or as part of commercial codes and, as such, are inaccessible to CFD research communities at large. Accordingly, there is no clear documented way of how to create such a software from scratch. The main objective of this study is to introduce a new software that is capable of generating computational grids in a graphical interactive three-dimensional environment. The software user is expected to build the geometry of the flow domain and the grid at the same time avoiding redundant effort and contradicting strategies. A distinguished feature of this software is that it offers the user the ability to build the grid based on his experience in CFD to ensure a grid that acheives the main requirements for the solver, namely stability, accuracy and economy.

Some of the widely adopted gridding techniques rely on partitioning the flow domain into quadrilateral and triangular cells as further shown in detail in Chapter II. The generated grid is always a mix of triangular and quadrilateral cells (in case of 2 dimensional space) or tetrahedral and hexahedral cells (in case of 3 dimensional space). The solver, for this type of mixed cell configuration must be capable of transferring its solution and descretization methodology between the two types of cells. The generated grids of these techniques are

called hexahedral dominant or tetrahedral dominant according to the major distribuion of the one of the cell types.

Although hybrid grids are being well established and used in commercial codes, the impact is always reflected on the solver ability to give accurate results for these grids. Whenever there is a performance drop and/or slow convergent, the computational grid is usually to blame for the defficencies. In the present study, this hybrid grid approach is avoided and focus is made on quadrilateral and hexahedral grids only as further detailed in Chapter II.

The software developed is intended to handle flow domains of complex geometry of any shape. The code should also be able to export files containing grid properties and geometrical quantities in a format readable by commercial packages. In order to acheive efficient performance, the developed program uses state of the art programming tools and takes advantage of GPU (Graphical Processing Unit), asynchronous programming, parallel, and vector processing.

1.4 THESIS LAYOUT

In Chapter II a review of the computational gird types and their generation techniques is presented.

Chapter III presents a concise review of the fundmentals of CFD with emphasis on the grid properties essential for stable and accurate solution of the flow governing equations.

Manifolds and homotopy are discussed in chapter IV. The basic concept have been extensively used in the methods adopted throughout development of the present grid generator.

Chapters V, and VI detail graphics programming, algorithms, and operations used in the present work in writing the grid generator package.

The grid generator algorithm is examined in detail in Chapter VII. The modeling and grid generation commands are explained with simple examples. The objective intended here is to aid the user in mastering the program and properties of generated grid. The results of the present study are presented also in chapter VII in the form of case studies on grid generation using the developed software. The cases examined began with the simplest of geometries (straight pipe) and moved to a more complex cases and ended with a poppet valve of internal combustion engines and the rotor of a centrifugal pump.

Conclusions and Recommendations are given in chapter VIII.

Three appendices A, B, and C are also included in the thesis. Appendix A, represents an easy reference for the basics of differential and integral calculus in general curvilinear coordinates. This is essential for understanding basic grid generation operations in geneal cuvilinear coordinates and subsequent algorithms.

Object Oriented Programming (OOP) details are presented in Appendix B while Appendix C contains listing of the important source code modules.

CHAPTER II

REVIEW OF GRID TYPES AND GENERATION TECHNIQUES

2.1 INTRODUCTION

Mesh generation is a corner stone in any CFD analysis. The grid must satisfy a number of geometric constraints as well as physical requirements to ensure stable and accurate solution. In complex geometry domains, grid generation becomes a formidable task in terms of effort and time. In such cases mesh generation algorithms have demanded an increasing level of automation in order to reduce the time consumed in this process. One of the first automatic methods used for quadrilateral mesh generation generated structured meshes and required domains with simple geometric shapes that could be mapped to Cartesian natural coordinate systems. The predominance of this method can be clearly noted in [9]. This technique is still widely used in commercial packages that take advantage of the relatively easy computational implementation of the method to shorten the pre-processing time. Commonly referred to as *algebraic mesh generation* or *transfinite interpolation*, the method performs the mapping transformation between the natural and physical domains by interpolating, with *blending functions*, the curves that define the physical boundaries [10, 11]. This transformation can use more complex schemes whenever smoother meshes with good control of the aspect ratio of their elements are desired. For this purpose, *elliptic generators* are used [12, 13]. In general, these approaches achieve the final mesh by applying an iterative routine to an existing algebraic mesh.

Unstructured mesh generators have also been used to automatically construct quadrilateral meshes. These methods require more complex algorithms to be implemented; however, they lend themselves to mesh in general geometries. *Grid superposition*, also known as quadtree, has been used to generate all-quadrilateral meshes. It consists of overlaying a uniform grid of points over the entire domain and properly connecting them to generate the mesh [14, 15]. Transformation from triangles is itself another method. Roughly speaking, the method combines two or more triangles and/or subdivides them to obtain quadrilateral elements.

Another technique used in conjuction with other schemes to generate quadrilateral meshes over complex geometries has been published under the title of *geometric decomposition*. This method subdivides the domain into simply connected polygons to which another method is applied to create the final mesh [16, 17, 18, 19] as discussed in detail in Chapter VI.

One of the most widely used unstructured methods, *advancing front,* has been published as automatic triangular mesh generator capable of meshing complex geometries. If the advance of the front is associated with a transformation-from-triangles scheme, a fully quadrilateral mesh generation method is obtained [20]. Recently, a *paving* method was developed by [21, 22, 23, 24], to generate quadrilateral elements directly in the front.

This chapter begins with the discussion of various types of computational grid used in CFD solution, then goes through the properties that control the grid quality, in addition to the grid classification.

Section 2.3 discusses the various common methods of grid generation in structured and unstructured grids.

Finally, section 2.4 discuss a new paradigm in solving the flow governing equations. This new paradigm contains many methods that can be best described by meshless (gridless) methods.

2.2 GRID TYPES AND CLASSIFICATIONS

There are two fundamental classes of grid popular in the numerical solution of boundary value problems in multidimensional regions: structured and unstructured grids. These classes differ in the way in which the mesh points are locally organized. In the most general sense, this means that if the local organization of the grid points and the form of the grid cells do not depend on their position but are defined by a general rule, the mesh is considered as structured. When the connection of the neighboring grid nodes varies from point to point, the mesh is called unstructured. As a result, in the structured case the connectivity of the grid is implicitly taken into account, while the connectivity of unstructured grids must be explicitly described by an appropriate data structure procedure.

The two fundamental classes of mesh give rise to three additional subdivisions of grid types: block-structured, overset, and hybrid. These types of mesh possess to some extent features of both structured and unstructured grids, thus occupying an intermediate position between the purely structured and purely unstructured grids.

2.2.1 Structured Grids

The most popular and efficient structured grids are those whose generation relies on a mapping concept. According to this concept the nodes and cells of the grid in an n-dimensional region $X^n \subset R^n$ are defined by mapping the nodes and cells of a reference (generally uniform) grid in some standard n-dimensional domain Ξ^n with a certain transformation

$$x(\xi) : \Xi^n \to X^n, \quad \xi = (\xi^1, \ldots, \xi^n), \quad x = \left(x^1, \ldots, x^n\right) \tag{2.1}$$

from Ξ^n onto X^n. The domain Ξ^n is referred to as the logical or computational domain.

The mapping concept was borrowed from examples of grids generated for geometries that are described by analytic coordinate transformations. In particular, two-dimensional transformations have often been defined by analytic functions of a complex variable. This is the case, for example, for the polar coordinate system in circular regions

$$x(\xi) = \exp(\xi^1)(\cos\xi^2, \sin\xi^2),\ r_0 \le \xi^1 \le r_1,\ 0 \le \xi^2 \le 2\pi.$$

As an illustrative example of a three-dimensional transformation, the following scaled cylindrical transformation may be considered:

$$x(\xi) : \Xi^3 \to X^3,\ \xi = (\xi^1, \xi^2, \xi^3),\ 0 \le \xi^i \le 1,\ \ i = 1, 2, 3$$

described by

$$\begin{aligned} x^1(\xi) &= r\cos\theta \\ x^2(\xi) &= r\sin\theta \\ x^3(\xi) &= H\xi^3 \end{aligned} \tag{2.2}$$

where

$$r = r_0 + (r_1 - r_0)\xi^1,\ \theta = \theta_0 + (\theta_1 - \theta_0)\xi^2,\ H > 0,$$

with

$$0 < r_0 < r_1,\ 0 \le \theta_0 < \theta_1 \le 2\pi$$

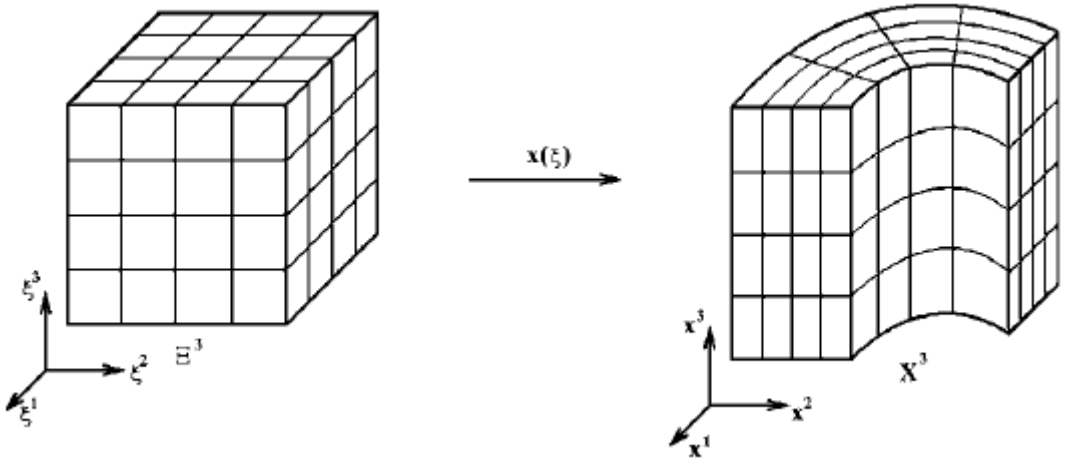


Figure 2.1: Cylindrical Structured Grid[25]

If $\theta_1 = 2\pi$ then this function transforms the unit three-dimensional cube into a space bounded by two cylinders of radii r_0 and r_1 and by the two planes $x^3 = 0$ and $x^3 = H$ as shown in Fig. (2.1). The reference uniform grid in Ξ^3 is defined by the nodes

$$\xi_{i,j,k} = (ih, jh, kh),\ 0 \leq i, j, k \leq N,\ h = 1/N,$$

where i, j, k and N are positive integers. The cells of this grid are the three dimensional cubes bounded by the coordinate planes $\xi_i^1 = ih$, $\xi_j^2 = jh$, and $\xi_k^3 = kh$. Correspondingly, the structured grid in the domain X^3 is determined by the nodes

$$x_{ijk} = x(\xi_{ijk}),\ 0 \leq i, j, k \leq N.$$

The cells of the grid in X^3 are the curvilinear hexahedrons bounded by the curvilinear coordinate surfaces derived from the parametrization $x(\xi)$, Fig. (2.1).

2.2.1.1 Coordinate Grids

Among structured grids, coordinate grids in which the nodes and cell faces are defined by the intersection of lines and surfaces of a coordinate system in X^n are very popular in finite difference and finite volume methods. The range of values of this system defines a computation region Ξ^n in which the cells are rectangular n-dimensional parallelepipeds, and the coordinate values define the function $x(\xi) : \Xi^n \to X^n$.

The simplest of such grids are the Cartesian grids obtained by the intersection of the Cartesian coordinates in X^n. The cells of these grids are rectangular parallelepipeds (rectangles in two dimensions). The use of Cartesian coordinates avoids the need to transform the physical equations. However, the nodes of the Cartesian grid do not coincide with the curvilinear boundary, which leads to difficulties in implementing the boundary conditions with second-order accuracy in flow domains of complex geometries.

2.2.1.2 Boundary-Fitted Grids

An important subdivision of structured grids is the boundary-fitted or boundary- conforming grids. These grids are obtained from one-to-one transformations $x(\xi)$ which map the boundary of the domain Ξ^n onto the boundary of X^n.

The most popular of these, have become the coordinate boundary-fitted grids whose points are formed by intersection of curved coordinate lines, while the boundary of X^n is composed of a finite number of coordinate surfaces (lines in two dimensions) $\xi^i = \xi_i^0$. Consequently, in this case the computation region Ξ^n is a rectangular domain, the boundaries of which are determined by $(n-1)$-dimensional coordinate planes in R^n, and the uniform grid in $\Xi^n$is the Cartesian grid. Thus the physical region is represented as a deformation of a rectangular domain and the generated grid as a deformed lattice as shown in Fig. (2.2).

These grids give a good approximation to the boundary of the region and are therefore suitable for the numerical solution of problems with boundary singularities, such as those

with boundary layers in which the solution depends very much on the accuracy of the approximation of the boundary conditions.

The requirements imposed on boundary-conforming grids are naturally satisfied with the coordinate transformations $x(\xi)$. The algorithm for the organization of the nodes of boundary-fitted coordinate grids consists of the trivial identification of neighboring points by incrementing the coordinate indices, while the cells are curvilinear hexahedrons. This kind of grid is very suitable for algorithms with parallel computing[1]. Its design makes it easy to increase or change the number of nodes as required for multigrid methods or in order to estimate the convergence rate and error, and to improve the accuracy of numerical methods for solving boundary value problems.

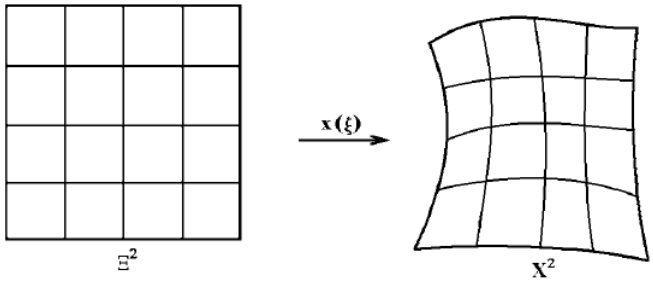


Figure 2.2: Boundary-conforming quadrilateral grid[25]

With boundary-conforming grids there is no necessity to interpolate the boundary conditions of the problem, and the boundary values of the region can be considered as input data to the algorithm, so automatic codes for grid generation can be designed for a wide class of regions and problems.

In the case of unsteady problems the most direct way to set up a moving grid is to do it via a coordinate transformation. These grids do not require a complicated data structure, since they are obtained from fixed domains, where the grid data structure remains intact.

2.2.1.3 Shape of Computational Domains

The idea of the structured approach is to transform a complex physical domain X^n to a simpler domain Ξ^n with the help of the parametrization $x(\xi)$. The region Ξ^n in Eq. (2.1), which is called the computational or logical region, can be either rectangular or of a different shape matching, qualitatively, the geometry of the physical domain; in particular, shape that can be triangular for $n = 2$ Fig. (2.3), or tetrahedral for $n = 3$. Using such parametrizations, a numerical solution of a partial differential equation in a physical region of arbitrary shape can be carried out in a standard computational domain, and codes can be developed that require only changes in the input.

The cells of the uniform grid can be rectangular or of a different shape. Schematic illustrations of two-dimensional triangular and quadrilateral grids are presented in Figs. (2.3

[1]Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel).

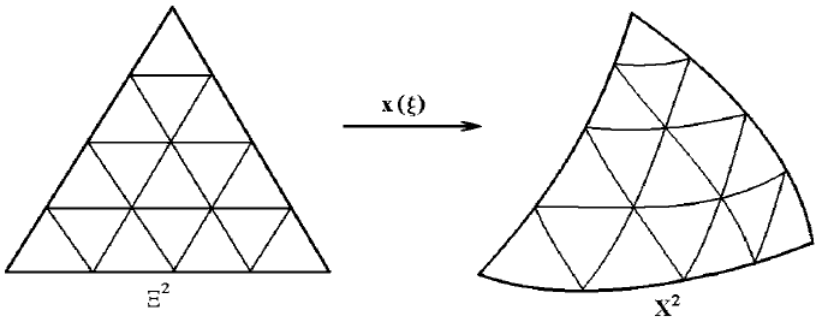


Figure 2.3: Boundary Conforming Triangular Grid[25]

and 2.4), respectively. Note that regions in the form of curvilinear triangles, such as that shown in Fig. (2.3), are more suitable for gridding in the structured approach by triangular cells than by quadrilateral ones.

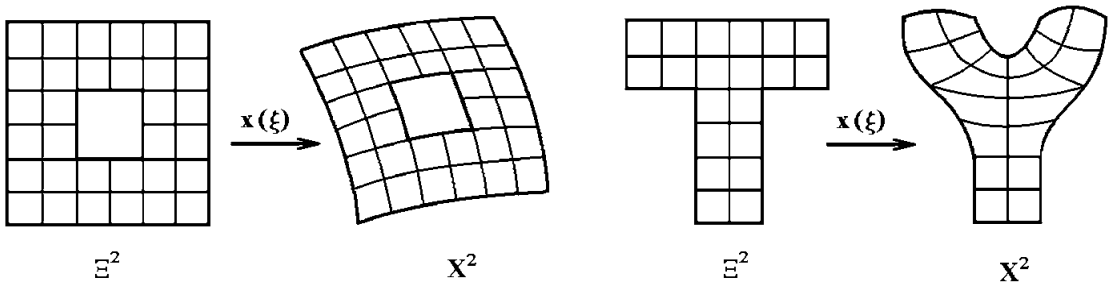


Figure 2.4: Computational Domains adjusted to the Physical Domains [25]

2.2.1.4 Stretching Methods

The stretching approach for generating structured grids is applied widely in the numerical solution of partial differential equations. Its major advantage is the rapidity of grid generation and direct control of grid spacing, while the main disadvantage is the necessity to explicitly select the zones where the stretching is needed. Of central importance in the method are intermediate transformations constructed on the basis of some standard stretching functions which provide the required spacing between the coordinate lines in selected zones.

A stretching method utilizing the standard stretching functions supplies one with a very simple means to cluster the nodes of the computational grid within the regions of steep gradients without an increase in the total number of grid nodes. This grid concentration improves the spatial resolution in the regions of large variation, thus enhancing the accuracy of the algorithms applied to the numerical solution of partial differential equations as further shown in Chapter VI.

2.2.1.5 Structured grid generation overview

The most efficient structured grids are boundary-conforming grids. The generation of these grids can be performed by number of approaches and techniques. Many of these methods are specifically oriented to the generation of grids for the finite-difference method.

A boundary-fitted coordinate grid in the region X^n is commonly generated first on the boundary of X^n and then successively extended from the boundary to the interior of X^n. This process is analogous to the interpolation of a function from a boundary or to the solution of a differential boundary value problem. On this basis there have been developed three basic groups of methods of grid generation:

- Algebraic Methods, which use various forms of interpolation or special functions.
- Differential methods, based mainly on the solution of elliptic, parabolic, and hyperbolic equations in a selected transformed region.
- Variational methods, based on optimiation of grid quality properties.

Algebraic method In the algebraic approach the interior points of the grid are commonly computed through formulas of transfinite interpolation. Methods like Transfinite Interpolation, Lagrange and Hermite Interpolations can be further seen in [25, 26, 13].

Algebraic methods are simple; they enable the grid to be generated rapidly and the spacing and slope of the coordinate lines to be controlled by the blending coefficients in the transfinite interpolation formulas. However, in regions of complicated shape the coordinate surfaces obtained by algebraic methods can become degenerate or the cells can overlap or cross the boundary. Moreover, they basically preserve the features of the boundary surfaces, in particular, discontinuties. Algebraic approaches are commonly used to generate grids in regions with smooth boundaries that are not highly deformed, or as an initial approximation in order to start the iterative process of an elliptic grid solver.

Differential method For regions with arbitrary boundaries, differential methods based on the solution of elliptic and parabolic equations are commonly used [25, 27]. The interior coordinate lines derived through these methods are always smooth, being a solution of these equations, and thus discontinuties on the boundary surface do not extend into the region. The use of parabolic and elliptic systems enables orthogonal and clustering coordinate lines to be constructed, while, in many cases, the maximum principle, which is typical for these systems, ensures that the coordinate transformations are nondegenerate. Elliptic equations are also used to smooth algebraic or unstructured grids.

In practice, hyperbolic equations are simpler than nonlinear elliptic ones and enable marching methods to be used and an orthogonal system of coordinates to be constructed, while grid adaptation can be performed using the coefficients of the equations. However, methods based on the solution of hyperbolic equations are not always mathematically correct and they are not applicable to regions in which the complete boundary surface is strictly defined. Therefore hyperbolic methods are mainly used for simple regions which have several lateral faces for which no special nodal distribution is required. Hyperbolic generation is particularly well suited for use with the overset grid approach. The marching procedure for the solution of hyperbolic equations allows one to decompose only the boundary geometry

in such a way that neighboring boundary grids overlap. Volume grids will overlap naturally if sufficient overlap is provided on the boundary. In practice, a separate coordinate grid around each subdomain can be generated by this approach.

Variational method Variational methods are used to generate grids which are required to satisfy more than one condition, such as nondegeneracy, smoothness, uniformity, near-orthogonality, or adaptivity, which cannot be realized simultaneously with algebraic or differential techniques [25, 27, 28]. Variational methods take into account the conditions imposed on the grid by constructing special functionals defined on a set of smooth or discrete transformations. A compromise grid, with properties close to those required, is obtained with the optimum transformation for a combination of these functionals.

At present, variational techniques are not widely applied to practical grid generation, mainly because their formulation does not always lead to a well-posed mathematical problem. However, the variational approach has been cited repeatedly as the most promising method for the development of future grid generation techniques, owing to its underlying, latent, powerful potential.

2.2.2 Block Structured Grids

In the commonly applied block strategy, the region is divided without holes or overlaps into a few contiguous subdomains, which may be considered as the cells of a coarse, generally unstructured grid. A separate structured grid is then generated in each block. The union of these local grids constitutes a mesh referred to as a block-structured or multi-block grid. Grids of this kind can thus be considered as locally structured at the level of an individual block, but globally unstructured when viewed as a collection of blocks. Thus a common idea in the block-structured grid technique is the use of different structured grids, or coordinate systems, in different regions, allowing the most appropriate grid configuration to be used in each region. This is further examined later in Chapter VI.

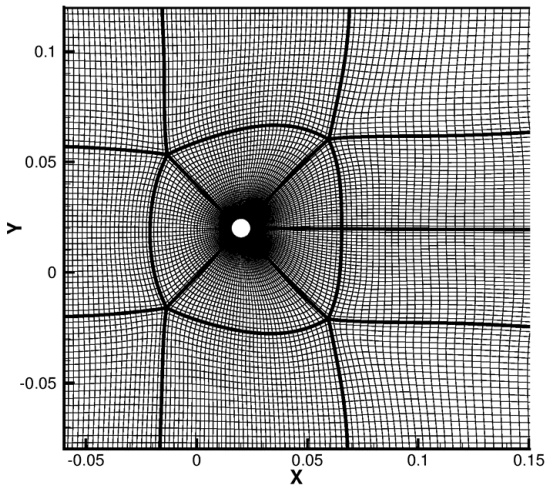


Figure 2.5: Block-Structured Grid[29]

Block-structured grids Fig. (2.5) are considerably more flexible in handling complex geometries than structured grids. Since these grids retain the simple regular connectivity pattern of a structured mesh on a local level, these block-structured grids maintain, in nearly the same manner as structured grids, compatibility with efficient finite-difference or finite-volume algorithms used to solve partial differential equations. However, the generation of block structured grids may take a fair amount of user interaction and, therefore, requires the implementation of an automation technique to lay out the block topology.

2.2.3 Unstructured Grids

Many field problems of interest involve very complex geometries that are not easily amenable to the framework of the pure structured-grid concept. Structured grids may lack the required flexibility and robustness for handling domains with complicated boundaries, or the grid cells may become too skewed and twisted, thus prohibiting an efficient numerical solution. An unstructured grid, Fig. (2.6), concept is considered as one of the appropriate solutions to the problem of producing grids in regions with complex shapes.

Unstructured grids have irregularly distributed nodes and their cells are not obliged to have any one standard shape. Besides this, the connectivity of neighboring grid cells is not subject to any restrictions; in particular, the cells can overlap or enclose one another. Thus, unstructured grids provide the most flexible tool for the discrete description of a geometry.

These grids are suitable for the discretization of domains with a complicated shape, such as regions around aircraft surfaces or turbomachinery blade rows. They also allow one to apply a natural approach to local adaptation, by either insertion or removal of nodes. Cell refinement in an unstructured system can be accomplished locally by dividing the cells in the appropriate zones into a few smaller cells. Unstructured grids also allow excessive resolution to be removed by deleting grid cells locally over regions in which the solution does not vary appreciably. In practice, the overall time required to generate unstructured grids in complex geometries is much shorter than for structured or block structured grids.

However, the use of unstructured grids complicates the numerical algorithm because of the inherent data management problem, which demands a special database to number and order the nodes, edges, faces, and cells of the grid, and extra memory is required to store information about the connections between the cells of the mesh. One further disadvantage of unstructured grids is that causes excessive computational work is associated with increased numbers of cells, cell faces, and edges in comparison with those for block structured meshes. Furthermore, moving boundaries or moving internal surfaces of physical domains are difficult to handle with unstructured grids. Besides, linearized difference scheme operators on unstructured grids are not usually band matrices, which makes it more difficult to use implicit schemes. As a result, the numerical algorithms based on an unstructured grid topology are the most costly in terms of operations per time step and memory per grid point.

Originally, unstructured grids were mainly used in the theory of elasticity and plasticity,

and in numerical algorithms based on finite-element methods. However, the field of application of unstructured grids has now expanded considerably and includes computational fluid dynamics.

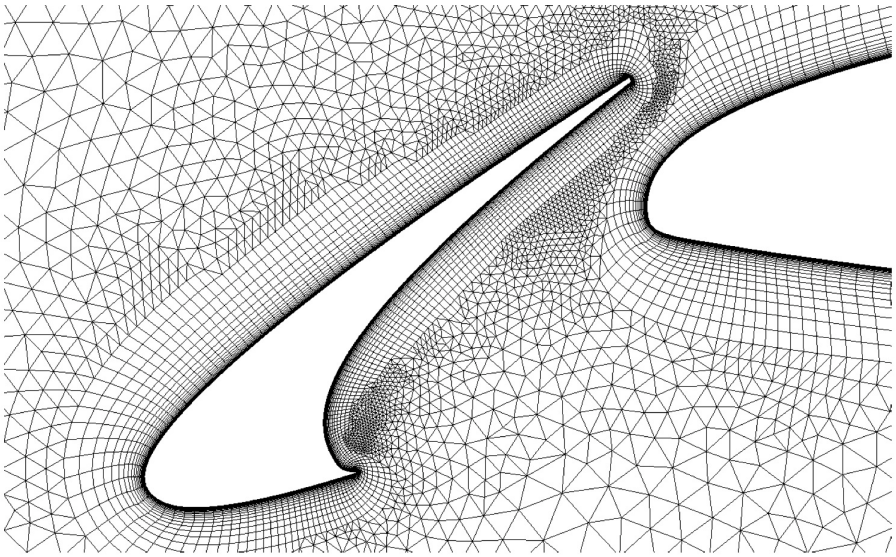


Figure 2.6: Unstructured Hybrid Grid[30]

2.2.4 Overset (Chimera) Grids

Overset grid, Fig. (2.7), consists of two layers or more of the basic grid types. This is a complex type of grid that needs a special care when being solved which in return requires a special solver for handling such configuration. The method evidently requires high demand on computer time and storage.

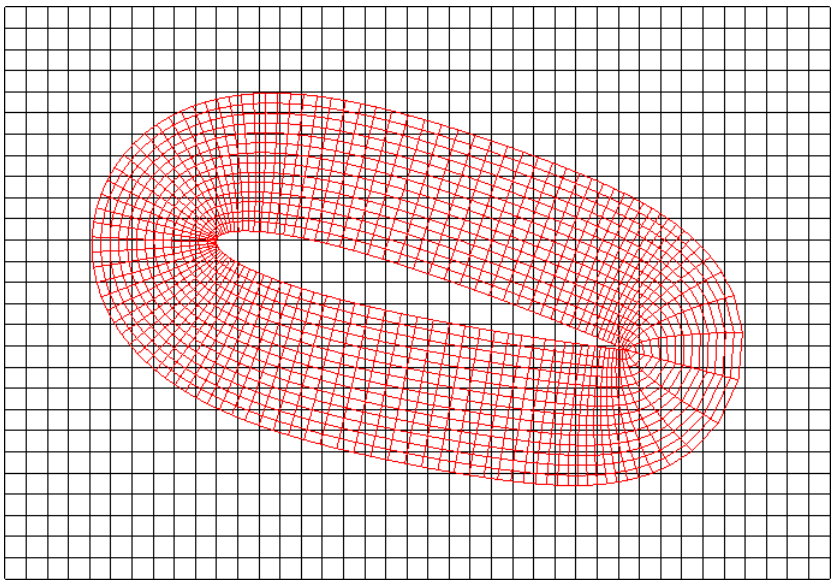


Figure 2.7: Overset Grid[31]

2.2.5 Hybrid Grids

Hybrid Grids are grids containing different types of cells in the same space. The need of hybrid grids were raised due to the lack of automatic grid generation to be used only for one grid type. Figure (2.8) shows one of these generated grids with quadrilateral on the boundaries and triangular cells in the rest of the gridded space. The solver should be capable of moving from one shape of grid to the others.

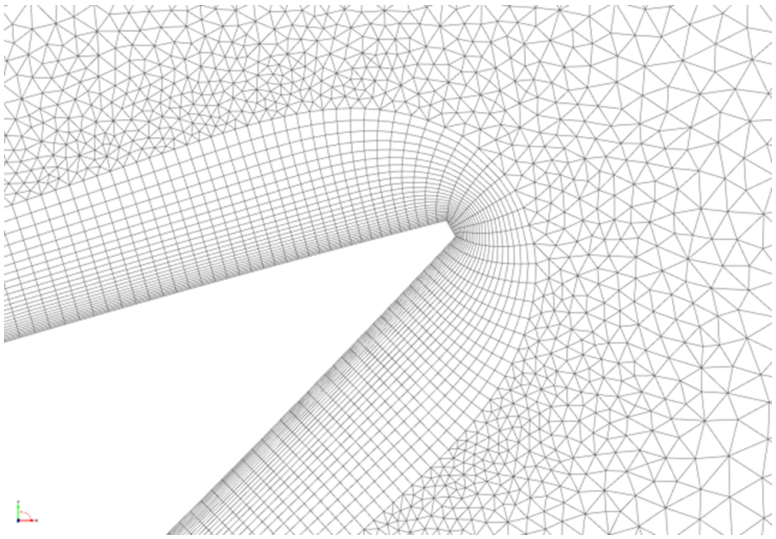


Figure 2.8: Hybrid Grid[32]

2.3 AUTOMATIC GRID GENERATION

2.3.1 Mapping Transformation

The mapping transformation was one of the first methods used in automatic quadrilateral mesh generation. The research and development that have been invested in this method produced a great number of versions widely used in the currently available CFD commercial packages [13, 28, 26, 33]. In this section, *transfinite interpolation*, *elliptic generators*, and *hyperbolic generator* are presented. Mapping techniques are relatively easy to implement; however, complex configurations depend on geometric decomposition and/or topological representation techniques in order to generate good meshes.

2.3.1.1 Transfinite Interpolation

The basic scheme of this method uses linear *blending functions* to map a natural domain into the physical domain. The natural domain is represented as a square region with natural coordinates (ξ, η) varying from zero to one. Each side of the natural domain is mapped into four parametric curves that enclose the physical domain [28, 34, 26]. Hence, any point $x = (x, y, z)$ inside the physical domain can be obtained as

$$
\begin{aligned}
x &= (1-\xi)\, f_1(\eta) + \eta f_2(\xi) + \xi f_3(\eta) + (1-\eta)\, f_4(\xi) \\
&\quad - (1-\xi)\,\eta x_{12} - \xi\eta_{23} - \xi\,(1-\eta)\, x_{34} - (1-\xi)\,(1-\eta)\, x_{41}
\end{aligned} \tag{2.3}
$$

where x_{ij} are the four nodes defined by the intersection of the four parametric curves $f_1(\eta), f_2(\xi), f_3(\eta), f_4(\xi)$, defined as

$$
f_i(t) = (x(t), y(t), z(t)) \qquad\qquad t = \eta, \xi \tag{2.4}
$$

By inspection of Eq. (2.3) one can easily conclude that *blending functions*, different from linear, can be used to control the aspect ratio of the elements in the mesh. Similar methodology is used in the present study and detailed in Chapter VI.

A similar approach is adopted in *Isoparametric Interpolation* a similar approach is adopted where only a few points of the boundaries are used [35]. This approach can also be used as a smoothing scheme to improve the mesh iteratively, in which the new position of a node is obtained as the average of its adjacent nodes [24].

2.3.1.2 Elliptic Generators

The method generates the mesh by solving an elliptic differential equation that describes the transformation between the natural and physical domains [13, 28, 26]. Typically, the Laplace equation written in the physical domain governs this transformation:

$$
\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2} = 0 \tag{2.5}
$$

where $\phi = \xi, \eta$. This method is also known as *Winslow* or homogeneous *Thompson-Thames-Mastic* (TTM) generator [26].

Laplace Equation, (2.5), is solved in the natural domain and the transformation yields

$$
g_{22}\frac{\partial^2\psi}{\partial\xi^2} - 2g_{12}\frac{\partial^2\psi}{\partial\xi\partial\eta} + g_{11}\frac{\partial^2\psi}{\partial\eta^2} = 0 \tag{2.6}
$$

where $\psi = x, y, z$, and

$$
g_{11} = \frac{\partial x}{\partial\xi}\cdot\frac{\partial x}{\partial\xi} = \left(\frac{\partial x}{\partial\xi}\right)^2 + \left(\frac{\partial y}{\partial\xi}\right)^2 + \left(\frac{\partial z}{\partial\xi}\right)^2 \tag{2.7}
$$

$$g_{12} = \frac{\partial x}{\partial \xi} \cdot \frac{\partial x}{\partial \eta} = \frac{\partial x}{\partial \xi}\frac{\partial x}{\partial \eta} + \frac{\partial y}{\partial \xi}\frac{\partial y}{\partial \eta} + \frac{\partial z}{\partial \xi}\frac{\partial z}{\partial \eta} \tag{2.8}$$

$$g_{22} = \frac{\partial x}{\partial \eta} \cdot \frac{\partial x}{\partial \eta} = \left(\frac{\partial x}{\partial \eta}\right)^2 + \left(\frac{\partial y}{\partial \eta}\right)^2 + \left(\frac{\partial z}{\partial \eta}\right)^2 \tag{2.9}$$

which are the components of the covariant metric tensor of the transformation, see Appendix A. The computational stencil, using a second-order centered finite difference scheme for the numerical approximation of the first and second derivatives, becomes

$$\psi_{ij} = C\left[\frac{g_{22}}{\Delta\xi^2}(\psi_{i+1,j} + \psi_{i-1,j}) - \frac{g_{12}}{2\Delta\xi\Delta\eta}(\psi_{i+1,j+1} - \psi_{i1,j+1} - \psi_{i+1,j-1} + \psi_{i-1,j-1}) \tag{2.10}$$

$$+\frac{g_{11}}{\Delta\eta^2}(\psi_{i,j+1} + \psi_{i,j-1})\Bigg] \tag{2.11}$$

where

$$C = \frac{1}{2\left(\frac{g_{22}}{\Delta\xi^2} + \frac{g_{11}}{\Delta\eta^2}\right)} \tag{2.12}$$

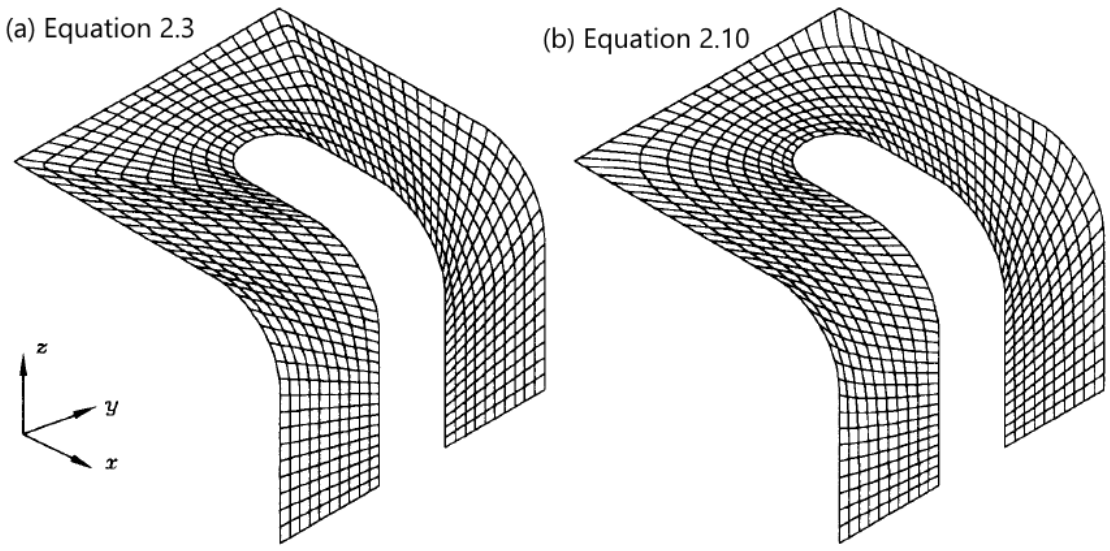


Figure 2.9: Transfinite Interpolation and Laplace Scheme[36].

Frequently, the final mesh is acheived by iteratively applying this stencil to an initial algebraic mesh, in which case the method is used as smoothing scheme. The final mesh displays a high level of orthogonality throughout the entire domain, but tends to concenrate nodes around convex sectors of the boundaries and move them away from concave sectors [28]. These properties can be noticed in Fig. (2.9)

The Poisson equation scheme provides an equal level of orthogonality with better control of the mesh along convex and concave sectors of the boundaries [13, 28]. The governing equations are obtained by writing a Laplace nonhomogeneous equation:

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}=Q_\phi \tag{2.13}$$

where $\phi=\xi,\eta$, and Q_ϕ are the weight functions that provide mesh control. This equation is also solved using

$$g_{22}\frac{\partial^2\psi}{\partial\xi^2}-2g_{12}\frac{\partial^2\psi}{\partial\xi\partial\eta}+g_{11}\frac{\partial^2\psi}{\partial\eta^2}=-g\left(Q_\xi\frac{\partial\psi}{\partial\xi}+Q_\eta\frac{\partial\psi}{\partial\eta}\right) \tag{2.14}$$

where

$$g=\left|\frac{\partial x}{\partial\xi}\times\frac{\partial x}{\partial\eta}\right|^2 \tag{2.15}$$

2.3.1.3 Hyperbolic Generators

Hyperbolic grid generation methods solve a hyperbolic set of equations to grow a grid from a boundary [33]. Fig. (2.10) shows a grid generated in this way. Typically the hyperbolic system is defined by imposing that the grid lines be orthogonal,

$$\frac{\partial x}{\partial r_\mu}\cdot\frac{\partial x}{\partial r_v}=0,\qquad \mu\neq v \tag{2.16}$$

and that the cell area is specified

$$\left|\frac{\partial x}{\partial r}\right|=\Delta$$

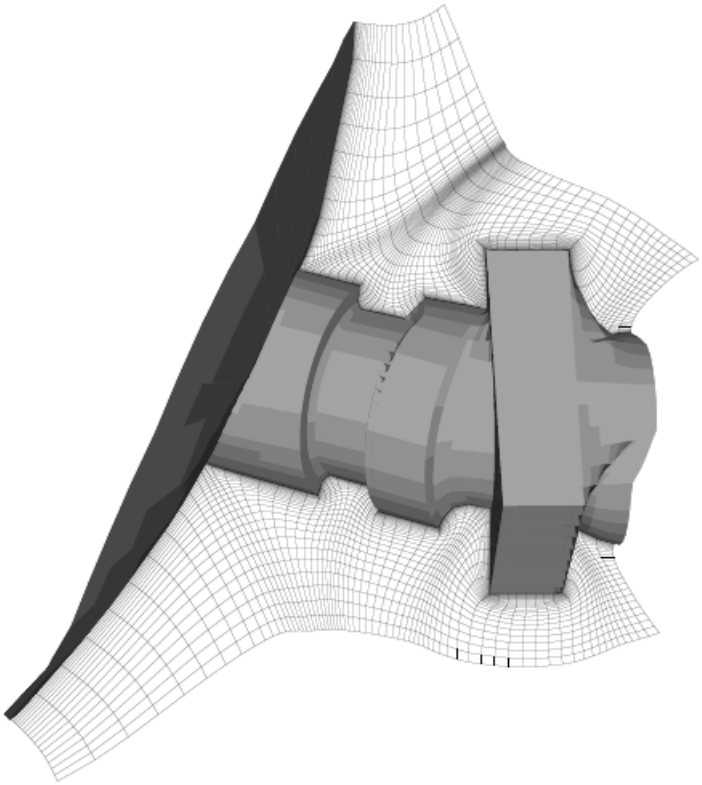


Figure 2.10: Hyperbolic Grid[37]

Hyperbolic methods usually add smoothing to prevent grid lines from prematurely crossing. The outer boundary of the grid is determined as the equations are solved, and thus the method is of limited use for block structured grids. The method is much faster than elliptic method since the grid is constructed by marching and is a useful technique in the context of overlapping grids.

2.3.2 Grid Superposition

The first step in the grid superposition method is to overlay an orthogonal grid of points over the entire domain and connect them to form an initial mesh. The points lying outside the boundaries are then eliminated, and the remaining nodal points operate as the core of the whole mesh. In general, grid superposition requires some transformation of triangles in order to acheive a fully boundary-fitted quadrilateral mesh. In the third stage, the initial mesh is connected to the boundary nodes using quadrilateral and trianglular elements. Finally, this mixed mesh is transformed into an all-quadrilateral mesh.

Grid superposition has also been published under the title of *quadtree*[38]. This name refers to the technique used to store the nodes of the overlaid mesh according to their spatial position. The method is combined with an advancing front scheme [39] to complete the mesh along the boundaries and used an a posteriori approach of subdividing the triangular elements to obtain quadrilateral elements.

A modified quadtree technique is proposed that divides the three-, four- and five-sided

polygons created in the initial mesh into quadrilateral elements. The possibility of creating three types of polygons results in good control of the mesh density.

2.3.3 Geometric Decomposition

Published as an automatic mesh generator, the method is, in reality, an auxiliary scheme that divides the domain into simple polygons which are then meshed using one of the methods discussed. In planar configurations, the medial axis technique has been used to divide complex geometries. It consists of a set of interconnected curves containing the center of all circles that can be inscribed in the geometry. It provides the basis for the final division of the domain. The medial axis technique along with other techniques was used [17] and [19] to generate the final decomposition. The medial axis technique was used as an aid to form an abstract representation of the geometry [18]. The medial axis technique forms a sketch graphic representation of the domain indicating which pieces seem to project out from the geometry and must be decomposed first. The decomposition technique was extended to the level of the final discretization by [40], creating meshes with high-density gradients.

A scheme that represents the surface to be meshed with meshed patches introduced by [41]. The first step consists of overlaying an initial set of regular meshed patches. Then, intersections between the boundaries and the patches are determined. In a third stage, the parts of the initial regular meshed patches that lay outside the surface are deleted. Finally, the patches are connected to the boundaries in order to form the mesh with each one of the regular meshed patches represents a division of the entire domain.

2.3.4 Transformation From Triangular Meshes

Any triangle can be divided into three quadrilaterals. This fact opens the possibility of using any existing triangular mesh generator to create a quadrilateral mesh [42]. Triangles can also be merged to produce quadrilateral elements [14]. The former scheme generates a mesh with finer density if compared to the inital triangular mesh, whereas the latter one ends up with a coarser density. However, the combination of both is often necessary to guarantee a quadrilateral mesh with reasonable element aspect ratios. Another algorithm that attempts to generate good quadrilateral elements whose quality is associated with quality of the initial triangular mesh have been proposed by [43]. Additional scheme to preserve the original density distribution during the transformation has been proposed by [44]. Basically, the scheme combines two trianlges to form a quadrilateral that is then divided into four quadrilaterals. The mesh density obtained is closer to the original density than it would be if the two triangles had been direclty divided into six quadrilaterals.

The above approaches require a relatively simple code, taking advantage of the speed and robustness of the existing trianglular mesh generators capable of discretizing very complex geometries.

2.3.5 Advancing Front Method

The advancing front method has been successfully used to automatically generate triangular meshes over general geometries. The robsutness of the technique is based on the fact that any polygon can be decomposed into triangles, so theat the closure of the mesh can always be acheived. Recently, some research has been conducted to generate quadrilateral elements with similar approaches. Another method that advances the front by creating and combining triangles, still in the front, to generate quadrilateral elements has been proposed by [20]. This algorithm requires a simply connected domain, i.e. all internal boundaries of the domain must be connected to its external boundary using *cut lines*.

Another method developed to directly generate quadrilateral elements in the front was introduced under the title of *paving* [21, 22, 23, 24]. This method advances the front by projecting rows of quadrilaterals inward, so that the elements near the contour tend to have a good aspect ratio, contributing to the orthogonality of the mesh along the boundaries (i.e., the perpendicularity of the lines of the mesh). In this method, as in [20], the closure is guaranteed if the front has an even number of nodes, which is acheived by maintaining this condition throughout the generation.

Paving has been used to automatically generate quadrilateral meshes over general planar geometries. The complexity of the algorithm, however, leads to a relatively low speed of generation if compared to the advancing front method used to produce triangular meshes.

2.4 MESHLESS METHODS

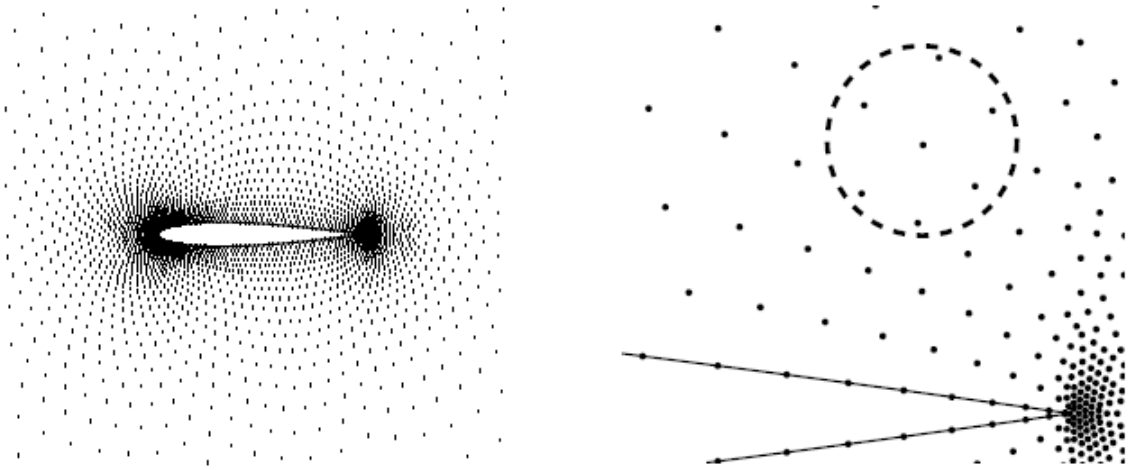
All the major fields of computational fluid mechanics, including finite element methods (FEM), finite difference methods (FDM), and finite volume methods (FVM), have traditionally relied on the use of elements, interlaced grids, or finite volumes as the underlying structures upon which the governing partial differential equations (PDE) are discretized[45, 46, 47].

Despite their varied names, all meshless schemes bypass the use of a conventional mesh to some degree. On the other hand, meshless schemes only require clouds of points, from which PDEs may be discretized. Local clouds for each point in a domain are proximity-based subsets of the global set of points. Local clouds of points replace the more traditional forms of connectivity found in FEM, FDM, and FVM. This loose definition of connectivity forms the basis for a wide variety of numerical methods for PDE's, some of the most notable of which are discussed here.

The motivation behind meshless methods lies in relieveing the burden of mesh generation. Since the application of computational methods to real world problems appears to be paced by mesh generation, alleviating this bottleneck potentially impacts enormous field of problems. It is not clear at this point how effective meshless methods will be at alleviating meshing problems. While a rigid mesh is not required, sufficiently dense point distributions are still required. Moreover, points must be grouped locally to form clouds. Obtaining opti-

mal clouds for different methods is also a non-trivial problem. However, recent progress in the area of point distribution and cloud generation has shown great promise [48, 46, 45].

A distinction should be made between global and local meshless methods clouds of points. A global cloud of points contains all points in a given domain, while a local cloud of points is a small subset of the global cloud. It is on the local cloud of points that PDEs are descritized. The use of a local cloud enables compact support with small bandwidth of the resulting linear system instead of large non-sparse systems[49, 50, 48]. Global and local clouds in two dimensions are illustrated in Fig. (2.11).



(a) Global point cloud. (b) Local point cloud.

Figure 2.11: Meshless discretization framework[51].

2.5 SUMMARY

Various types of computational grid has been reviewed with main classifications from the geometrical point of view, i.e. structured and unstructured grids. Hybrid and overset grids have also been discussed along with the generation techniques associated with each type and class. Short review of the meshless methods has been given to conclude the review of previous work on grid types and generation techniques.

CHAPTER III

COMPUTATIONAL FLUID DYNAMICS AND COMPUTATIONAL GRIDS

3.1 INTRODUCTION

Computational fluid dynamics, usually abbreviated as CFD, is a branch of fluid mechanics that uses numerical methods and algorithms for the analysis of systems involving fluid flow, heat transfer and associated phenomena such as chemical reactions by means of computer-based simulation. With high-speed supercomputers, better solutions can be achieved. Ongoing research yields software that improves the accuracy and speed of complex simulation scenarios such as transonic or turbulent flows.

The technique is very powerful and spans a wide range of industrial and research applications. Some examples are:

- aerodynamic forces on aircrafts and vehicles.
- hydrodynamics of ships.
- power plants; combustion in IC engines and gas turbines.
- turbomachinery and electronic engineering: cooling of equipment including micro-circuits.
- chemical process engineering: mixing and separation, polymer molding.
- external and internal environment of buildings: wind loading and heating/ventilation.
- marine engineering: loads on off-shore structures.
- environmental engineering: distribution of pollutants and effluents.
- hydrology and oceanography: flows in rivers, estuaries, oceans.
- meterology: weather prediction.
- biomedical engineering: blood flows through arteries and veins.

The ultimate aim of developments in the CFD field is to provide a capability comparable in accuracy and performance to other CAE (Computer-Aided Engineering) tools such as

stress analysis codes. For a long time CFD has been lagging behind other CAE codes mainly because of the tremendous complexity of the underlying behavior of the governing equations, which precludes a description of fluid flows. The availability of friendly interfaces have led to a recent upsurge of interest and CFD was poised to make an entry into the wider industrial community early in the 1980s.

The fundmental concepts and methodology of CFD are discussed in this chapter with emphasis on the weight placed on computational grids and their properties affecting solution integrity. In section 3.2 a breif derivation of the governing equations of the fluid flow is presented. In section 3.3 the Finite Volume descritiztion Method (FVM), is presented.

3.2 THE GOVERNING EQUATIONS

The main objectives of CFD is to obtain numerical values across the flow field of its various variables. These variables are physically related by a mathematical model referred to as "the governing equations" represented by:

- The conservation of Mass equation (Continuity Equation).
- The conservation of linear Momentum equation (Momentum Equation).
- The conservation of Enegry equation (Energy Equation).

In the following, the governing equations are written for easy reference when the solution procedure is discussed. For detailed derivation of these equations, the reader may refer to text books on CFD, eg[52, 2].

In tensor notation[7], the governing equations may be written as:

i) Continuty Equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{V}) = S_m \tag{3.1}$$

ii) Momentum Equation:

$$\frac{D\,(\rho \mathbf{V})}{Dt} + \nabla\,(\rho \mathbf{V} \otimes \mathbf{V} - \mathbb{T}) = S_v \tag{3.2}$$

iii) Energy Equation:

$$\frac{\partial h}{\partial t} + \nabla \cdot (h\mathbf{V} - \mathbf{q}) = S_\phi \tag{3.3}$$

Where:

$\mathbf{V}$	Fluid Velocity Vector
ρ	Fluid Density
$\mathbb{T}$	Stress Tensor given by, see (3.1)

$$\mathbb{T} = -\left(p - \frac{2}{3}\mu\nabla\cdot\mathbf{V}\right)\mathbb{I} + 2\mu\mathbb{D} \tag{3.4}$$

$$\mathbb{T} = \sigma_{ij} = \left\{\begin{array}{ccc} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{array}\right\} = \left\{\begin{array}{ccc} \sigma_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_{yy} & \tau_{yz} \\ \tau_{zx} & \sigma_{zy} & \sigma_{zz} \end{array}\right\} \tag{3.5}$$

$\mathbf{q}$ Flux Vector usually given by a Fourier type law

$\mathbf{q} = \Gamma_\phi \nabla\phi$

Γ_ϕ Diffusion Coefficient.

p Hydrostatic Pressure

$\mathbb{I}$ Unit Tensor

$\mathbb{D}$ Rate of Strain Tensor given by:

$$\mathbb{D} = \epsilon_{ij} = \left\{\begin{array}{ccc} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{array}\right\} = \left[\begin{array}{ccc} \frac{\partial u}{\partial x} & \frac{1}{2}\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right) & \frac{1}{2}\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial z}\right) \\ \frac{1}{2}\left(\frac{\partial v}{\partial y} + \frac{\partial v}{\partial x}\right) & \frac{\partial v}{\partial y} & \frac{1}{2}\left(\frac{\partial v}{\partial y} + \frac{\partial v}{\partial z}\right) \\ \frac{1}{2}\left(\frac{\partial w}{\partial z} + \frac{\partial w}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial w}{\partial z} + \frac{\partial w}{\partial y}\right) & \frac{\partial w}{\partial z} \end{array}\right] \tag{3.6}$$

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{3.7}$$

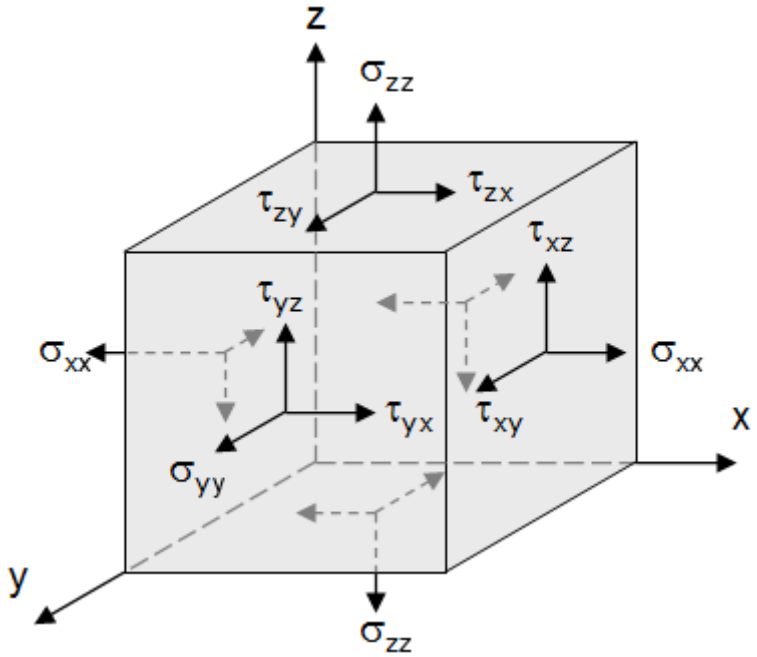


Figure 3.1: Stress Tensor Components

The terms on the right-hand side of governing Equations (3.1 - 3.3) represent sources or sinks of the basic variable considered. These equations, however, can be re-written in the general coordinate free form [7]

$$\frac{D\left(\rho\psi\right)}{Dt} + \nabla \cdot \left(\rho\psi \otimes \mathbf{V} - \mathbf{\Omega}\right) = S \tag{3.8}$$

where ψ and S are tensor fields of the same order and Ω is a tensor field of one order higher than that of ψ and S. The assigned value of ψ, Ω, and S, to yield the mass and, momentum equations are shown, respectively, in Table (3.1)

Transport Equation	ψ	Ω	S
Mass	1	0	S_m
Momentum	$\mathbf{V}$	$\mathbb{T}$	S_v
Scalar	ϕ	$\mathbf{q}$	S_ϕ

Table 3.1: Definition of Quantities in Eq. (3.8)

3.3 FINITE VOLUME DESCRITIZATION OF THE GOVERNING EQUATIONS

The governing Eqs. (3.1 - 3.3) are, by nature, non linear and do not have an exact solution. Most solution methods adopted attempt at obtaining an approximate solution at predefined points in the flow domain. For this purpose, a gird is superimposed on the flow domain and values of the variables are sought at the grid intersection points. The partial derivatives in the governing equations are next approximated to algebraic formulae that can be solved numerically to yield the flow parameters at the grid nodes. This approximation process is referred to as ***descritization*** of the governing equation. Here, only Finite Volume Methodology (FVM) of descritization will be reviewed.

The finite volume method (FVM) is an increasingly popular numerical method for approximate solution of partial differential equations (PDEs)[2, 3]. In this method, the flow domain is divided into control volumes (computational cells) by a suitable grid generator. The governing equations are integrated over the computational cells and then cast in a form suitable for numerical solution.

The method offers the following advantages:

1. The governing equations are presented in their integral form which is often how they are derived from the underlying physical laws.

2. The FVM naturally conserves the conserved quantities when applied to PDEs expressing conservation laws since, as two neighboring cells share a common interface, the total flux of a conserved quantity out of one cell will be the same as that entering the next cell.

3. The computational grids employed in FVM usually offer flexible descritization as they can accommodate irregularly shaped boundaries to reduce geometry errors, as well as the possibility of local refinement for better resolution in regions of high gradients.

One disadvantage of FVM is that there is no easily accessible underlying theory for formal accuracy.

Because of the nature of FVM [2], the governing equations are integrated first before descritization takes place. For a two-dimensional steady incompressible flow, the momentum equation may be written as:

$$\frac{\partial J_x}{\partial x} + \frac{\partial J_y}{\partial y} = S \tag{3.9}$$

where J_x and J_y are the total (convective and diffusive) fluxes defined by

$$J_x \equiv \rho u \psi - \Gamma \frac{\partial \psi}{\partial x} \tag{3.10}$$

and

$$J_y \equiv \rho v \psi - \Gamma \frac{\partial \psi}{\partial y} \tag{3.11}$$

where u and v denote the velocity components in the x and y directions and Γ is the diffusivity coefficient. The integration of Eq. (3.9) over the control volume shown in Fig. (3.2) gives

$$J_e - J_w + J_n - J_s = (S_C + S_P \psi_P)\, \Delta x \Delta y \tag{3.12}$$

where the source term has been linearized according to the recommendations of [6]. The quantities J_e, J_w, J_n, and J_s are the integrated total fluxes over the control-volume faces; that is, J_e stands for $\int J_x\, dy$ over the interface e, and so on.

In a similar manner, the continuity equation (3.1) can be integrated over the control volume to yield

$$F_e - F_w + F_n - F_s = 0 \tag{3.13}$$

where F_e, F_w, F_n, and F_s are the mass flow rates through the faces of the control volume.

If ρu at point e is taken to prevail over the whole interface e, one can write

$$F_e = (\rho u)_e \, \Delta y \tag{3.14}$$

Similarly,

$$F_w = (\rho u)_e \, \Delta y \tag{3.15}$$

$$F_n = (\rho v)_n \, \Delta x \tag{3.16}$$

$$F_s = (\rho v)_s \, \Delta x \tag{3.17}$$

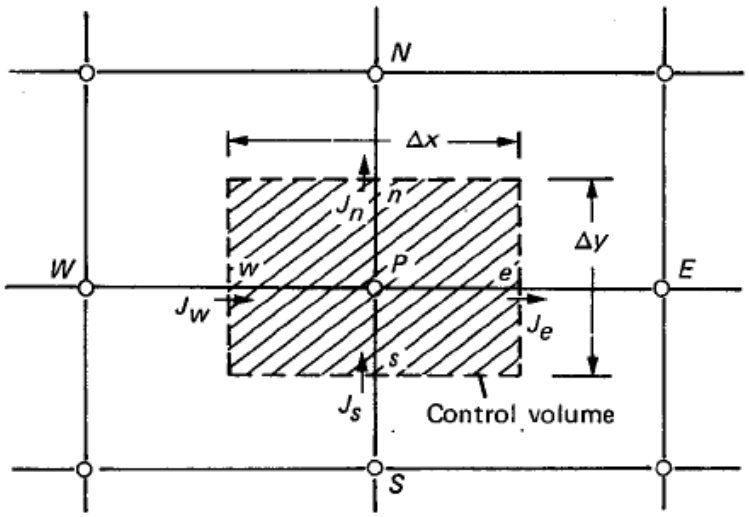


Figure 3.2: Control Volume for the Two-Dimensional Situation[6]

Subtracting Eq. (3.13) from Eq. (3.12) become:

$$(J_e - F_e\psi_P) - (J_w - F_w\psi_P) + (J_n - F_n\psi_P) - (J_s - F_s\psi_P) = (S_C + S_P\psi_P)\,\Delta x \Delta y \tag{3.18}$$

Equation (3.18) may be cast in the final form

$$a_P\phi_P = a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S + b \tag{3.19}$$

where the ϕ's are the values of the variable under consideration at the nodal points shown in Fig. (3.2) and the a's are the convective and diffusive fluxes at the cell faces e, w, n, and s as shown[6].

Equations (3.19) are solved iteratively to yield values of ϕ's at every nodal point. However, special treatment for cell face fluxes, velocity pressure coupling schemes and closure models have to be implemented as detailed in textbooks on CFD methodology [2, 6, 52].

3.4 THE COMPUTATIONAL GRID

As mentioned earlier, the governing equations (3.19) are solved iteratively over a foundation computational grid to yield approximate values of the flow parameters at grid nodes. Creating the foundation grid is considered to be the most critical stage in the CFD simulation process (and the most time and effort consuming) as mentioned before. Without a properly adjusted fine grid, inaccurate results are likely to be obtained and may lead to a defected design.

There are two general notions of a grid in an n-dimensional bounded domain. One of these considers the grid as a set of algorithmically specified points of the domain. The points are called the grid nodes. The second considers the grid as an algorithmically described collection of standard n-dimensional volumes covering the solution domain. The standard volumes are referred to as the grid cells. The cells are bounded curvilinear volumes, whose boundaries are divided into a few segments which are $(n-1)$-dimensional cells. Therefore they can be formulated successively from one dimension to higher dimensions. The boundary points of the one-dimensional cells are called the cell vertices. These vertices are the grid nodes. Thus the grid nodes are consistent with the grid cells in that they coincide with the cell vertices.

3.4.1 Desired Properties of Computational Grid

As already noted, the accuracy of the numerical solutions, and the stability and the rate of convergence of the solution procedure all depend to a certain degree on particular properties of the computational grid. These properties, the way of achieving them and the degree of their influence will be described in this section.

The most influencial properties of the grid on the solution are:

- Orthogonality,
- Grid Spacing,
- Smoothness, and
- Alignment of grid lines with stream lines.

Orthogonality

The very reason for developing computational methods to solve flow cases in complex geometries is to enable working with non-orthogonal grids, which is desirable, if not essential in most practical situations. However, it is still usefull to identify the advantages which arise from minimizing the departure from orhogonality, all other factors being equal.

First of all, if orthogonality prevails, then all cross-derivative terms in the governing equations vanish. This means there will be no explicit "sources" in the descritised equations arising from these terms, and the "neglected" cross-derivatives terms in flux-corrections used to derive the pressure-correction equation are also formally absent [7]. The result is a more stable and faster converging numerical solution procedure.

The accuracy of numerical solution is not affected by grid non-orthogonality alone, but also depends rather strongly on some other grid properties. However, strong non-orthogonality does affect the stability and convergence rate of the solution, and therefore should be avoided were possible[1].

Grid Spacing

The accuracy of a choosen interpolation practices (differencing schemes) is usually judged by means of the Taylor Series Truncation Analysis. This shows that all schemes lose one order of accuracy if the grid spacing is non-uniform. From this consideration it could be concluded that uniform spacing is a desirable property of a computational grid. However, this goal is neither easy nor economical to achieve in every situation. In practical flow configurations there are regions of very steep gradients where very fine resolution is needed, and regions of low gradients, where more coarse spacing is adequate. For this reason a non-uniform grid is usually preferable, and it is desirable that a grid generation procedure should facilitate the achievement of such an arrangement.

In constructing non-uniform grids it is desirable to keep the grid expansion ratios in each coordinate direction under control. These ratios are defined as (see Fig. (3.3))

$$r_e^1 = \frac{\delta x_E^{(1)}}{\delta x_P^{(1)}} \quad ; \quad r_e^2 = \frac{\delta x_E^{(2)}}{\delta x_P^{(2)}} \tag{3.20}$$

also the grid aspect ratios, defined as

$$r_a = \left(\frac{\delta x_P^{(i)}}{\delta x_P^{(j)}} \right)_{i \neq j} \tag{3.21}$$

[1]This, however, is not a shortcoming of the method in general, but a consequnce of the adopted treatment of the cross-derivative terms.

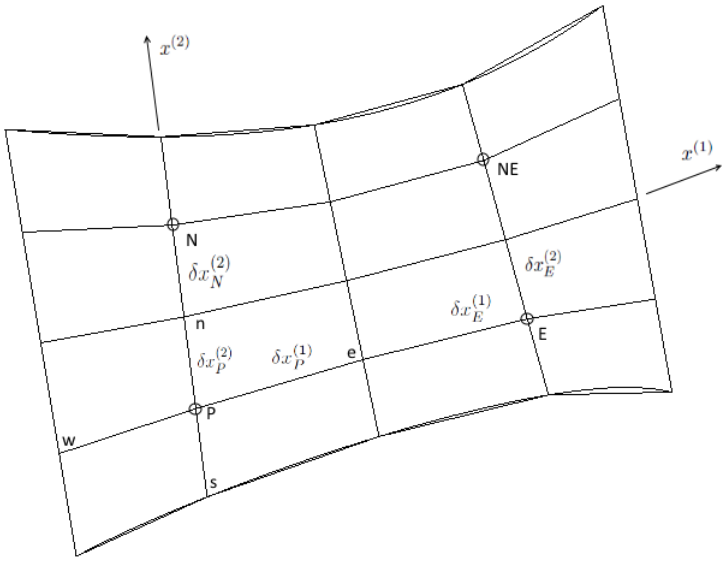


Figure 3.3: Definition of grid expansion and aspect ratios

should not exceed ~ 10. These ratios influence the stablility of the solution method (and also the boundedness of solutions) [7, 1] due to the fact that they can generate negative cross-derivatives coefficients.

Smoothness

Methods which use curvilinear velocity components that requires the calculation of curvature terms, will suffer significantly from sensitivity to grid smoothness[2]. The geometrical information about the grid is commonly limited to the cartesian coordinates of the cell vertices, and hence these may and are taken to be connected by segments of straight lines. Therefore, the grid lines are not - and need to be - continous.

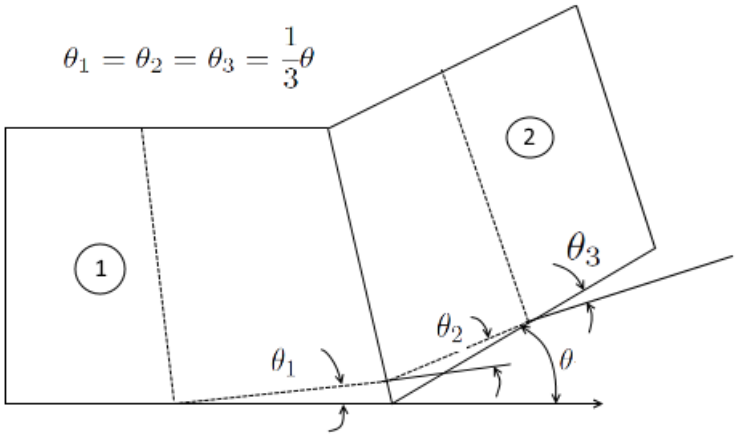


Figure 3.4: Definition of grid smoothness

Here, however, the term smoothness is taken to refer to the relative change in grid line direction from one cell to another, characterized by the angle θ shown in Fig. (3.4). This

[2] "Smooth" in this respect assumes continuity of the grid lines and their first derivatives.

does have an influence on the accuracy of the interpolation formulae used to obtain dependent variable values at locations other than the principal nodes. Higher-order interpolation practices may help in reducing the errors introduced by non-smoothness. However, they are computationaly more expensive and can generate more negative coefficients, thus affecting the stability of the solution process. Often it is possible to avoid large θ's (i.e. to obtain a smoother grid) by local refinement. Fig. (3.4) shows how large θ between cells (1) and (2) can be reduced by one-third by halving the two cells.

It follows that, at least when linear interpolation is used, it is desirable that the grid should be as smooth as the geometry of the solution domain allows. The method of grid generation should clearly offer the possibility of controlling this grid property.

Alignment of Grid Lines with Stream Lines

"Numerical diffusion" is a function of the skewness of grid and stream lines, and is reduced to minimum or completely eleminated if one set of grid lines is closely aligned with the streamline. This, therefore, is a very desirable property of the computational grid, and is probably the one which most affects numerical accuracy, especially if a lower-order discretization schemes are used[6].

However, in order to achieve alignment, it is necessary at the time of grid generation to know the stream line pattern, which is usually not the case. Therefore, the generation of an ideal grid should be an adaptive process, carried out during the course of the flow calculation. Adaptive grid generation is, however, very complex and difficult to achieve.

A crude but effective alternative is to perform the adaptation manually, by generating a solution on an initial grid and then using it to improve alignment for a subsequent more accurate calculations. Of course, this is possible only if the grid generation procedure employed offers this flexibility.

All of the forementioned desirable properties of a computational grid cannot, in general, be satisfied simultaneously. However, the two which are of the highest importance for the most solution methods are alignment of grid lines and stream lines and, high resolution in regions of high gradient - do not conflict, and usually can be satisified to a much higher degree than when regular or orthogonal grid are used.

3.5 CONCLUSIONS

The chapter started with the derivation of the conservation equations governing fluid flow along with Finite Volume Method of descritization.

Computational Grid properties that usually influence the stability and accuracy, as well as, economy of the iterative solution procedure have been discussed in some detail. Therefore, subsequent generation of computaional grids, detailed in Chapter VI, will have to keep these properties under tight control.

CHAPTER IV

MANIFOLDS AND HOMOTOPY

4.1 INTRODUCTION

This chapter is concerned with the definition of Manifolds as an abstracted term for curves and surfaces in any dimension, and any space.

The chapter begins with the definition of Manifolds, then goes through the homoeomorphism (continuous maps with continuous inverses) between topological manifolds, and functions.

The chapter finally discusses the most important one-dimensional manifolds used in this thesis, beside an important algorithm of dividing the circle circumference into a number curves of bezier curves is presented.

4.2 TOPOLOGICAL SPACES

In topology and related branches of mathematics, a topological space is a set of points, along with a set of neighbourhoods for each point, that satisfy a set of axioms relating points and neighbourhoods [53]. The definition of a topological space relies only upon set theory and is the most general notion of a mathematical space that allows for the definition of concepts such as continuity, connectedness, and convergence. Other spaces, such as manifolds and metric spaces, are specializations of topological spaces with extra structures or constraints.

4.3 MANIFOLDS

In mathematics, a manifold is a topological space that resembles Euclidean space near each point [54]. More precisely, each point of an n-dimensional manifold has a neighbourhood that is homeomorphic to the Euclidean space of dimension n. Lines and circles, but not figure eights, are one-dimensional manifolds. Two-dimensional manifolds are also called surfaces. Examples include the plane, the sphere, and the torus, which can all be realized in three dimensions, but also the Klein bottle and real projective plane which cannot. The surface of the Earth requires (at least) two charts to include every point.

Although near each point, a manifold resembles Euclidean space, globally a manifold might not. For example, the surface of the sphere is not a Euclidean space, but in a region it can be charted by means of geographic maps: map projections of the region into the Eu-

clidean plane. When a region appears in two neighbouring maps (in the context of manifolds they are called charts), the two representations do not coincide exactly and a transformation is needed to pass from one to the other, called a transition map.

The concept of a manifold is central to many parts of geometry and modern mathematical physics because it allows more complicated structures to be described and understood in terms of the relatively well-understood properties of Euclidean space. Manifolds naturally arise as solution sets of systems of equations and as graphs of functions. Manifolds may have additional features. One important class of manifolds is the class of differentiable manifolds. This differentiable structure allows calculus to be done on manifolds. A Riemannian metric on a manifold allows distances and angles to be measured.

4.4 HOMOTOPY

Homotopy is the continuous transformation from one function to another. Homotopy between two functions f and g from a space X to a space Y is a continuous map G from $X \times [0,1] \mapsto Y$ such that $G(x,0) = f(x)$ and $G(x,1) = g(x)$, where $\times$ denotes set pairing. Another way of saying this is that a homotopy is a path in the mapping space $Map(X,Y)$ from the first function to the second.

Two mathematical objects are said to be homotopic if one can be continuously deformed into the other. The concept of homotopy was first formulated by Poincaré around 1900 [55].

Formally, a homotopy between two continuous functions f and g from a topological space X to a topological space Y is defined to be a continuous function $H : X \times [0,1] \to Y$ from the product of the space X with the unit interval [0,1] to Y such that, if $x \in X$ then $H(x,0) = f(x)$ and $H(x,1) = g(x)$.

If one thinks of the second parameter of H as time then H describes a continuous deforming of f into g: at time 0 we have the function f and at time 1 we have the function g.

An alternative notation is to say that a homotopy between two continuous functions $f, g : X \to Y$ is a family of continuous functions $h_t : X \to Y$ for $t \in [0,1]$ such that $h_0 = f$ and $h_1 = g$, and the map $t \mapsto h_t$ is continuous from [0,1] to the space of all continuous functions $X \to Y$. The two versions coincide by setting $h_t(x) = H(x,t)$.

Continuous functions f and g are said to be homotopic if and only if there is a homotopy H taking f to g as described above. Being homotopic is an equivalence relation on the set of all continuous functions from X to Y. This homotopy relation is compatible with function composition in the following sense: if f_1, $g_1 : X \to Y$ are homotopic, and f_2, $g_2 : Y \to Z$ are homotopic, then their compositions $f_2 \circ f_1$ and $g_2 \circ g_1 : X \to Z$ are also homotopic.

4.4.1 Homotopy of Functions

In the special case of the $\mathbb{R}^n$ valued functions one can actually shows that all continuous maps are homotopic to one another. Adding little pieces of f to little pieces of g, where the

portion of f shrinks, and the portion of g grows, as t goes from 0 to 1. The homotopy is defined as:

$$F(s,t) = (1-t)\cdot f(s) + t\cdot g(s) \tag{4.1}$$

It's easy to see that Eq. (4.1) satisfies the conditions for a homotopy as when $t = 0$, $F(s,0) = f(s)$, and when $t = 1$ $F(s,1) = g(s)$. Since this is composition of continuous functions (addition, substraction, multiplication, and the maps f and g), this is also a continuous function. This is known as the straight-line homotopy since one can imagine taking the curve f and the curve g and drawing lines between associated points Fig. (4.1) (i.e., drawing a line between $f(s)$ and $g(s)$ for each s), then the homotopy just moves f to g along those lines. This result applies to convex subspaces of $\mathbb{R}^n$ as well.

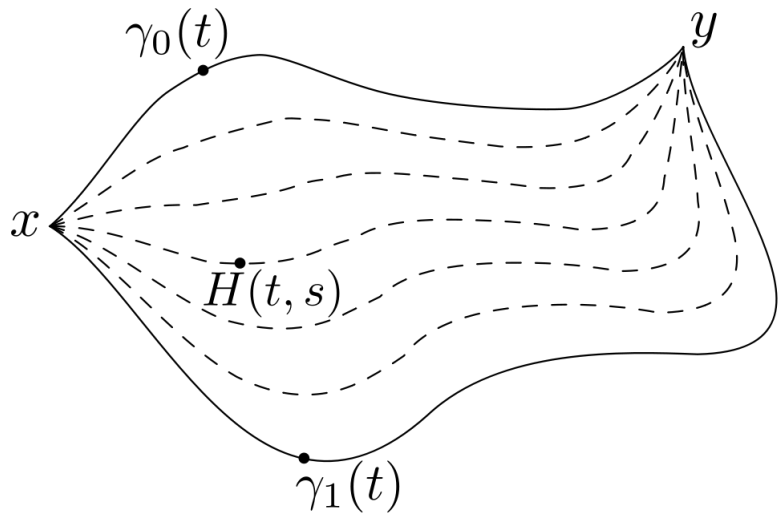


Figure 4.1: Homotopy of Functions[56]

4.5 BEZIER CURVES

A Bézier curve is defined by a set of control points $\mathbf{P}_0$ through $\mathbf{P}_n$, where n is called its order ($n = 1$ for linear, 2 for quadratic, etc.). The first and last control points are always the end points of the curve; however, the intermediate control points (if any) generally do not lie on the curve.

4.5.1 Generalization

Bézier curves can be defined for any degree n. A recursive definition for the Bézier curve of degree n expresses it as a point-to-point linear combination (linear interpolation) of a pair of corresponding points in two Bézier curves of degree $n-1$.

Let $\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}$ denote the Bézier curve determined by any selection of points $P_0, P_1, ..., P_n$. Then to start,

$$\begin{aligned}\mathbf{B}_{\mathbf{P}_0}(t) &= \mathbf{P}_0, \text{ and} \\ \mathbf{B}(t) &= \mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_{n-1}}(t) + t\mathbf{B}_{\mathbf{P}_1\mathbf{P}_2\ldots\mathbf{P}_n}(t)\end{aligned} \tag{4.2}$$

The formula can be expressed explicitly as follows:

$$\begin{aligned}\mathbf{B}(t) =& \sum_{i=0}^{n} \binom{n}{i}(1-t)^{n-i}t^i\mathbf{P}_i \\ =&(1-t)^n\mathbf{P}_0 + \binom{n}{1}(1-t)^{n-1}t\mathbf{P}_1 + \cdots \\ &\cdots + \binom{n}{n-1}(1-t)t^{n-1}\mathbf{P}_{n-1} + t^n\mathbf{P}_n, \quad t \in [0,1]\end{aligned} \tag{4.3}$$

where $\binom{n}{i}$ are the binomial coefficients.
For example, for n = 5 Fig. (4.4):

$$\begin{aligned}\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2\mathbf{P}_3\mathbf{P}_4\mathbf{P}_5}(t) = \mathbf{B}(t) = (1-t)^5\mathbf{P}_0 + 5t(1-t)^4\mathbf{P}_1 + 10t^2(1-t)^3\mathbf{P}_2 \\ + 10t^3(1-t)^2\mathbf{P}_3 + 5t^4(1-t)\mathbf{P}_4 + t^5\mathbf{P}_5, \quad t \in [0,1]\end{aligned} \tag{4.4}$$

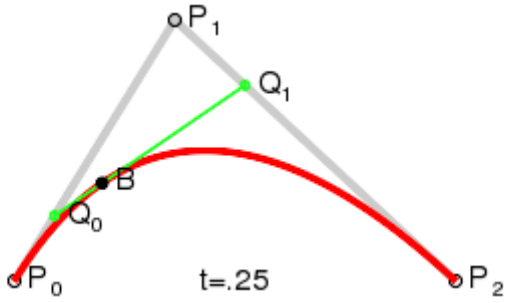


Figure 4.2: Quadratic Bezier Curve[57]

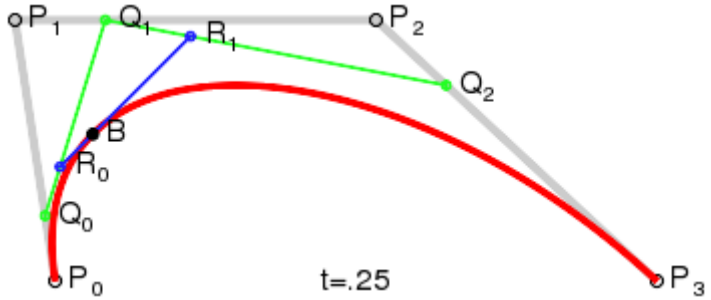


Figure 4.3: Cubic Bezier Curve[57]

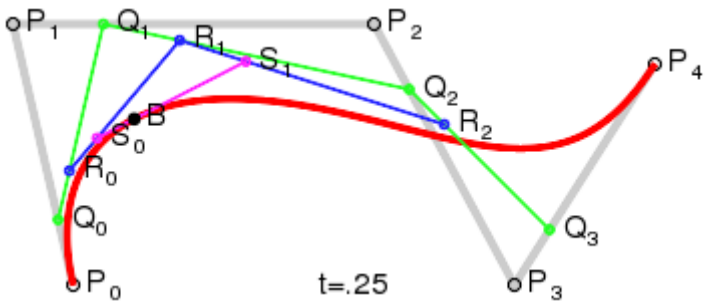


Figure 4.4: Quartic Bezier Curve[57]

4.5.1.1 Linear Bézier Curves

Given points $\mathbf{P}_0$ and $\mathbf{P}_1$, a linear Bézier curve is simply a straight line between those two points. The curve is given by

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \ , \ t \in [0,1] \tag{4.5}$$

and is equivalent to linear interpolation.

4.5.1.2 Quadratic Bézier Curves

A quadratic Bézier curve Fig. (4.2) is the path traced by the function $\mathbf{B}(t)$, given points $\mathbf{P}_0$, $\mathbf{P_1}$, and $\mathbf{P_2}$,

$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \ , \ t \in [0,1] \tag{4.6}$$

which can be interpreted as the linear interpolant of corresponding points on the linear Bézier curves from $\mathbf{P}_0$ to $\mathbf{P}_1$ and from $\mathbf{P}_1$ to $\mathbf{P}_2$ respectively. Rearranging the preceding equation yields:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 \ , \ t \in [0,1] \tag{4.7}$$

The derivative of the Bézier curve with respect to t is

$$\mathbf{B}'(t) = 2(1-t)(\mathbf{P}_1 - \mathbf{P}_0) + 2t(\mathbf{P}_2 - \mathbf{P}_1) \tag{4.8}$$

from which it can be concluded that the tangents to the curve at $\mathbf{P}_0$ and $\mathbf{P}_2$ intersect at $\mathbf{P}_1$. As t increases from 0 to 1, the curve departs from $\mathbf{P}_0$ in the direction of $\mathbf{P}_1$, then bends to arrive at $\mathbf{P}_2$ from the direction of $\mathbf{P}_1$.

The second derivative of the Bézier curve with respect to t is

$$\mathbf{B}''(t) = 2(\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0) \tag{4.9}$$

A quadratic Bézier curve is also a parabolic segment. As a parabola is a conic section, some sources refer to quadratic Béziers as "conic arcs".

4.5.1.3 Cubic Bézier Curves

Four points $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$ in the plane or in higher-dimensional space define a cubic Bézier curve Fig. (4.3). The curve starts at $\mathbf{P}_0$ going toward $\mathbf{P}_1$ and arrives at $\mathbf{P}_3$ coming from the direction of $\mathbf{P}_2$. Usually, it will not pass through $\mathbf{P}_1$ or $\mathbf{P}_2$; these points are only there to provide directional information. The distance between $\mathbf{P}_0$ and $\mathbf{P}_1$ determines "how long" the curve moves into direction $\mathbf{P}_2$ before turning towards $\mathbf{P}_3$.

Writing $B_{\mathbf{P}_i,\mathbf{P}_j,\mathbf{P}_k}(t)$ for the quadratic Bézier curve defined by points $\mathbf{P}_i$, $\mathbf{P}_j$, and $\mathbf{P}_k$, the cubic Bézier curve can be defined as a linear combination of two quadratic Bézier curves:

$$\mathbf{B}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2}(t) + t\mathbf{B}_{\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3}(t) \ , \ t \in [0,1] \tag{4.10}$$

The explicit form of the curve is:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \ , \ t \in [0,1] \tag{4.11}$$

For some choices of $\mathbf{P}_1$ and $\mathbf{P}_2$ the curve may intersect itself, or contain a cusp.

Any series of any 4 distinct points can be converted to a cubic Bézier curve that goes through all 4 points in order. Given the starting and ending point of some cubic Bézier curve, and the points along the curve corresponding to t = 1/3 and t = 2/3, the control points for the original Bézier curve can be recovered.

4.5.2 Cubic Bezier Curve Solution

The bezier curves of cubic degree Fig. (4.3) has been selected in this study to be implemented throughout the software code. This proved to be the most effective and flexible form of Bezier curve.

A cubic Bezier curve is defined by four points. Two are endpoints. (x_0, y_0) is the origin endpoint. (x_3, y_3) is the destination endpoint. The points (x_1, y_1) and (x_2, y_2) are control points. Two equations define the points on the curve.

Both are evaluated for an arbitrary number of values of t between 0 and 1. One equation yields values for x, the other yields values for y. As increasing values for t are supplied to the equations, the point defined by $x(t)$, $y(t)$ moves from the origin to the destination.

The parametric representaion of curve can be expressed in $\mathbf{x}(t)$ and $\mathbf{y}(t)$ where $t = [0, 1]$. To obtain

$$x(t) = \mathbf{a}_x t^3 + \mathbf{b}_x t^2 + \mathbf{c}_x t + x_0 \tag{4.12}$$

$$y(t) = \mathbf{a}_y t^3 + \mathbf{b}_y t^2 + \mathbf{c}_y t + y_0 \tag{4.13}$$

To obtain the control points $\mathbf{P}_1 = (x_1, y_1)$, $\mathbf{P}_2 = (x_2, y_2)$, and $\mathbf{P}_3 = (x_3, y_3)$ from the cubic parametric representation of the bezier curve as follow for x

$$\begin{aligned} x_1 &= x_0 + \mathbf{c}_x/3 && (4.14)\\ x_2 &= x_1 + (\mathbf{c}_x + \mathbf{b}_x)/3 && (4.15)\\ x_3 &= x_0 + \mathbf{c}_x + \mathbf{b}_x + \mathbf{a}_x && (4.16) \end{aligned}$$

and as follow for my

$$\begin{aligned} y_1 &= y_0 + \mathbf{c}_y/3 && (4.17)\\ y_2 &= y_1 + (\mathbf{c}_y + \mathbf{b}_y)/3 && (4.18)\\ y_3 &= y_0 + \mathbf{c}_y + \mathbf{b}_y + \mathbf{a}_y && (4.19) \end{aligned}$$

This method of definition can be reverse-engineered to give up the coefficient values $\mathbf{c}, \mathbf{b}$, and $\mathbf{a}$ based on the points described above:

$$\begin{aligned} \mathbf{c}_x &= 3(x_1 - x_0) && (4.20)\\ \mathbf{b}_x &= 3(x_2 - x_1) - \mathbf{c}_x && (4.21)\\ \mathbf{a}_x &= x_3 - x_0 - \mathbf{c}_x - \mathbf{b}_x && (4.22) \end{aligned}$$

$$\begin{aligned} \mathbf{c}_y &= 3(y_1 - y_0) && (4.23)\\ \mathbf{b}_y &= 3(y_2 - y_1) - \mathbf{c}_y && (4.24)\\ \mathbf{a}_y &= y_3 - y_0 - \mathbf{c}_y - \mathbf{b}_y && (4.25) \end{aligned}$$

4.6 TRANFORMATION OF CIRCULAR ARCS INTO BEZIER ARCS

The software in this study is using bezier curves to apply homotopy between their parametric equations. Therefore any one topological shape should be partitioned into series of bezier curves to satisfy this requirement. The following sections describe two methods of obtaining a bezier curve from the circle. The first method is an analytical method[58] that divide the circle into 4 equal bezier curves. The second method is a numerical method that has been invented specially for the purpose of this study. This method has been improved to get the bezier curve from any two points on the circle.

4.6.1 Analytical Method

It is impossible to draw an absolutely exact circle with one Bezier curve. But we can approximate a unit quarter of a circle (90^0 arc) by a cubic Bezier curve with an error 1.96×10^{-4} in the radius, what is ac-ceptable for most practical cases.

To approximate, one should divide the circle into four arcs and convert each of them separately

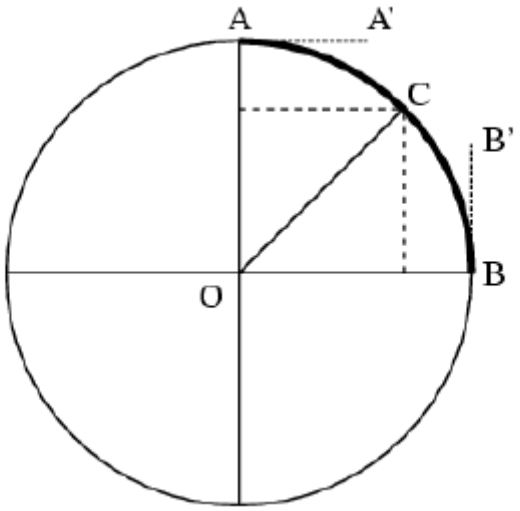


Figure 4.5: Bezier Curve From Circle Arc[58]

Let us consider only the upper right segment (the arc from point A to point B) , because we can convert other segments in the similar way (only some values will be negative). Since the angle AOB is of 90 degrees, the Bezier control line AA' is horizontal, and the Bezier control line BB' is vertical. The radius r of the circle is equal to the length of the lines OA, OB, as well as OC. The point C is on the middle of the arc AB, so the angles AOC and COB equal 45 degrees. The length d of AA' and BB' is unknown, however, it can be expressed as $d = r * k$, where k is a constant (in the literature this constant very often is called as "magic number").

Let us assume that $r = 1$ and the coordinates of the center point $O = [0, 0]$. In this case, $d = k$, so the coordinates of the four points, defining the Bezier curve, are:

$A = [0, 1]$

$A' = [k, 1]$

$B' = [1, k]$

$B = [1, 0]$

According to Eq. (4.3) cubic degree of bezier curve equation is

$$C(t) = (1-t)^3 A + 3t(1-t)^2 A' + 3t^2(1-t)B' + t^3 B \tag{4.26}$$

Since the point C lies at $t = 0.5$, $(1-t) = 0.5$ and the relative x coordinate of C equals the relative y coordinate of C, we can write the following two equations for C:

$$C = \frac{8}{1}A + \frac{8}{3}A' + \frac{8}{3}B' + \frac{8}{1}B \tag{4.27}$$

$$C = \sqrt{1/2} = \sqrt{2}/2 \tag{4.28}$$

Solving the two equations on the x axis (the same result would be for y axis as well) we obtain:

$$\frac{0}{8} + \frac{3}{8}kl + \frac{3}{8} + \frac{1}{8} = \sqrt{2}/2$$

$$k = \frac{4}{3}\left(\sqrt{2} - 1\right) = 0.5522847498 \tag{4.29}$$

So, the control points of the cubic Bezier curve for the upper right arc of a circle with radius r are:

$A = [0,\ r]$

$A' = [r * k,\ r]$

$B' = [r,\ r * k]$

$B = [r,\ 0]$

Consider an arc of less than 90 degree and radius r. Assume that we have to approximate it by one segment of a cubic Bezier curve.

the CW (clockwise) arc is symmetric along the positive x axis. The resulting Bezier curve connects $\mathbf{P}_1$ and $\mathbf{P}_4$ and its boundary tangents are collinear with the vectors $(\mathbf{P}_1 - \mathbf{P}_2)$ at the start point and $(\mathbf{P}_4 - \mathbf{P}_3)$ at the end point. The variation of tangent magnitude L is

within the domain [0, $R * tan(\beta/2)$], The coordinates of the control points $\mathbf{P}_2$ and $\mathbf{P_3}$.

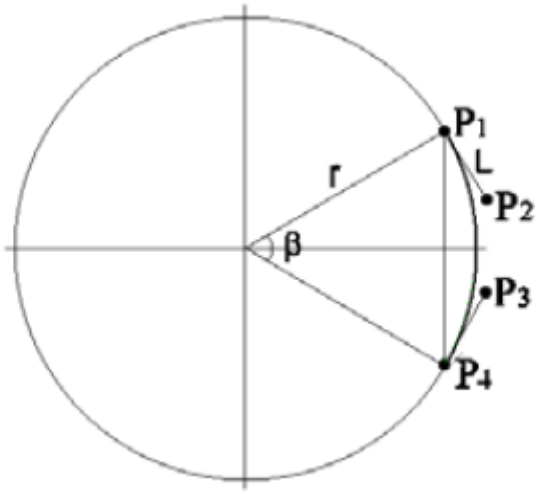


Figure 4.6: Bezier Curve From Two points on Circle[58]

Let the coordinates of the arc start point P_1 and end point P_4 be (x_1, y_1) and (x_4, y_4), respectively. From the elementary geometry, the coordinates of the cubic Bezier control points are $\mathbf{P}_2 = (x_2, y_2)$ $\mathbf{P}_3 = (x_3, y_3)$:

$$x_2 = x_1 + kR\sin(\varphi) \tag{4.30}$$
$$y_2 = y_1 – kR\cos(\varphi) \tag{4.31}$$

$$x_3 = x_4 + kR\sin(\varphi) \tag{4.32}$$
$$y_3 = y_4 + kR\cos(\varphi) \tag{4.33}$$

4.6.2 Numerical Method

In the present study, it is assumed that a cubic bezier curve of four control points can be approximated into a circle arc by taking two tangents from two selected points on the specified circle as shown in Fig. (4.7)

The curve is constructed across the smaller angle ϕ between the two points $\mathbf{P}_1$ and $\mathbf{P}_2$

The two tangents are taken to have the same length i.e.

$$\mathbb{P}_1\mathbb{P}_2 = \mathbb{P}_3\mathbb{P}_4 = g \tag{4.34}$$

This length however is assumed in the begining of calculation to be 0, g is then increased

by an increment Δv fraction of the perimeter of the circle which can be formulated as

$$\Delta v = 2\pi R/300 \approx 0.021R \tag{4.35}$$

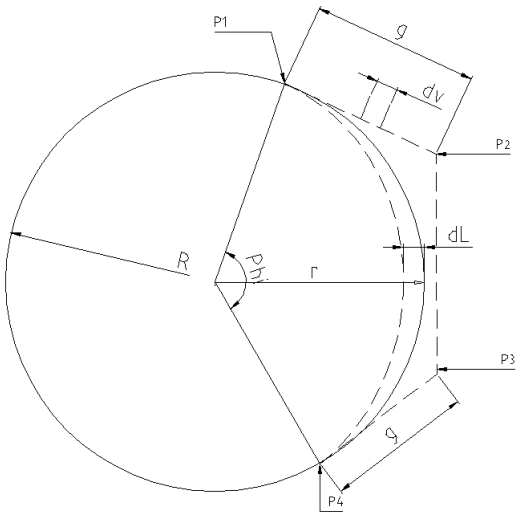


Figure 4.7: Approximation of Circular Arc to Bezier Curve

The bezier curve may then be constructed using the four points $\mathbb{P}_1$, $\mathbb{P}_2$, $\mathbb{P}_3$, and $\mathbb{P}_4$.

The accuracy of the curve in representing the circular arc is determined from the maximum peak radius r of the constructed bezier curve measured to the radius R of the specified circle

$$\frac{r}{R} = 1 - \frac{dL}{R} \leq 0.99 \tag{4.36}$$

If this ratio is smaller than the set value 99%, the length g of the tangents may be increased again by Δv, and a new Bezier curve is constructed.

This procedure is repeated until r/R lies within the accepted level of accuracy. For the algorithm steps, the user is advised to refer to the appendix (C.1.1) for a complete algorithm description.

4.7 CONCLUSIONS

In this chapter the concept of manifolds as a higher abstracted representaion of curves is introduced followed by the concept of homotopy between functions and their applications on curves as a first degree manifold.

The bezier curve (as a sample of one dimensional manifold) has been discussed in more detail. Finally a new numerical solution have been introduced for the conversion of any circular arc into bezier curve of cubic degree.

CHAPTER V

COMPUTER GRAPHICS PROGRAMMING

5.1 INTRODUCTION

Graphics programming is the process of sending commands to the computer graphic card for the sake of displaying geometrical shapes on the computer screen. There are two famous libraries for drawing 3D objects, DirectX (which is a microsoft technlology), OpenGL (which is a de-facto standard). In this thesis OpenGL library has been choosed to accomodate the requirements for the thesis software package.

OpenGL is a software interface to graphics hardware. The interface consists of about 150 distinct commands that are used to specify the objects and operations needed to produce interactive three-dimensional graphics.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL. Instead, one must work through whatever windowing system that controls the particular hardware being used. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow one to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, the desired model must be built up from a small set of geometric primitives - points, lines, and polygons.

A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS[1] curves and surfaces. GLU is a standard part of every OpenGL implementation.

First section of this chapter begins the discussion of OpenGL philosophy in drawing 3D objects, then goes through OpenGL Pipe Line and define some important OpenGL intrinsic keywords.

Second section deals with the concept of representing complex 3D objects into memory by defining the word **Model**. Models are the conceptual entities that are stored in the computer memory, which needs special attention to map precisely between what is seend on

[1]Non-Uniform Rational Basis Spline (NURBS) is a mathematical model commonly used in computer graphics for generating and representing curves and surfaces. It offers great flexibility and precision for handling both analytic (surfaces defined by common mathematical formulae) and modeled shapes.

the screen and what is stored in the computer RAM.

Final section in this chapter discuss model snappings. The snapping of line is always expressed by its first and end points. The section will illustrate the mathematical model behind finding the tangent point, first and end points of the shape under the mouse.

5.1.1 OpenGL as A State machine

OpenGL is a state machine, in the sense that various states (or modes) that are put into it remain in effect until changed. For example, the current color is a state variable, that is used in drawing every object until explicitly changed. The current color is only one of many state variables that OpenGL maintains. Other state variables control such things as the current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes, pixel-packing conventions, ... etc. Many state variables refer to modes that are enabled or disabled with the command **glEnable()** or **glDisable()**.

Each state variable or mode has a default value, and at any point one can query the system for each variable's current value. Typically, one of the six following commands may be used to do this: **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, or **glIsEnabled()**. Some state variables have a more specific query command such as **glGetLight()**, **glGetError()**, or **glGetPolygonStipple()**. In addition, one can save a collection of state variables on an attribute stack using **glPushAttrib()** or **glPushClientAttrib()**, temporarily modify them, and later restore the values with **glPopAttrib()** or **glPopClientAttrib()**. For temporary state changes, these commands should be used rather than any of the query commands, since they are likely to be more efficient.

5.1.2 OpenGL Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering, as shown in Fig. (5.1), is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do.

Geometric data (vertices, lines, and polygons) follow the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer.

Framebuffer[59] is the portion of memory reserved for holding the complete bit-mapped image that is sent to the screen. Typically the frame buffer is stored in the memory chips on the video adapter. In some instances, however, the video chipset is integrated into the motherboard design, and the frame buffer is stored in general main memory.

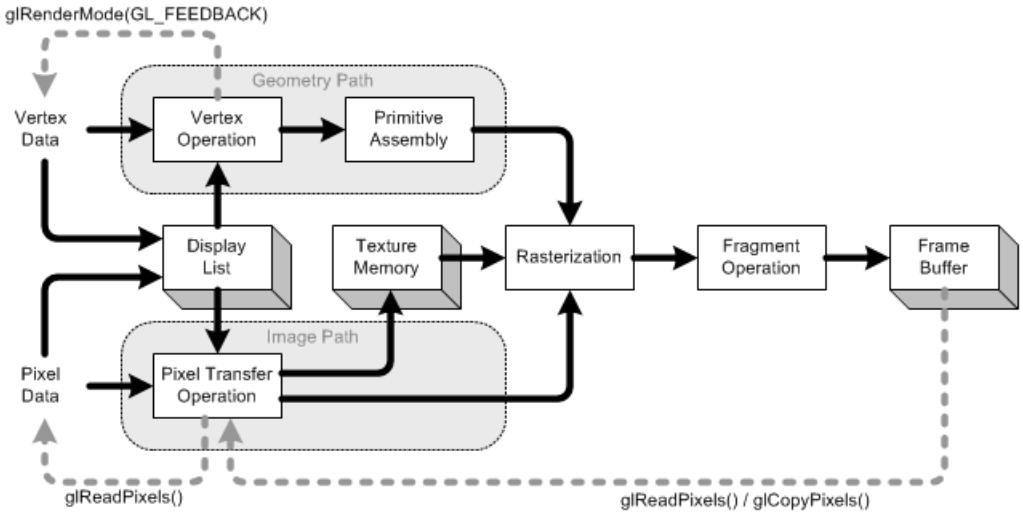


Figure 5.1: OpenGL 1.1 Pipe Line[60]

In the following, a quick review of the various processes of the OpenGL pipeline shown in Fig. (5.1) is presented.

a- Display Lists

All data, whether describing geometry or pixels, can be saved in a display list for current or later use. (The alternative to retaining data in a display list is processing the data immediately - also known as immediate mode.) When a display list is executed, the retained data is sent from the display list in the next process just as if it were sent by the application in immediate mode.

b- Evaluators

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points.

c- Per Vertex Operations and Primitive Assembly

The "per-vertex operations" stage, converts the vertices into primitives[2]. Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on the screen.

If advanced features are enabled, this stage is even busier. If texturing[3] is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting cal-

[2]Primitives are ways that OpenGL interprets vertex streams, converting them from vertices into triangles, lines, points, and so forth.

[3]A texture is an OpenGL Object that contains one or more images that all have the same image format.

culations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

Clipping, which is the process of eliminating unseen portions of geometry that fall outside the viewed space, or in case that is defined by a plan, considered a major part of primitive assembly.

Point clipping simply passes or rejects vertices. Line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped.

The viewport and depth (z coordinate) operations are then applied in the perspective division which makes distant geometric objects appear smaller than closer objects.

If culling[4] is enabled and the primitive is a polygon, then it may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines.

The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step which is reviewed below.

d- Pixel Operations

While geometric data take one path through the OpenGL rendering pipeline, pixel data take a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step.

If pixel data is read from the frame buffer, for further processing, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. These results are then packed into an appropriate format and returned to an array in system memory.

There are special pixel copy operations to copy data in the framebuffer to other parts of the framebuffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the framebuffer.

e- Texture Assembly

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them.

Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

[4]Triangle primitives after all transformation steps have a particular facing. This is defined by the order of the three vertices that make up the triangle, as well as their apparent order on-screen. Triangles can be discarded based on their apparent facing, a process known as Face Culling.

f- Rasterization

Rasterization is the conversion of both geometric and pixel data into fragments. Each fragment square corresponds to a pixel in the framebuffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support antialiasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

g- Fragment Operations

Before values are actually stored into the framebuffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled.

The first operation which may be encountered is texturing, where a texel (texture element) is generated from texture memory for each fragment and applied to that fragment.

In addition to the texturing, there are other operations that can be calculated:

- Scissor Test: The Scissor Test is a Per-Sample Processing operation that discards Fragments that fall outside of a certain rectangular portion of the screen.
- Alpha Test: The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value.
- Stencil Test: The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (x_w, y_w) and a reference value.
- Depth Buffer Test: The depth buffer test discards the incoming fragment if a depth comparison fails (the depth buffer is for hidden-surface removal).

Failing an enabled test may end the continued processing of a fragment's square. Then, blending, dithering, logical operation, and masking by a bitmask may be performed.

Finally, the thoroughly processedfragment is drawn into the appropriate buffer, where it has finally advanced to be a pixel and achieved its final resting place.

5.2 GRAPHICAL OBJECT MODEL

Drawing separate points and lines doesn't have a specific meaning until its being encapsulated into a higher conceptual entities. The Graphical Model is the higher conceptual entity that logically contains more than one point, or surfaces group that act as a unique entity.

This unique entity is transforming as a whole object, and it contains a specific information like center of gravity, or the color of its subset elements.

The graphical model undergoes a whole transformation of its sub elements and act as a one object throughout the life of the running program.

5.2.1 Graphical Model Naming

There are two naming (coding) of the graphical model when running into the program, used to distinguish between models in memory. The first name is a color name that consists of 4 bytes. The second name is a universal unique name generated by GUID (Globally Unique Identifier) used in offline sessions and when saving and retrieving the model into the current program.

Color Name

This special naming is done automatically for each newly created model in the viewport. The name is used in the viewport operations like sensing the model under mouse.

This name consists of four bytes that are corresponding to Red, Green, Blue, and Alpha components (or RGBA). The 32 bit naming buffer can hold numbers from 0 to $2^{32} = 4,294,967,296$ or more than 4 billion identifier which is very sufficient to store our models in the viewport.

Universal Name

The universal naming is generated randomly by a GUID number (Globally Unique Identifier). The term GUID typically refers to various implementations of the universally unique identifier (UUID) standard.

GUIDs are usually stored as 128-bit values, and are commonly displayed as 32 hexadecimal digits with groups separated by hyphens, such as {21EC2020-3AEA-1069-A2DD-08002B30309D}. GUIDs generated from random numbers sometimes contain 6 fixed bits saying they are random and 122 random bits; the total number of unique such GUIDs is 2122 or 5.3×1036. This number is so large that the probability of the same number being generated randomly twice is negligible; however other GUID versions have different uniqueness properties and probabilities, ranging from guaranteed uniqueness to likely non-uniqueness. Assuming uniform probability for simplicity, the probability of one duplicate would be about 50% if every person on earth owned 600 million GUIDs.

The universal name is very important when saving and retrieving the models to a persistent sorage (i.e. files on disk). Due to the expected widespread usage of the thesis software, people may share their created geometries with each other, and may also accumulate their work together. Dealing with this naming of models will allow a future integration between all of these people and all of their master pieces.

5.2.2 Picking Viewport Models

Selection of models in viewport[5] incorporates the process of knowing the model under the mouse tip. In CAD programs, it is also required to sense the model under the mouse while

[5] A rectangle on the raster graphics screen (or interface window) defining where the image will appear, usually the entire screen or interface window.

the mouse is moving, and to highlight this model to notify the user that this model is ready to be selected. an example of models that need to be selected and move around in the viewport, are the control points of bezier curve.

In the process of modifying the bezier curve shape, two operations are needed. First operation is to click on the bezier curve, this brings out the bezier control points and display them on the viewport. Second operation is to move it by dragging the control point with mouse. The dragging operation is defined as (Mouse Button Down -> Mouse Move -> Mouse Button Up (Release)). During this dragging operation, the control point is following the mouse movement to any location the user desires.

Figure 5.2: Back Buffer (Color Naming View)

The mechanism of knowing which model is under the mouse is done through a color picking technique. This technique has been first mentioned in the OpenGL Red Book[59]. The whole process is done in the OpenGL Frame Buffer with one special rule that this buffer will never make it to the user eyes as shown in Fig. (5.2), this buffer is only intended to render the current scene with the color naming of the model.

The graphical picking algorithm can be summarized as follows:

- Draw the models with their associating naming colors Fig. (5.2).
- Read the pixel where the mouse tip location is pointing at.
- Process this color and get the corresponding memory model for this color.
- Notify the model that it is now under the mouse to do its associated behaviour.

- Do NOT swap buffers, otherwise the user will see the color coding.
- Render the models again as normal into the back buffer and present this buffer to the user Fig. (5.3).

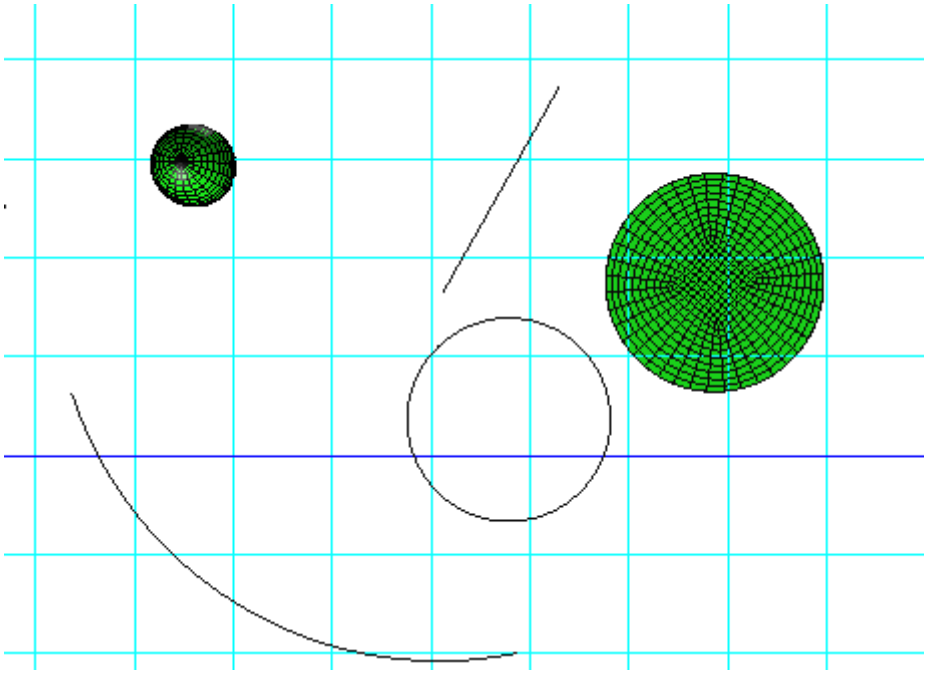


Figure 5.3: Presented Buffer User View

5.2.3 Models Interactive Operations

The viewport is a living space, each model in the space can sense the mouse movement over its representing shape, and respond to this movement. When the mouse goes near any model in the view port, this model will respond to the mouse pointer by highlighting itself to notify the user that it can be selected by the mouse left button.

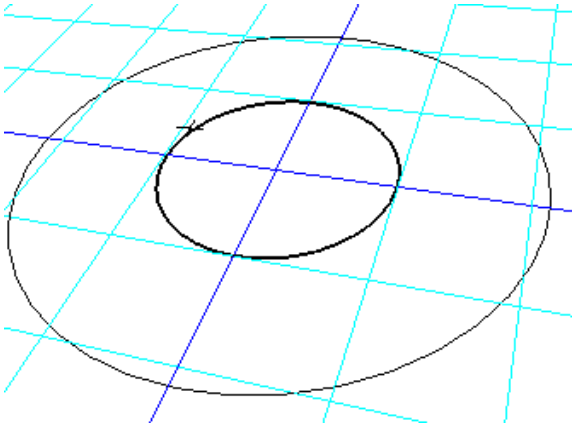


Figure 5.4: Highlighted Inner Circle.

Fig. (5.4), illustrate two circles (outer, and inner), the inner circle is highlighted by a thick represented line due to the mouse tip approach to its shape. While Fig. (5.5) illustrate the same two circles but with highlighted outer circle.

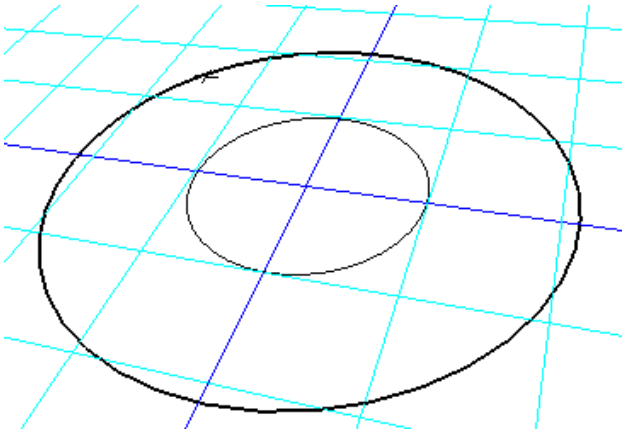


Figure 5.5: Highlighted Outer Circle

Highlighting Algorithm can be done by taking care of the model selection buffer[6]. The shape thickness in the selection buffer is intentionally increased to 3 points more than the usual drawing of 1 point thickness[7] to ensure more thick shape edges higher than default value of 1.

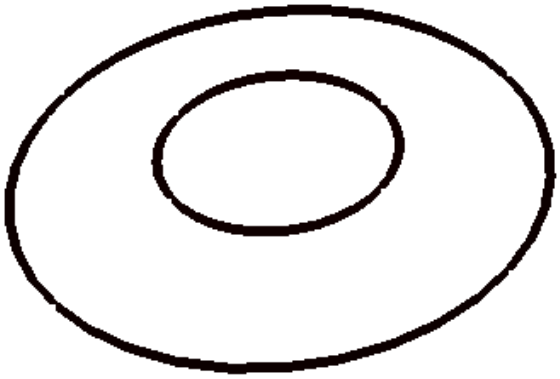


Figure 5.6: Thick Shapes in Selection Buffer as seen by mouse.

Fig. (5.6), illustrate two circles with thick shape edges that are rendered in the Selection Back Buffer. The mouse hovering on the edge will result in knowing that the mouse got near the shape and a visual feedback to the user can take place.

[6]Selection Buffer here is referring to the pass that all models are rendered with their naming colors.
[7]Thickness value representation is a hardware and driver responsibility.

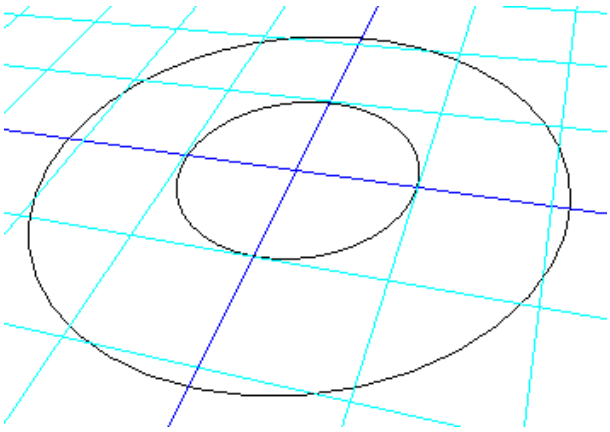


Figure 5.7: Normal Shapes in Presented Buffer as seen by user.

Fig. (5.7), illustrate the same two circles but in their final viewing form on the screen. The two circles state now are not selected nor being approached by the mouse pointer.

5.3 MODEL SNAPPING OPTIONS

Snapping while drawing is a vital functionality in the CAD applications in general. Whenever the user tries to connect two lines together, or to pick a certain point over the model topology, Snapping options come in handy to make it easier for the user to perform such task.

There are three snapping options implemented in the current study:

- Nearest Point Fig. (5.8).
- Start - End Points Fig. (5.9).
- Middle Point Fig. (5.10).

When one or more of these snapping options is enabled, a red point is highlighted near the mouse pointer during its movement to indicate that the active point is the current snap point.

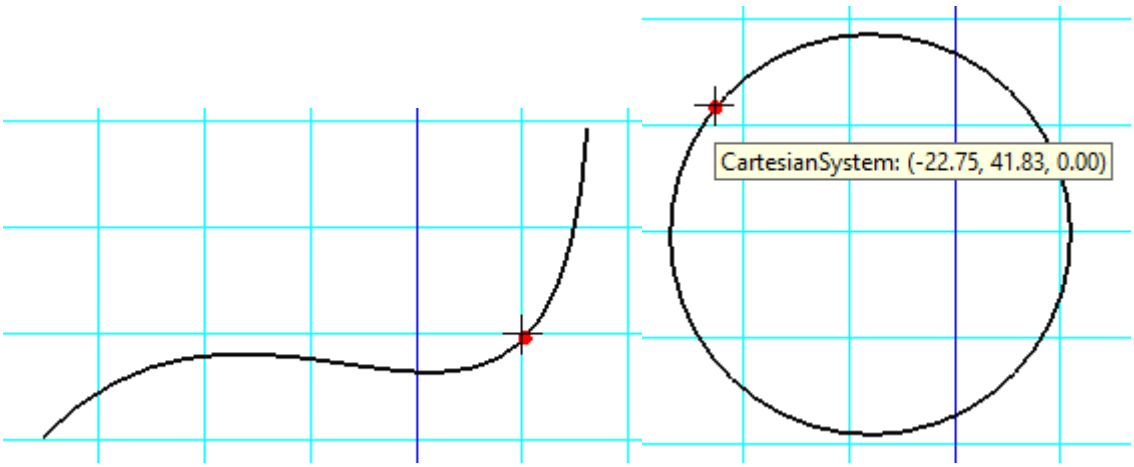


Figure 5.8: Nearest Point Snapping

The snapping options also differ in behaviour when applied to certain shapes. For example the circle doesn't have start-end points but it has a near point that can be calculated, also the middle point snapping can be applied to the circle center point.

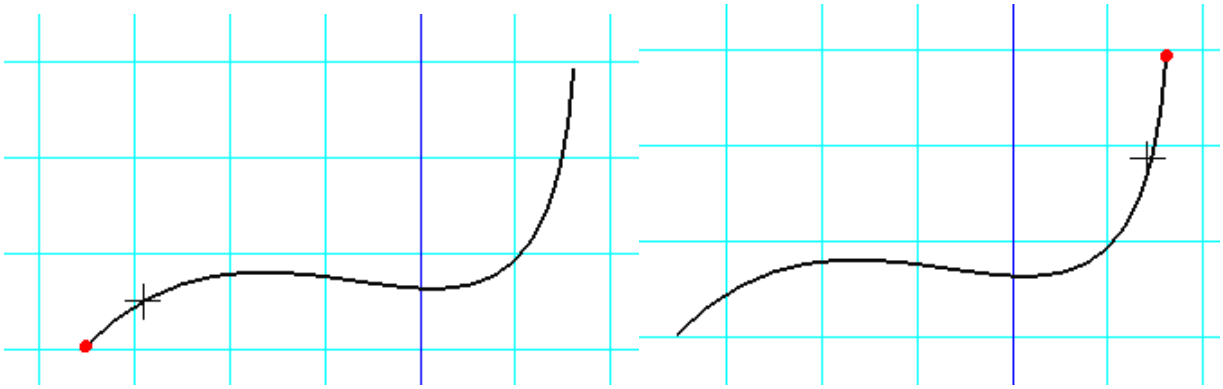


Figure 5.9: Snapping to first, and last points of the curve

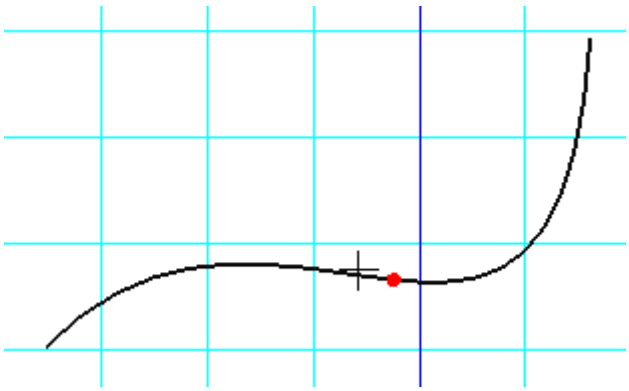


Figure 5.10: Middle Point Snapping

5.3.1 Snapping Options Calculations

Each shape model has a different algorithm in calculating its snapping points. Giving the fact that the snapping point is only calculated when the mouse pointer is near the shape and lie on its terriority, the first information of the current location of mouse pointer is obtained.

From this location, the nearest point of the shape is calculated depending on the shape type, following that step, the program checks if the snapping option of start-end, or middle points are checked and then calculate their values.

There are four distinct shapes that are supported in this thesis:

- Line

- Circle
- Bezier Curve
- Arc

5.3.2 Line Segment

Nearest Point

The nearest point can be calculated from the first point of the line x_1, y_1 to the location of the near point p_x, p_y as illustrated in Fig. (5.11).

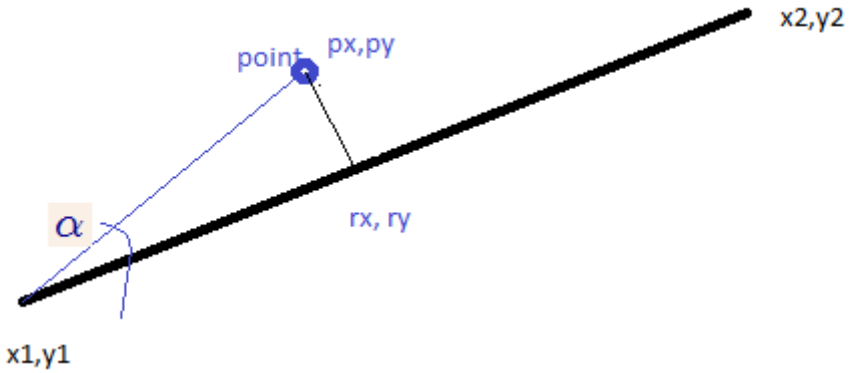


Figure 5.11: Nearest Calculation for Line Segment

Considering that the mouse point is too close to the line, the α angle is very small, and can be neglected, hence the mouse point location can be considerd lieing on the line itself. for this situation the length between the projection of $\mathbf{p}$ on the line at r_x, r_y can be considered nearly equal the length between the point $\mathbf{p}$ and x_1, y_1.

The calculation algorithm is done as follows:

- Length between mouse point and the line first point is determined as

$$|\mathbf{p} - (x_1, y_1)| \tag{5.1}$$

- Calculting the parameter of the parameteric equation of the line is calculated as

$$t = \frac{|\mathbf{p} - (x_1, y_1)|}{|(x_2, y_2) - (x_1, y_1)|} \tag{5.2}$$

- substituting the parameter t in the line parametric equation

$$(r_x, r_y) = \mathbf{r}(t) = x(t) + y(t) \tag{5.3}$$

  giving the exact location for $\mathbf{r}$.

- The point on $\mathbf{r}$ is then highlighted with red to the end user.

First-End Points

If this snapping option is enabled as shown above, a simple comparison is done over the calculated parameter t. If $t > 0.5$ then the snapping point is calculated based on the last point location of the line $\mathbf{r} = (x_2, y_2)$, and for $t < 0.5$ the snapping point is calculated based on the first point of the line $\mathbf{r} = (x_1, y_1)$.

Middle Point

The middle point discovery also depends on the information of the mouse point that is very near to the line. In this case the line parametric equation is calculated by giving 0.5 as a parameter $(r_x, r_y) = \mathbf{r}(0.5) = x(0.5) + y(0.5)$.

5.3.3 Circle

To calculate the near point, a parametric equation for the circle is obtained through the entire shape. The parametric equation should begin from 0 and ends with 1, a conversion is made by multiplying with 2π

$$\mathbf{r} = r(s) = x(s) + y(s) = \rho\cos(s) + \rho\sin(s) \tag{5.4}$$

Where $s = 0 \rightarrow 2\pi$, ρ is the circle radius.

again the mouse pointer is near the circle edge which can be considered that the mouse pointer is approximately over the circle shape line. Calculating the $\mathbf{r}$ point Fig. (5.12), on the circle from $\mathbf{p}$ mouse location point can be summarized in the following algorithm:

- Determining the angle α between vector $\mathbf{p} - \mathbf{c}$ and x axis.
- $s = \alpha/2\pi$
- $\mathbf{r} = \begin{cases} r(s) & p_y > 0 \\ r(-s) & p_y < 0 \end{cases}$

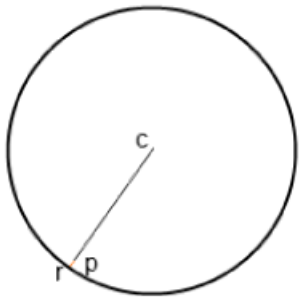


Figure 5.12: Circle Tanget Snapping

5.3.4 Bezier Curve

To find the near point of the bezier curve, giving that the mouse point location is near the curve edge; a digital algorithm is conducted to discover the parameter of curve (that lies between $0 \rightarrow 1$).

The problem with bezier curve is that they lack a central point like circle, instead its curvature is changing all over its path line. The algorithm depends on the error approximation between the mouse point and curve point in such that giving the minimum length between the two points. The condition that satisfy the minimum length between the two points will be selected as the nearest point.

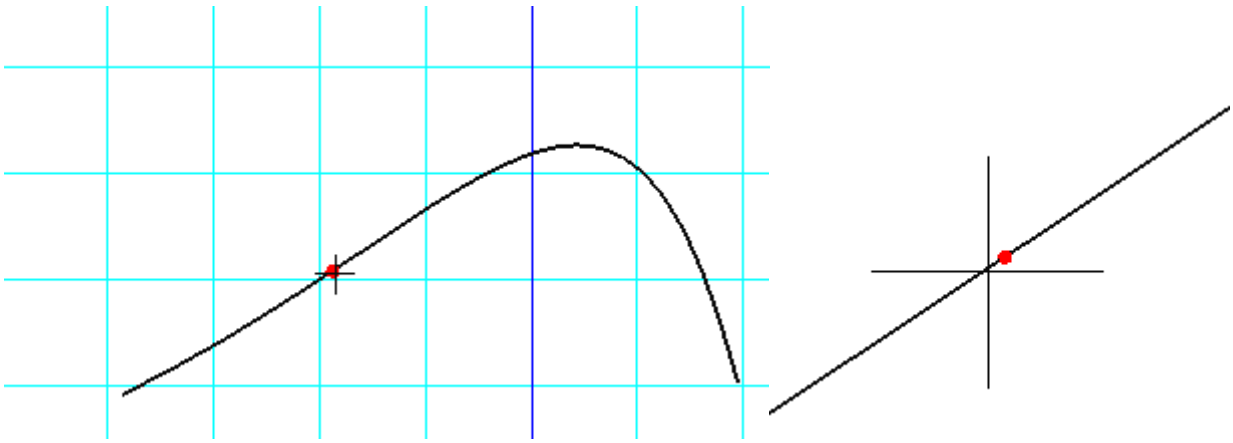


Figure 5.13: Bezier Curve Nearest Point

Minimum length between mouse location and curve point is calculated as seen in Fig. (5.14) by determining the shortest path between mouse and curve.

- WHILE ($\delta e > 0.1$)
  - Increase parametric value s with Δs
  - Calculate $\delta new = |\mathbf{r}_s - \mathbf{p}|$
  - if the new length $> \delta e$ then decrease the value of Δs by half and back to previous step.
- END

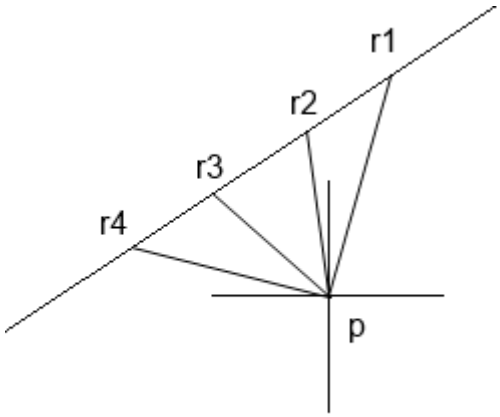


Figure 5.14: Bezier Curve Nearest Point Suggested Locations

The algorithm ends when the δe error is being between $0 \rightarrow 0.1$, and the corresponding $\mathbf{r} = r_s$ is selected as the desired nearest point.

Snapping for first-end, and middle points is simply obtained by substitiuing the parametric equation of curve with 0, 1, and 0.5 values.

5.3.5 Circular Arc

Considering the Arc as a subset of the circle, with three points, center of the arc, first point, and last point. When mouse goes near the arc the arc is highlighted and then the snapping calculation takes control if enabled.

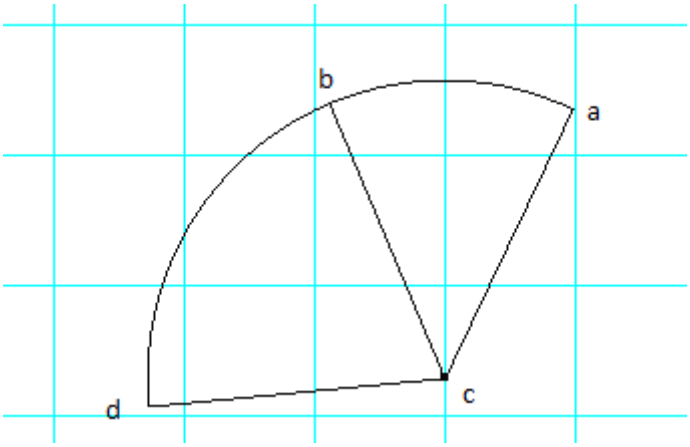


Figure 5.15: Arc Snapping Calculation

The nearest point (b) is always near the arc, however the interest of the arc is to obtain the first and last points of the arc. This can be calculated by the following algorithm:

- Obtaining Normalized Vectors $\tilde{\mathbf{ca}} = a - c$, $\vec{\mathbf{cb}} = b - c$, $\vec{\mathbf{cd}} = d - c$
- Calculating the total arc angle θ_{total} between $\tilde{\mathbf{ca}}$, and $\vec{\mathbf{cd}}$.
- Calculating the angle θ_{acb} between $\tilde{\mathbf{ca}}$, and $\vec{\mathbf{cd}}$.

- Calculating the fraction $s = \frac{\theta_{acb}}{\theta_{total}}$.
- if $s < 0.5$ highlight first point
- if $s > 0.5$ highlight last point.

5.4 CONCLUSIONS

In this chapter the OpenGL library for 3D graphics visualization has been introduced. The model concept of graphical representation has been disucssed.

Selection of models in viewport and sensing the mouse locations has been discussed with their corresponding algorithms.

Finally, snapping of points in certain shapes have been introduced to illustrate how CAD programs respond to the mouse pointer movement on the screen.

CHAPTER VI

GRIDDING ALGORITHMS AND OPERATIONS

6.1 INTRODUCTION

The gridding technique in this chapter assumes the ability to divide the fluid flow domain into quadratic shapes. The partitioning process is a user specific task that aims at providing a grid with the appropriate properties discussed in Chapter II, and III. (regions that contain empty spaces and have their edges as bezier curves) as shown in Fig. (6.1).

The chapter discusses the gridding operation of the quadratic shapes in addition to the special algorithms for making two dimensional and three-dimensional grids.

The algorithm goes into the domain partitions and create the grid lines based on the user input of the number and concenration of cells in these regions.

Section 6.2 discusses a single quadratic shape, and how the gridding operation is carried out on this shape to get the desired grid. After that, the discussion goes through the concentration of the grid lines in terms of expansion and contration of lines based on single or double stretching functions.

In Section 6.3 The connectivity between neighbouring quadratic shapes is discussed. This calls for special attention as it plays a crucial role in extrusion and rotation of quadratic shape grid.

Section 6.4 discusses the operation of creating three dimensional grids (Hexahedral Cells) from two dimensional grids (Quadrilateral Cells) with the aid of extrusion and revolving of the quadrilateral grids created earlier.

Finally, a special algorithm is discussed for revolving axisymmetric shapes to obtain hexahedral cells from rotating the domain a 90 degree around the axis of symmetry.

6.2 TWO-DIMENSIONAL GRIDDING OPERATIONS

Fig. (6.1), shows a flow domain divided into 6 shapes, intended to be covered with quadratic cells. The sides of quadratic shape in Fig. (6.2a), sides (a), (b), (c), and (d), are bezier curves with four control points each. This allows the quadratic shape to take many shapes by modifing the control points of the bezier sides. The shapes can be circles, rectangles, or even hyperbolic curves.

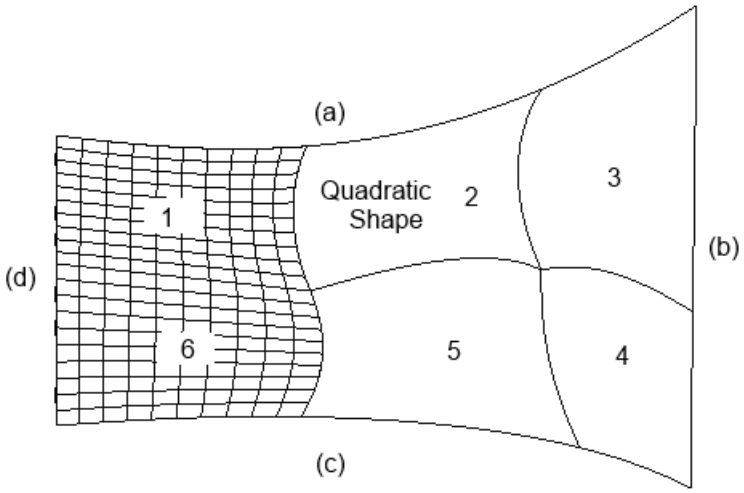
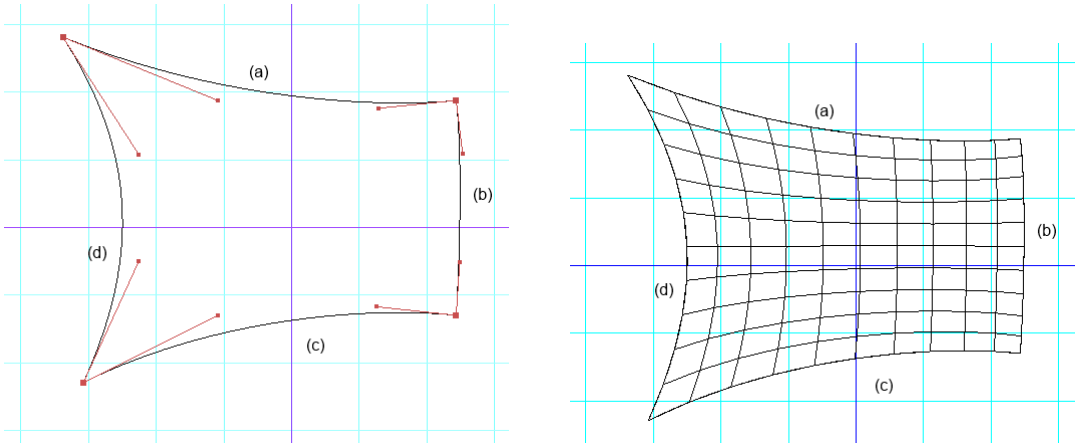


Figure 6.1: Partioned Flow Domain



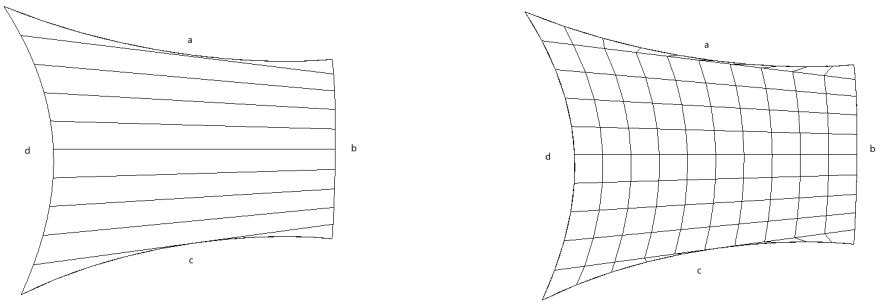
(a) Empty Quadratic Shape (b) Gridded Quadratic Shape

Figure 6.2: Empty and Gridded Quadratic Shape

Fig. (6.2b), shows the quadratic shape after gridding, the process is discussed below.

6.2.1 Linear Gridding

First attempt in gridding the quadrilateral shape is carried out using direct lines calculations between two opposite sides of the shape. The calculated lines Fig. (6.3a) between (d), and (b) curves, serves as the starting point for the calculation of the transversal curve points between (a), and (c) Fig. (6.3b).



(a) Straight Lines Calculations

(b) Second Step Transversal Curves Calculations

Figure 6.3: Linear Gridding

A major problem in this technique lies in the use of the straight lines generation itself. In the case of too many cells, the generated lines may overlap the other opposite curved sides as shown in Fig. (6.4).

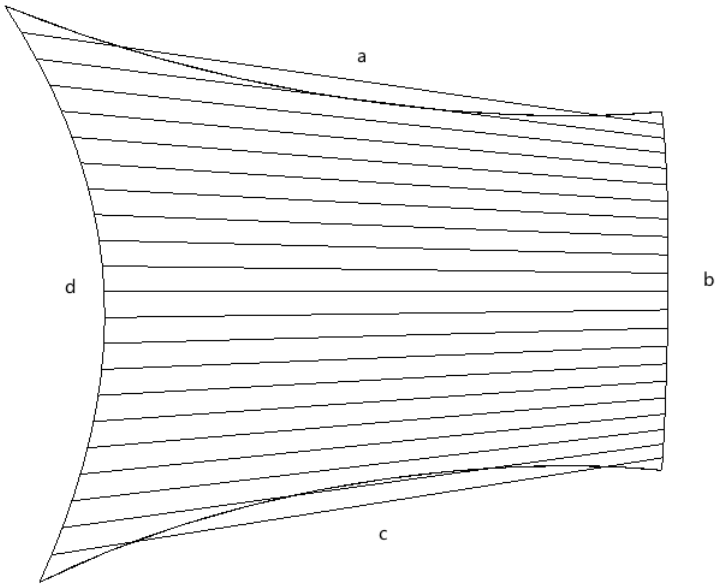


Figure 6.4: Overlapping of Straight Lines

This problem however can be solved by using homotopy between functions of the quadratic shape sides as will be illustrated next.

6.2.2 Homotopy and Gridding Operations

Gridding the quadrilateral shape is carried out using homotopy between the quadratic equations representing the opposite bezier curves (a/c) or (b/d), Fig. (6.5). First, homotopy is done between curves (a), and (c) in terms of the parametric equations discussed in Sec. 4.5, resulting in new unadjusted curves (a', c') as shown in Fig. (6.5).

The number and locations of these curves are determined by the user depending on how many rows and columns of cells should the quadrilateral shape contain. The unadjusted

curves do not necessarily end and/or start on curves (b), and (d) as shown in Fig. (6.5). These curves have to be adjusted to begin and terminate exactly on curves (b) and (d).

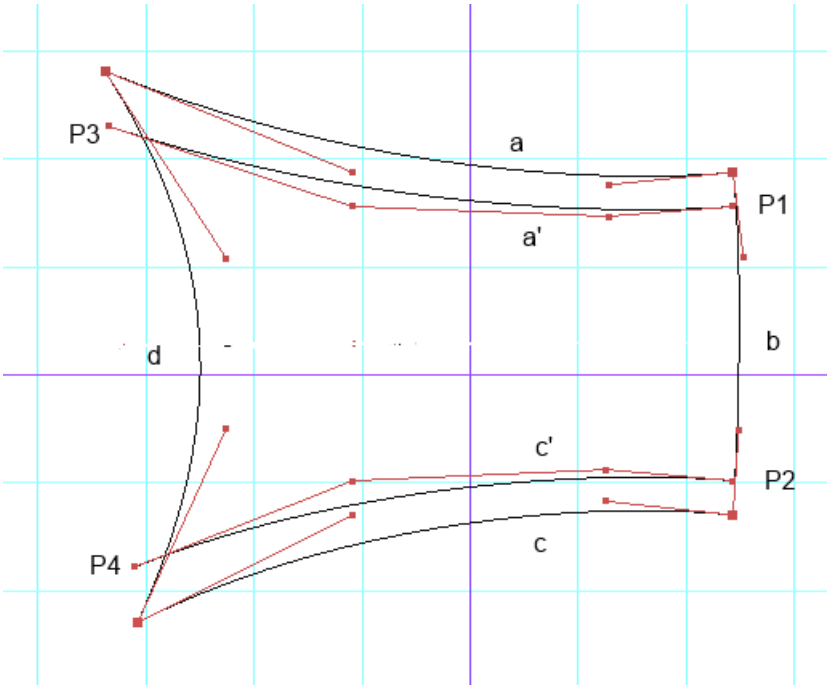


Figure 6.5: Two Un-adjusted Curves at 0.1, and 0.9 between Top and Bottom Curves

Adjustment is carried out by locating the terminal points of curves a', and c' on curves (b) $(P1, P2)$, and on (d) $(P3, P4)$ using the values of parameter t in Equ. (4.12, 4.13) used to create the curves (a', c'). The Bezier curves a', and c' are recalculated using the new terminal points as shown in Fig. (6.6).

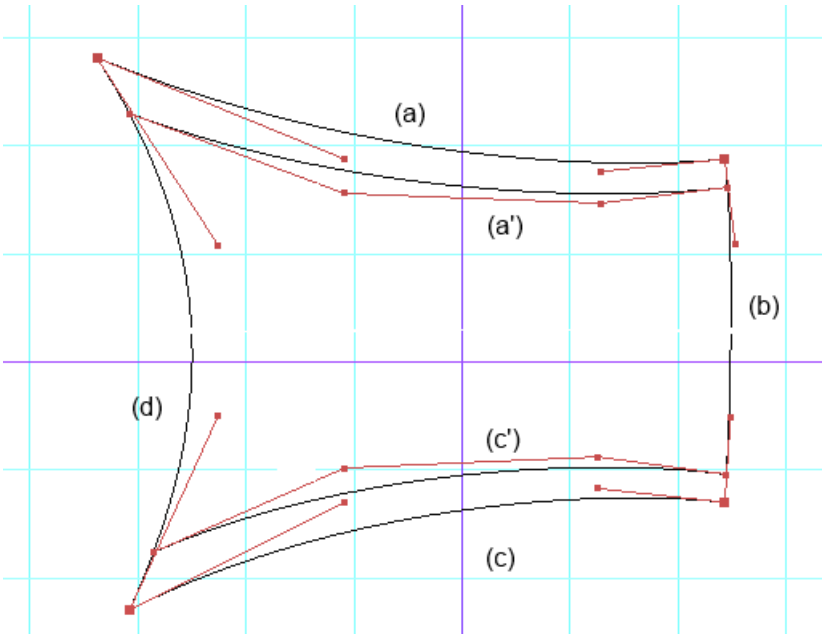


Figure 6.6: Two adjusted curves at 0.1, and 0.9 between Top and Bottom Curves

After adjusting all generated curves between (a), and (c) with reference to curves (b),

and (d) the quadrilateral shape will have all of its inner curves finalized as shown in Fig. (6.7).

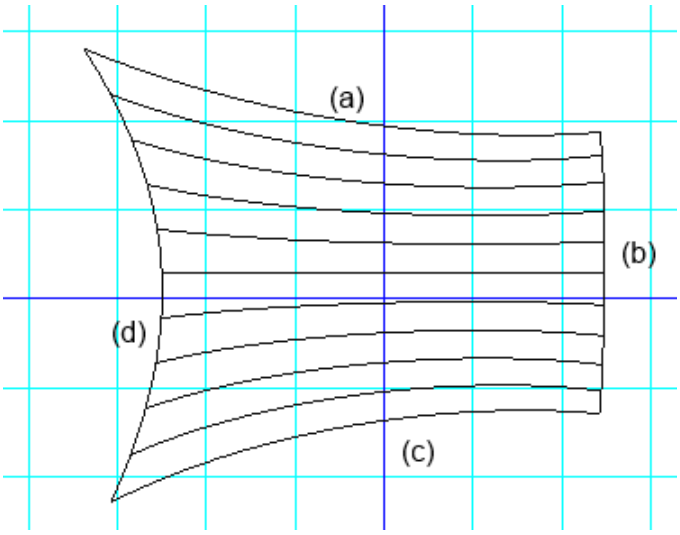


Figure 6.7: All adjusted curves (a), and (c), in quadratic shape

The last step of gridding the quadrilateral domain is to calculate the inner transversal curves between curves (b), and (d). This process is easily done by using the previous adjusted curves as follow; Beginning with curve (a) going through inner curves until reaching curve (c):

- Calculate a point on each curve at the known value of t (ranging between $0 \rightarrow 1$) based on the current location of desired curve.
- After calculating all transversal points Fig. (6.8), a transversal curve is generated by connecting these points together by straight lines.
- This operation is repeated again with different values of t until the area from curve (b) to curve (c) is covered.

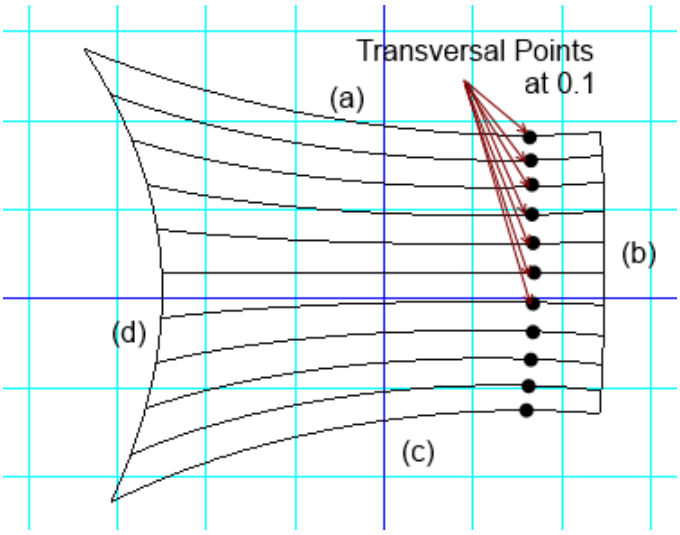


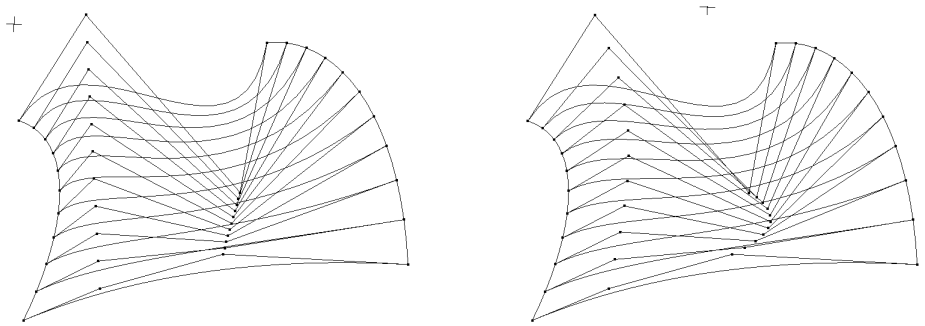
Figure 6.8: Transversal Points Calculation

Finally after all points are connected transversally, the whole quadrilateral shape is now fully gridded into quadratic cells as shown in Fig. (6.2b).

The function responsible for making the homotopy curve between curves (a), and (c), and adjusting its terminals on curves (b), and (d) is listed in App. (C.2.1).

6.2.3 Enhanced Homotopy Gridding

Moving the terminal points of the bezier curve changes the curve profile to accomodate to the desired shape Fig. (6.9a). However, a more accurate adjustment of the curve profile is further needed to get a smoother transition by moving the two middle points of the bezier curve as shown in Fig. (6.9b).



(a) Homotopy curves after adjusting terminal points

(b) Homotopy curves after adjusting the middle points for smoother transition

Figure 6.9: Enhancing intermediate curves smoothness

The algorithm used in modifying the middle points Fig.(6.10) of the generated cuve can be summarized as follows:

- Length between terminal points P_0 and P_1 is determined $L_{01} = |P_0P_1|$.
- Terminal point P_0 is then transformed to the side curve at its designated location.
- The length between the new point and middle point is determined $L'_{01} = \left|P'_0P_1\right|$.
- Ratio between new length and old length is obtained $r = L'_{01}/L_{01}$.
- Point P_1 location is extended along the line $\overline{P_0P_1}$ with the calculated ratio r.
- Point P_2 location is extended along the line $\overline{P_1P_2}$ with the calculated ratio r.
- The same steps are repeated for points P_3 and P_2 along the lines $\overline{P_3P_2}$, and $\overline{P_2P_1}$ with the same ratio (accumulated over previous calculations).

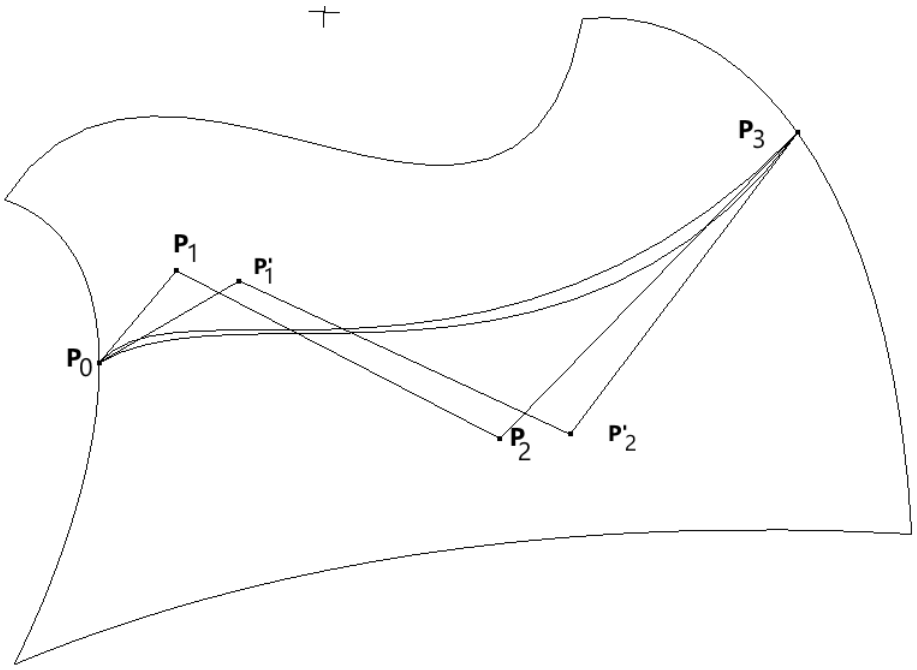


Figure 6.10: Enhanced Homotopy by Middle Points Transformation

6.2.4 Gridding Techniques Comparison

A simple comparison between the three types of gridding techniques is found in the Table (6.1).

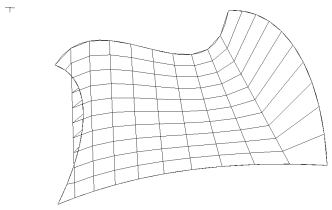
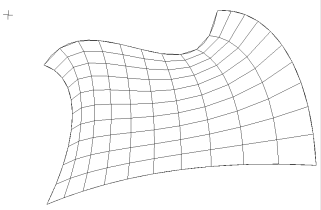
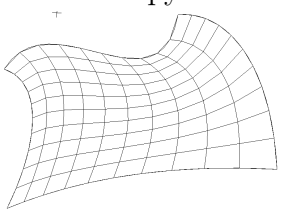
	Linear	Homotopy	Enhanced Homotopy
			
Aspect Ratio	Max Mean Min 4.596 1.963 1.098	Max Mean Min 2.543 1.766 1.018	Max Mean Min 2.231 1.631 1.004
Skewness	Max Mean Min 0.973 0.206 0.013	Max Mean Min 0.520 0.190 0.029	Max Mean Min 0.666 0.255 0.047
Curves Alignment	Very Poor	Good	Very Good

Table 6.1: Gridding Techniques Comparison

6.2.5 Contracting / Stretching Function

The gridding in the quadrilateral shape depends on a stretching function that is based on geometric series extension, or contraction, represented by:

$$\sum_{k=0}^{n-1} ar^k = a\frac{1-r^n}{1-r} \tag{6.1}$$

Where,

r Expansion or Contraction Coefficient,

n Number of Grid Lines,

a Series coefficient calculated from total length of 1, and,

$\sum_{k=0}^{n-1} ar^k$ is the length of the line segment to be divided.

The amount of extension or contraction depends on the value of r in Eqn.(6.1) where $0 < r < 1$ is employed for contracting grid, and $r > 1$ for expanding grid as shown in Fig. (6.11-6.12, 6.13),.

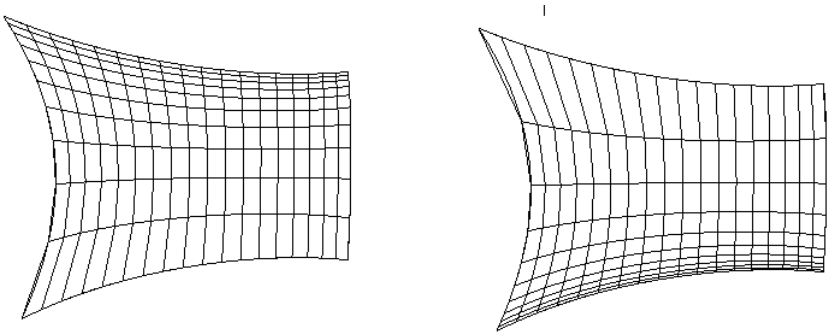


Figure 6.11: Top and Bottom Single Stretched Function

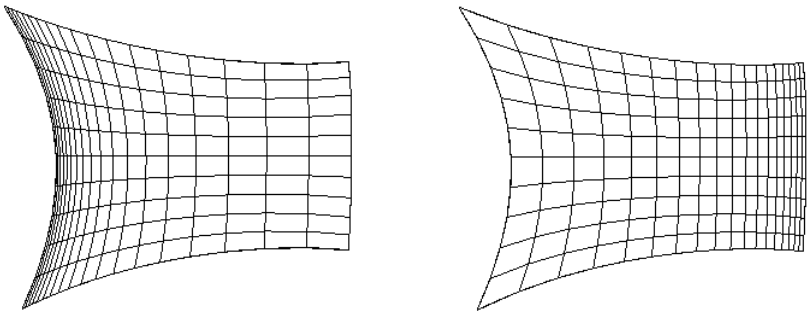
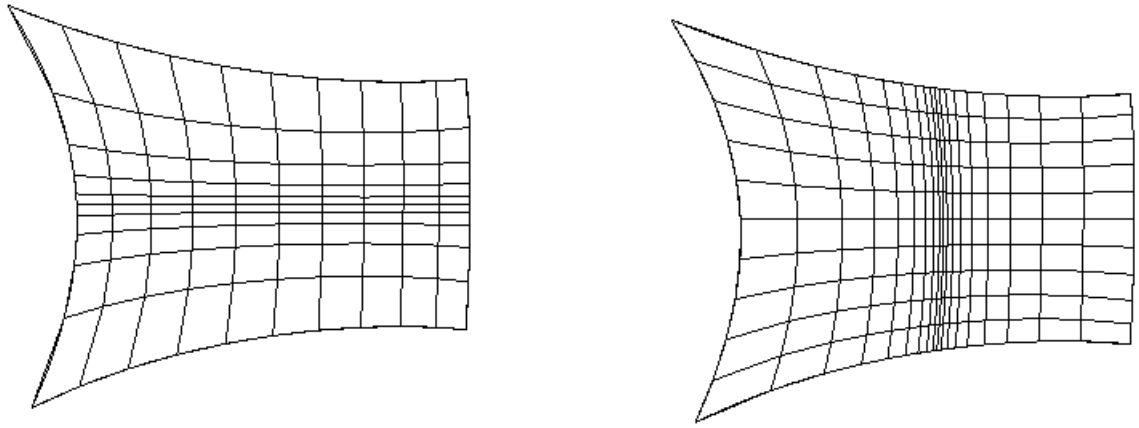


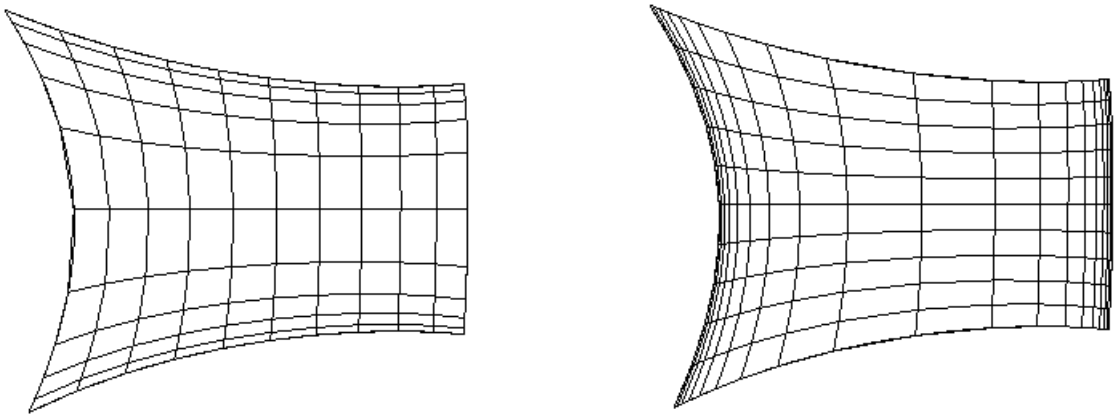
Figure 6.12: Left and Right Single Stretched Function

In case of single concentration stretching function, the values from 0 to 1, or above 1 will concentrate the lines on specific edge as shown in Fig. (6.11), and Fig. (6.12).

Fig. (6.13) Shows double stretching and contracting grids where Eqn. (6.1) is used with the aid of the current value of r.



(a) Double Stretched gridding in the middle



(b) Double stretched gridding in the edges

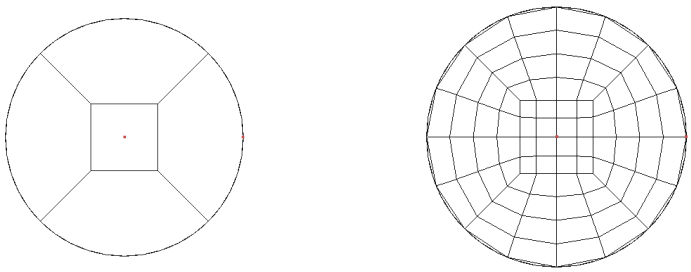
Figure 6.13: Double stretched function on quadratic cell

The subprogram written to accomodate grid concentration is listed in App. (C.2.2).

6.2.6 Circle Gridding

To avoid problems associated with poles, circle gridding requires that the center be surrounded by a quadrilateral shape. Accordingly the circle is divided into an inner zone featured in the square surrounding the center, and an outer zone divided into four regions as shown in Fig. (6.14). The inner square is then gridded by a suitable number of grid lines as explained earlier. The outer four regions are also gridded as explained above in Section (6.2.2) to yield the grid shown in Fig. (6.14b). The circumference of the circle is approximated by straight lines as shown. The more the number of cells the smoother the resulting circle circumference. The number of cells in the outer regions is connected with the number in the inner zone and, thus increasing the number of cells in the outer regions for smoother

circle circumference will inevitably increase the number of cells in the inner zone. This may not always be wise strategy in terms of computational overheads.

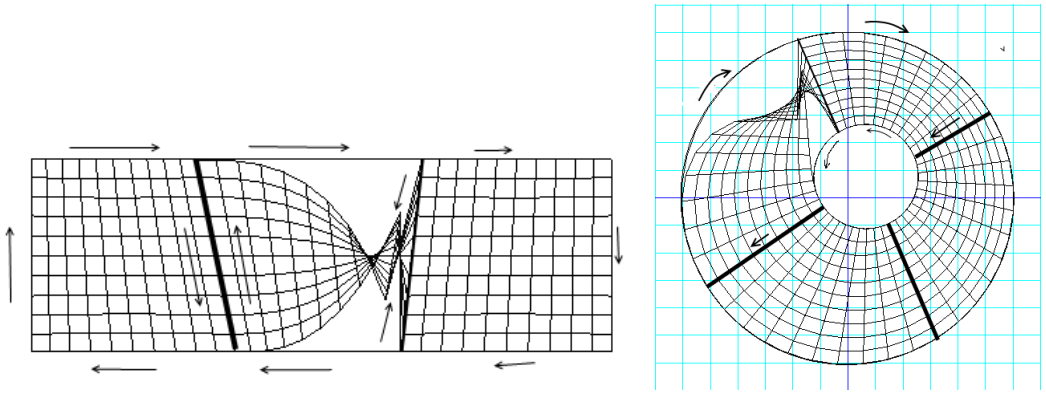


(a) Sub Domains (b) Sub Domains Gridded

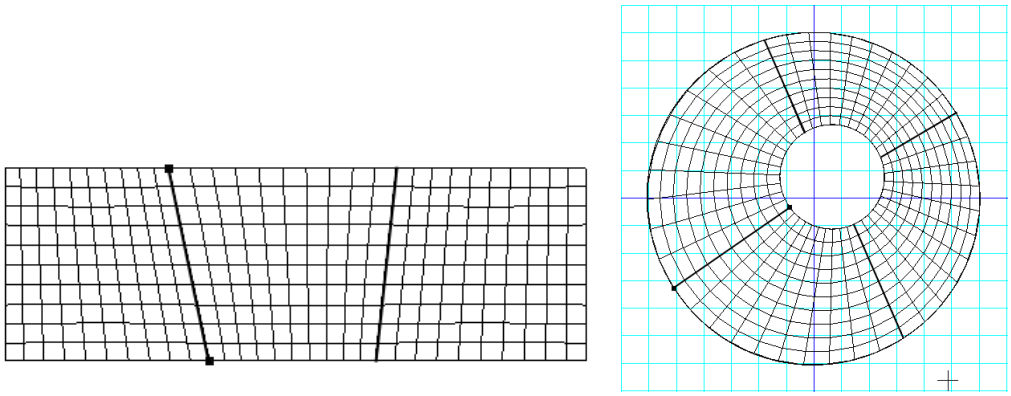
Figure 6.14: Circle Gridding

6.2.7 Edge Senses and Coding

When gridding the quadratic shapes, there are important aspects of the orientation in the edges of the bezier curves defining the shape. These orientations (edge senses) if not handled correctly, would result in a deformed grid shown in Fig. (6.15a).



(a) Deformation in cell due to sense difference between edges



(b) No deformation after applying sense algorithm

Figure 6.15: The effect of sense deformation

The correct gridding can be seen in Fig. (6.15b), which applies the sense algorithm for

correcting the generated lines based on the edges senses.

To be able to define the edges senses, a certain coding has been applied to the quadratic cell to distinguish between the different orientation of the bezier curve.

This was coded as [C, First {Curve Index Point Index}, Second {Curve Index Point Index}]. For example when C0310 is true, this means that bezier curve at index 0 and the last control point at index 3 are the same point on the bezier curve of index 1 and first point on this cuve at index 0, and the orientation of the cell of these coding appears as seen in Fig. (6.16). The coding of these senses is listed in App. (C.2.3).

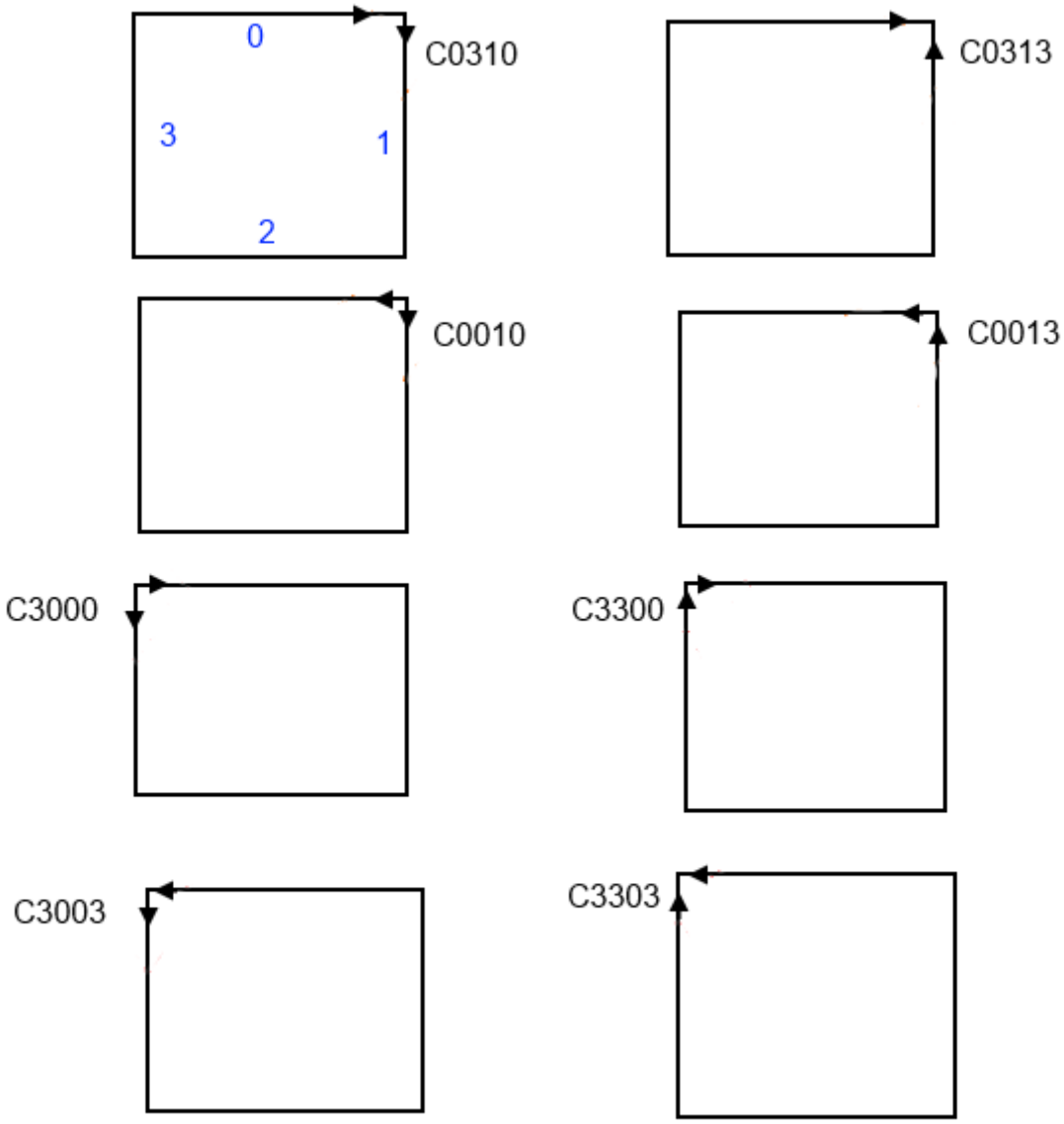


Figure 6.16: Quadratic Cell Senses of Upper Points

6.3 CONNECTIVITY BETWEEN QUADRILATERAL SHAPES

The grid in the quadrilateral shape has four other quadrilateral neighbours at each side. These neighbours are sharing their points with this quadrilateral shape. The shared points on the edges of the quadrilateral shapes are important to acheive the continuity of cell elements between blocks . The discovery of neighbours is the process of finding the connectivity nodes (points) between the quadrilateral shapes.

6.3.1 Discovery of Neighbours

Qudrilateral shapes cover the flow domain and share their sides between each other. The process of finding the neighbours are carried out by iterating over the shape sides, testing if it is shared to another quadrilateral shape or not, by examining its first and last points.

It is worth mentioning that index of the side on the quadrilateral shape can be different on the target shape, and it requires a special attention in the gridding between the adjacent quadrilateral shapes. The code listing that determine the neighborhood quadrilateral shapes is listed in App. (C.2.4).

6.3.2 Grouping

Grouping of quadrilateral shapes is the next process after discovering the whole connected neighbours. Grouping is a logical process that tells that those neighbour shapes are indeed acting as a one unit in transformation.

The grouped quadrilateral shapes always refer to a complete 2D gridded flow domain, which permits the user to use this grid in further calculations (for more information of the grouping rechnique the reader may refer to the App. (C.2.5)). The grouping also servers as the start point of more higher 3D operations like revolving and extrusion to the grouped shapes.

6.3.3 Gridding between Neighbours

Special attention of grouped shapes should be taken during gridding of any sub quadratic shape. Continuity of grid lines that passes through the shapes and its neighbours should be the same number when viewed from any side.

Fig. (6.17) shows five connected shapes having 6x6 cells. when changing the horizontal number of cells in Shape C, the number of horizontal cells in Shapes B, and D should be adapted to the same number of cells, as shown in Fig. (6.18).

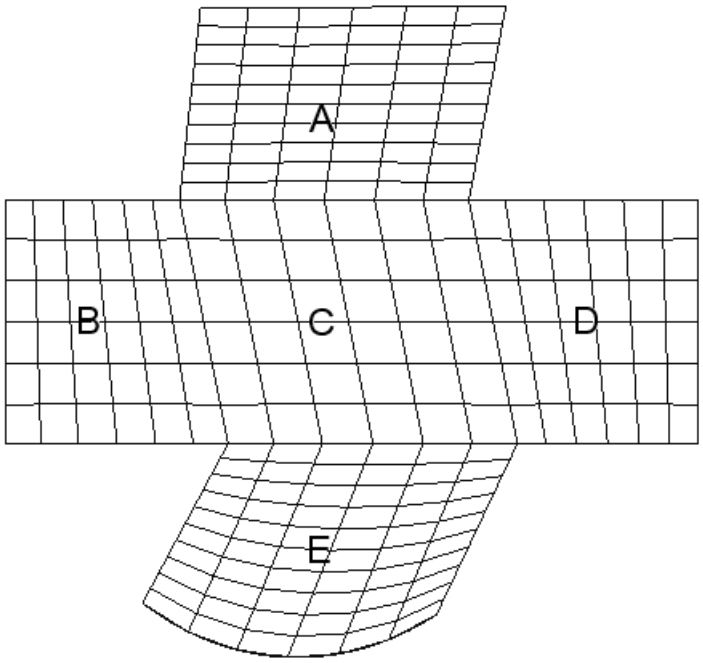


Figure 6.17: Five Connected shapes with 6x6 cells

The same adaptation occur also in the case of changing the vertical number of cells in any sub quadratic shape as shown in Fig. (6.19).

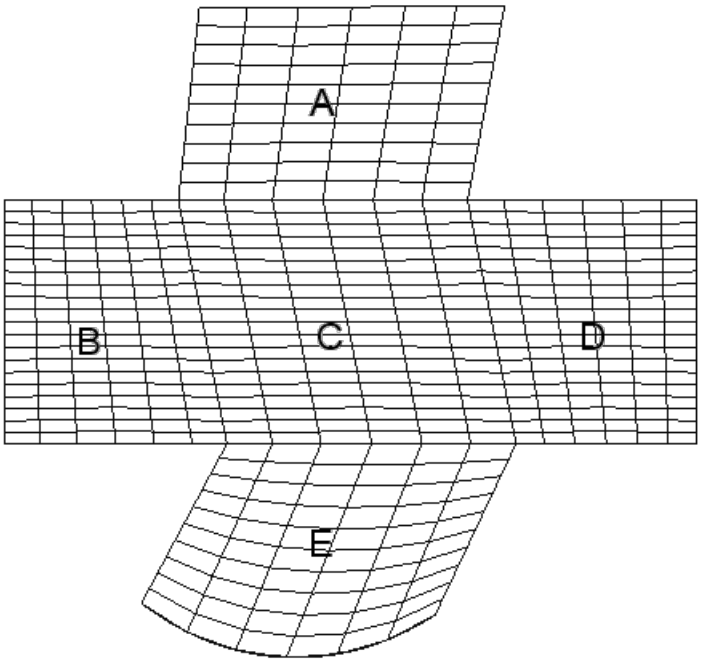


Figure 6.18: Five Connected Shapes. B, C, and D shapes gridded as 6x20 cells

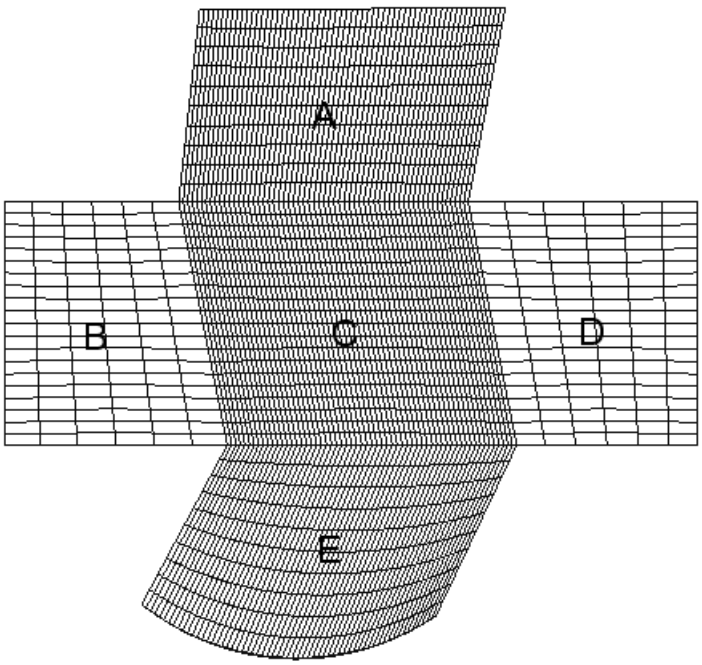


Figure 6.19: Five Connected Shapes A,C, and E Shapes are vertically densed gridded.

6.4 THREE DIMENSIONAL GRIDS

The creation of connected two dimensional grids can be extended into 3D grids by applying two primary operations, extrusion, and revloving. The two operations will convert the quadrilateral mesh elements into hexahedral mesh elements (hexahedron element). Hexahedral elements are connected from its points and sharing their side faces with the other elements. The face is either an internal face with owner and neighbour cell, or external face (face that can have boundary condition).

6.4.1 Extrusion Operation

Extrusion operation extend the mesh nodes positions with the aid of the normal vector direction into new nodes with an arbitary length value selected by the user. Segmentation is applied to the arbitary length to produce layers of the grid nodes, each layer of nodes connects to the previous and next layer of nodes to form a 3D hexahedral cells from the 2D quadrilateral grid.

$\Delta\mathbf{v}$ vector for the position node with the arbitary length Δl, should be calculated as follow

$$\Delta x = \Delta l \times \cos\theta_x = \Delta l \times \frac{\mathbf{v}\cdot\mathbf{i}}{\|\mathbf{v}\|\|\mathbf{i}\|} \tag{6.2}$$

$$\Delta y = \Delta l \times \cos\theta_y = \Delta l \times \frac{\mathbf{v} \cdot \mathbf{j}}{\|\mathbf{v}\|\|\mathbf{j}\|} \tag{6.3}$$

$$\Delta z = \Delta l \times \cos\theta_z = \Delta l \times \frac{\mathbf{v} \cdot \mathbf{k}}{\|\mathbf{v}\|\|\mathbf{k}\|} \tag{6.4}$$

$$\Delta \mathbf{v} = \Delta x\,\mathbf{i} + \Delta y\,\mathbf{j} + \Delta z\,\mathbf{k} \tag{6.5}$$

The new grid node position is calculated

$$\mathbf{v}_{new} = \mathbf{v}_{old} + \Delta \mathbf{v}$$

This algorithm is applied to all the nodes in the 2D quadrilateral grid.

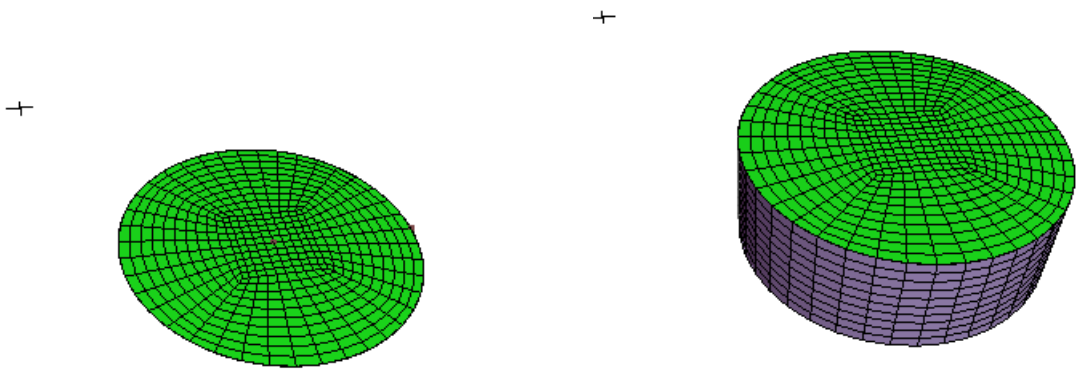


Figure 6.20: Extrusion Operation

6.4.2 Revolve Operation

Revolve operation revolves the mesh nodes positions with the aid of a rotational axis into a new nodes with an arbitary angle value selected by the user. Segmentation is applied to the arbitary angle to produce layers of the grid nodes, each layer of nodes connects to the previous and next layer of nodes to form a 3D hexahedral cells from the 2D quadrilateral grid. Quaternions algera are used in applying the rotation as follow:

1. Initial quaternion is calculated based on the axis and zero angle
2. Final quaternion is calculated based on the axis and final angle
3. Total quaternion is calculated by multiplying initial and final quaternions.

4. Total rotation matrix is obtained from the total quaternion.

5. The rotation matrix is multiplied with the node coordinates to obtain the new coordinates after rotation.

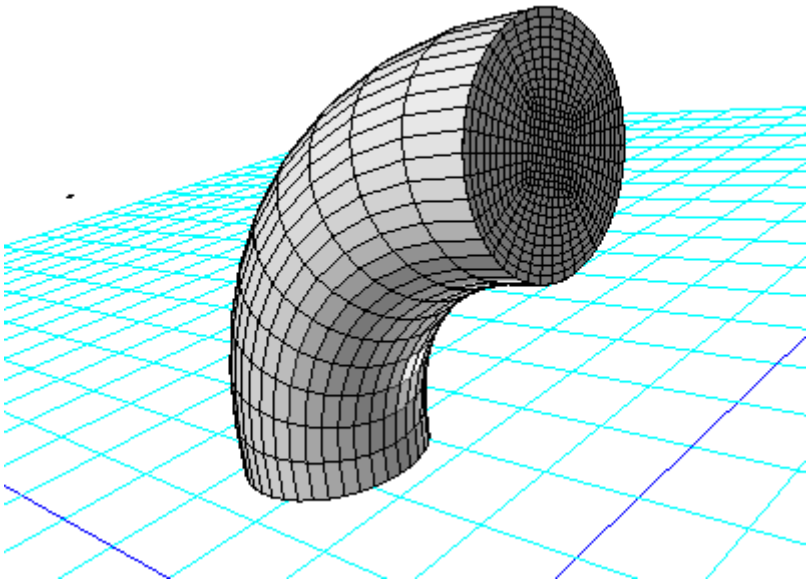


Figure 6.21: Revolve Operation around y-axis of a gridded circle

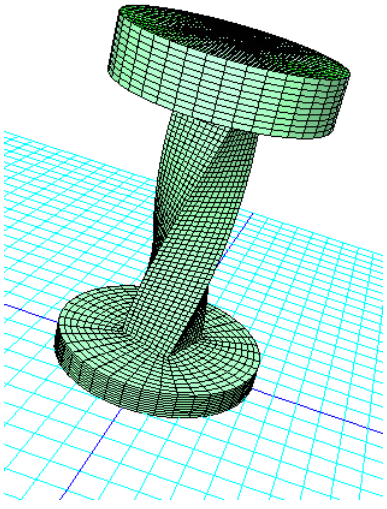


Figure 6.22: Extrusion with Twising

6.4.3 Auxilary Operations

Two auxilary operations can be accumulated to the extrude, and revolve basic operations. Twisting, and scaling operation are used to control the resultant 3D geometry.

6.4.3.1 Twisting Operation

Twisting operation revolve the mesh nodes around the normal axis Fig. (6.22).

6.4.3.2 Scaling Operation

Scaling operation expand or contract the layer nodes to the normal axis. Fig. (6.23).

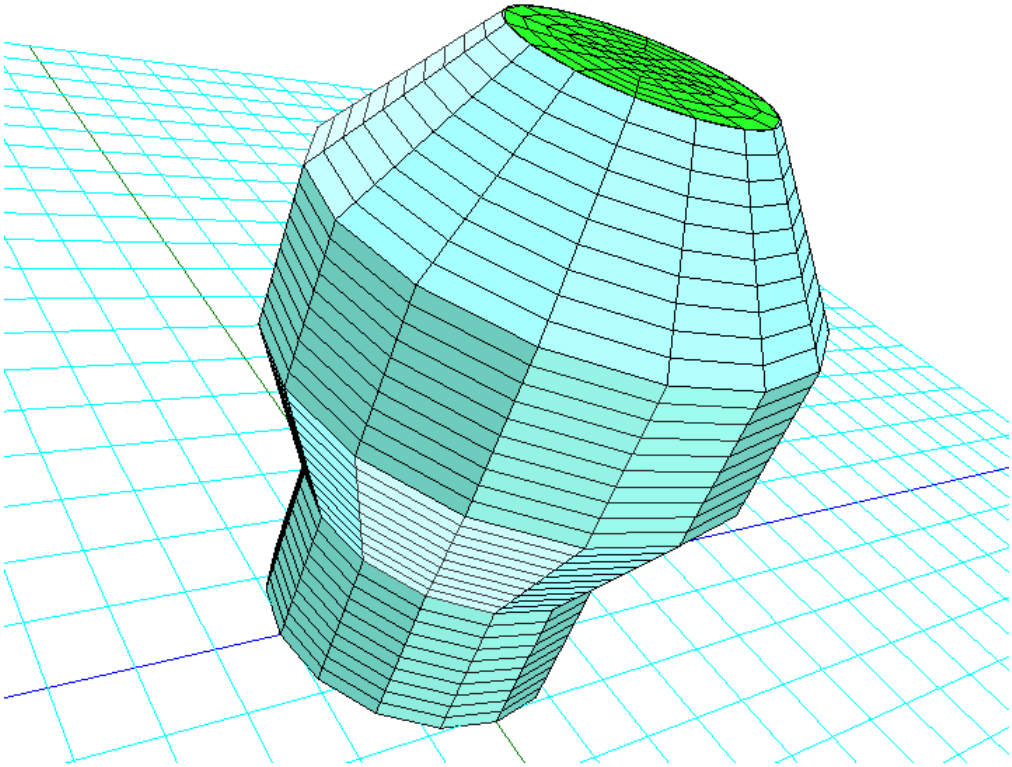


Figure 6.23: Geometry Double Scaling

6.4.4 Axisymmetric Operation

Axisymmetric operation is done on the 2D grids that one if its sides completely lies on the x-axis Fig. (6.24). These points are considered singularity points. The grids of this types can be rotated as follow:

1. Grid rows to be divided into lower and upper parts with 6:4 ratio respectively.
2. Singular point is at $y = 0$ and will not be rotated.
3. Ceiling point of the lower part rotated two times with half and complete angle respectively, forming two lines.
4. Ceiling point of the upper part rotated with the required amount of segments to form the upper ceiling nodes.
5. The grid lines are being connected from the lower part to the upper part forming one grid face for one grid column. Fig. (6.25).
6. The operation is repeated for all columns of the two dimensional grids.

7. The generated faces are then connected to form hexahedreal grid. Fig. (6.26).

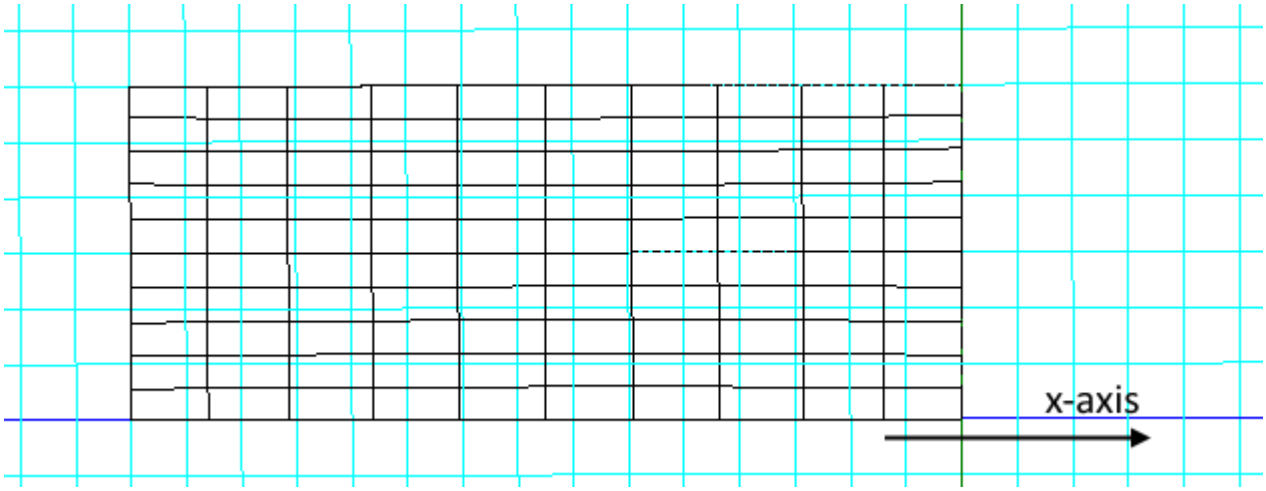


Figure 6.24: Two Dimensional Grid on x-axis

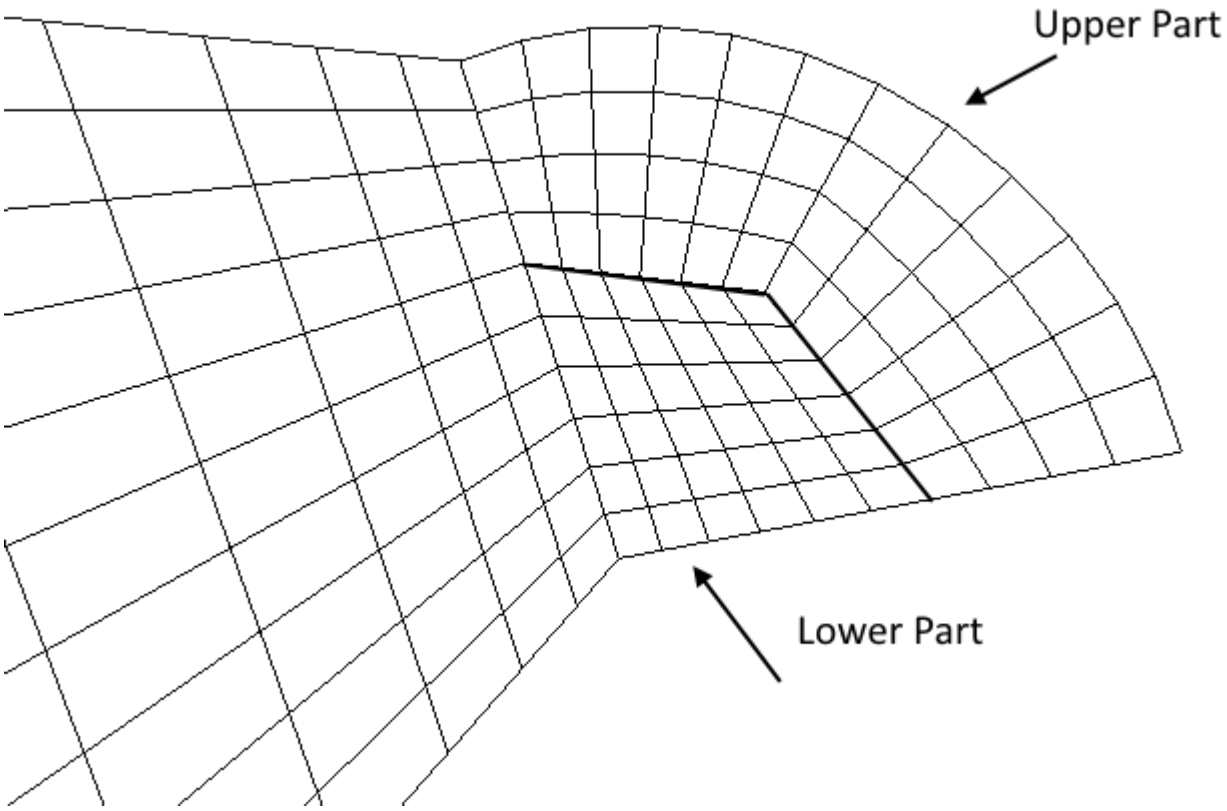


Figure 6.25: First Column Grid Face

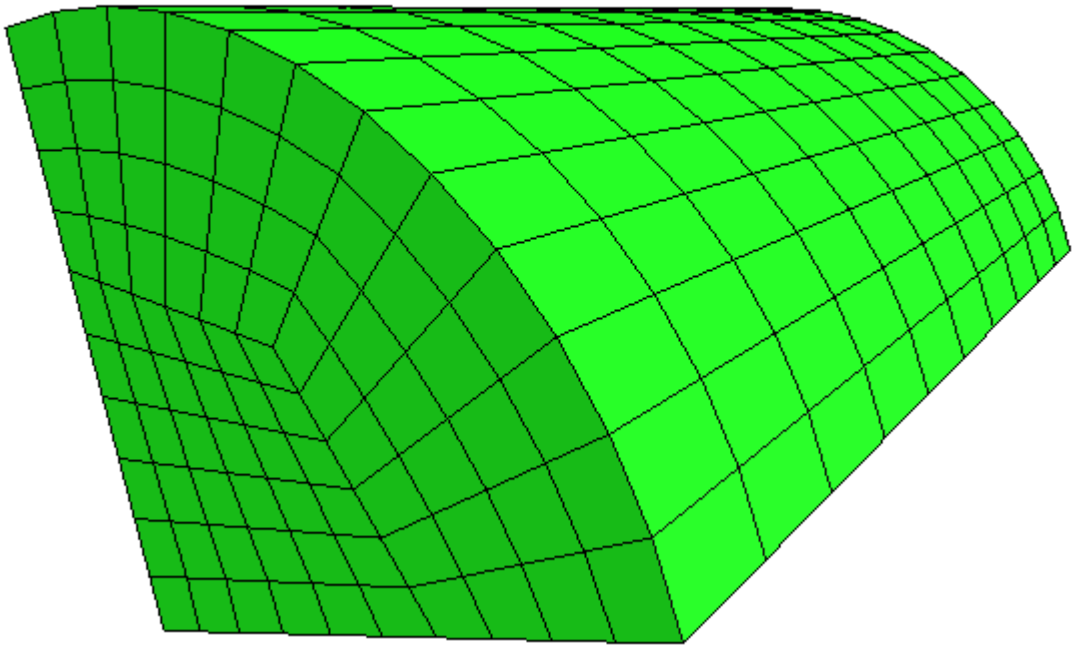


Figure 6.26: Axisymmetric Grid

6.5 CONCLUSIONS

In this chapter, the generation of two-dimensional grid between four curves has been discussed and illustrated using the homotopy between the curves. The four curves form a quadrilateral sub-domain. The sub-domain needs to discover the grid points of its neighbouring sub-domains and connect its points with them. Neighbour discovery and grouping of sub-domains have been discussed throughout the chapter. Finally, the three dimensional grid creation algorithms and operations have been illustrated and discussed.

However, the final grid distribution and intensity depend almost entirely on the experience and his attempt to acheive high quality grid from the computational point of view. No standard quality measures are known for grid generation software, the quality of the grid is judged by the performance of the solver of the governing equations, i.e., accuracy, rate of convegence, and stability.

CHAPTER VII

PROGRAM STRUCTURE AND CASE STUDIES

7.1 INTRODUCTION

In this chapter, the program features are presented and discussed in detail. This aims at explaining and demonstrating the various functionalities of the software. Case studies have been carefully selected to illustrate the package performance and agility in drawing hexahedral grids. No specific dimensions were attached to the geometries presented since the objective of the excersice is to illustrate the sequence with which the grid could be generated in different cases.

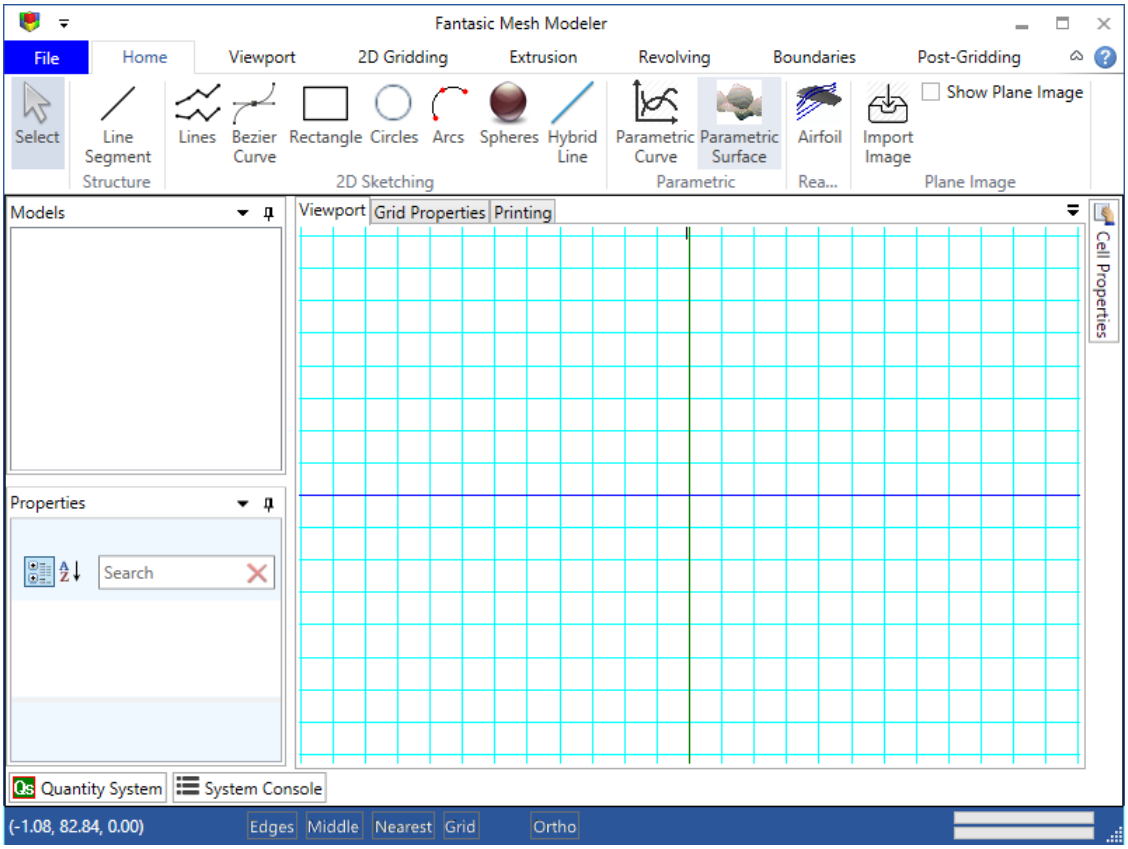


Figure 7.1: Software Main Screen

This software program adheres to the block structured technique. The technique requires that the domain be divided into a number of sub-domains contained within four curves each. These sub-domains in essence, serve as the starting point of the gridding operation which

requires the user to select the four points that enclose each sub-domain. When the user completes the selection of four points, a grid appears in that block and connects itself with the neighbour block grid if existed from a previous operation.

Functions of saving the work done into external files and retrieving it along with completed grids are of great value to the program user. The program has the ability to export the final grid into two commercial package formats namely Fluent Ansys®, and Pro Star®. This can be extended to any number of commercial packages if needed.

The chapter is divided into two main sections, the first describes the user interface menus and icons of the most important parts that serve as the key concepts to the software usage. The second section contains a number of case studies carefully selected to demonstrate the capabilities of the developed software program.

Case Studies have been split into two categories. First category include some basic shapes involving simple operations needed for grid generation. Second category contains more complex shapes that require a more careful handling from the user.

7.1.1 Program Structure and Layout

Immediately after opening the program, the main screen Fig. (7.1) appears. The program main screen is divided into disctinct areas. These areas are being discussed on the following sections.

7.1.2 Ribbon Menu Bar

This is the Ribbon Menu Bar. This menu adheres to the latest microsoft guide lines of UI (User Interface) experience by employing the Ribbon Menu concept that can be found in the latest editions of Microsoft Office® and built in programs of Windows® operating system.

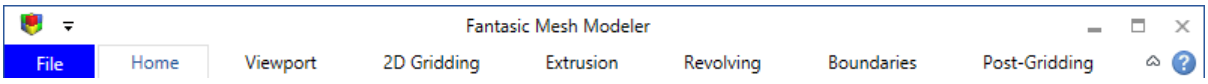


Figure 7.2: Ribbon Menu Bar

The menu Fig. (7.2) contains the following titles:

File File Operations Menu

Home Main drawing functionalities

Viewport Functionality related to the appearence of the shapes and grid.

2D Gridding Functions of creating two dimensional grids.

Extrusion Functions of extruding the two dimensional shapes into three dimensional objects.

Revolving Functions of creating three dimensional objects by rotation of two dimensional shapes around certain axes.

Boundaries Functions of specifying and selecting the boundary conditions of the final grid.

Post-Gridding Special functions that only applied to the final three dimensional grids.

7.1.3 Operations Area

This area Fig. (7.3) is the one that most of the work is done into it. This area supports automatic docking of its windows for more flexibility in adjusting the area based on the user prereferences.

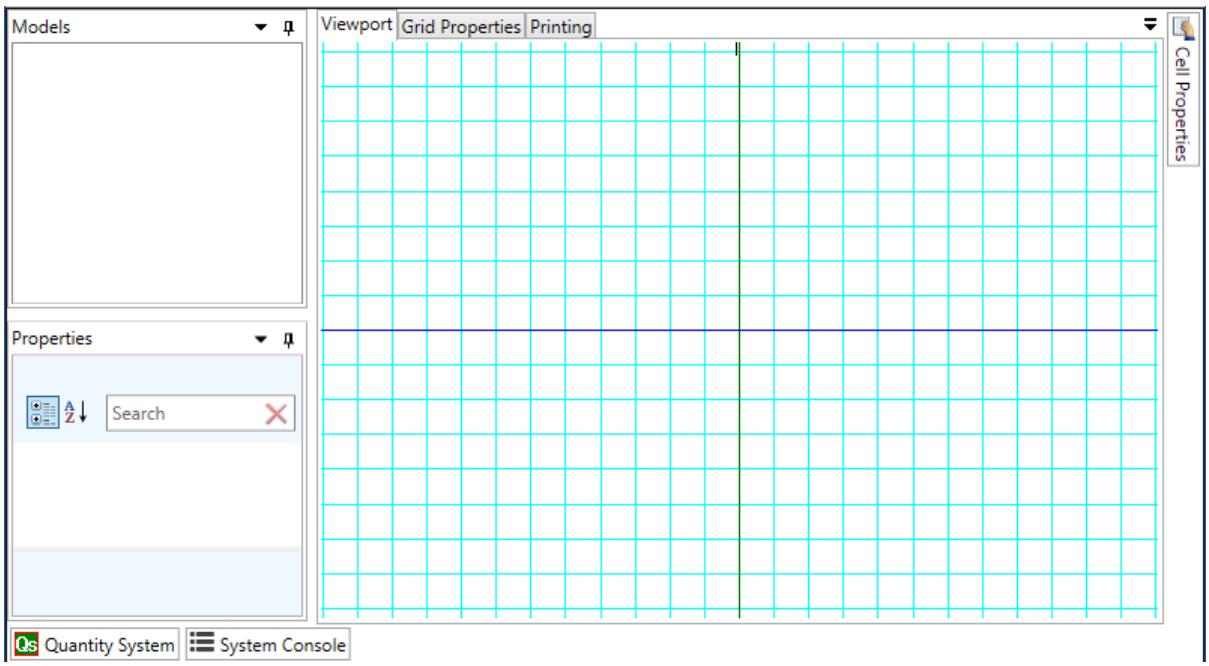


Figure 7.3: Operation Area

Fig. (7.4) shows the complete opened tool windows. The **System Console** window is a special window for showing messages to the user during his work with the program in addition to any warnings.

Quantity System window is an expression evaluator window that can make quick calculations through writing mathematical experssions like 4*sin(30<deg>). The window also can access the viewport models by their names and can call certain functions that aren't available on the user interface. However rest of the tool windows are discussed in general in the next sub sections.

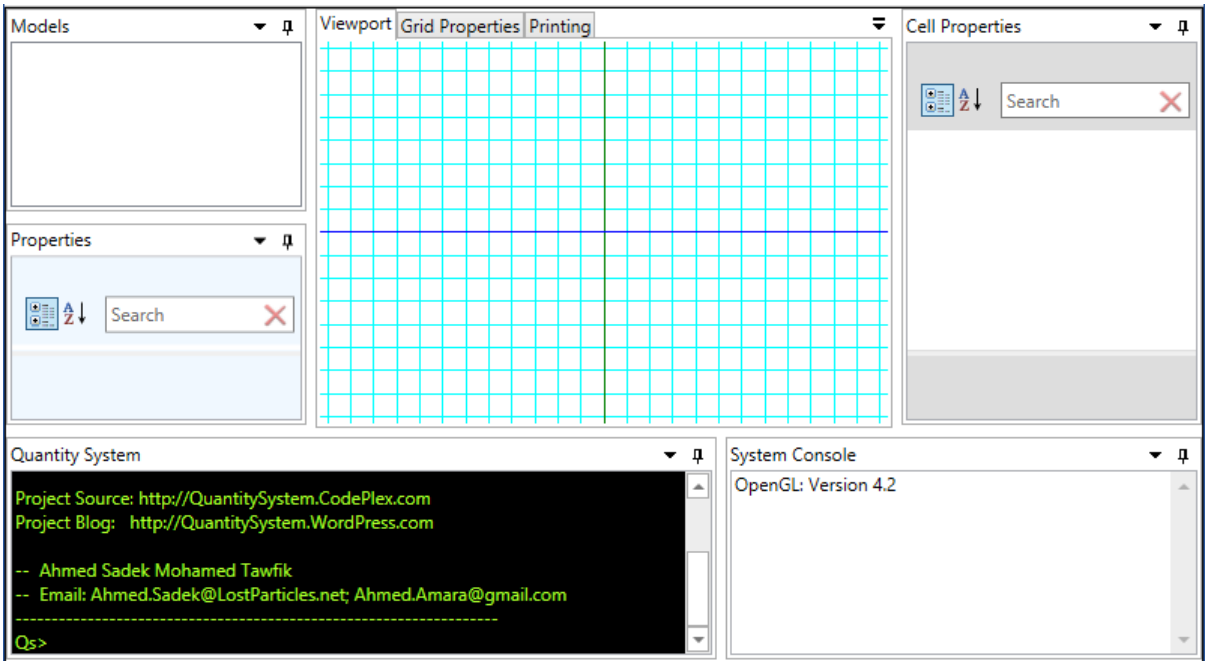


Figure 7.4: Opeartion Full Area Sides

7.1.3.1 Viewport and Models Windows

This is the window Fig.(7.5) that any drawing take place on. It supports the user interaction with mouse during drawing and viewing the shapes and 3D objects. In addition the Models windowlist all models that appears in the viewport drawing area. Clicking on model in this window will select this model to view its properties in the properties window.

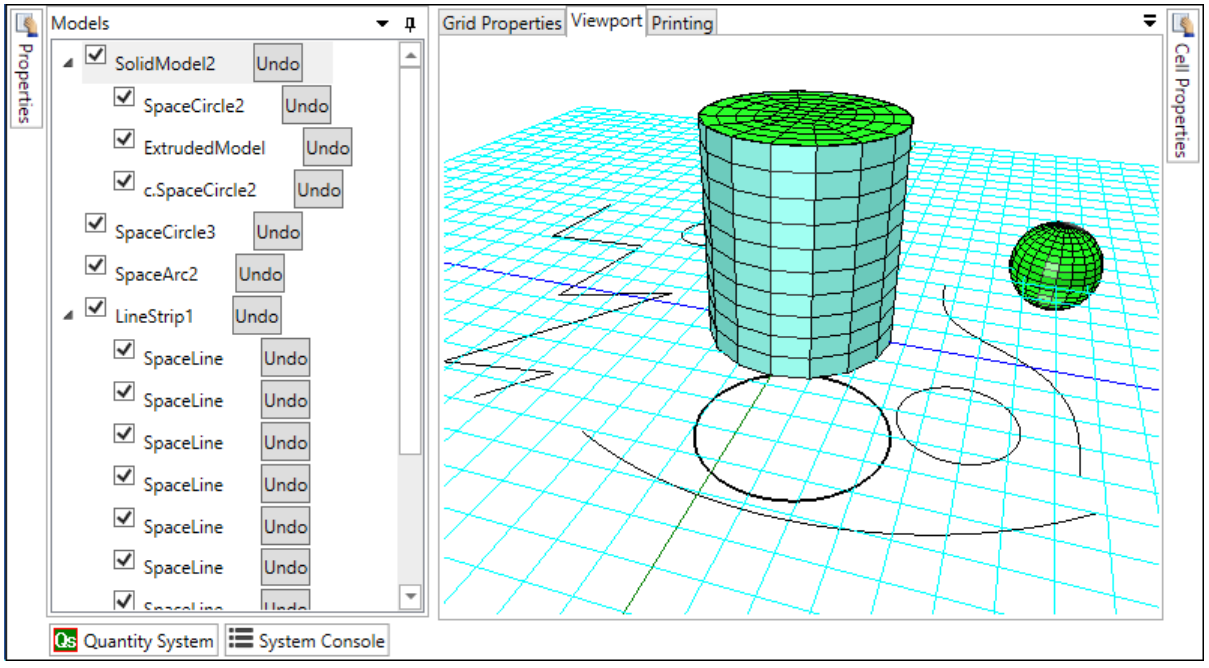


Figure 7.5: Viewport and Models windows

7.1.3.2 Properties Window

This window display a list of the available properties of the currently selected model. Fig. (7.6) shows the properties of MorphedQuad model. This object for example has its M and N Properties set to 10. This window update itself whenever the user selects a model from the viewport window or the models window.

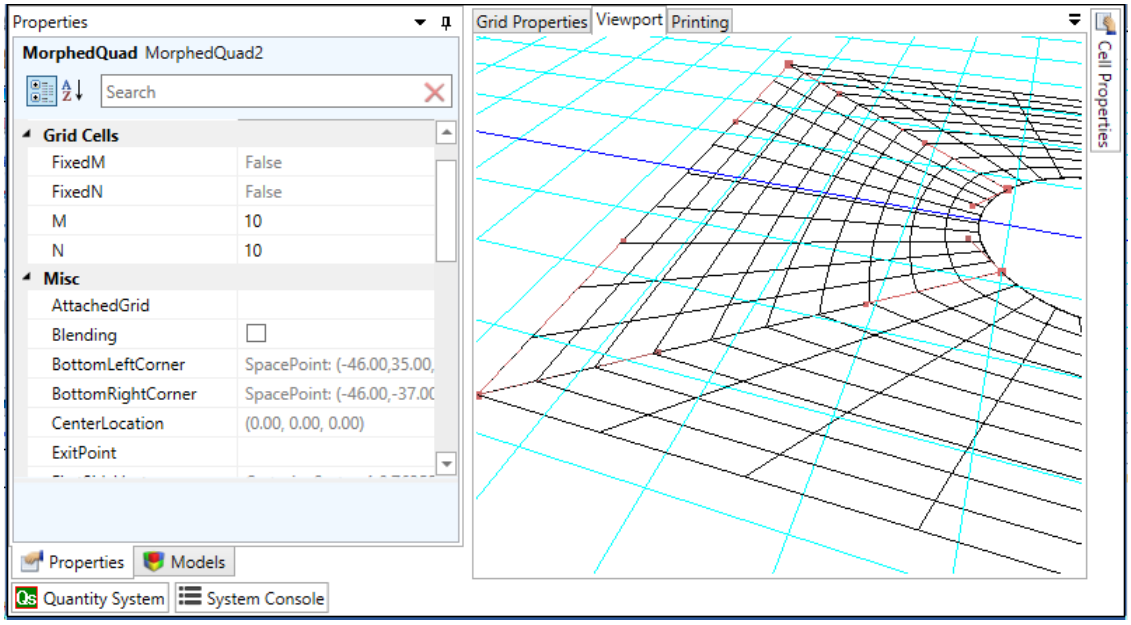


Figure 7.6: Model Properties Window

7.1.3.3 Cell Properties Window

This window Fig.(7.7) display the properties of selected face element on the final 3D grid to view its properties.

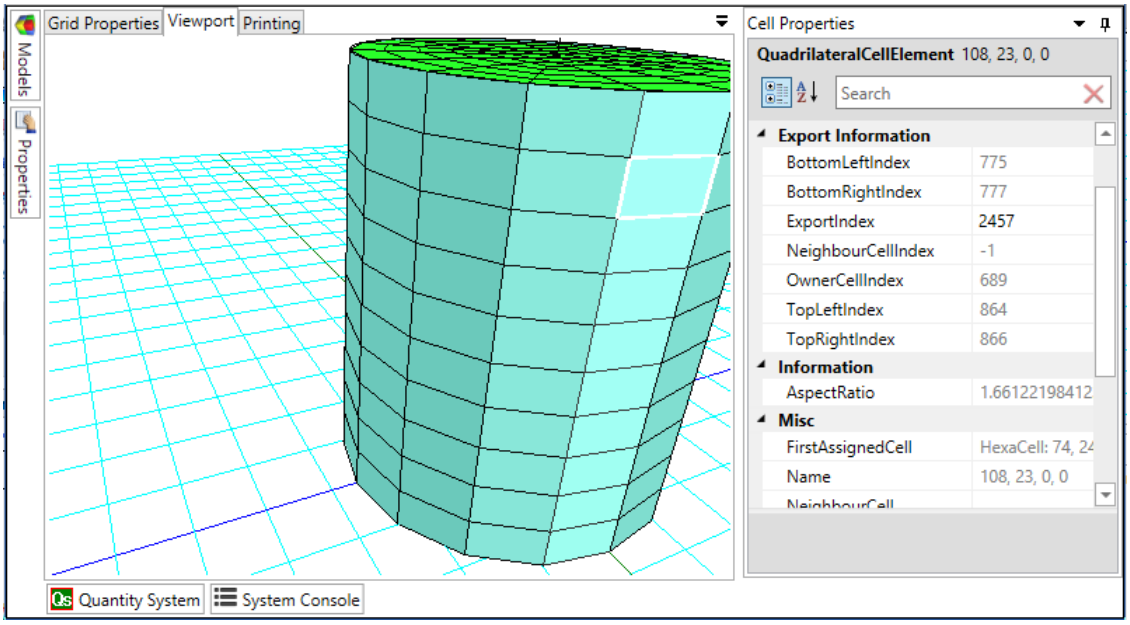


Figure 7.7: Cell Properties Window

7.1.3.4 Grid Properties

This window Fig. (7.8) display the grid data stored internally in the program. These data include number of cells, faces, and nodes of the selected grid, in addition to the ability to export the data to the various programs packages. The tabular view can be copied and pasted into excel files also. The window display a detailed information about the grid points, faces, and cells. It also point out the internal and boundary faces.

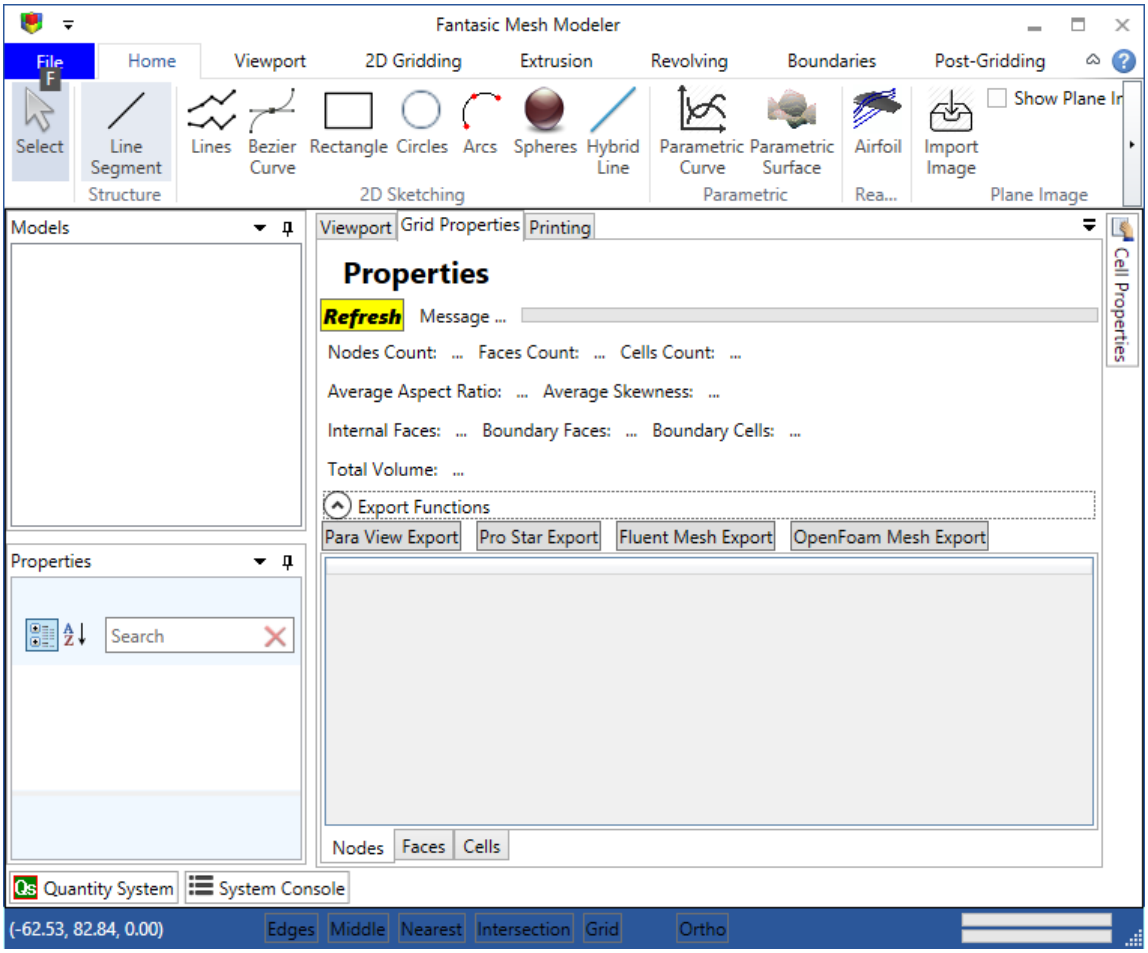


Figure 7.8: Grid Properties Window

7.1.4 Task Bar Area

This area Fig. (7.9) contains the important information about the current operation progress and the drawing snapping options of points while drawing. The program features two progress bars that shows main operations in addition to sub operations in case of such ones.

The left part of task bar shows the current coordinate projection on the viewport for helping the user to know the current mouse location. The snapping options of points are very important during working with shapes for creating two dimensional grids and connecting shapes together. Snapping is the process of highlighting certain points on the shapes while mouse is moving over them. These points serve as connection nodes between shapes, and grids. Following are a breif description for each snapping option and its usage:

Edges Highlight the start, and end points of lines, bezier curves, and circular arcs, in addition to the corner points of Morphed Quad[1] objects.

Middle Highlight the middle point of supported shapes.

Nearest Highlight the nearest point to the mouse that lie on the shape.

Grid Highlight the nodes found on the 3D grid visible surfaces.

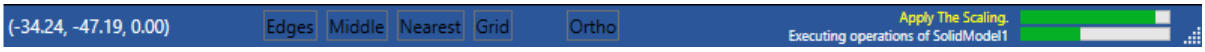


Figure 7.9: Task Bar Area

7.2 PROGRAM MENUS

7.2.1 File Menu

File menu as shown in Fig. (7.10) contains the basic operations file management. New, Open, and Save commands are for clearing, retrieving files, and saving them.

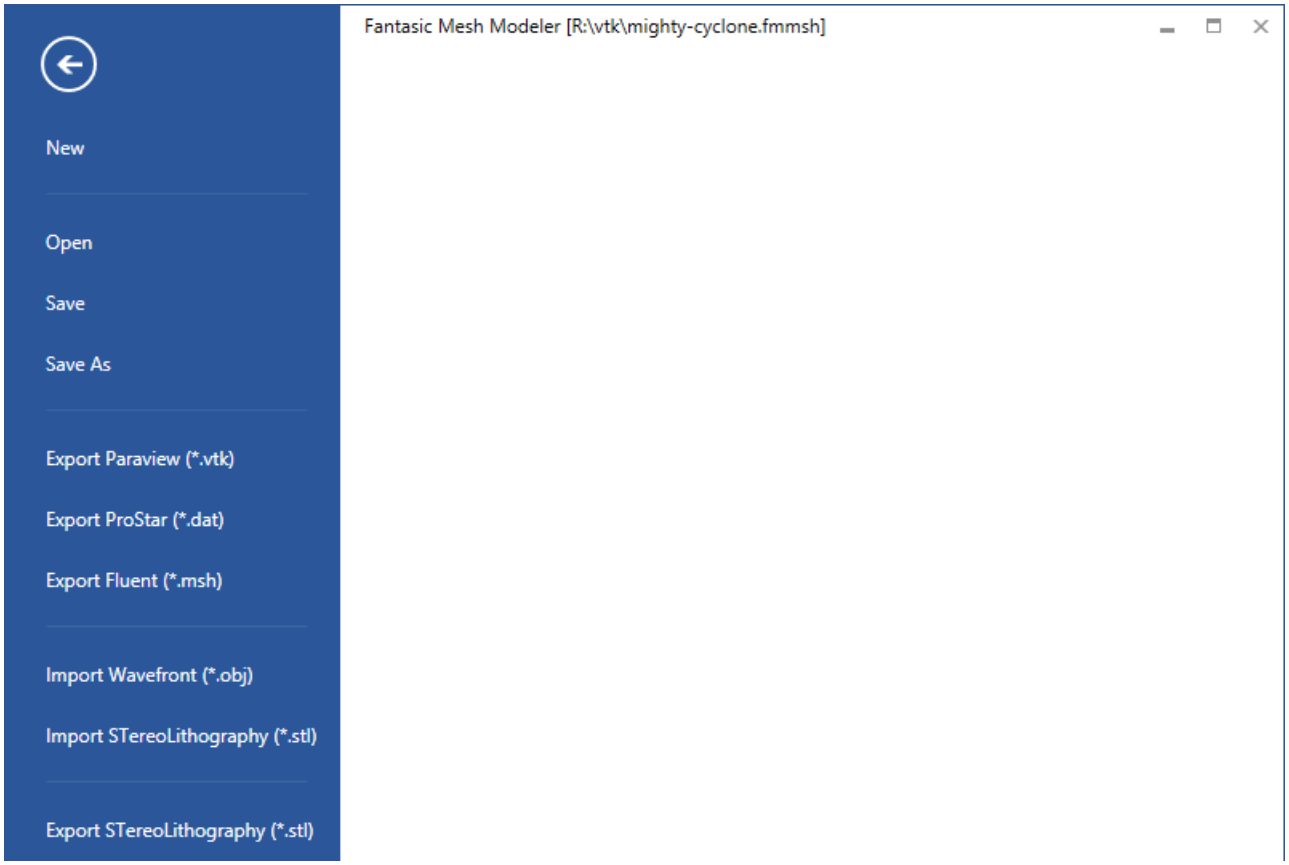


Figure 7.10: File Menu

There are also the Export functionalities into ParaView VTK format, Pro Star format, and finally Ansys Fluent format. The program also has the ability to import and view

[1]Morphed Quad (Morphe Quadrilateral) are the foundation shape for building 2D Grids.

common 3d file formats of STereo Lithography® (stl) and Wavefront® (obj) files[2]. Solid Models that contains the 3D grids can export its surface into the STereo Lithogtaphy® (stl) format as shown in the menu.

7.2.2 Home Menu

Home Menu is the primary menu of drawing sketches on the base grid plane.

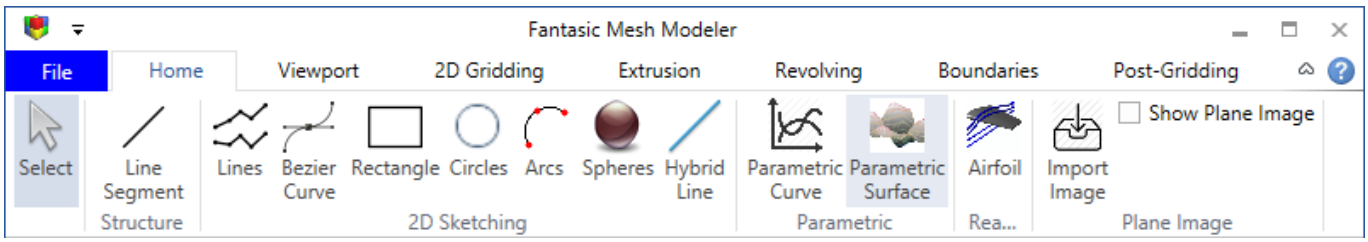


Figure 7.11: Home Menu

7.2.2.1 2D Sketching

Line Segmets Draws line segments with two points.

Lines Strips Draw line strips (many lines connected in their first and end points with other lines).

Bezier Curve Draw a bezier curve with 4 control points.

Circle Draw circles with center point and radius point.

Sphere Draw a sphere with center and radius point.

Hybrid Line A special modeling that draws line strips in addition to bezier curves in the same object.

7.2.2.2 Parametric Sketching

Parametric Curve This button is used to enter an arbitary curve equation $F(t) = x(t)\mathbf{i} + y(t)\mathbf{j} + z(t)\mathbf{k}$ where $t = [0, 1]$

Parametric Surface This button is used to enter arbitary surface equation is on the form $F(u, v) = x(u, v), y(u, v), z(u, v)$ where $u, v = [0, 1]$

7.2.2.3 Air Foil

This button allows the access of massive database of common airfoil profiles Fig. (7.12) that can be selected to appear on the viewport.

[2]The program can only view these files, however there is no support of editing them or creating automatic meshes.

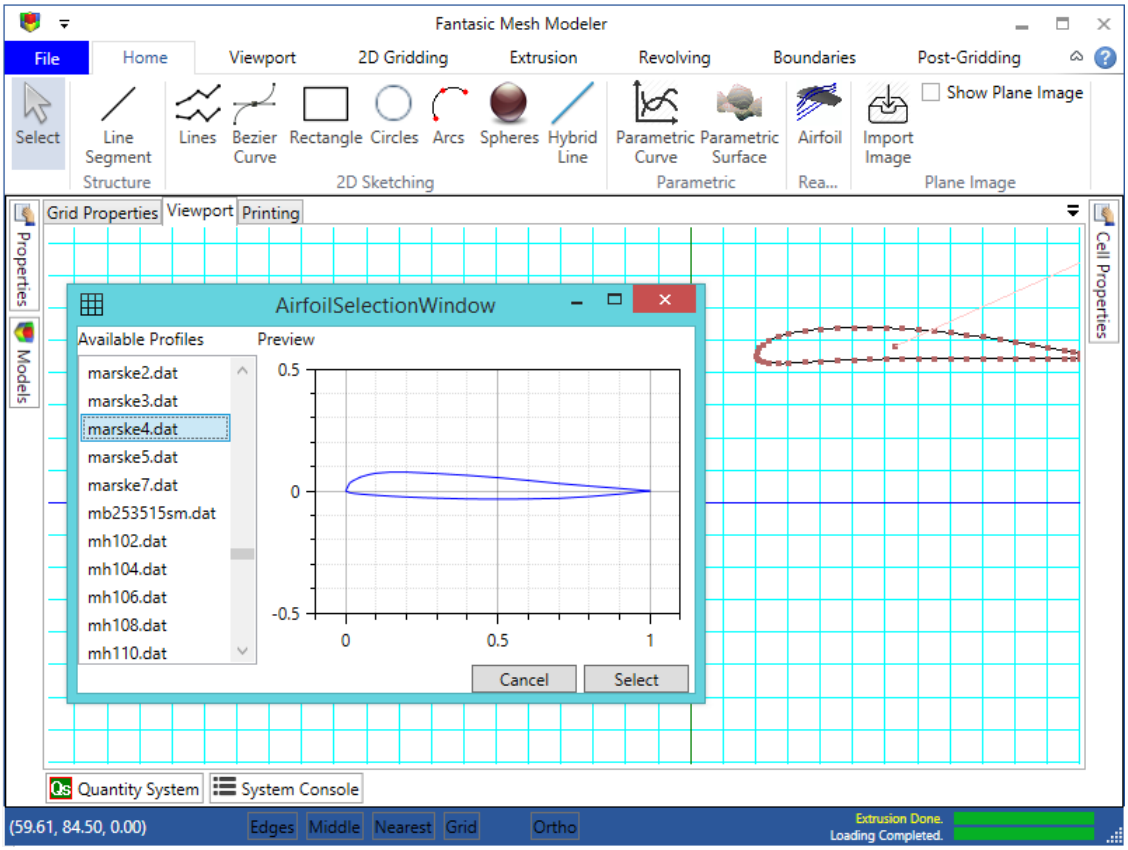


Figure 7.12: Air Foil Profiles

7.2.2.4 Plane Image

The program has the ability to load an image from the file system to display it on the drawing grid Fig. (7.13). The use in return can draw over this imported image to form the required sketch and grid.

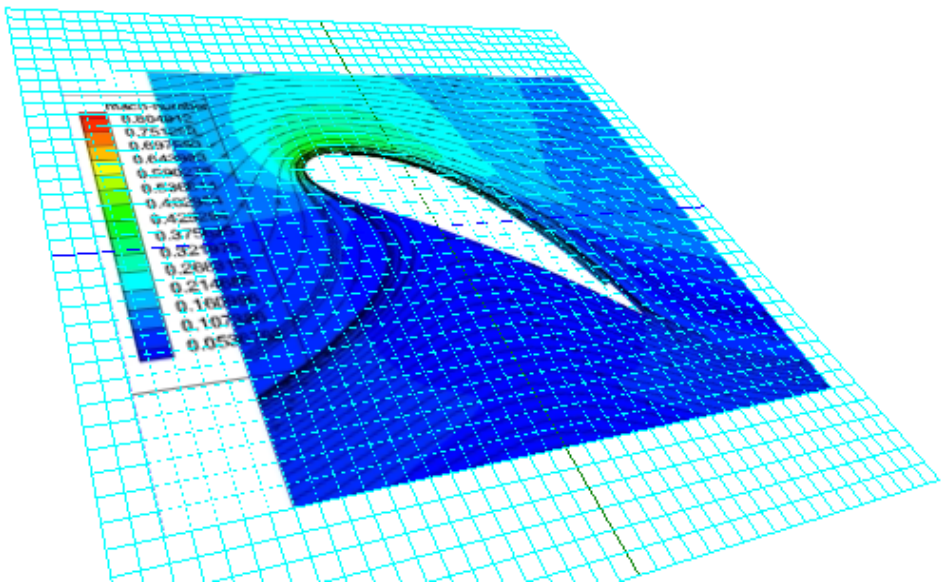


Figure 7.13: Drawing Grid Plane Image

7.2.3 Viewport Menu

Viewport is the area in the program where sketching and gridding take place. This area contains also the drawing plane which receive mouse movements and clicks.

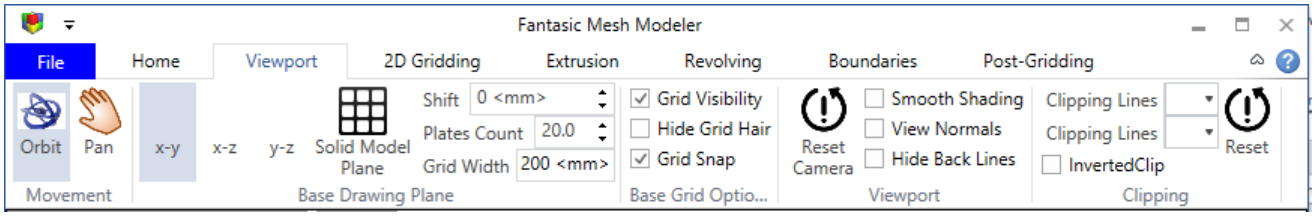


Figure 7.14: Viewport Menu

7.2.3.1 Viewport Movement

Orbit Rotating of the viewport around it self[3]. The same functionality can be achieved by holding SHIFT while draggin the mouse around.

Pan Translating of the whole viewport to the right, left, up, or down. The same functionality can be achieved by holding SHIFT+CTRL while dragging the mouse.

7.2.3.2 Base Drawing Plane

The base drawing plane is the visual representation of the current drawing plane. The drawing plane can be shifted up or down by changing the Shift field value. The drawing plane can be set into x-y, x-z, and y-z planes, this allows more flexibility in drawing sketches. The common properties of Base Drawing Plane are:

Shift move the base grid in the direction of its normal vector above or below based on the value of shift[4].

Plates Count Number of visible tiles.

Grid Width The width the base drawing plane cover in the viewport.

There are also some attributes control the rendering behaviour of the base drawing plane:

Grid Visibility Unchecking this field will hide the base grid from the viewport.

Hide Grid Hair Whenever the mouse move on the grid, there is always a small hair line that is projected on the drawing plane. In some cases, the user decides to hide this hair line, he may uses this button.

Grid Snap Mouse movement on the base grid has an infinite precision when grabbing x, y, z values. Checking this control limit these values into their integer parts.

[3]The rotation is done by the virtual track ball algorithm.

[4]The shift value is actually the fourth item in the plane equation $ax + by + cz + d = 0$, in this equation shift is the d variable

Reset Camera Pressing this button resets all movements that were done on the viewport and get them back to the original location[5].

In addition to the normal rendering, there is a clipping option in the viewport that can be selected from Clipping Lines as shown in Fig.(7.15).

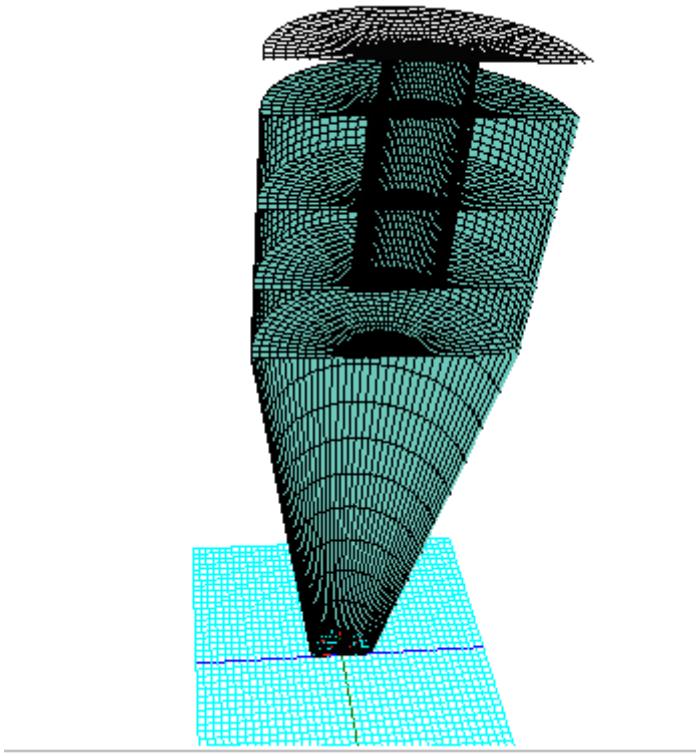


Figure 7.15: Clipping along x-axis (yz-plane)

7.2.4 2D Gridding Menu

Gridding the sketch involves selection of certain points on the viewport to form up the blocks that form the complete domain. In order to define these blocks and complete the domain grid, the user may use the following buttons.

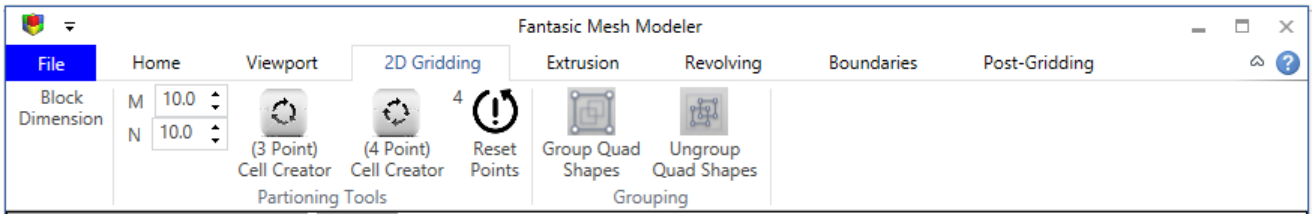


Figure 7.16: 2D Gridding Menu

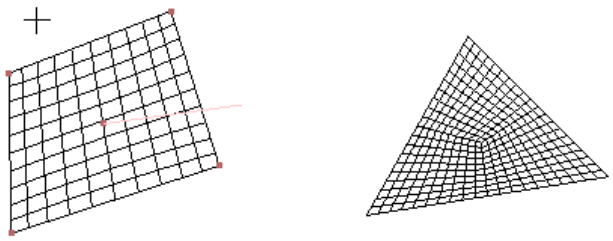
Block Dimension (M x N) The fields predefine the number of grid cells in the x, and y-directions.

(4 Point) Cell Creator Pressing this button put the viewport in a state to receive 4 points to form a cell (block that will be gridded) as shown in Fig. (7.17a).

[5]Usually this is at $x = 0, y = 0, z = -100$

(3 Point) Cell Creator The same functionality of creating grid but with 3 points and it generate 3 connected grid blocks as shown in Fig. (7.17b).

Reset Points In circumstances of wrong point selection, this button can reset the process as if it were no points has been selected yet.



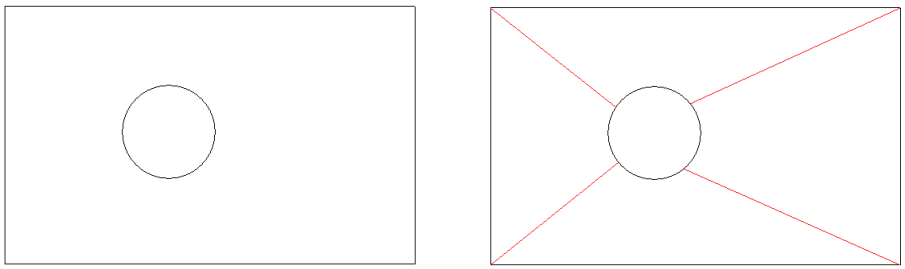
(a) (b)

Figure 7.17: Single Grid Generated from 4 Point, and 3 Point Buttons.

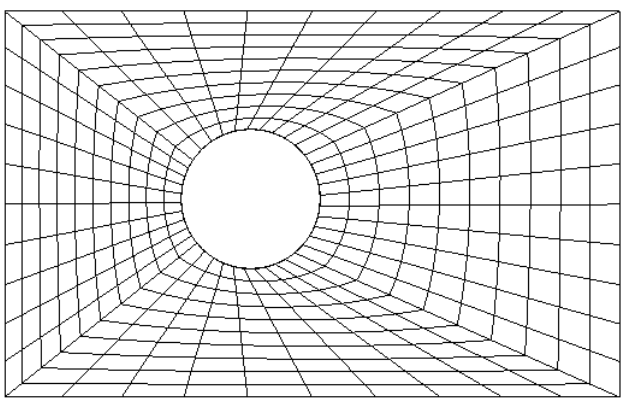
7.2.4.1 Gridding Sequence

Figure (7.18) illustrate the sequence of creating a grid for a simple domain Fig. (7.18a) by:

1. Creating 4 sub-domains.
2. Gridding the 3 sub-domains one after the other using the block see Fig. (7.18c).



(a) Empty Domain (b) Sub Domains Divisions



(c) Final Grid.

Figure 7.18: Grid Creation Sequence

After the grid creation and making sure that they are all connected, and adhering to the required guidelines, the user can click **GroupQuadShapes** button for grouping these blocks into one of the closed shapes in the viewport as shown in Fig. (7.19).

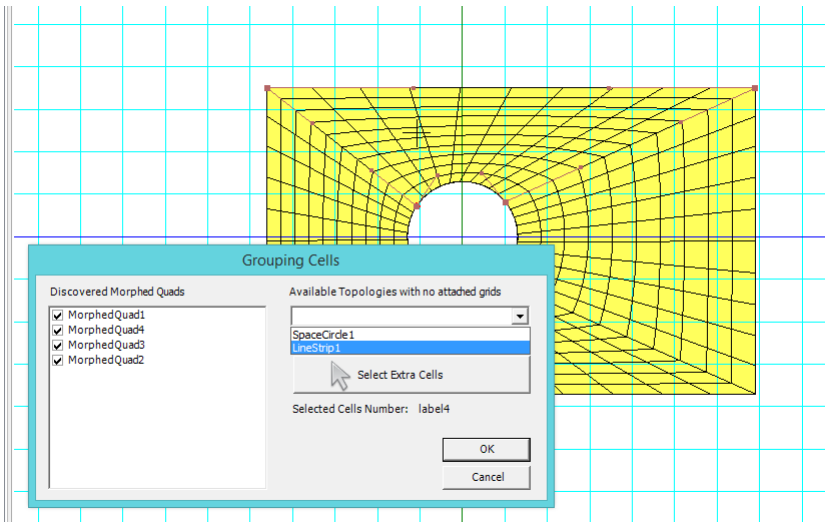


Figure 7.19: Selecting the container of the grouped grids.

Once grouping is done, the container shape is now ready for subsequent 3D operations to complete the desired grid.

7.2.5 Extrusion Menu

Extrusion is the process of conveting the 2D shape into 3D shape. For example the extrusion of a circle will result into a cylinder by extending the circle points with the aid of the normal vector, z in this case, to the required height and segmenting this height with the number of required partitions.

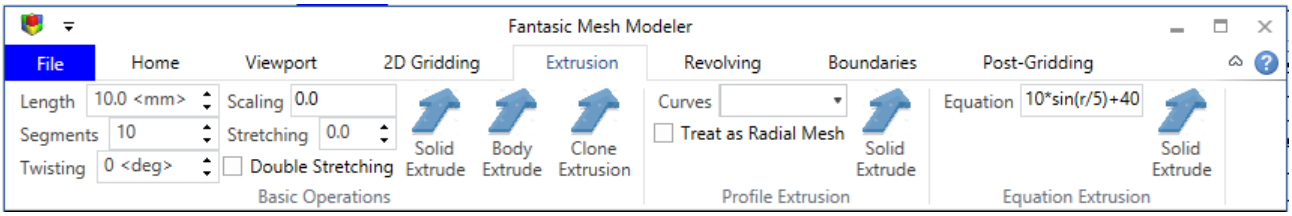


Figure 7.20: Extrusion Menu

7.2.5.1 Basic Extrusion Processes

This is the regular extrusion behaviour and can be controlled by the following field values:

Length The length of the extrusion.

Segments The number of layers covering the extrusion length.

Twisting $[-2\pi, 2\pi]$ When applied in degrees, each layer will rotate around the normal vector of the extrusion with a value = (Twisting Angle / Number of Segments)

Scaling $[-1, 1]$ When applied, the points at each layer contract ($Scaling < 0$), or expand ($Scaling > 0$) according to its corresponding point on the normal vector line.

Stretching $[0, 2]$ When applied, the height between layers is changing based on this value to simulate contraction and expansion between layers. When $Stretching > 1$ contraction occur in first layers, while $Stretching < 1$ result in contraction at ending layers.

Double Stretching This check box change the stretching behaviour to affect both ends of the extruded shape. When $Stretching > 1$ contraction occur at the edges , while $Stretching < 1$ result in contraction occur in the middle of the extruded shape.

Solid Extrude Extrudes the object with its inner grid that was assigned to it when blocks were grouped together. The operation produces R^3 manifold

Body Extrude Extrudes the 2D shape only which result in a surface of this shape. R^2 manifold

Clone Extrude Copys the shape with the given number of segments along the normal vector.

7.2.5.2 Profile Extrusion

Shapes that are drawn on the x-y plane can be extruded with the guidance of another curve that is drawn on y-z plane or x-z plane as shown on Fig. (7.21)

Curves Drop down list of all curves that can be used in the profile extrusion process.

Treat as Radial Mesh When this value is true (checked) the extrusion depends on the radius of the point extruded to get the required height Fig. (7.22).

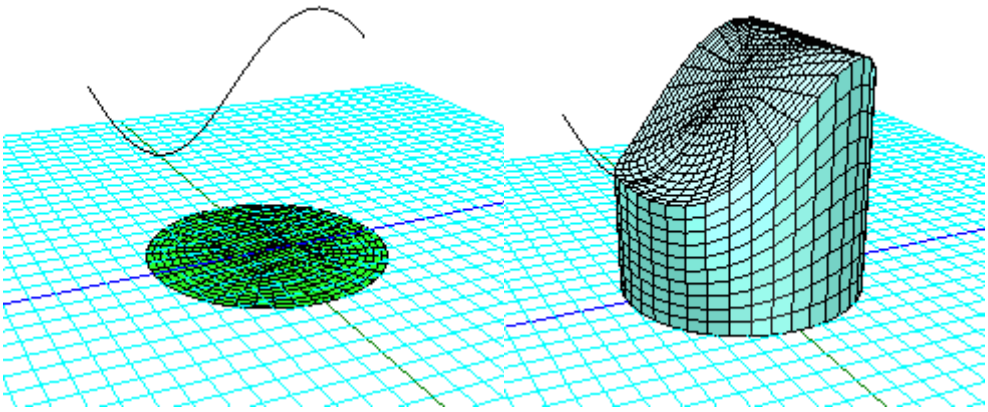


Figure 7.21: Profile Extrude Process

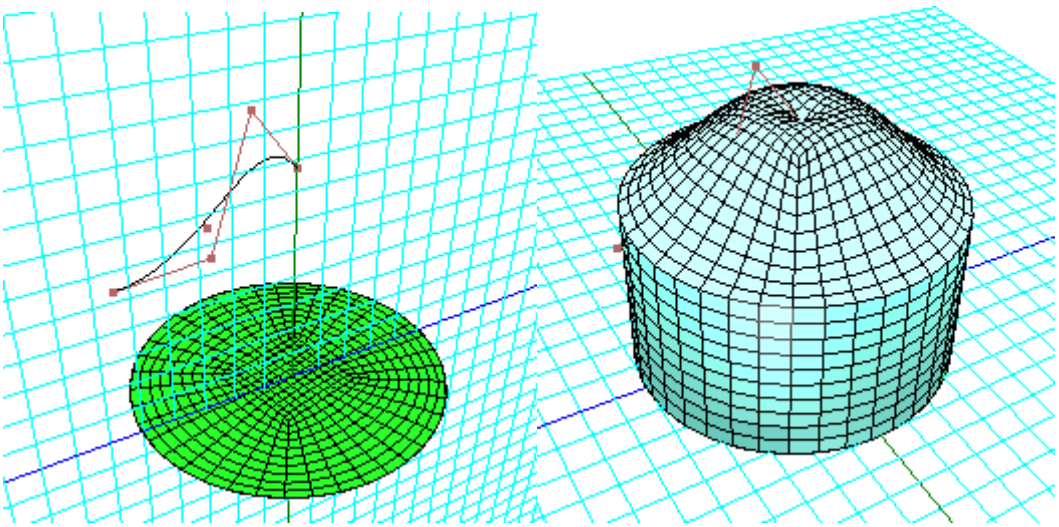


Figure 7.22: Radial Profile Extrude

7.2.5.3 Equation Extrusion

The ability of the user to write arbitary equation for the extrusion Fig. (7.23). The equation is expecting one parameter r to be written inside the expression. The program will run the the equation for each point on the mesh by giving its radius value.

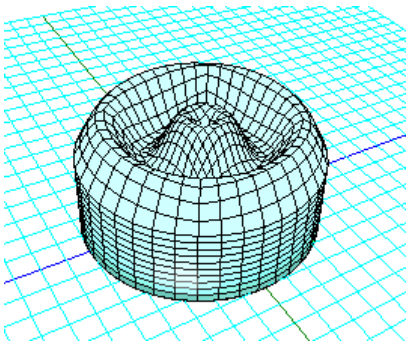


Figure 7.23: Extrude by equation $10sin\,(r/5) + 40$

7.2.6 Revolving Menu

Revolving operation can be considered an extrusion process but involve a rotational axis and an angle of rotation.

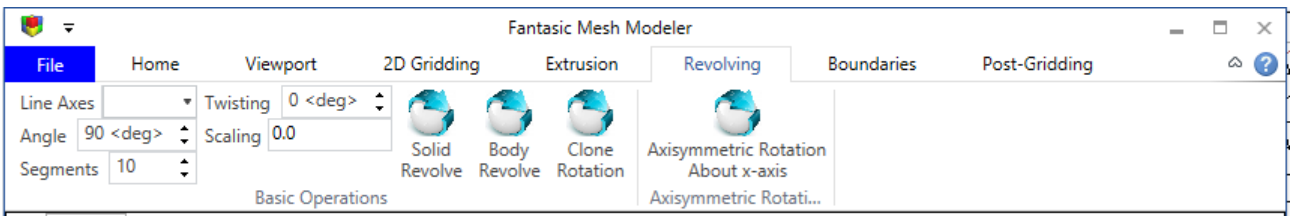


Figure 7.24: Revolving Menu

7.2.6.1 Basic Operations

The following field values control the revolving process:

Line Axis Dropdown list of available Lines in the viewport that can act as rotation axis, in addition to the x, y, z axes.

Angle $[0, 2\pi]$ The whole angle of revolving.

Segments Number of layers.

Twisting $[-2\pi, 2\pi]$ When applied in degrees, each layer will rotate around the normal vector of the extrusion with a value = (Twisting Angle/Number of Segments)

Scaling $[-1, 1]$ When applied, the points at each layer contract ($Scaling < 0$), or expand ($Scaling > 0$) according to its corresponding point on the normal vector line.

Solid Revolve Revolves the shape with its inner grid that was assigned to it when blocks were grouped together. The operation produce a R^3 manifold

Body Revolve Revolves the 2D shape only which result in a surface of this shape. R^2 manifold

Clone Revolve Copys the shape with the same number of segments Fig. (7.25).

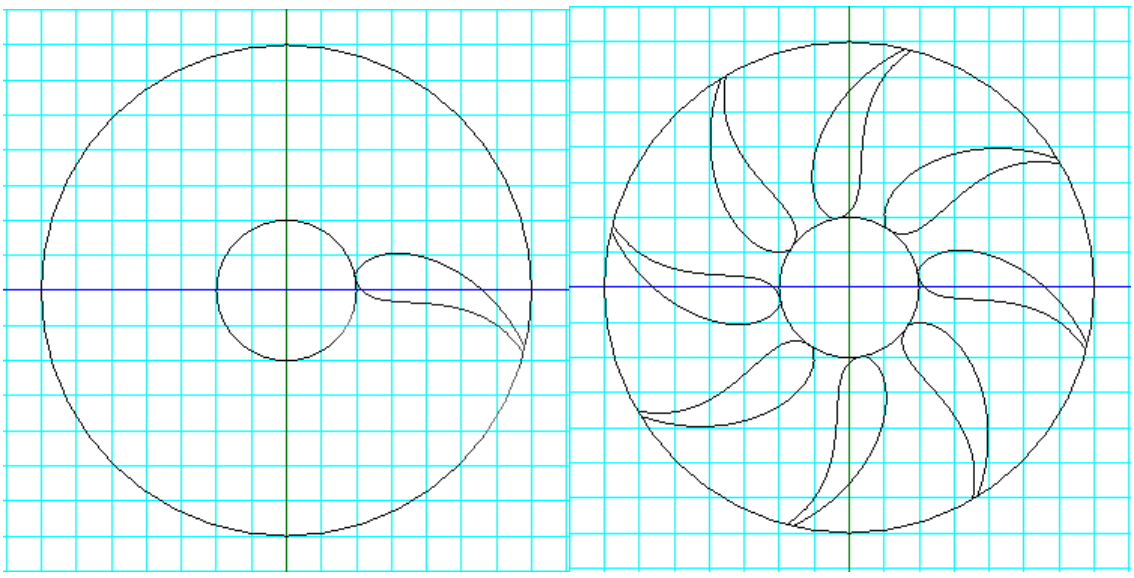


Figure 7.25: Clone Revolve with angle 360 degree and 8 segments

7.2.6.2 Axisymmetric Revolve

Any grid that has one or more side lying on the x-axis can be reolved axisymmetrically around the x-axis with 90° Fig. (7.26).

A special calculation is taken into consideration for avoiding the singularity point in this case.

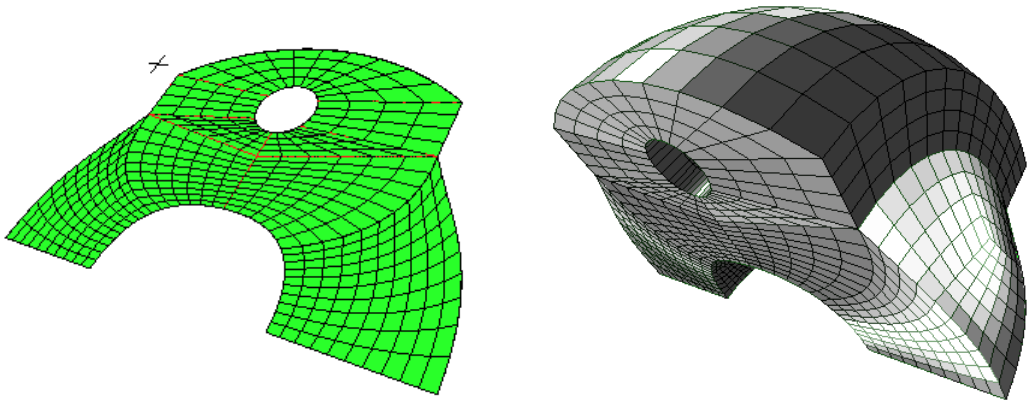


Figure 7.26: Axisymmetric Revolve

7.2.7 Boundaries Menu

Boundary conditions are of extreme importance to the CFD analysis. The menu is responsible for selecting the boundary surfaces and assigning them to known boundary types. The menu is divided into 3 parts:

1. Patch Type: Specify the active boundary condition.

2. Visibility

   (a) Regions Visibility: Toggle On/Off the display of boundary conditions on the model.

   (b) Delete All Regions: Complete remove of all specified boundary conditions on the model.

3. Selection

   (a) Element Information: Allow the user to select a surface face to view its information on the Cell Properties window.

   (b) Individual Elements: Changes single surface face boundary condition to the active boundary type.

   (c) Vertical strip Elements: Changes one strip of vertical elements boundary conditions to the active boundary type.

   (d) Horizontal strip Elements: Changes one strip of horizontal elements boundary conditions to the active boundary type.

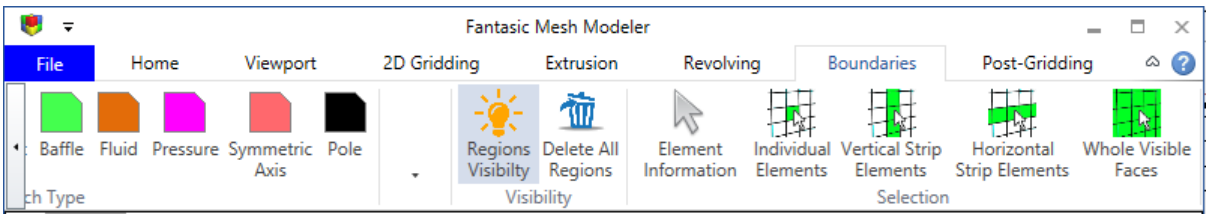


Figure 7.27: Boundaries Menu

7.2.8 Post-Gridding Menu

These operations can be done on the 3D Grid.

Cells Level Number of cells to be removed.

Remove Cell Removes a cell from the hexahedral grid appearing on the viewport.

Surface Plane Set the drawing plane to the surface selected.

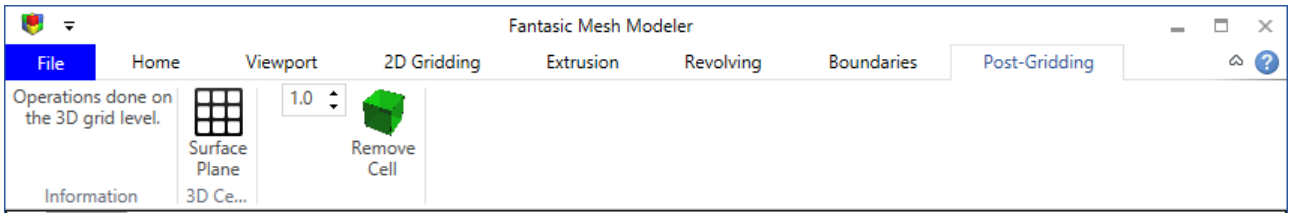


Figure 7.28: Post Gridding Menu

7.3 CASE STUDIES

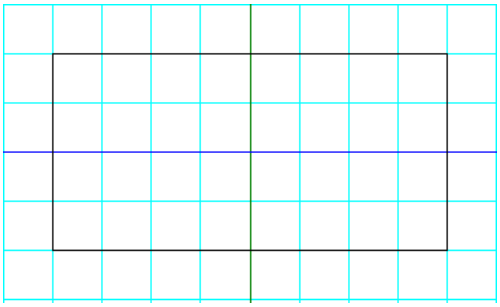
7.3.1 Simple Cases

7.3.1.1 Rectangular Duct with Baffle and 90° Bend

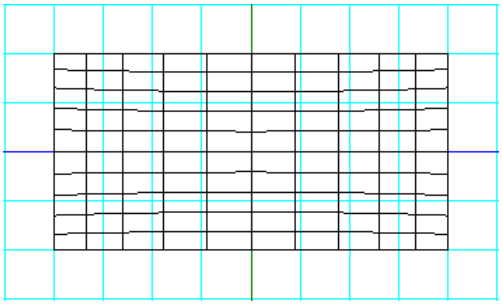
The Program was used to generate a straight duct grid Fig. (7.32). Following steps have been executed to reach this result:

1. Home Menu -> Rectangle (Cross Section of the duct) Fig. (7.29a).
2. Task Bar -> Check Edges.
3. 2D Gridding -> Select the four corners of the rectangle in counter clockwise or clockwise order.
4. Grid Generated (default N x M = 10 x 10) Fig. (7.29b).
5. In Properties Window -> Select MFactor = 1.4 and NFactor = 1.4 with Double Stretching over M and N. Fig.(7.29c).
6. Menu Bar -> Extrusion -> Length = 80, Segments = 30 -> Solid Extrude. Fig.(7.30a)

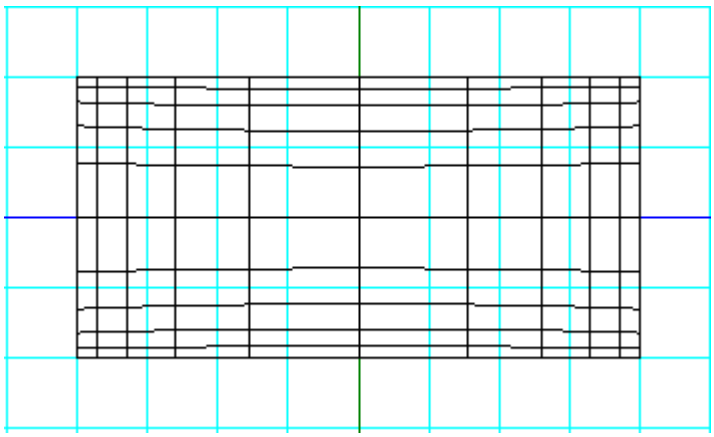
7. Menu Bar -> Home -> Line Segment -> Draw line parallel to the x-axis. Fig.(7.30b).
8. Menu Bar -> Revolving -> Solid Revolve around axis = SpaceLine1 with angle = 90° and 30 Segments.Fig.(7.30b).
9. Menu Bar -> Extrusion -> Extrude for 80 in z-direction with 30 Segments.
10. Menu Bar -> Post-Gridding -> Remove Cell Depth = 4 Width of cells removed as shown in Fig. (7.32).



(a) Duct Cross Section

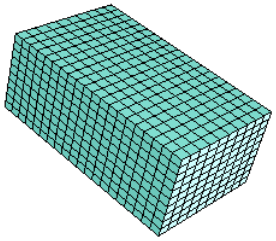


(b) Gridded Duct Section

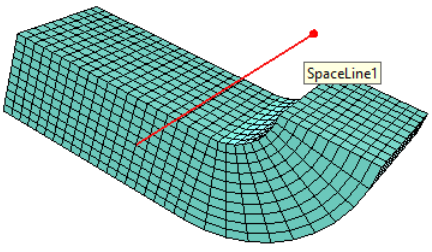


(c) Contracted Duct Grid

Figure 7.29: Empty and Gridded Domain



(a) First Extrude



(b) Rotation Around SpaceLine1

Figure 7.30: Bending Duct Operations

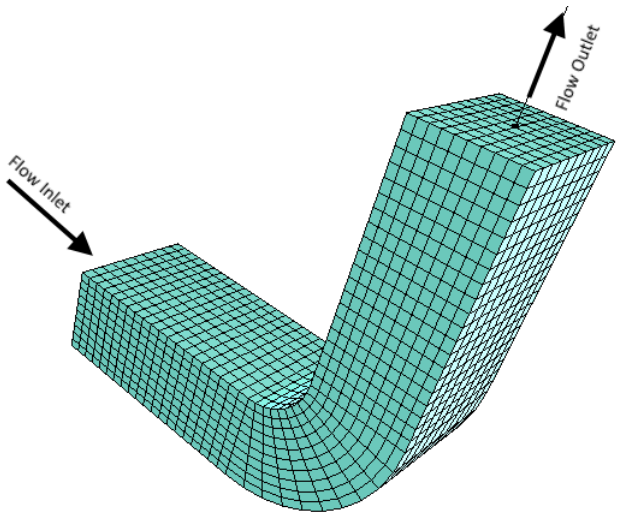


Figure 7.31: Second Extrusion Operation

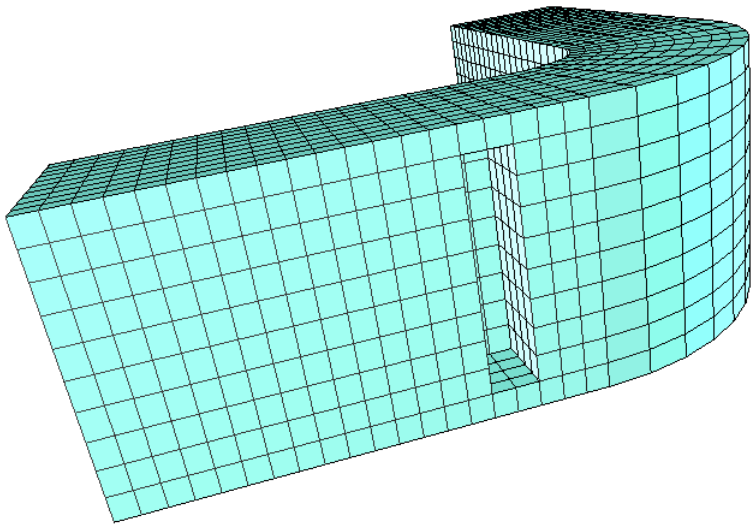


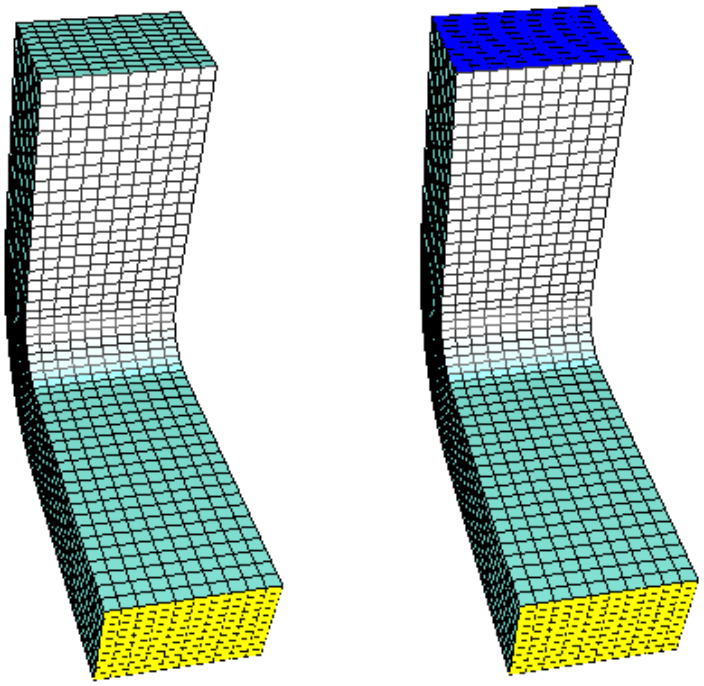
Figure 7.32: Duct Baffle

The inlet and outlet boundaries are defined as follows:

11. Rotate the duct such that both inlet and outlet planes become in view.

12. Menu Bar -> Boundaries -> Whole Visible Faces -> Inlet -> Move the pointer to the inlet face and press left button. Fig. (7.33a).

13. Menu Bar -> Boundaries -> Whole Visible Faces -> Exit -> Move the pointer to the outlet face and press left button. Fig. (7.33b).



(a) Selection of Inlet Boundary (b) Selection of Outlet Bonudary

Figure 7.33: Duct Boundaries Definition

7.3.1.2 Straight Pipe I

1. Menu Bar -> Home -> Circle (Draw circle with required diameter and centre of pipe). Fig. (7.34a).

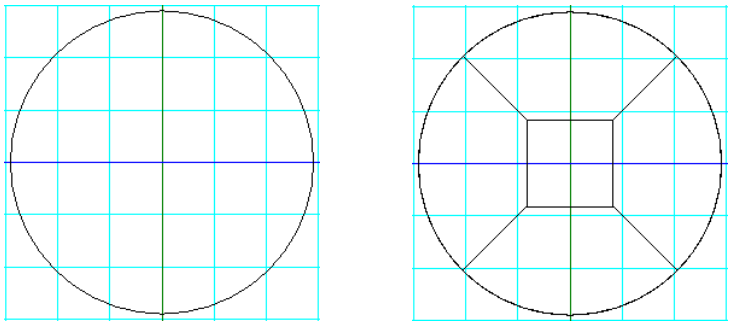
2. Menu Bar -> Home -> Rectangle (Draw square centred with pipe cross-section). Fig. (7.34b).

3. Menu Bar -> Home -> Line Segment (Divide the pipe cross-section into the shown sub-domains). Fig. (7.34b).

4. Menu Bar -> 2D Gridding -> 4 Point Cell Greator -> Edges from the Task Bar -> (Highlight the corners points of each sub-doman in a C-W manner)

5. Grids are created with the default M x N (10 x 10). This can be changed together with the contraction or the expansion factors in the Properties Window. Fig. (7.35).

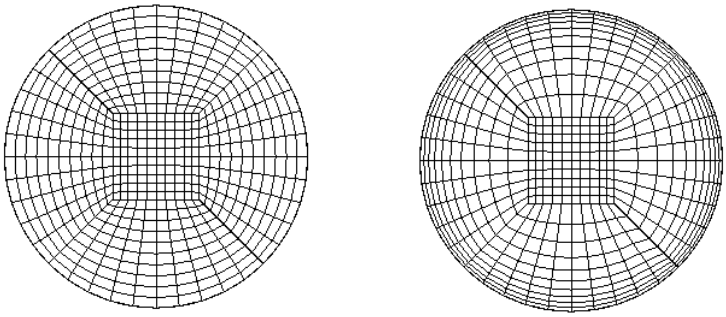
6. Menu Bar -> Extrusion -> Solid Extrude with the required length and number of segments. Fig. (7.36). (Extrusion can be effected with either expansion expansion or contraction function selected from the Stretching field).



(a) Pipe Cross-Section

(b) Pipe Cross-Section Sub-domains

Figure 7.34: Straight Pipe Sketch Initial Steps



(a) Gridding Sub-Domains with 10x10 cells

(b) Wall Increased Cells Density by Applying Stretching

Figure 7.35: Pipe Gridded Cross-Section

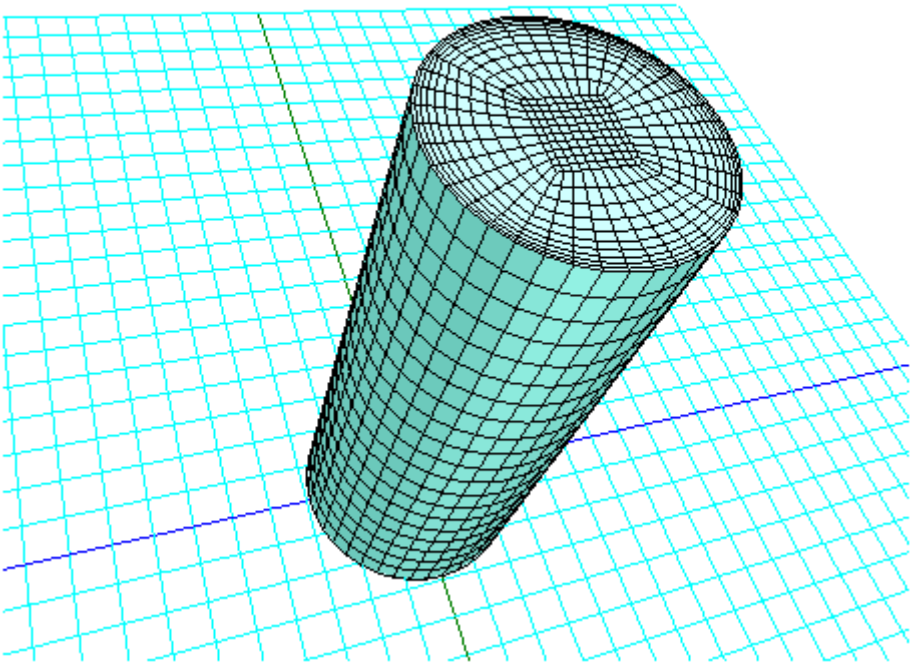


Figure 7.36: Straight Pipe from Basic Extrusion

Bending with 90°

1. Click on the pipe model by mouse.
2. Menu Bar -> Viewport -> Solid Model Plane (The drawing plane should now be on the top of the pipe model)
3. Menu Bar -> Home -> Line Segment (Draw line segment that will serve as a rotation axis)
4. Menu Bar -> Revolving -> Line Axes -> Select Last Space Line from the list.
   (a) Modify Angle field = 90<deg>
   (b) Modify Segments
   (c) Click the Pipe Model
   (d) Click Solid Revolve Button
5. Model is revolved around the selected space line.
6. Menu Bar -> Extrusion -> Solid Extrude. Fig. (7.37).

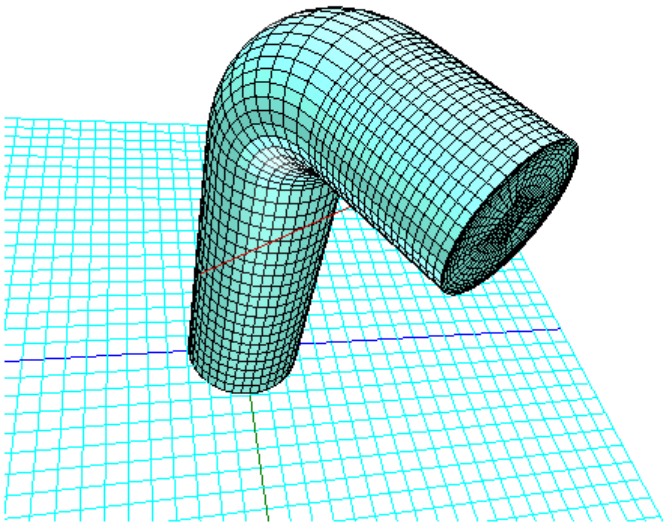


Figure 7.37: Straight Pipe with 90° Bending

7.3.1.3 Straight Pipe II

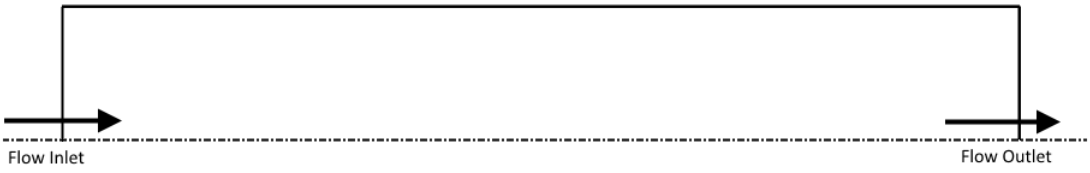


Figure 7.38: Schematic of an Axisymmetric Pipe

In addition to the possibility of generating grid covering a straight pipe by extruding a circle representing its cross section, another option is to revolve around the x-axis a rectangle with the pipe radius as its height and the pipe length as its width as shown in Fig. (7.38). The following steps illustrate the sequence of operations needed for creating the axisymmetric grid:

1. Menu Bar -> Home -> Rectangle (draw a rectangle that its bottom line lie completely on the x-axis)

2. Menu Bar -> 2D Gridding -> 4 Point Cell Creator with Edge snapping activated -> Highlight corners and click them in clock-wise manner.

3. Properties Window -> MFactor is modified as shown in Fig. (7.39).

4. Menu Bar -> Revolving -> Axisymmetric Rotation About x-axis. Fig. (7.40).

5. Three kinds of boundaries need to be declared; Inlet, Exit, and Symmetry Planes, in addition to the pipe walls.

6. Menu Bar -> Boundaries -> Inlet -> Whole Visible Faces -> Select the Inlet plane by clicking the mouse left button. Fig. (7.41a).

7. Menu Bar -> Boundaries -> Exit -> Whole Visible Faces -> Select the Outlet plane by clicking the mouse left button.Fig. (7.41a).

8. Menu Bar -> Boundaries -> Symmetric Plane -> Whole Visible Faces -> Select the First Symmetry plane. Fig. (7.41b).

9. Menu Bar -> Boundaries -> Symmetric Plane -> Whole Visible Faces -> Select the Second Symmetry plane. Fig. (7.41c).

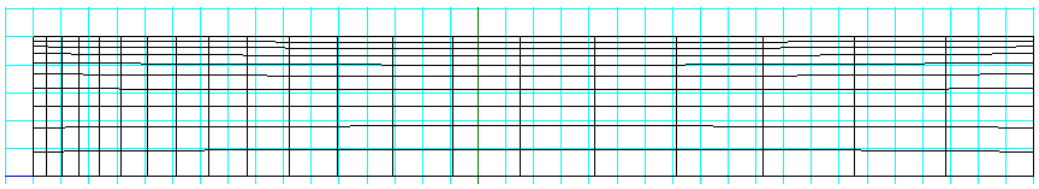


Figure 7.39: Axisymmetric Pipe Cross-Section

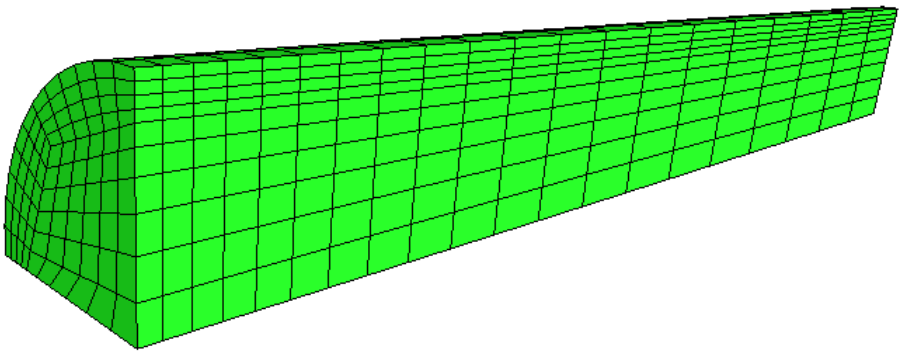
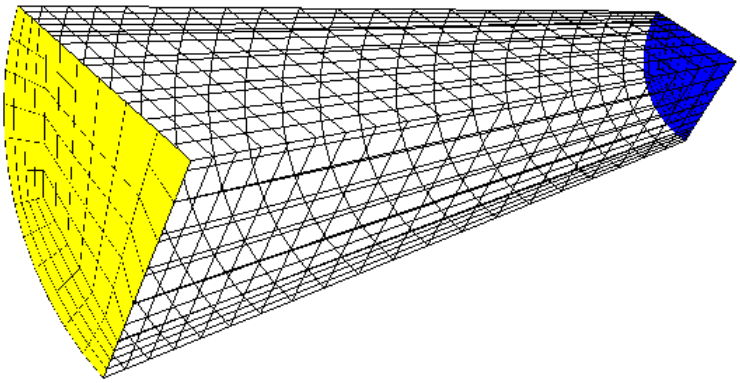
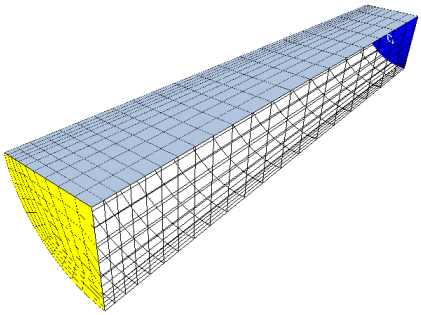


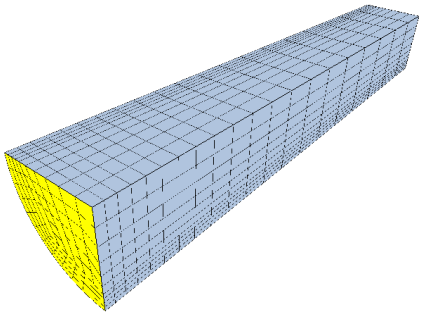
Figure 7.40: Axisymmetric Straight Pipe Grid



(a) Inlet and Outlet Boundaries



(b) Symmetry Plane 1



(c) Symmetry Plane 2

Figure 7.41: Definition of Boundary Conditions

7.3.1.4 S Shape Pipe

1. Starting from a cross-section Fig. (7.35).
2. Menu Bar -> Extrusion -> Solid Extrude.
3. Menu Bar -> Viewport -> Solid Model Plane (Moves the drawing plane to the top of the solid model)
4. Menu Bar -> Line Segment (Draw a line to be used as an axis line) Fig. (7.42a).

5. Menu Bar -> Revolving -> Solid Revolve with Line Axis = SpaceLine1, Angle = 45°. Fig. (7.42b).

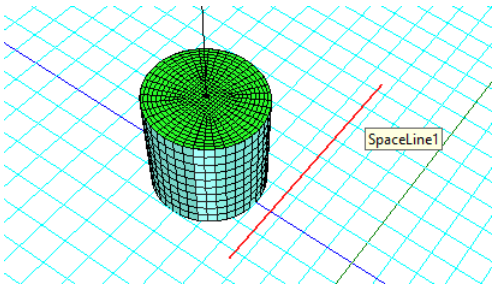
6. Menu Bar -> Extrude -> Solid Extrude

7. Menu Bar -> Viewport -> Solid Model Plane (The plane moves to the top location)

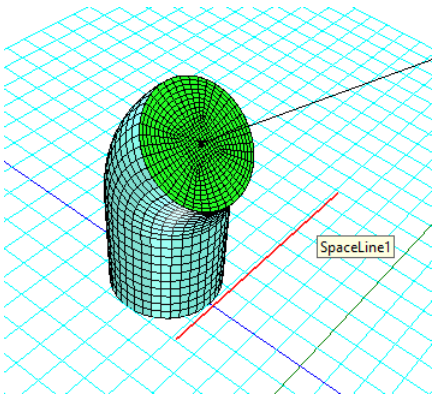
8. Menu Bar -> Line Segment (Draw SpaceLine2 second rotation axis)

9. Menu Bar -> Revolving -> Solid Revolve. Fig. (7.42c).

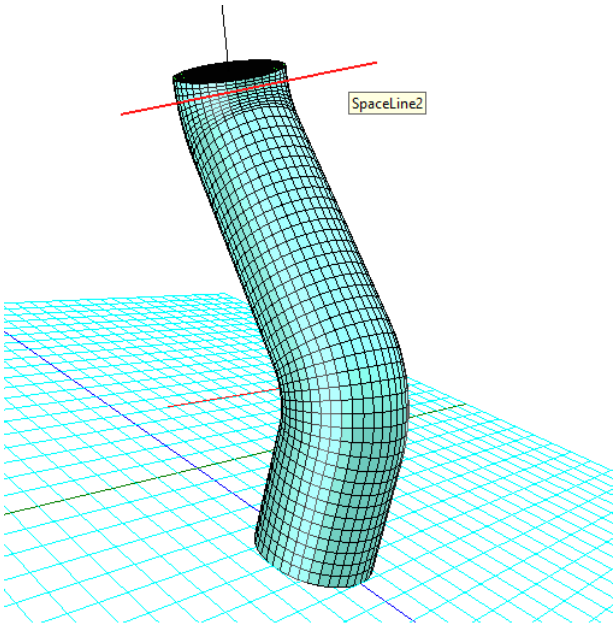
10. Menu Bar -> Extrude -> Solid Extrude. Fig. (7.42d).



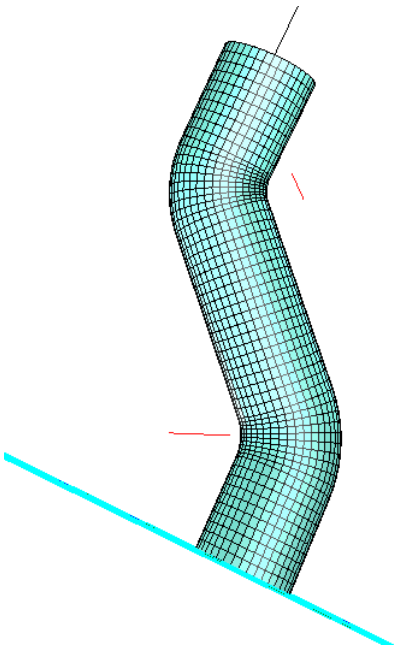
(a) Space Line Axis



(b) First Revolve Operation



(c) Second Revolve Operation



(d) Final Mesh

Figure 7.42: S Shape Pipe Grid Modeling Operations

7.3.2 Composite Cases

7.3.2.1 Axisymmetric Sudden Expansion-Contraction

Following steps were carried out to reach a complete axisymmetric sudden expansion-contraction. Fig. (7.46a).

1. Menu Bar -> Home -> Line Strip (Draw Cross-Section that its lower side completey lie on the x-axis). Fig. (7.43)

2. Menu Bar -> Home -> Line Segment (Specify sub-domains region). Fig. (7.44).

3. Menu Bar -> 2D Gridding -> 4 Point Cell Creator (Highlight sub-domains corners in C-W to create target grid). Fig. (7.45).

4. Menu Bar -> Revolve -> Axisymmetric Rotation About x-axis (to finish the creation 3D axisymmetric grid). Fig. (7.46a).

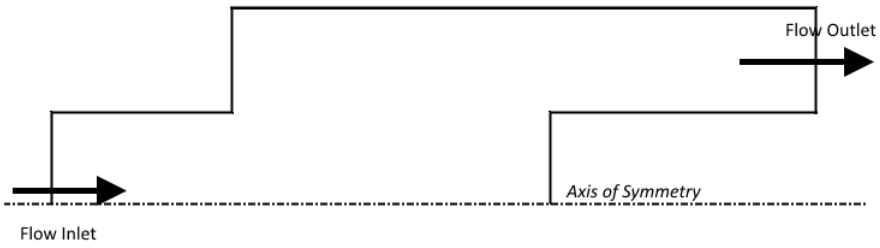


Figure 7.43: Sudden Expansion / Contraction Cross-Section

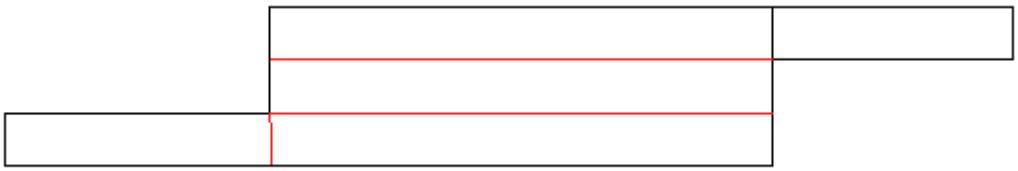


Figure 7.44: Sudden Expansion Contraction Sub-domains

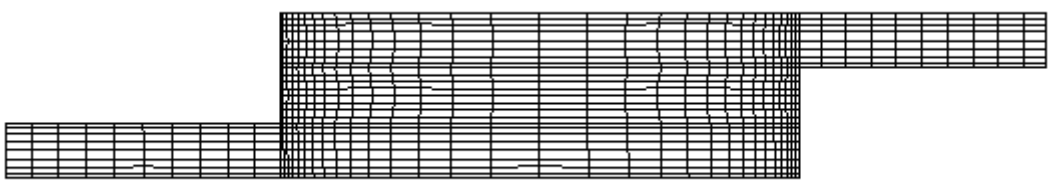
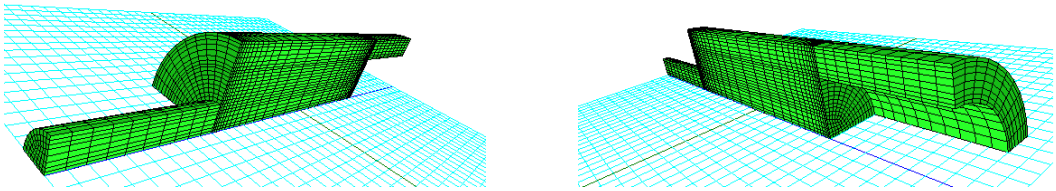
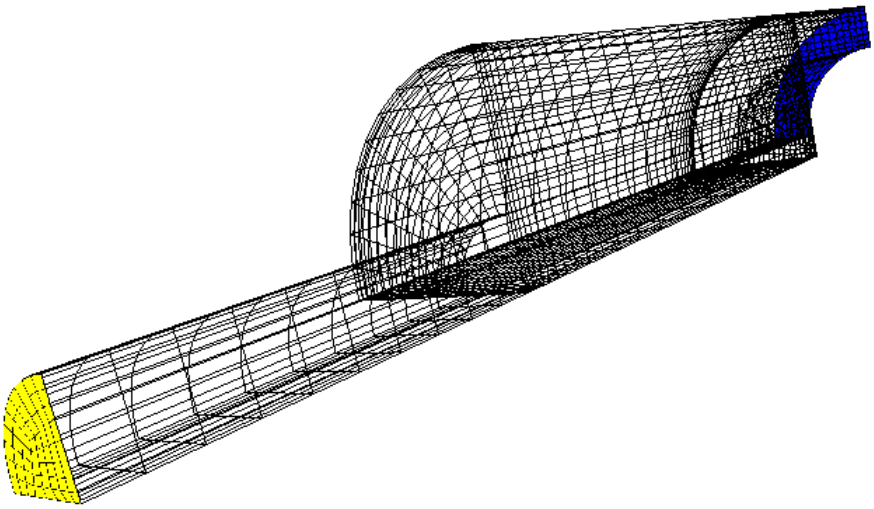


Figure 7.45: Sudden Expansion Contraction Gridded

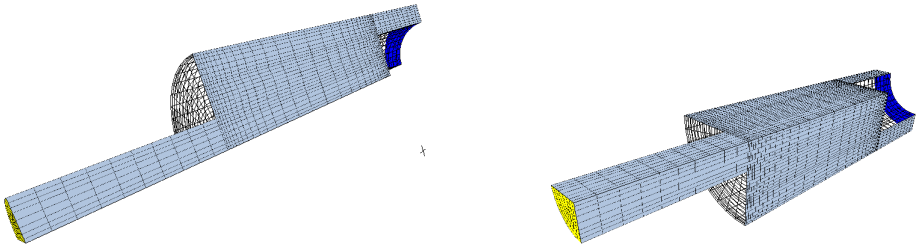


(a) Grid Front View (b) Grid Rear View

Figure 7.46: Expansion Contraction Grid



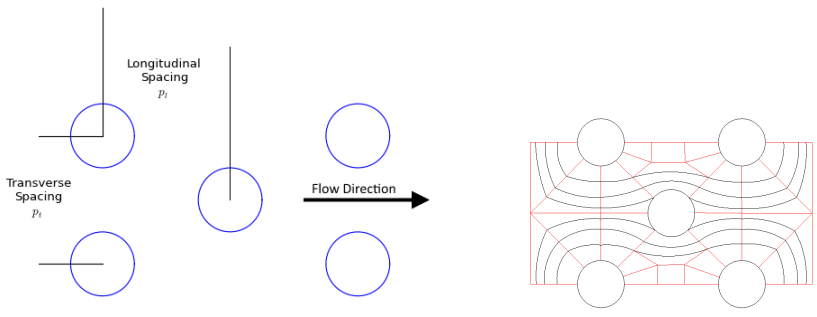
(a) Inlet and Outlet Boundaries



(b) Symmetry Plane 1 (c) Symmetry Plane 2

Figure 7.47: Definition of Bonudary Conditions

7.3.2.2 Bank of Tubes



(a) Tube Arrangement (b) Sub-domains

Figure 7.48: Bank of Tubes

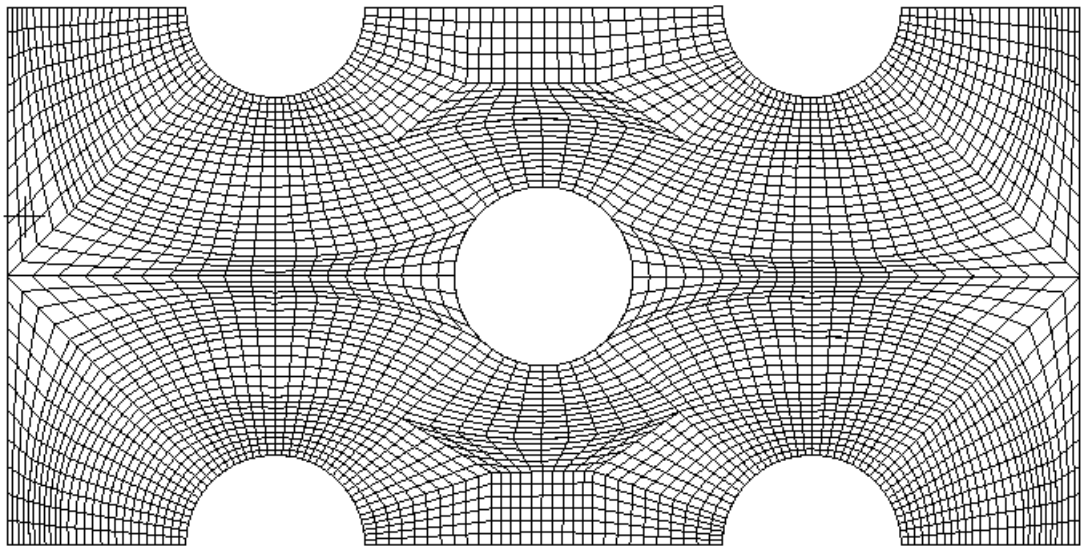


Figure 7.49: Gridded Bank of Tubes

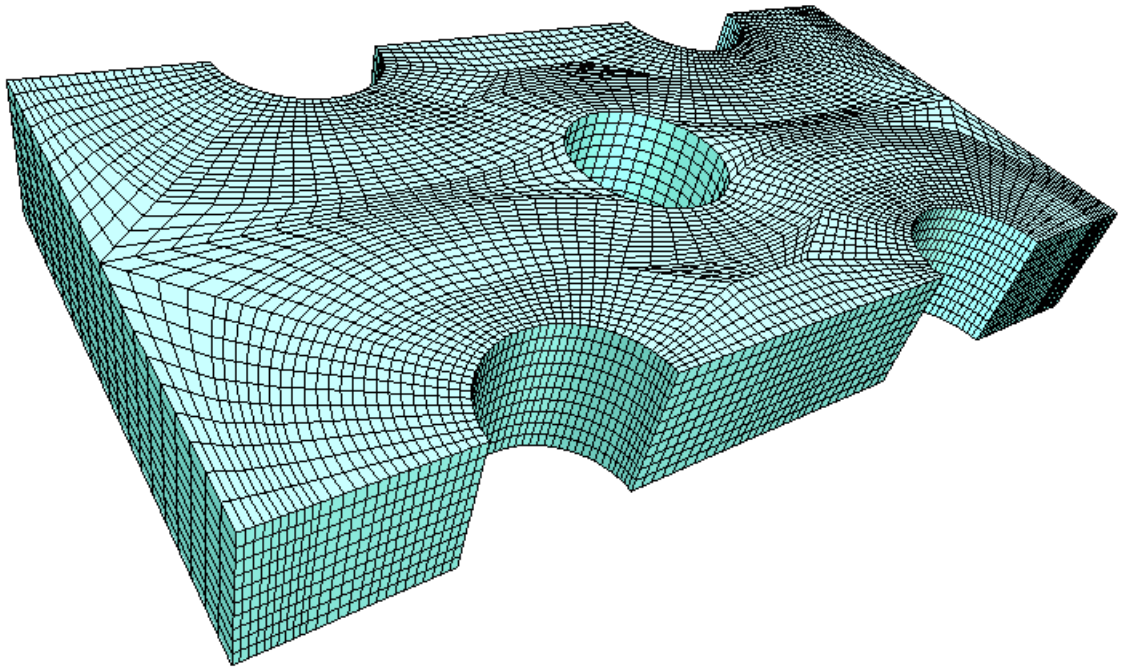
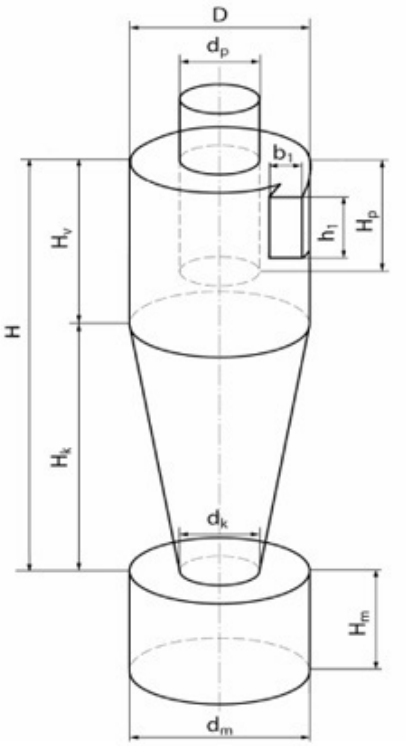


Figure 7.50: Extruded Bank of Tubes

7.3.2.3 Hydro Cyclone

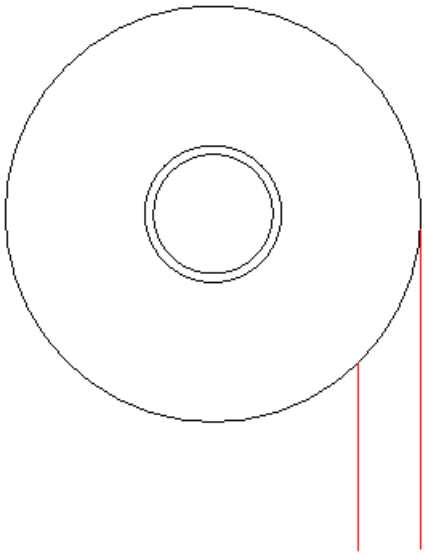


D	d_p	d_k	b_1	h_1	H_v	H_k	H_p	H_c	d_m	H_m
200	65	85	40	90	262	520	178	782	150	110

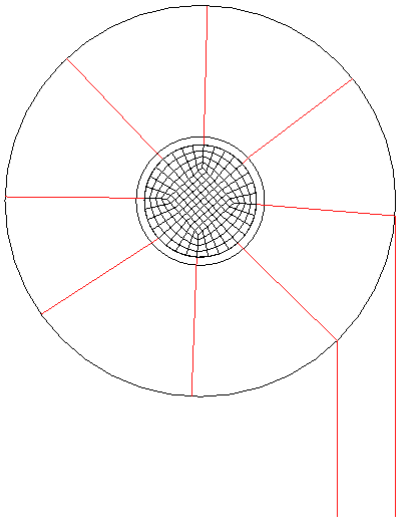
Figure 7.51: Hydro Cyclone Schematic Diagram[61]

Sketching and Gridding

1. We begin the drawing of hydro cyclone with 2D sketching as shown in Fig. (7.52a).
   (a) Menu Bar -> Home -> Circle (Draw circle with $d_k = 85$)
   (b) Draw inner circle with the ratio d_p/D
   (c) Draw another inner circle with less diameter to form an inner tube cross section as shown in Fig. (7.52a).
2. Inner Circle -> Right Click -> Make Grid ... -> Four Cores with 8 Exterior Points -> Press Ok (Circle is automatically gridded with 8 snapping points on its exterior)
3. Menu Bar -> Home -> Line Segment -> Edges, Nearest, and Grid from the Task Bar -> (For each snapping point on the gridded circle draw a line segment to the outer circle) Fig. (7.52b).
4. Menu Bar -> 2D Gridding -> 4 Point Cell Creator -> Edges from Task Bar -> (Highlight the corners points of each sub-doman in a C-W manner)
5. Grids are created with the default M x N (10 x 10). Fig. (7.53).



(a) Cyclone Sketch



(b) Gridded Inner Circle with sub-domains

Figure 7.52: Cyclone Sketching and Sub-Domains

3D Operations After the complete gridding of the cyclone cross-section, following steps are required to generate the 3D mesh of the Cyclone:

1. Double click the flap to unselect it from the sub-domains that are going to be extruded.
2. Menu Bar -> Extrusion -> Solid Extrude with Length = 520, Scaling = 2.35, and Segments = 10. Fig. (7.54).
3. Menu Bar -> Extrusion -> Solid Extrude with Length = 84. Fig. (7.55).
4. Double click sub-domains of the inner tube.
5. Menu Bar -> Extrusion -> Solid Extrude with Length = 88. Fig. (7.56).
6. Double click the flap to select it for next process

7. Menu Bar -> Extrusion -> Solid Extrude with Length = 90. Fig. (7.57).

8. Unselect all other sub-domains excluding ones inside the inner tube.

9. Menu Bar -> Extrusion -> Solid Extrude with Length = 50 Fig. (7.58)

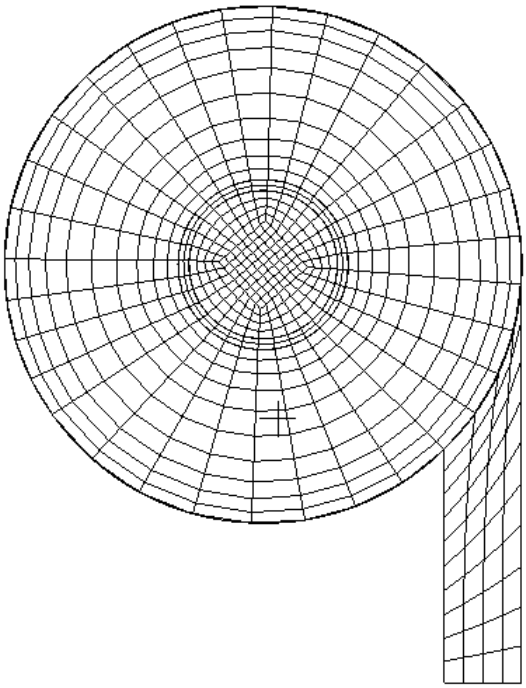


Figure 7.53: Cyclone Cross-Section Completely gridded

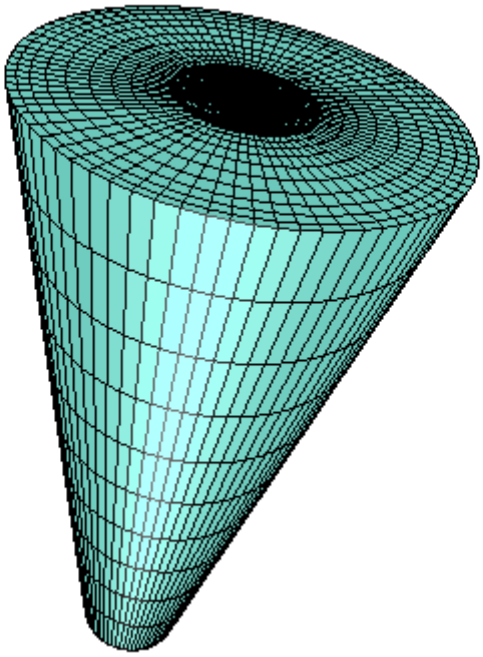


Figure 7.54: Cyclone First Extrusion Process

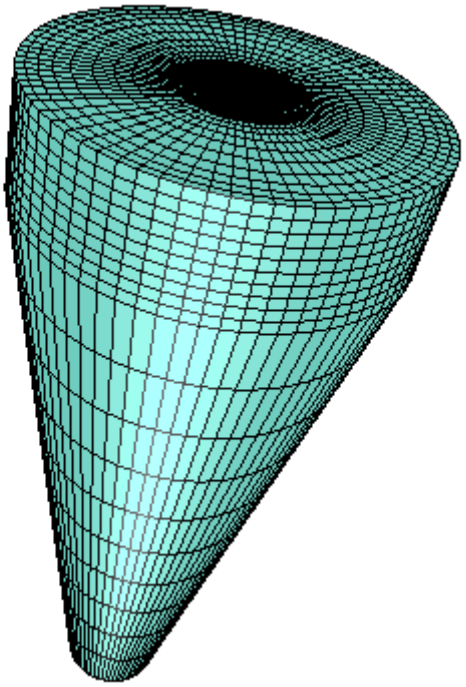


Figure 7.55: Cyclone Second Extrusion Process

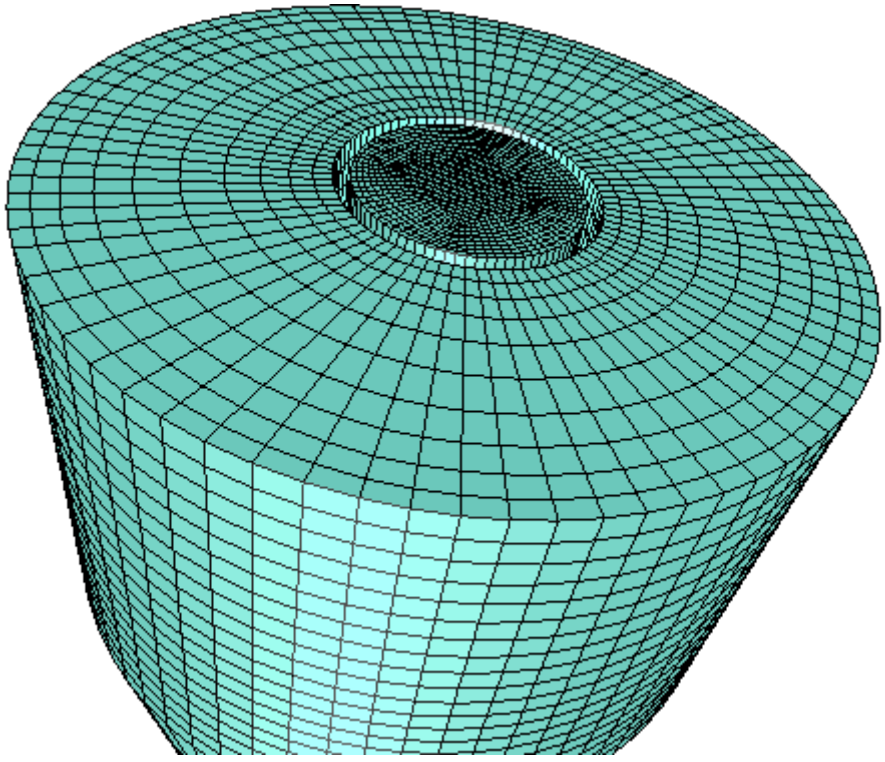


Figure 7.56: Cyclone Third Extrusion Process

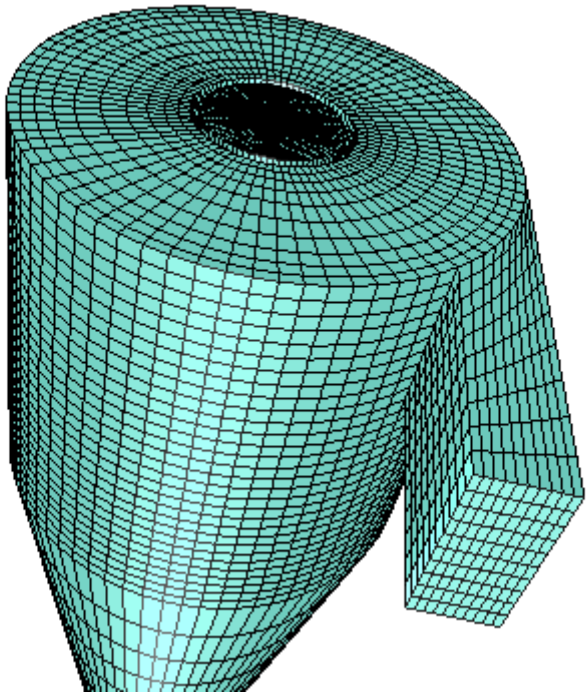


Figure 7.57: Cyclone Fourth Extrusion Process

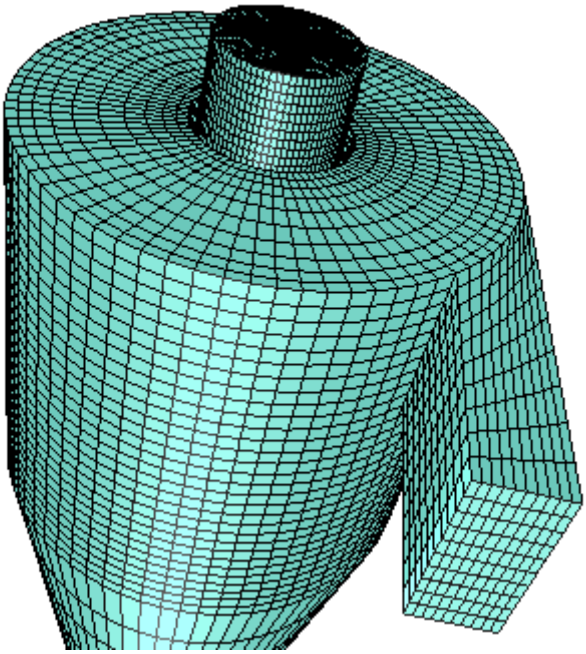


Figure 7.58: Cyclone Fifth Extrusion Process

Cyclone Lower Part

1. Draw Circle at the base of the cyclone.

   (a) Activate Grid, and Nearest Snapping Options.

2. Line Segments -> Draw Line from highlighted node on the cyclone grid on bottom to the nearest point on the circle

3. Repeat last process 3 times to complete the sub-domains. Fig. (7.59)
4. Menu Bar -> 2D Gridding -> 4 Point Cell Creator -> Highligh points with Edges Snap Option activated. Fig. (7.60).
5. Right Click Bottom Circle -> Invert Normal (Extrusion takes place along the normal).
6. Menu Bar -> Extrusion -> Solid Extrude with Length = 100 Fig. (7.61)

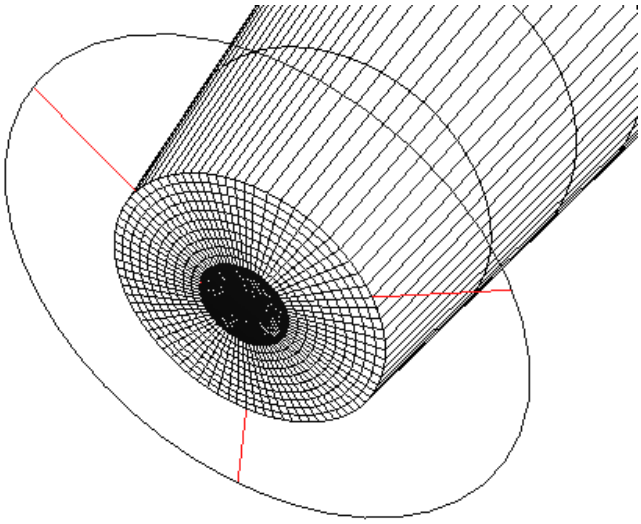


Figure 7.59: Cyclone Lower Part Sub-domains

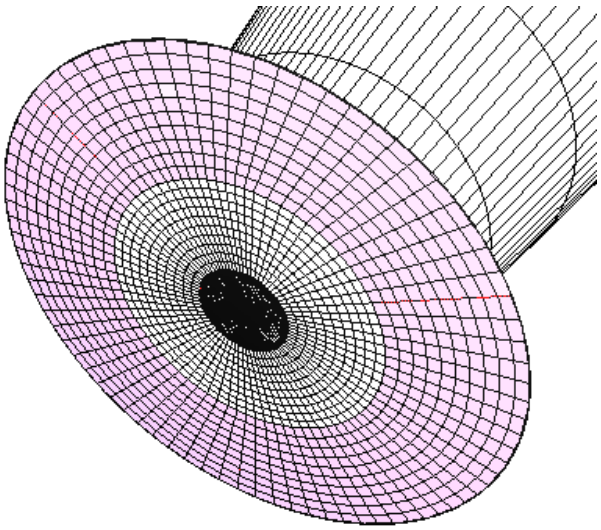


Figure 7.60: Cyclone Lower Part Gridded

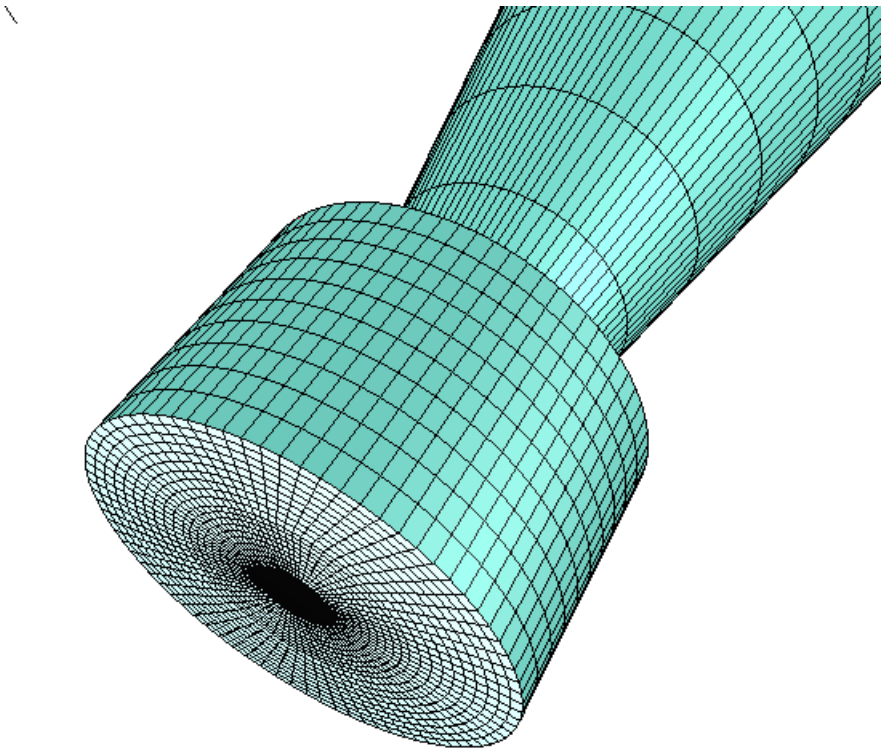


Figure 7.61: Cyclone Lower Part First Extrusion

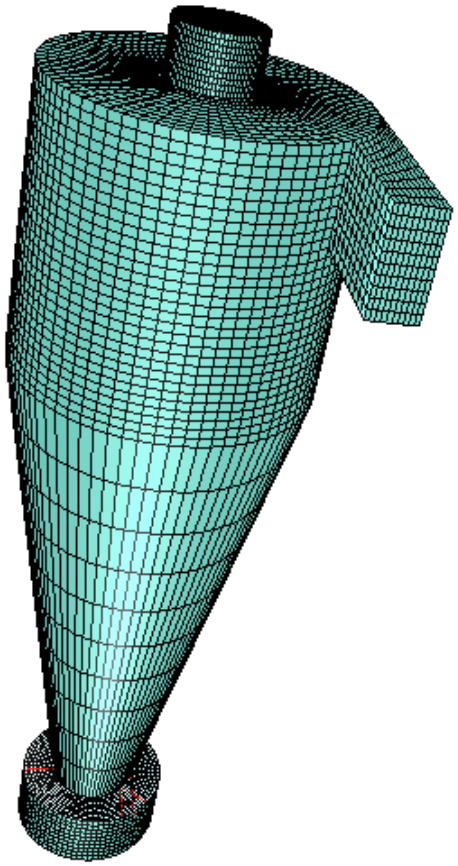
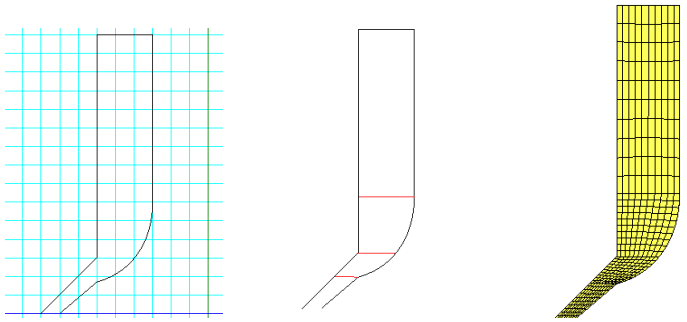


Figure 7.62: Final Hydro Cyclone Mesh

7.3.2.4 Internal Combusion Engine Poppet Valve

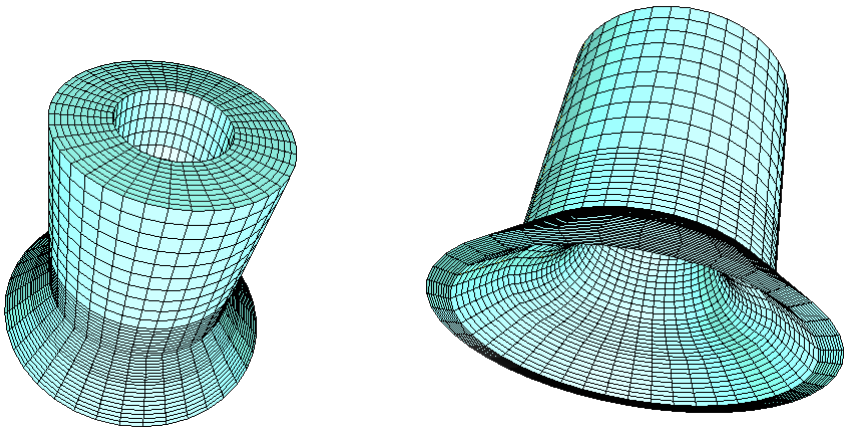
A poppet valve is a valve typically used to control the timing and quantity of gas or vapour flow into an engine. It consists of a hole, usually round or oval, and a tapered plug, usually a disk shape on the end of a shaft also called a valve stem. The portion of the hole where the plug meets with it is referred as the 'seat' or 'valve seat'. The shaft guides the plug portion by sliding through a valve guide. In exhaust applications a pressure differential helps to seal the valve and in intake valves a pressure differential helps open it.

It is required to grid the space around the poppet valve for the sake of CFD analysis. The process includes two parts, first one is the modeling of the poppet valve space itself. Second one is the modeling of the upper space connected to the valve that take or intake gases.



(a) Valve Cross-Section (b) Geometry Sub-domains (c) Valve Gridding

Figure 7.63: Poppet Valve Profile and Sub-domains



(a) (b)

Figure 7.64: Poppet Valve Grid

Poppet Valve Mesh Generation

1. Home Menu -> Line Strip (Draw the cross section to the left of the y-axis). Fig. (7.63a).

2. Home Menu -> Line Segment (Divide the area into sub-domains). Fig. (7.63b).

3. 2D Gridding Menu -> 4 Point Cell Creator (Build the grid by selecting sub-domains corners in clockwise manner). Fig. (7.63c).

4. Revolving Menu -> Solid Revolve with Angle = 360°, Line Axis = Y-Axis. Fig. (7.64).

Upper Part Gridding

1. Menu Bar -> Post-Gridding -> Surface Plane (Click on any cell surface at the top of the model).

2. Menu Bar -> Home -> Circle (Draw Circle on the plane).

3. Menu Bar -> Home -> Line Segment (Draw 3 Line with Grid and Nearest Snapping Activated). Fig. (7.65).

4. Menu Bar -> Home -> Bezier Curves (Draw curve coming out from the circle). Fig. (7.65).

5. Menu Bar -> 2D Gridding -> 4 Point Cell Creator (Building the grid by highlighting Edges Snapping points). Fig. (7.66).

6. Menu Bar -> Extrusion -> Solid Extrude. Fig. (7.67).

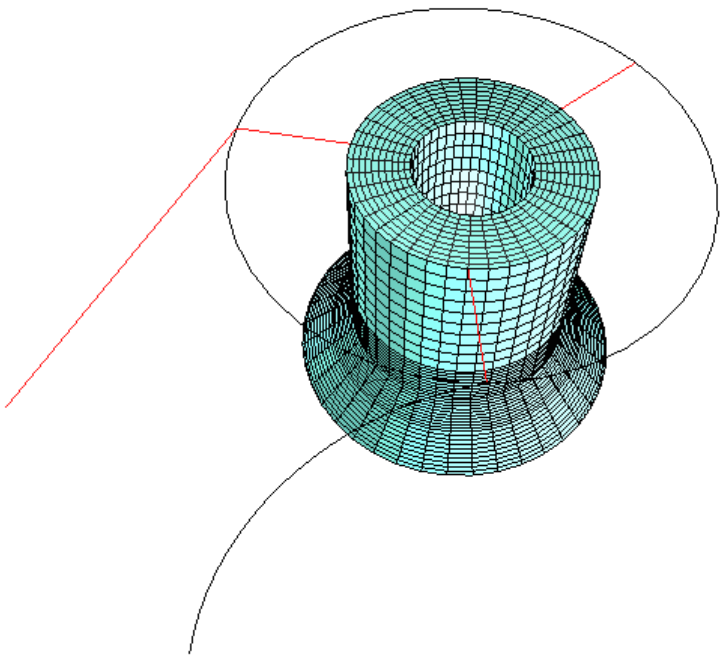


Figure 7.65: Poppet Upper Sketch with sub-domains

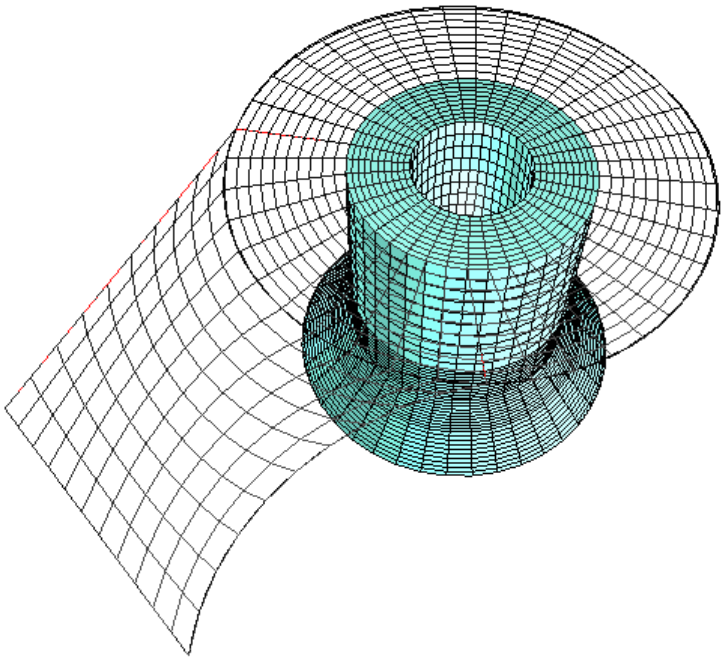


Figure 7.66: Poppet Upper Part Gridded

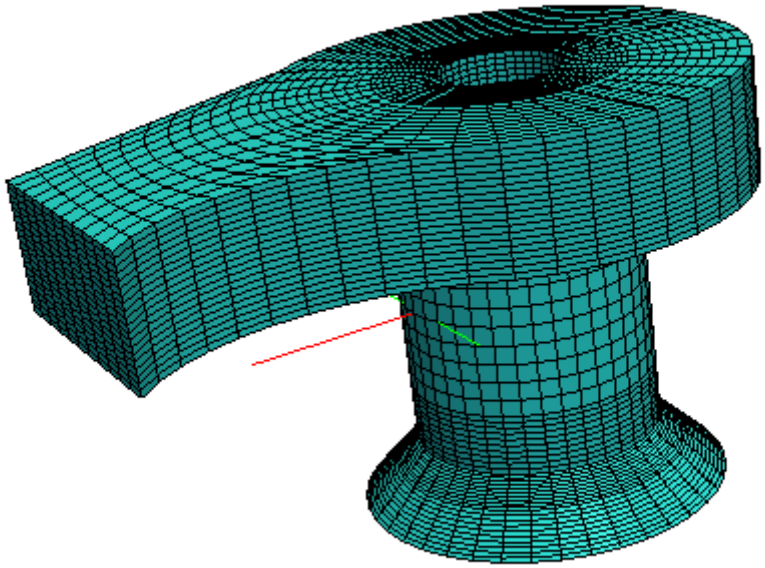


Figure 7.67: Poppet Valve 3D Mesh

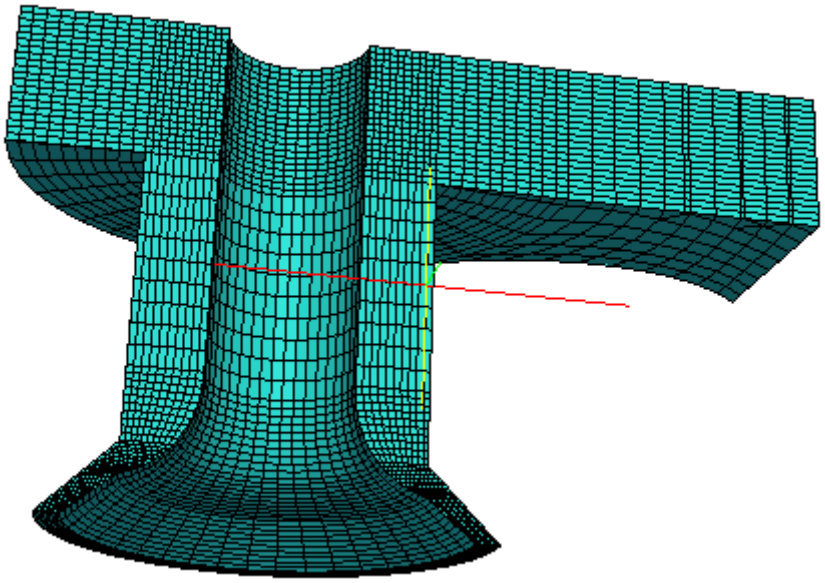


Figure 7.68: Poppet Valve Grid Cross Section

7.3.2.5 Composite Valve

The valve has 3 operations modes in which allow three different routes of the fluid flow:

Middle Valve seat positioned in the middle between the axial plug and annulus plug. Fig. (7.69).

Left Axial plug completely close the valve with the valve seat. Fig. (7.70).

Right Annulus plug completely close the valve with the valve seat. Fig. (7.71).

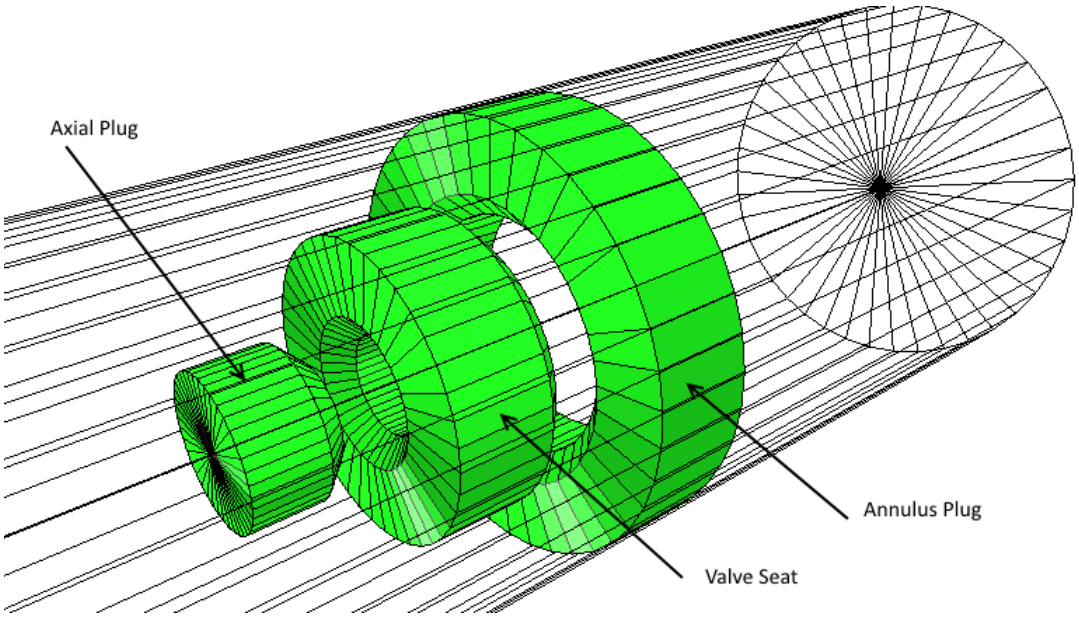


Figure 7.69: Isometric Valve Seat at Middle

Figs. (7.72 - 7.74) Shows the cross section of the valve at the three operations mode. These drawings are the starting point of the gridding operation.

The steps of building the mesh can summarized as follows:

1. Menu Bar -> Home Menu -> Line Segments (Draw Lines with Edges and Nearest snap options activated). Figs. (7.75, 7.78, 7.82).

2. Menu Bar -> 2D Gridding -> 4 Point Cell Creator -> Edges from the Task Bar -> (Highlight the corners points of each sub-doman in a C-W manner).

3. Generated Grids Stretching factors MFactor, and NFactor, should be modified in the Properties Window. Figs. (7.76, 7.79, 7.83).

4. Menu Bar -> Revolving -> Axisymmetric Rotation About x-axis (Creating the final 3D Mesh of the valve). Figs. (7.77, 7.80, 7.84).

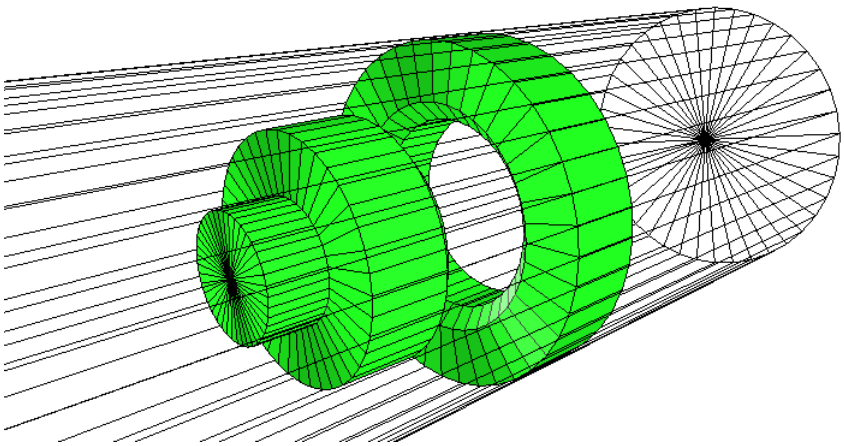


Figure 7.70: Isometric Valve Closed by Axial Plug

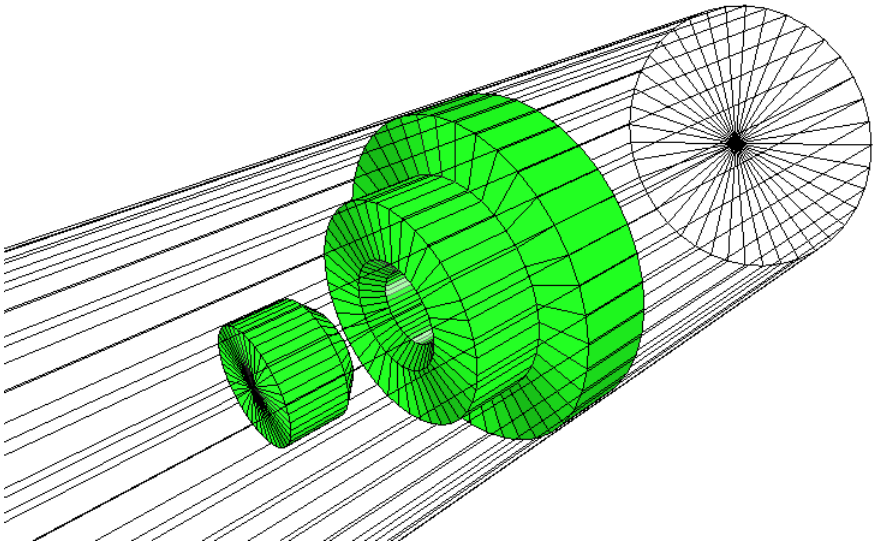


Figure 7.71: Isometric Valve Closed by Annulus Plug

Valve Cross Sections

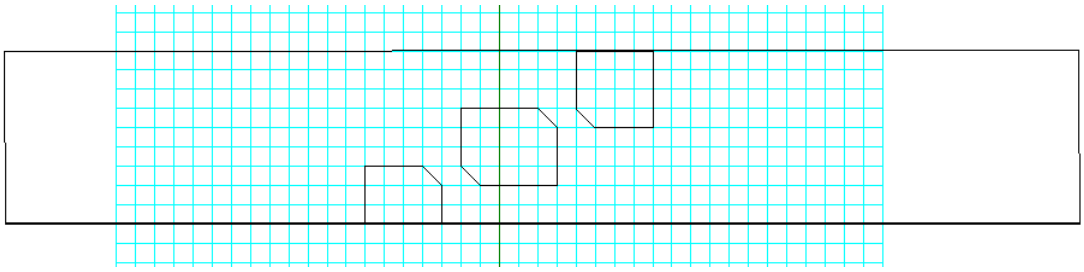


Figure 7.72: Valve Seat at Middle Cross-Section

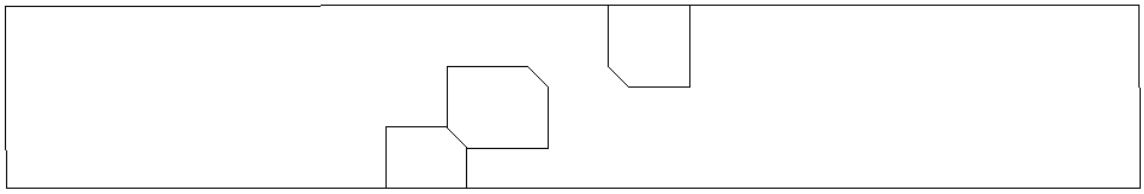


Figure 7.73: Valve Closed by Axial Plug Cross-Section

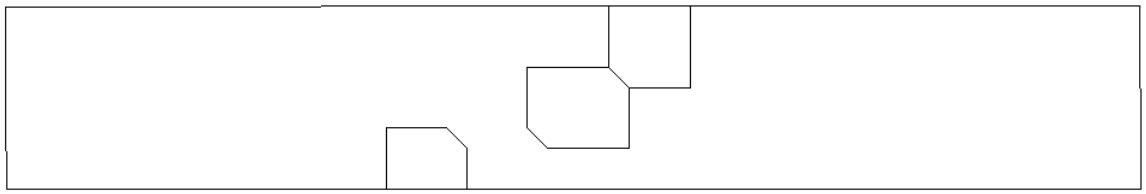


Figure 7.74: Valve Closed by Annulus Plug Cross Section

Valve at Mid-Way

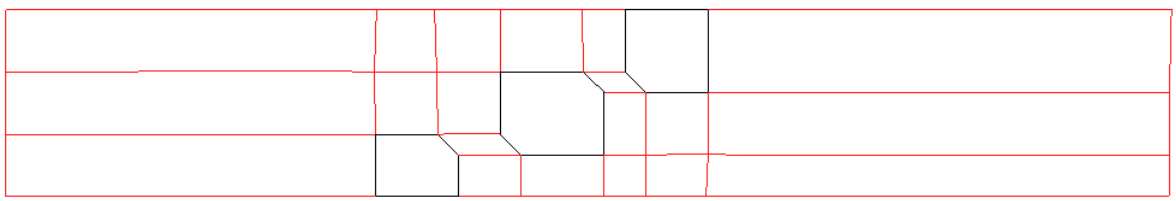


Figure 7.75: Middle Valve Seat Sub-domains

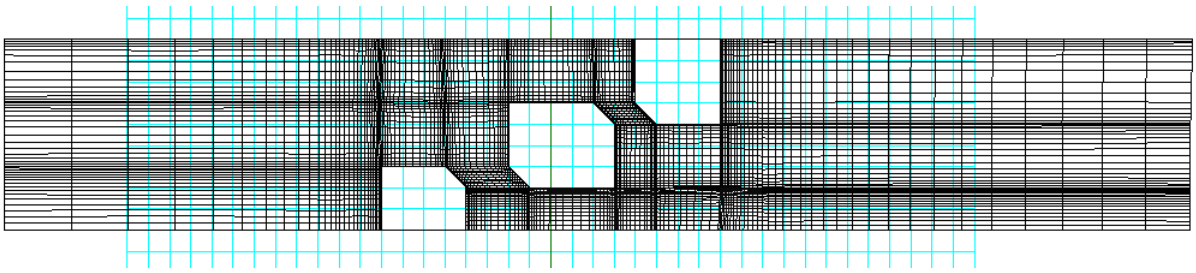


Figure 7.76: Middle Valve Seat Gridded

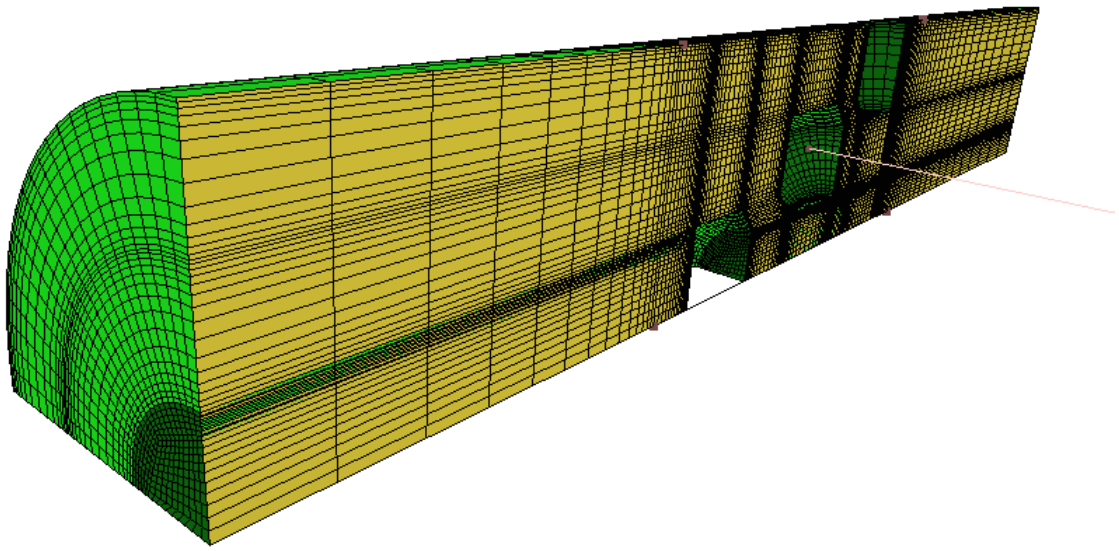


Figure 7.77: Middle Valve Seat Axysimmetric Revolve

Valve Half-Closed by Axial Plug

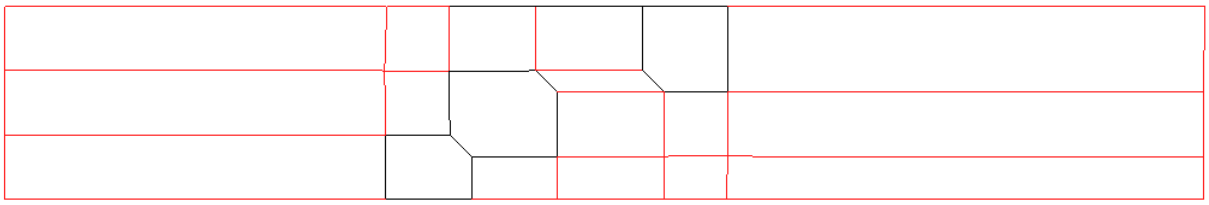


Figure 7.78: Valve Left Axial Plug Sub-domains

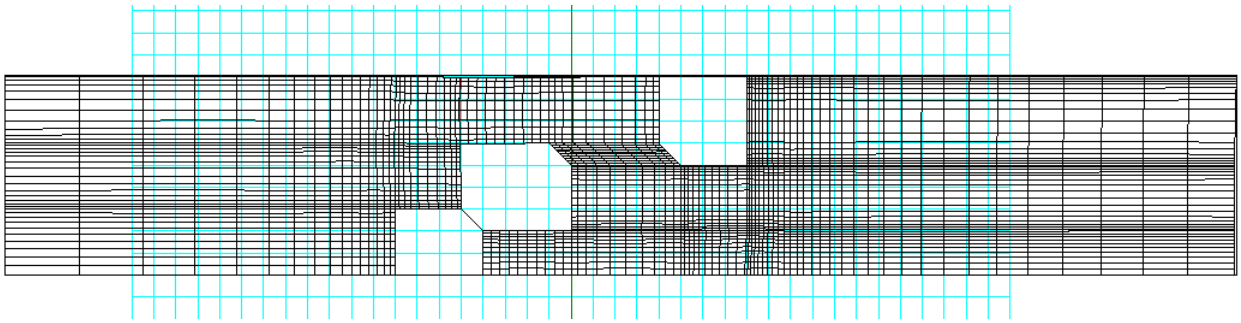


Figure 7.79: Valve Left Axial Plug Gridded

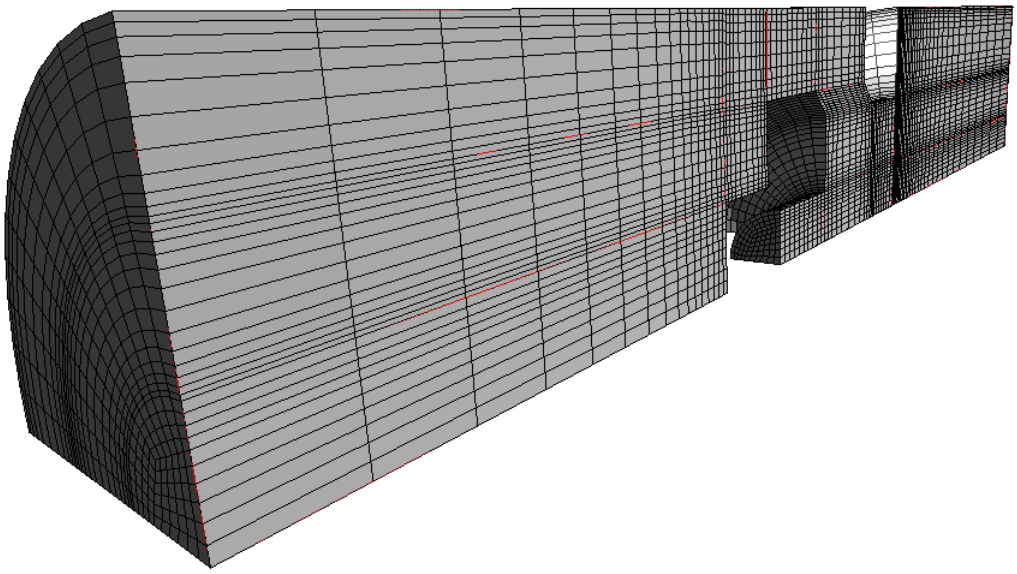


Figure 7.80: Valve Left Axial Plug Axysimmetric Revolve

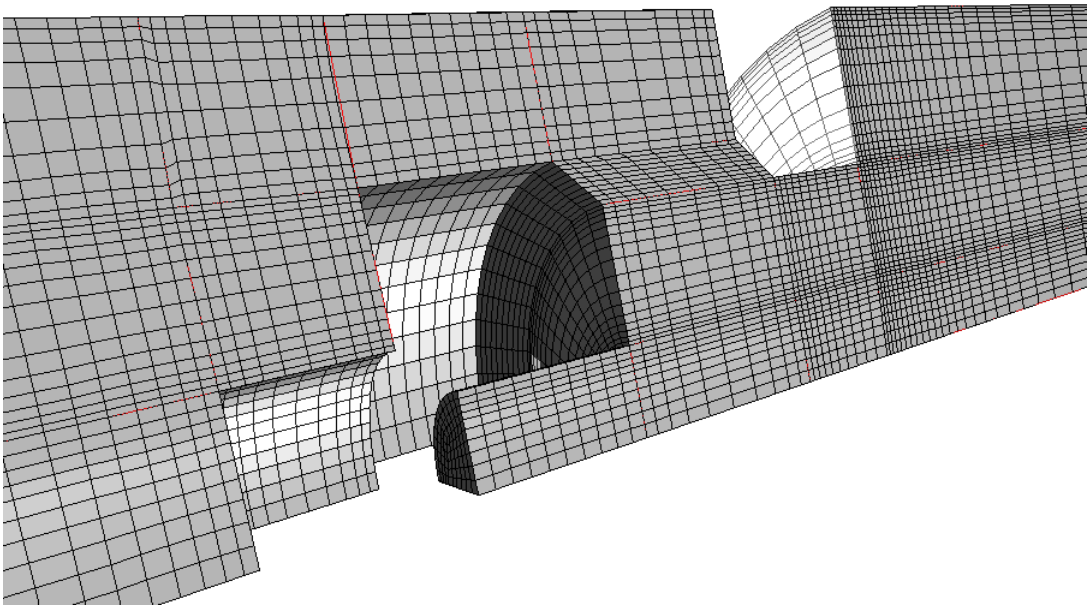


Figure 7.81: Valve Left Axial Plug Close View

Valve Half-Closed by Annulus Plug

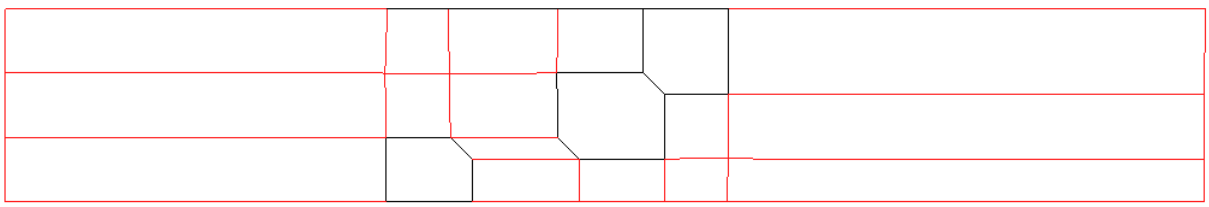


Figure 7.82: Valve Right Annulus Plug Sub-domains

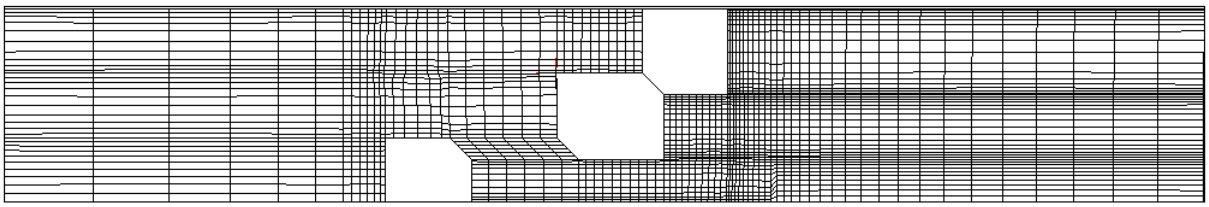


Figure 7.83: Valve Right Annulus Plug Gridded

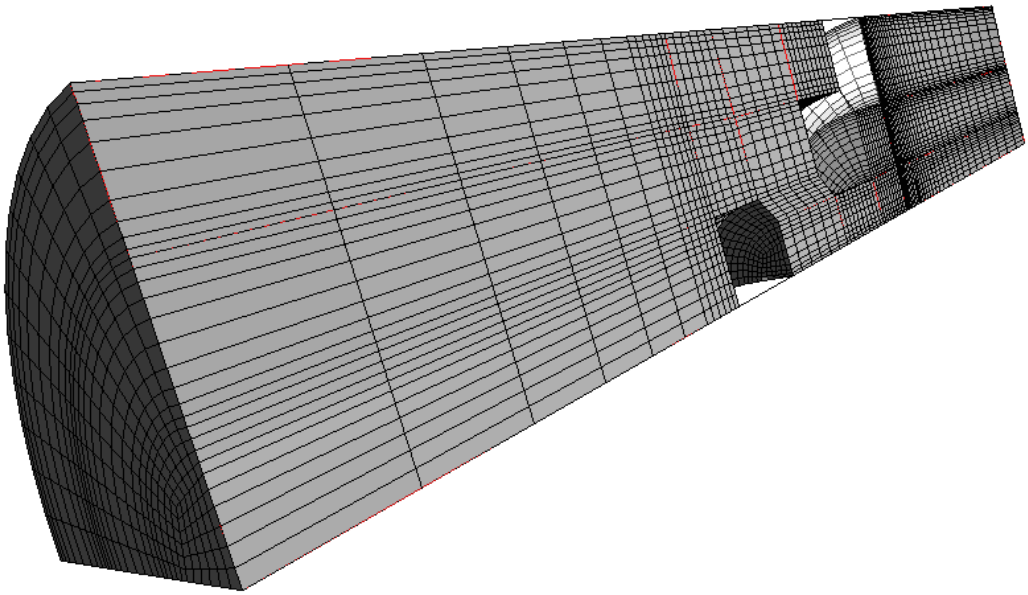


Figure 7.84: Valve Right Annulus Plug Axysimmetric Revolve

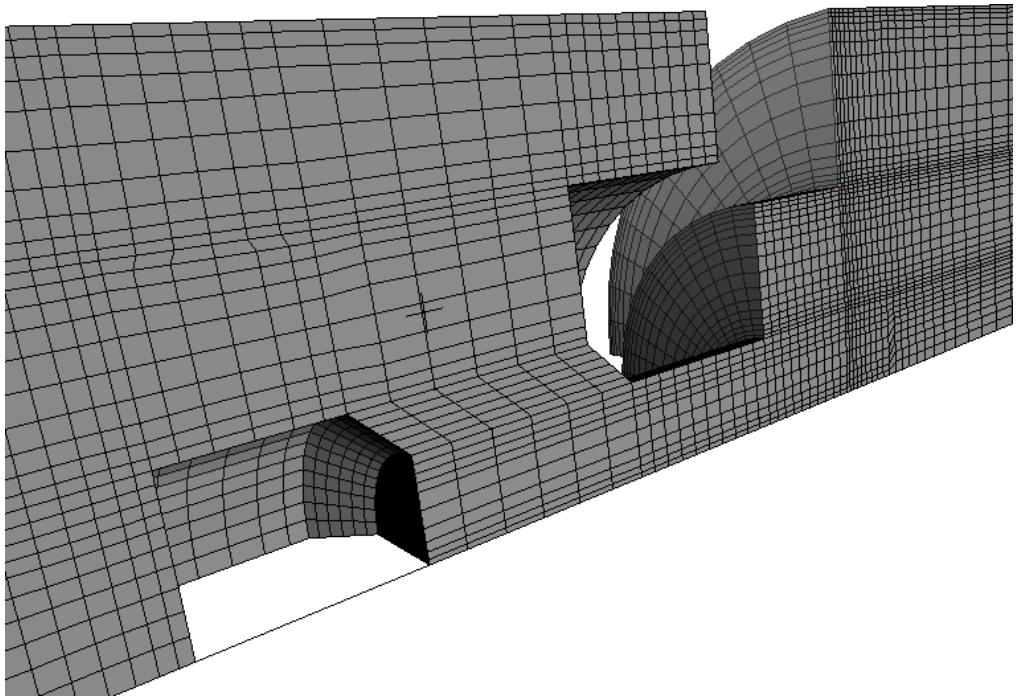


Figure 7.85: Valve Right Annulus Plug Close View

7.3.2.6 Pump rotor

Fig. (7.86) shows the pump rotor cross section to be gridded.

1. Menu Bar -> 2D Gridding -> Activate Edges Snapping Option -> Highlight four edges C-W for each subdomain. Fig. (7.88).
2. Menu Bar -> Viewport -> x-z (Drawing plane transforms to the new location)
3. Menu Bar -> Home -> Hybrid Line -> Draw first line by placing two points, then Press SPACE (Bezier Mode), draw bezier curve, finally straight line. Fig. (7.89).
4. Menu Bar -> Extrusion -> Profile Extrusion Sub Menu -> Solid Extrude with Curves = HybridLineCurveStrip1, and [Treat as Radial Mesh] enabled. Fig.(7.90).

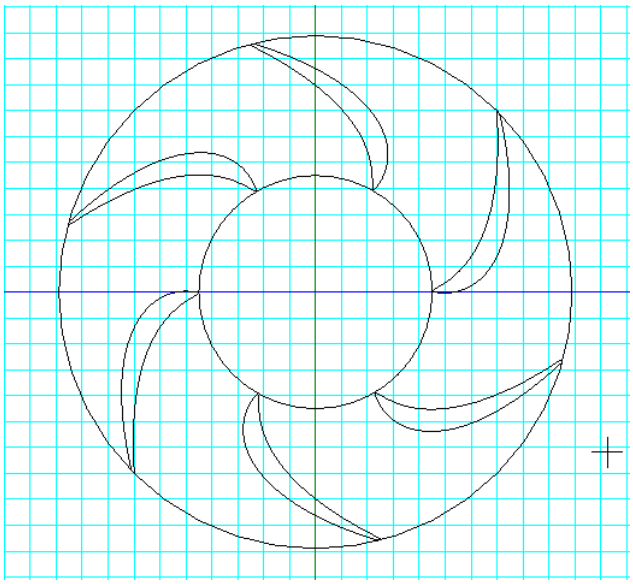


Figure 7.86: Pump Rotor Sketch

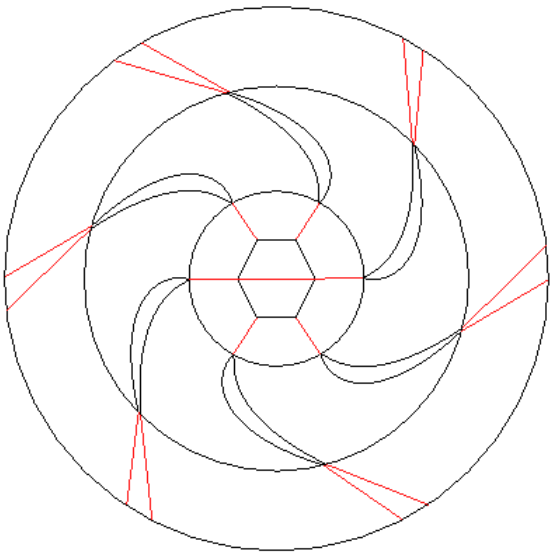


Figure 7.87: Pump Rotor Sub-domains

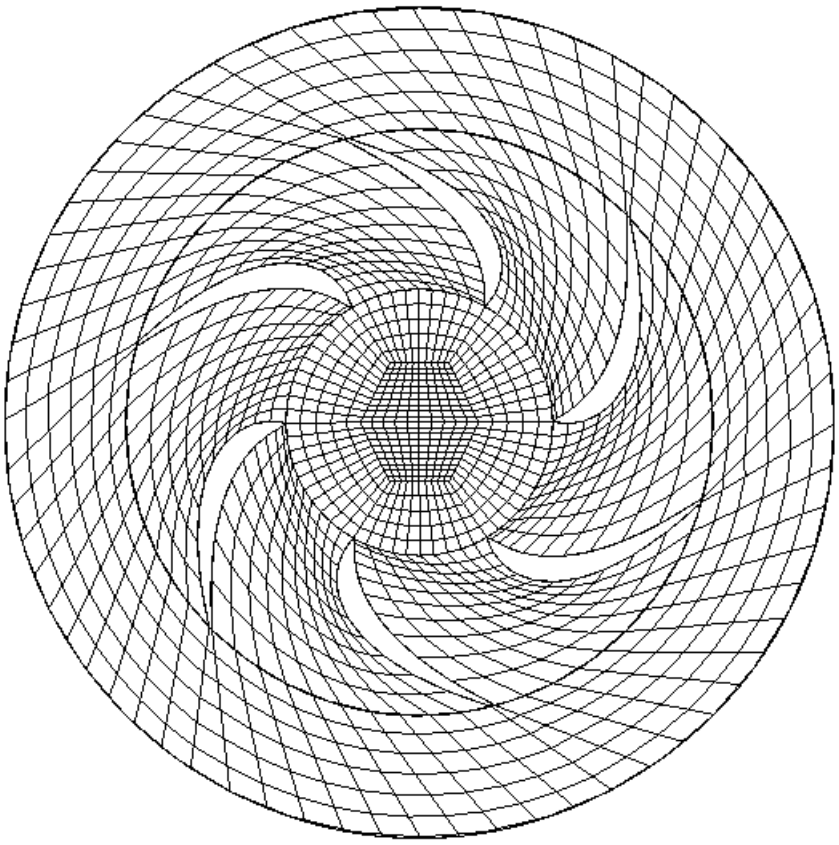


Figure 7.88: Pump Rotor Grid

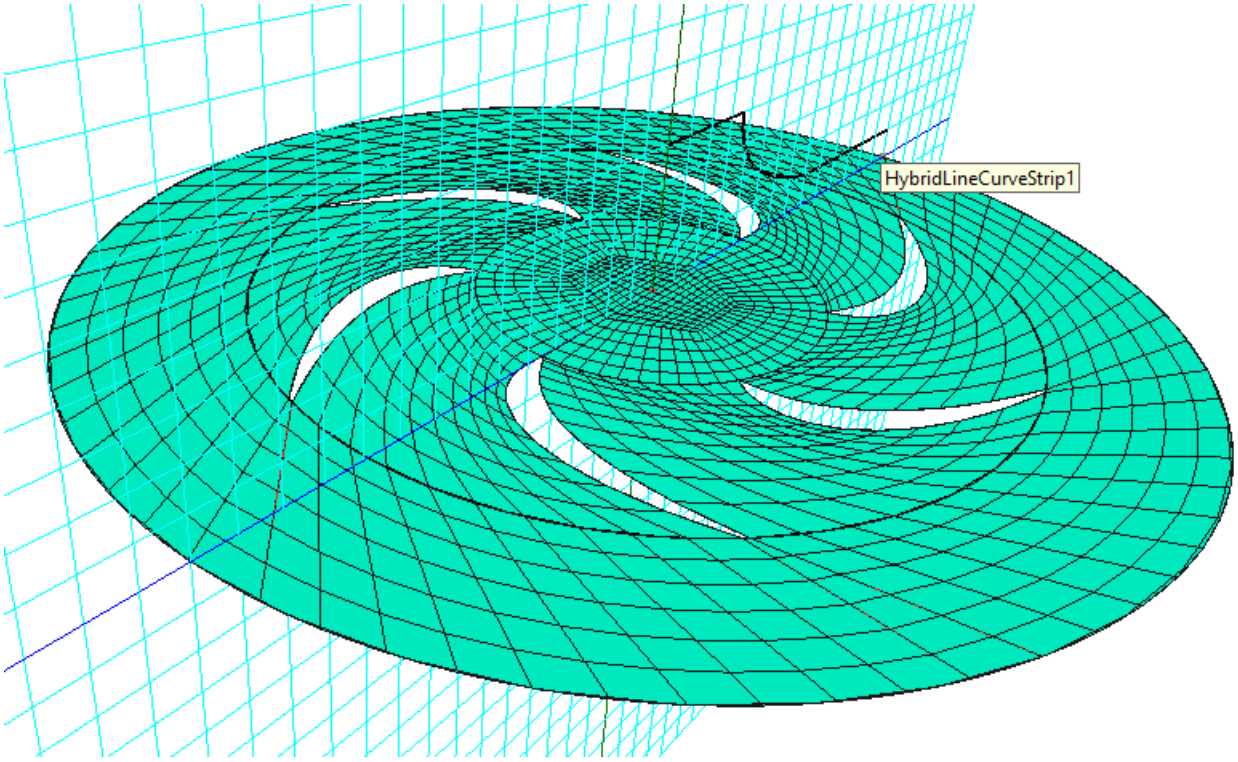


Figure 7.89: Extrude Line Profile

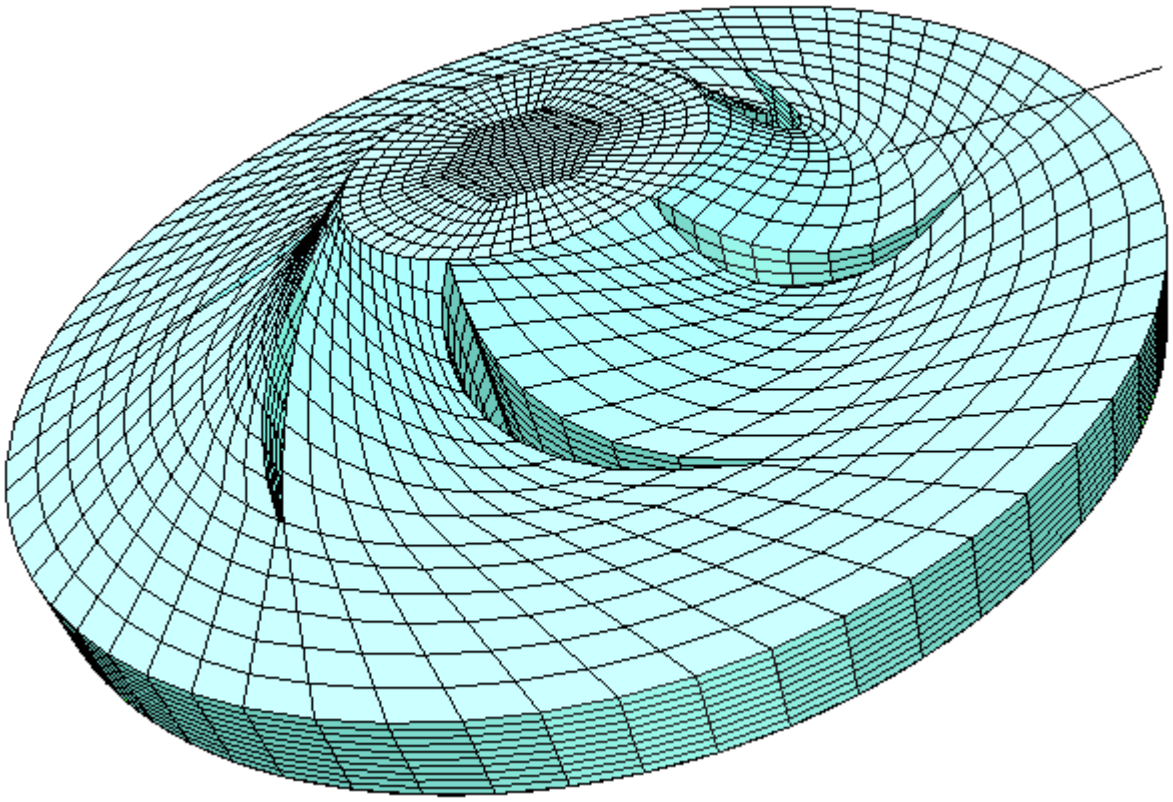


Figure 7.90: Pump Rotor Profile Extrusion

7.4 EXPORTING GRID FILE

The exported file of the generated grid usually includes four elements:

1. Grid Nodes (Cell Vertices).
2. Cell Faces.
3. Grid Cells.
4. Boundary Faces and Boundary Types.

The format of the exported file depends entirely on the format of the target package. In this respect, the developed program has three formats compatible with Paraview® for plotting purposes as well as Star-CD®, OpenFOAM®, and Fluent® preprocessors and solver packages.

Access to the export menu is available through the workspace panel from the Export Functions command Fig. (7.8). Figure (7.91) shows the folder structure for an OpenFOAM case. The Generated or Exported Grid Information section contains separate files for grid nodes (points), cells, faces, and boundaries. For details on the format of the various files dealing with the grid information, the reader may refer to [62].

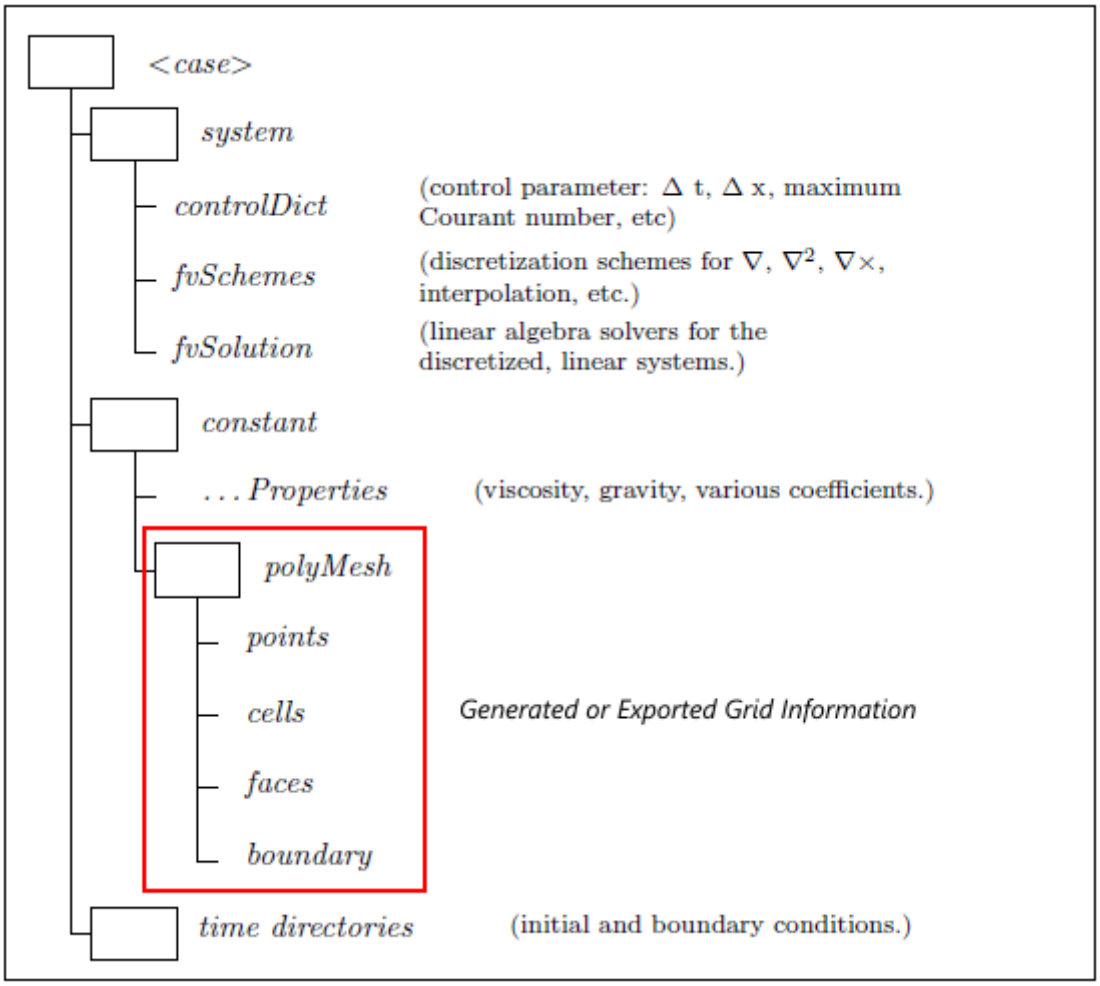


Figure 7.91: OpenFOAM Case Structure[62]

CHAPTER VIII

CONCLUSIONS AND RECOMMENDATIONS

8.1 CONCLUSIONS

From the work presented throughout the thesis, and in conjuction with objective detailed in chapter I, the following conclusions may be drawn:

1. Software Package has been developed and successfully used to generate quadrilateral and hexahedral grids for CFD applications in flow domains of varying complexities.

2. The program relies on visual interaction with the user and a sophisticated graphics library and menu driven commands to create the desired grid.

3. The approach adopted for generating the grid relies on the block structured concept where the domain is divided into a number of sub-domains contained within four curves each. When the user selects the four intersection points of the sub-domain curves, a structured grid appears in that sub-domain or block and connects itself with the neighbouring block grids if existed.

4. Case studies with variable degrees of geometrical complexity were carried out using the developed package. The sequence of grid generation operations demonstrated the package attributes intended at the design stage namely agility, flexibility, powerful control, and ease of use.

8.2 RECOMMENDATIONS FOR FUTURE WORK

1. Implementation of the coordinate transformation of surfaces that enables the calculations of geometric properties of the grid sub-domains using metric tensor and associated curvilinear operators. This is expected to enhance the speed and accuracy of grid generation especially when handling sizeable grids.

2. Employing the hyperbolic grid generation techniques to enhance smoothness of the grid especially at the boundaries of structured sub-domains.

3. Extension of the program to intersecting three-dimensional bodies (e.g. two intersecting cylinders).

4. Extending the package ability to export the grid data to files compatible with more CFD packages formats.

5. Writing extensive manual for the software package.

LIST OF REFERENCES

[1] M. Peric. *A Finite Volume Method for The Prediction of Three-Dimensional Fluid Flow in Complex Ducts. PhD thesis.* PhD thesis, University of London, 1985.

[2] H. K. Versteeg and W. Malalasekera. *An introduction to computational fluid dynamics.* Harlow, 1995.

[3] R. J. LeVeque. *Finite volume methods for hyperbolic problems.* Cambridge: Cambridge University Press, 2002.

[4] Tim Tilford. Computational fluid dynamics. `http://ammsc.gre.ac.uk/coursinf/cfd_exp.html`, 2013.

[5] Hadi Mohammadi. Flow structure around the bileaflet heart valve prostheses. `http://www.hadi-mohammadi-ubc.com`, 2009.

[6] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow.* New York: McGRAW-HILL, 1980.

[7] I. A. Demirdzic. *A Finite Volume Method for Computation of Fluid Flow in Complex Geometris. PhD thesis.* PhD thesis, University of London, 1982.

[8] NASA NTRS. Cfd vision 2030 study: A path to revolutionary computational aerosciences. Technical report, NASA, 2014.

[9] W. R. Buell and B. A. Bush. Mesh generation - a survey. *ASME*, 72-WA/DE-2:332–338, 1972.

[10] V. N. Kaliakin. A simple coordinate determination scheme for two-dimensional mesh generation. *Computers & Structures*, 43(3):505–515, 1992.

[11] R. Haber, M. S. Shephard, J. F. Abel, R. H. Gallagher, and D. P. Greenberg. A general two-dimensional, graphical finite element perprocessor utilizing discrete transfinite mappings. *International Journal for Numerical Methods in Engineering*, 17:1015–1044, 1981.

[12] K. L. Lin and H. J. Shaw. Two-dimensional orhthogonal grid generation techniques. *Compouters & Structures*, 41(4):569–583, 1991.

[13] M. Montgomery and S. Fleeter. A locally analytic technique applied to grid generation by elliptic equations. *International Journal for Numerical Methods in Engineering*, 38:421–432, 1995.

[14] J. M. Tembulkar and B. W. Hanks. On generating quadrilateral elements from a triangular mesh. *Computer & Structures*, 42(4):665–667, 1992.

[15] S. H. Lo. Generating quadrilateral elements on plane and over curved surfaces. *Computers & Structures*, 31(3):421–426, 1989.

[16] J. A. Talbert and A. R. Parkinson. Development of an automatic, two-dimensional finite element mesh generator using quadrilateral elements and bezier curve boundary definition. *International Journal for Numerical Methods in Engineering*, 29:1551–1567, 1990.

[17] T. K. H. Tam and C. G. Armstrong. 2d finite element mesh generation by medial axis subdivision. *Advances in Engineering Software*, 13(5/6):313–324, 1991.

[18] T. D. Blacker, M. B. Stephenson, J. L. Mitchiner, L. R. Phillips, and Y. T. Lin. Automated quadrilateral mesh generation: A knowledge system approach. *ASME*, 88-WA/CIE-4:0, 1988.

[19] C. S. Krishnamoorth, B. Raphael, and S. Mukherjee. Meshing by successive superelement decomposition (msd) - a new approach to quadrialteral mesh generation. *Finite Element in Analysis and Design*, 20:1–37, 1995.

[20] J. Z. Zhu, O. C. Zienkiewicz, E. Hinton, and J. Wu. A new approach to the development of automatic quadrilateral mesh generation. *International Journal for Numerical Methods in Engineering*, 32:849–866, 1991.

[21] T. D. Blacker and M. B. Stephenson. Paving: A new approach to automated quadrilateral mesh generation. *International Journal for Numerical Methods in Engineering*, 32:811–847, 1991.

[22] T. D. Blacker, J. Jung, and W. R. Witkowski. An adaptive finite element technique using element equilibrium and paving. *ASME*, 90-WA/CIE-2:0, 1990.

[23] T. D. Blacker, M. B. Stephenson, and S. Canann. Analysis automation with paving: A new quadrilateral meshing technique. *Advances in Engineering Software*, 13(5/6):332–337, 1991.

[24] T. D. Blacker and M. B. Stephenson. Paving: A new approach to automatic quadrilateral mesh generation. *Sandia National Laboratories*, SAND-90-0249:0, 1990.

[25] V. D. Liseikin. *Grid generation methods (2nd ed.)*. Dordrecht: Springer, 2010.

[26] P. Knupp and S. Steinberg. *Fundmentals of Grid Generation*. CRC Press, Boca Raton, 1994.

[27] M. Farrashkhalvat. and J. P. Miles. *Basic structured grid generation with an introduction to unstructured grid generation*. Oxford: Butterworth Heinemann, 2003.

[28] J. F. Thompson, Z. U. A. Warsi, and C. W. Mastin. *Numerical Grid Generation - Foundations and Applications*. North-Holland, New York, 1985.

[29] Douglas Alan Schwer and Philip Buelow. Diagonalized upwind navier stokes code. `http://duns.sourceforge.net/tutorial3.html`, 2013.

[30] LiU. Research at computational mathematics. `http://www.liu.se/mai/bm/forskning%3Fl=en`, 2014.

[31] CST. Overset grid assembly process. `http://celeritassimtech.com/%3Fpage_id=113`, 2011.

[32] Pointwise. Aerodynamic optimization with pointwise and friendship-framework. `http://blog.pointwise.com/2014/06/09/aerodynamic-optimization-with-pointwise-and-friendship-framework/`, 2014.

[33] W. M. Chan and J. L. Steger. Enhancements of a three-dimensional hyperbolic grid generation scheme. *Applied Mathematics and Computation*, 51:181–205, 1992.

[34] Paul L. George. *Automatic Mesh Generation - Application to Finite Element Methods*. Wiley, 1991.

[35] P. A. F. Martins and M. J. M. B. Marques. Model3 - a three-dimensional mesh generator. *Computers & Structures*, 42(2):511–529, 1992.

[36] S. C. S. Antonio. *On the Generation of Quadrilateral Element Meshes for General CAD Surfaces*. PhD thesis, Massachusetts Institute of Technology, 1996.

[37] William D. Henshaw. Automatic grid generation. *Acta Numerica*, 5:121–148, January 1996.

[38] P. L. Baehmann, S. L. Wittchen, M. S. Shephard, K. R. Grice, and M. A. Yerry. Robust, geometrically based, automatic two-dimensional mesh generation. *International Journal for Numerical Methods in Engineering*, 24:1043–1078, 1987.

[39] A. Tezuka. Adaptive process with quadrilateral finite elements. *Advances in Engineering Software*, 15:185–201, 1992.

[40] F. Cheng, J. W. Jaromczyk, J. R. Lin, S. S. Chang, and J. Y. Lu. A parallel mesh generation algorithm based on vertex label assignment scheme. *International Journal for Numerical Methods in Engineering*, 28:1429–1448, 1989.

[41] L. T. Souza and M. Gattass. A new scheme for mesh generation and mesh refinement using graph theory. *Computers & Structures*, 46(6):1073–1084, 1993.

[42] G. Xie and J. A. H. Ramaekers. Graded mesh generation and transformation. *Finite Element in Analysis and Design*, 17:41–55, 1993.

[43] B. P. Johnston, J. M. Sullivan Jr., and A. Kwasnik. Automatic conversion of triangular finite element meshes to quadrilateral elements. *International Journal for Numerical Methods in Engineering*, 31:67–84, 1991.

[44] E. Rank, M. Scheweingruber, and M. Sommer. Adaptive mesh generation and transformation of triangular to quadrilateral elements. *Communications in Numerical Methods in Engineering*, 9:121–129, 1993.

[45] G. R. Liu. *Mesh Free Methods: Moving Beyond the Finite Element Method*. CRC Press, 2003.

[46] G. R. Liu and Y. T. Gu. *An Introduction to Meshfree Methods and Their Programming*. Springer, 2005.

[47] J. J. Monaghan and R. A. Gingold. Shock simulation by the particle method sph. *Journal of Computational Physics*, 52:374–389, 1983.

[48] A. J. Katz. *Meshless methods for computational fluid dynamics*. Stanford University, 2009.

[49] A. J. Ferreira. *Progress on meshless methods*. Dordrecht: Springer, 2009.

[50] M. Griebel. *Meshfree methods for partial differential equations*. New York: Springer, 2010.

[51] J. K. Aaron. *MESHLESS METHODS FOR COMPUTATIONAL FLUID DYNAMICS*. PhD thesis, STANFORD UNIVERSITY, 2009.

[52] D. B. Spalding and S. V. Patankar. *Numerical prediction of flow, heat transfer, turbulence, and combustion: selected works of Professor D. Brian Spalding*. Oxford: Pergamon Press, 1983.

[53] C. Berge. *Topological Spaces Including a Treatment of Multi-Valued Functions, Vector Spaces and Convexity*. New York: Dover, 1997.

[54] G. E. Bredon. *Topology & Geometry*. New York: Springer-Verlag, 1995.

[55] G. P. Collins. The shapes of space. *Sci. Amer*, 291:94–103, July 2004.

[56] Wikipedia. Homotopie. `http://fr.wikipedia.org/wiki/Homotopie`, 2009.

[57] Wikipedia. Bezier curve. `http://en.wikipedia.org/wiki/B%C3%A9zier_curve`, 2014.

[58] Aleksas Riskus. Approximation of a cubic bezier curve by circular arcs and vice versa. *INFORMATION TECHNOLOGY AND CONTROL*, 35(4):1, 2006.

[59] D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 2.1 (6th ed.)*. NJ: Addison-Wesley, 2008.

[60] S. R. Buss. *3D computer graphics a mathematical introduction with OpenGL*. Cambridge: Cambridge University Press, 2003.

[61] J. Cernecky and K. Plandorova. The effect of the introduction of an exit tube on the separation efficiency in a cyclone. *Brazilian Journal of Chemical Engineering*, 30:627–641, July/Sept 2013.

[62] OpenFOAM. Openfoam. `http://www.openfoam.com/`, 2004.

APPENDIX A

REVIEW OF DIFFERENTIAL AND INTEGRAL CALCULUS IN GENERAL CURVILINEAR COORDINATES

A.1 INTRODUCTION

In geometry, curvilinear coordinates are a coordinate system for Euclidean space in which the coordinate lines may be curved. These coordinates may be derived from a set of Cartesian coordinates by using a transformation that is locally invertible (a one-to-one map) at each point. This means that one can convert a point given in a Cartesian coordinate system to its curvilinear coordinates and back. The name curvilinear coordinates, coined by the French mathematician Lamé, derives from the fact that the coordinate surfaces of the curvilinear systems are curved.

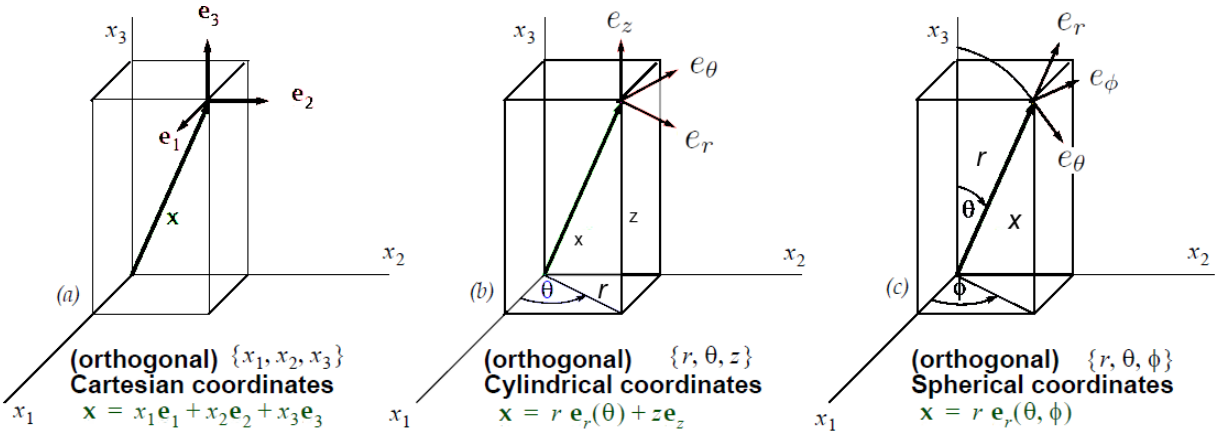


Figure A.1: Common Coordinate Systems.

Well-known examples of curvilinear systems are Cartesian, cylindrical and spherical polar coordinates, for R3, where R is the 3D space of real numbers.

Note that all three systems are orthogonal because the associated base vectors are mutually perpendicular. The cylindrical and spherical coordinate systems are inhomogeneous because the base vectors vary with position. As indicated $\mathbf{e}_r$, depends on θ for cylindrical coordinates and $\mathbf{e}_r$ depends on both θ and Φ for spherical coordinates.

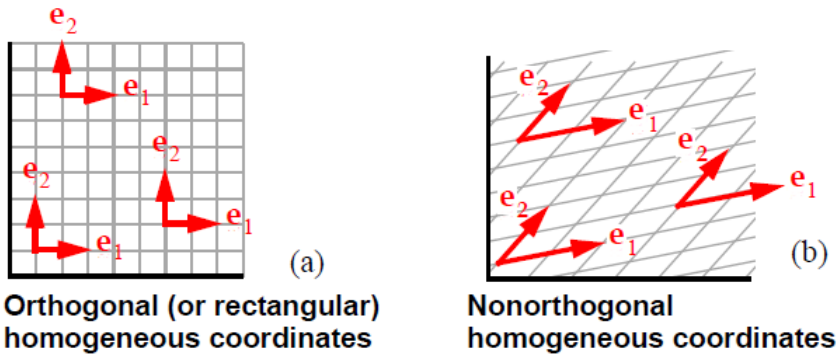


Figure A.2: Homogeneous coordinates.

A coordinate system is called "homogeneous" if the associated base vectors are the same throughout space. A basis is "orthogonal" (or "rectangular") if the base vectors are everywhere mutually perpendicular. Most authors use the term "Cartesian coordinates" to refer to the conventional orthonormal homogeneous right-handed system of Fig. (A.2). As seen in Fig. (A.3), a homogeneous system is not required to be orthogonal. Furthermore, no coordinate system is required to have unit base vectors. The opposite of homogeneous is "curvilinear," and Fig. (A.3) below shows that a coordinate system can be both curvilinear and orthogonal. In short, the properties of being "orthogonal" or "homogeneous" are independent (one does not imply or exclude the other).

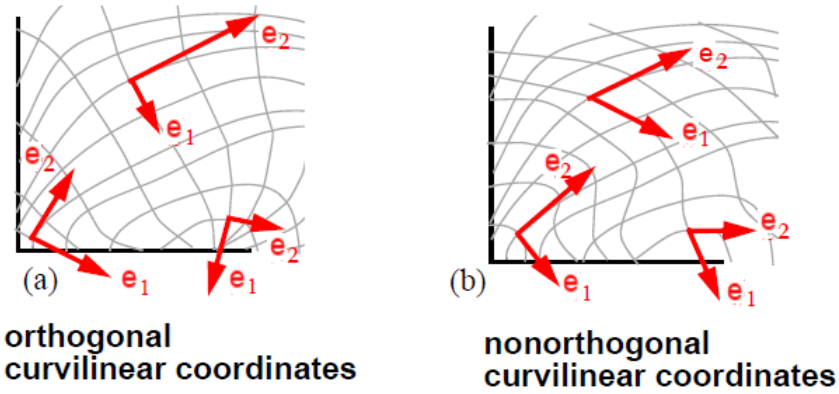


Figure A.3: Orthogonal and Non-Orthogonal Curvilinear Coordinates.

A.2 CURVILINEAR COORDINATES

The coordinate grid is the family of lines along which only one coordinate varies. If the grid has at least some curved lines, the coordinate system is called "curvilinear," and, as shown in Fig. (A.3), the associated base vectors (tangent to the grid lines) necessarily change with position, so curvilinear systems are always inhomogeneous. The system in Fig. (A.3)a has base vectors that are everywhere orthogonal, so it is simultaneously curvilinear and orthogonal. Note from Fig. (A.1) that conventional cylindrical and spherical coordinates are both orthogonal and curvilinear. Incidentally, no matter what type of coordinate system is used, base vectors need not be of unit length; they only need to point in the direction that

the position vector would move when changing the associated coordinate, holding others constant. We will call a basis "regular" if it consists of a right-handed orthonormal triad. The systems in Fig. (A.3) have irregular associated base vectors. The system in Fig. (A.3) can be "regularized" by normalizing the base vectors. Cylindrical and spherical systems are examples of regularized curvilinear systems.

A.2.1 Base Vectors

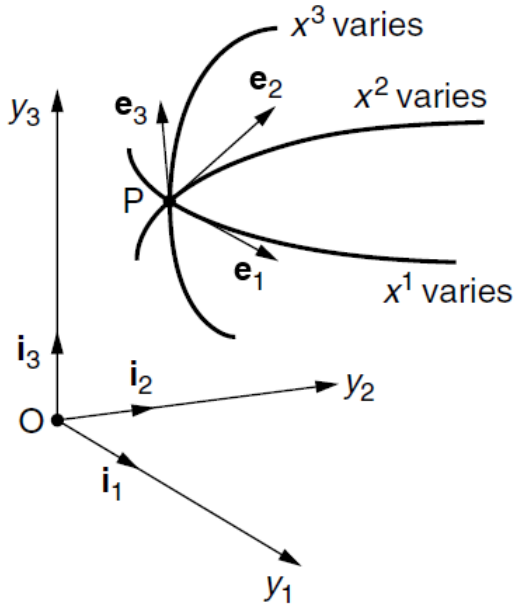


Figure A.4: Covariant base vectors at point P in three dimensions.[27]

The position vector $\mathbf{r}$ of a point $\mathbf{P}$ in space with respect to some origin O may be expressed as

$$\mathbf{r} = y_1 i_1 + y_2 i_2 + y_3 i_3 \tag{1.1}$$

where $\{\mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3\}$, alternatively written as $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$, are unit vectors in the direction of the rectangular cartesian axes. Assuming an invertible relationship between set of cartesian coordinates and set of curvilinear coordinates, i.e.

$$y_i = y_i\left(x^1, x^2, x^3\right), \quad i = 1, 2, 3. \tag{1.2}$$

with the inverse relationship

$$x^i = x^i\left(y_1, y_2, y_3\right), \quad i = 1, 2, 3. \tag{1.3}$$

A.2.1.1 Transformation Equation

We will refer to any curvilinear coordinates with x^i that represent ρ, θ, z in cylindrical or ρ, θ, ϕ in spherical

where i = 1,2,3.

Cylindrical Equation Transformation

Representing Cylindrical From Cartesian Coordinates

$$\mathbf{r} = r(x, y, z)\,\mathbf{e}_r + \theta(x, y, z)\,\mathbf{e}_\theta + z\,\mathbf{e}_z \tag{1.4}$$

$$\mathbf{r} = \sqrt{x^2 + y^2}\,\mathbf{e}_r + \tan^{-1}\left(\frac{y}{x}\right)\,\mathbf{e}_\theta + z\,\mathbf{e}_z \tag{1.5}$$

Then the inverse is

$$\mathbf{r} = x\,(r, \theta, z)\;\mathbf{i} + y\,(r, \theta, z)\;\mathbf{j} + z\,(r, \theta, z)\;\mathbf{k} \tag{1.6}$$

$$\mathbf{r} = r\cos(\theta)\;\mathbf{i} + r\sin(\theta)\;\mathbf{j} + z\,\mathbf{k} \tag{1.7}$$

A.2.1.2 Covariant Base Vectors

Covariant Base Vectors can be obtained from the transformation from Curvilinear to Cartesian Coordinates.

$$\mathbf{e}_i = \frac{\partial \mathbf{r}}{\partial x^i} \tag{1.8}$$

$$\mathbf{e}_i = \frac{\partial y_1}{\partial x^j} + \frac{\partial y_2}{\partial x^j} + \frac{\partial y_3}{\partial x^j} \tag{1.9}$$

$$(\mathbf{e}_i)_j = \frac{\partial y_j}{\partial x^i} \tag{1.10}$$

Where y_i components of cartesian transformation vector

A.2.1.3 Contravariant Base Vectors

Contravariant Base Vectors can be obtained from the transformation from Cartesian to Curvilinear Coordinates.

$$\mathbf{e}^i = \frac{\partial x^i}{\partial \mathbf{r}} \tag{1.11}$$

$$\mathbf{e}^i = \frac{\partial x^i}{\partial y_1} + \frac{\partial x^i}{\partial y_2} + \frac{\partial x^i}{\partial y_3} \tag{1.12}$$

$$(\mathbf{e}^i)_j = \frac{\partial x^i}{\partial y_j} \tag{1.13}$$

A.2.1.4 Jacobian Matrix

the Jacobian **J** can be computed by taking the determinant of the Cartesian transformation tensor or by simply taking the triple scalar product of the covariant base vectors, whichever method is more convenient.

$$\mathbf{J} = \mathbf{e}_1 \cdot (\mathbf{e}_2 \times \mathbf{e}_3) \tag{1.14}$$

A.2.1.5 Gradient Operator in General Curvilinear Coordinates

$$\nabla = \mathbf{i}_k \frac{\partial}{\partial y_k} = \mathbf{i}_1 \frac{\partial}{\partial y_1} + \mathbf{i}_2 \frac{\partial}{\partial y_2} + \mathbf{i}_3 \frac{\partial}{\partial y_3} \tag{1.15}$$

$$\nabla x^i = \mathbf{i}_k \frac{\partial x^i}{\partial y_k} = \mathbf{i}_i \frac{\partial x^i}{\partial y_1} + \mathbf{i}_2 \frac{\partial x^i}{\partial y_2} + \mathbf{i}_3 \frac{\partial x^i}{\partial y_3} = \mathbf{e}^i \tag{1.16}$$

Gradient for any scalar value

$$\nabla \varphi = \mathbf{i}_k \frac{\partial \varphi}{\partial y_k} \tag{1.17}$$

multiply this by components of curvilinear components $\frac{\partial x^j}{\partial x^j}$

$$\nabla \varphi = \mathbf{i}_k \frac{\partial \varphi}{\partial x^j} \frac{\partial x^j}{\partial y_k} = \mathbf{e}^j \frac{\partial \varphi}{\partial x^j} \tag{1.18}$$

A.2.1.6 Representaion of Vector Components

Because of curvilinear coordinates and dual basis behavior we are able to describe the vector $\mathbf{r}$ with reference to these base vectors.

$$\mathbf{r} = r^1\mathbf{e}_1 + r^2\mathbf{e}_2 + r^3\mathbf{e}_3 = r^i\mathbf{r}_i \tag{1.19}$$

$$\mathbf{r} = r_1\mathbf{e}^1 + r_2\mathbf{e}^2 + r_3\mathbf{e}^3 = r_i\mathbf{e}^i \tag{1.20}$$

$$\mathbf{r} = r^i\mathbf{e}_i = r_i\mathbf{e}^i \tag{1.21}$$

Which means Contravriant Components of vector is represented based on Covariant Base Vectors and vice versa.

Later in this chapter see A.2.3 this formation of the equation will be very important when getting chrstoffel symbols.

A.2.2 Metric Tensor

Covariant Metric Tensor

$$g_{ij} = \mathbf{e}_i \cdot \mathbf{e}_j \tag{1.22}$$

When the space is Euclidean, the base vectors can be expressed as linear combinations of the underlying orthonormal laboratory basis and the above set of dot products can be computed using the ordinary orthonormal basis formulas.[1]

$$g_{ij} = \frac{\partial y_k}{\partial x^i}\frac{\partial y_k}{\partial x^j} \tag{1.23}$$

$$g_{ij} = \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \tag{1.24}$$

[1]Note: If the space is not Euclidean, then an orthonormal basis does not exist, and the metric coefficients g_{ij} must be specified a priori. Such a space is called Riemannian. Shell and membrane theory deals with 2D curved Riemannian manifolds embedded in 3D space. The geometry of general relativity is that of a four-dimensional Riemannian manifold.

Contravariant Metric Tensor

$$g^{ij} = \mathbf{e}^i \cdot \mathbf{e}^j \tag{1.25}$$

it is important to

$$\mathbf{e}^i \cdot \mathbf{e}_j = \delta^i_j = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.26}$$

Geometrically, 1.24 requires that the first contravariant base vector $\mathbf{e}^1$ must be perpendicular to both $\mathbf{e}_2$ and $\mathbf{e}_3$, so it must be of the form $\mathbf{e}^1 = \alpha\,(\mathbf{e}_2 \times \mathbf{e}_3)$. The as-yet undetermined scalar α is determined by requiring that $\mathbf{e}^1 \cdot \mathbf{e}_1$ equal unity.

$$\mathbf{e}^1 \cdot \mathbf{e}_1 = [\alpha\,(\mathbf{e}_2 \times \mathbf{e}_3) \cdot \mathbf{e}_1] = \alpha \mathbf{e}_1 \cdot (\mathbf{e}_2 \times \mathbf{e}_3) = \alpha J = 1 \tag{1.27}$$

In the second-to-last step, we recognized the triple scalar product, , to be the Jacobian defined in 1.14. In the last step we asserted that the result must equal unity. Consequently, the scalar is merely the reciprocal of the Jacobian:

$$\alpha = \frac{1}{\mathbf{J}} \tag{1.28}$$

All three contravariant base vectors can be determined similarly to eventually give the final result:

$$\mathbf{e}^1 = \frac{1}{\mathbf{J}}\,(\mathbf{e}_2 \times \mathbf{e}_3)\,, \qquad \mathbf{e}^2 = \frac{1}{\mathbf{J}}\,(\mathbf{e}_3 \times \mathbf{e}_1)\,, \qquad \mathbf{e}^3 = \frac{1}{\mathbf{J}}\,(\mathbf{e}_1 \times \mathbf{e}_2) \tag{1.29}$$

A.2.2.1 Transformation with Metric Tensors

$$r^i = \mathbf{r} \cdot \mathbf{e}_i = g^{ij} r_j \tag{1.30}$$

$$r_i = \mathbf{r} \cdot e^i = g_{ij} r^j \tag{1.31}$$

Metric Tensor can convert the covariant base vector into contravariant base vector and

vice versa as in following equation

$$\mathbf{e}_i = g_{ij}\mathbf{e}^j \tag{1.32}$$

$$\mathbf{e}^i = g^{ij}\mathbf{e}_j \tag{1.33}$$

also convert covariant vector component into contravariant vector component

$$r_i = g_{ij}r^j \tag{1.34}$$

$$r^i = g^{ij}r_j \tag{1.35}$$

A.2.3 Christoffel Symbols

Differentiating the base vectore for the second time give us:

$$\frac{\partial \mathbf{e}_i}{\partial x^j} = \frac{\partial \mathbf{e}_1}{\partial x^j}\,\mathbf{i} + \frac{\partial \mathbf{e}_2}{\partial x^j}\,\mathbf{j} + \frac{\partial \mathbf{e}_3}{\partial x^j}\,\mathbf{k} \tag{1.36}$$

$$\frac{\partial \mathbf{e}_i}{\partial x^j} = \Gamma^1_{ij}\,\mathbf{e}_1 + \Gamma^2_{ij}\,\mathbf{e}_2 + \Gamma^3_{ij}\,\mathbf{e}_3 \tag{1.37}$$

$$\frac{\partial \mathbf{e}_i}{\partial x^j} = \Gamma^k_{ij}\,\mathbf{e}_k \tag{1.38}$$

we can transfer the g_k to the other side with dot product and raising the index

$$\frac{\partial \mathbf{e}_i}{\partial x^j} \cdot \mathbf{e}^k = \Gamma^k_{ij} \tag{1.39}$$

some properties
indexing order in second derivative is “I don’t know”

$$\frac{\partial \mathbf{e}_i}{\partial x^j} = \frac{\partial^2 \mathbf{r}}{\partial x^j \partial x^i} = \frac{\partial^2 \mathbf{r}}{\partial x^i \partial x^j} = \frac{\partial \mathbf{e}_j}{\partial x^i} \tag{1.40}$$

$$\Gamma^k_{ij} = \Gamma^k_{ji} \tag{1.41}$$

its the same as $g_{ij} = g_{ji}$
lets take metric tensor and differentiate it again

$$\frac{\partial g_{ij}}{\partial x^k} = \frac{\partial \mathbf{e}_i}{\partial x^k} \cdot \mathbf{e}_j + \mathbf{e}_i \cdot \frac{\partial \mathbf{e}_j}{\partial x^k} \tag{1.42}$$

$$\frac{\partial g_{ij}}{\partial x^k} = \Gamma^l_{ik}\, \mathbf{e}_l \cdot \mathbf{e}_j + \mathbf{e}_i \cdot \Gamma^l_{jk} \mathbf{e}_l \tag{1.43}$$

however $\mathbf{e}_l \cdot \mathbf{e}_j$ and $\mathbf{e}_i \cdot \mathbf{e}_l$ are metric tensors and equal to each other

$$\frac{\partial g_{ij}}{\partial x^k} = \Gamma^l_{ik}\, g_{lj} + \Gamma^l_{jk} g_{il} \tag{1.44}$$

and this is an important equation because indices can be shuffled CCW

$$\frac{\partial g_{jk}}{\partial x^i} = \Gamma^l_{ji}\, g_{lk} + \Gamma^l_{ki} g_{jl} \tag{1.45}$$

$$\frac{\partial g_{ki}}{\partial x^j} = \Gamma^l_{kj}\, g_{li} + \Gamma^l_{ij} g_{kl} \tag{1.46}$$

Applying this formula make this 1.45+1.46-1.44 give us:

$$\frac{\partial g_{jk}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^k} = 2\Gamma^l_{ij} g_{kl} \tag{1.47}$$

Rest of items removes each other

$$\Gamma^l_{ij} g_{kl} = \frac{1}{2}\left[\frac{\partial g_{jk}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^k}\right] \tag{1.48}$$

knowing that Kroneker Delta can be obtained from multiplying contravariant and covariant metric tensor

$$g_{kl} g^{mk} = \delta^m_l \tag{1.49}$$

Then Christoffel Symbols becomes

$$\Gamma^l_{ij} g_{kl} g^{mk} = \frac{1}{2} g^{mk} \left[\frac{\partial g_{jk}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^k} \right] \tag{1.50}$$

$$\Gamma^l_{ij} \delta^m_l = \Gamma^l_{ij} = \frac{1}{2} g^{mk} \left[\frac{\partial g_{jk}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^k} \right] \tag{1.51}$$

A.2.3.1 Second Kind Christoffel Symbols

$$[ij,\, k] = \frac{\partial \mathbf{e}_i}{\partial x^j} \cdot \mathbf{e}_k \tag{1.52}$$

$$\Gamma^k_{ij} = \frac{\partial \mathbf{e}_i}{\partial x^j} \cdot \boldsymbol{e}^k \tag{1.53}$$

$$\Gamma^k_{ij} = \frac{\partial \mathbf{e}_i}{\partial x^j} \cdot \left(g^{kl} \mathbf{e}_l \right) = g^{kl} \, [ij,\, l] \tag{1.54}$$

$$[ij,\, l] = g_{kl} \Gamma^l_{ij} \tag{1.55}$$

$$[ij,\, l] = \frac{1}{2} \left[\frac{\partial g_{jk}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^k} \right] \tag{1.56}$$

A.2.3.2 Covariant Derivative

$$\frac{\partial \mathbf{r}}{\partial x^j} = \frac{\partial}{\partial x^j} \left(r^i \mathbf{e}_i \right) = \frac{\partial r^i}{\partial x^j} \mathbf{e}_i + r^i \frac{\partial \mathbf{e}_i}{\partial x^j} \tag{1.57}$$

$$\frac{\partial \mathbf{r}}{\partial x^j} = \frac{\partial r^i}{\partial x^j} \mathbf{e}_i + r^i \Gamma^k_{ij} \mathbf{e}_k \tag{1.58}$$

notice that we made index shuffle in christoffel part

$$\frac{\partial \mathbf{r}}{\partial x^j} = \left(\frac{\partial r^i}{\partial x^j} + \Gamma^i_{kj} r^k \right) \mathbf{e}_i \tag{1.59}$$

Covariant derivative of contravariant vector r^i

$$\frac{\partial \mathbf{r}}{\partial x^j} = r^i_{,j}\mathbf{e}_i \tag{1.60}$$

where

$$r^i_{,j} = \frac{\partial r^i}{\partial x^j} + \Gamma^i_{kj} r^k \tag{1.61}$$

Covariant derivative of covariant vector r_i

$$\frac{\partial \mathbf{r}}{\partial x^j} = r_{i,j}\mathbf{e}^i \tag{1.62}$$

where

$$r_{i,j} = \frac{\partial r^i}{\partial x^j} - \Gamma^k_{ij} r_k \tag{1.63}$$

A.2.4 Div, Gradient, and Curl

$$\nabla \cdot \mathbf{r} = e^j \cdot \frac{\partial \mathbf{r}}{\partial x^j} \tag{1.64}$$

$$\nabla \cdot \mathbf{r} = \frac{\partial r^j}{\partial x^j} + \Gamma^j_{jk} r^k \tag{1.65}$$

$$\nabla \cdot \mathbf{r} = \frac{\partial r^j}{\partial x^j} + \frac{1}{\sqrt{g}} \frac{\partial}{\partial x^j} \left(\sqrt{g}\right) r^j \tag{1.66}$$

$$\nabla \cdot \mathbf{r} = \frac{1}{\sqrt{g}} \frac{\partial}{\partial x^j} \left(\sqrt{g} r^j\right) \tag{1.67}$$

lets introduce other form

$$\frac{\Delta}{\Delta x^j}(\cdots) \equiv \frac{1}{\sqrt{g}} \frac{\partial}{\partial x^j} \left[\sqrt{g}\,(\cdots)\right] \tag{1.68}$$

$$\frac{\Delta}{\Delta x^{(j)}}(\cdots) \equiv \sqrt{\frac{g_{jj}}{g}} \frac{\partial}{\partial x^{(j)}} \left[\sqrt{\frac{g}{g_{jj}}}\,(\cdots)\right] \tag{1.69}$$

A.3 ORTHOGONAL CURVILINEAR COORDINATES

Orthogonal coordinates never have off-diagonal terms in their metric tensor. In other words, the infinitesimal squared distance ds^2 can always be written as a scaled sum of the squared infinitesimal coordinate displacements

$$ds^2 = \sum_{k=1}^{d} \left(h_k \, dq^k\right)^2 \tag{1.70}$$

where d is the dimension and the scaling functions (or scale factors)

$$h_k(\mathbf{q}) \overset{\text{def}}{=} \sqrt{g_{kk}(\mathbf{q})} = |\mathbf{e}_k| \tag{1.71}$$

equal the square roots of the diagonal components of the metric tensor, or the lengths of the local basis vectors $\mathbf{e}_k$ described below. These scaling functions hi are used to calculate differential operators in the new coordinates, e.g., the gradient, the Laplacian, the divergence and the curl.

A.3.1 Covariant Basis

In Cartesian coordinates, the basis vectors are fixed (constant). In the more general setting of curvilinear coordinates, a point in space is specified by the coordinates, and at every such point there is bound a set of basis vectors, which generally are not constant: this is the essence of curvilinear coordinates in general and is a very important concept. What distinguishes orthogonal coordinates is that, though the basis vectors vary, they are always orthogonal with respect to each other. In other words $\mathbf{e}_i \cdot \mathbf{e}_j = 0 \quad \text{if} \quad i \neq j$.

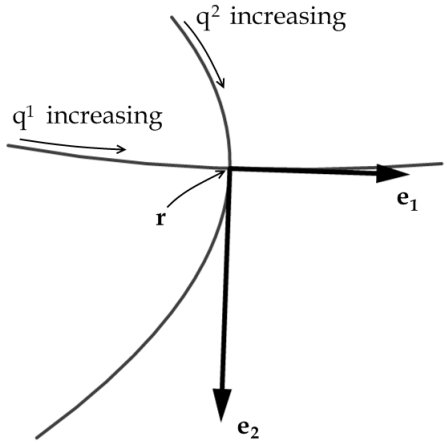


Figure A.5: Orthogonal Coordinates

These basis vectors are by definition the tangent vectors of the curves obtained by varying

one coordinate, keeping the others fixed:

$$\mathbf{e}_i = \frac{\partial \mathbf{r}}{\partial q^i} \tag{1.72}$$

where $\mathbf{r}$ is some point and q^i is the coordinate for which the basis vector is extracted. In other words, a curve is obtained by fixing all but one coordinate; the unfixed coordinate is varied as in a parametric curve, and the derivative of the curve with respect to the parameter (the varying coordinate) is the basis vector for that coordinate.

The vectors are not necessarily of equal length. The normalized basis vectors are notated with a hat and obtained by dividing by the length:

$$\hat{\mathbf{e}}_i = \frac{\mathbf{e}_i}{|\mathbf{e}_i|} \tag{1.73}$$

A vector field may be specified by its components with respect to the basis vectors or the normalized basis vectors, one must be sure which case is dealt. Components in the normalized basis are most common in applications for clarity of the quantities (for example, one may want to deal with tangential velocity instead of tangential velocity times a scale factor); in derivations the normalized basis is less common since it is more complicated.

The useful functions known as scale factors (sometimes called Lamé coefficients, this should be avoided since some more well known coefficients in linear elasticity carry the same name) of the coordinates are simply the lengths of the basis vectors.

A.3.2 Contravariant Basis

The basis vectors shown above are covariant basis vectors (because they "co-vary" with vectors). In the case of orthogonal coordinates, the contravariant basis vectors are easy to find since they will be in the same direction as the covariant vectors but reciprocal length (for this reason, the two sets of basis vectors are said to be reciprocal with respect to each other):

$$\mathbf{e}^i = \frac{\hat{\mathbf{e}}_i}{h_i} = \frac{\mathbf{e}_i}{h_i^2} \tag{1.74}$$

this follows from the fact that, by definition, $\mathbf{e}_i \cdot \mathbf{e}_j = \delta_i^j$, using the Kronecker delta. Note that:

$$\hat{\mathbf{e}}_i = \frac{\mathbf{e}_i}{h_i} = h_i \mathbf{e}^i = \hat{\mathbf{e}}^i \tag{1.75}$$

To avoid confusion, the components of the vector x with respect to the e_i basis are

represented as x^i, while the components with respect to the $\mathbf{e}^i$ basis are represented as x^i:

$$\mathbf{x} = \sum x^i \mathbf{e}_i = \sum x_i \mathbf{e}^i \tag{1.76}$$

The position of the indices represent how the components are calculated (upper indices should not be confused with exponentiation). The components are related simply by:

$$h_i^2 x^i = x_i \tag{1.77}$$

A.3.3 Dot Product

The dot product in Cartesian coordinates (Euclidean space with an orthonormal basis set) is simply the sum of the products of components. In orthogonal coordinates, the dot product of two vectors x and y takes this familiar form when the components of the vectors are calculated in the normalized basis:

$$\mathbf{x} \cdot \mathbf{y} = \sum x_i \hat{\mathbf{e}}_i \cdot \sum y_i \hat{\mathbf{e}}_i = \sum x_i y_i \tag{1.78}$$

This is an immediate consequence of the fact that the normalized basis at some point can form a Cartesian coordinate system: the basis set is orthonormal.

For components in the covariant or contraviant bases,

$$\mathbf{x} \cdot \mathbf{y} = \sum h_i^2 x^i y^i = \sum \frac{x_i y_i}{h_i^2} = \sum x^i y_i = \sum x_i y^i \tag{1.79}$$

This can be readily derived by writing out the vectors in component form, normalizing the basis vectors, and taking the dot product. For example, in 2D:

$$\begin{aligned} \mathbf{x} \cdot \mathbf{y} &= \left(x^1 \mathbf{e}_1 + x^2 \mathbf{e}_2\right) \cdot \left(y_1 \mathbf{e}^1 + y_2 \mathbf{e}^2\right) \\ &= \left(x^1 h_1 \hat{\mathbf{e}}_1 + x^2 h_2 \hat{\mathbf{e}}_2\right) \cdot \left(y_1 \frac{\hat{\mathbf{e}}^1}{h_1} + y_2 \frac{\hat{\mathbf{e}}^2}{h_2}\right) = x^1 y_1 + x^2 y_2 \end{aligned} \tag{1.80}$$

where the fact that the normalized covariant and contravariant bases are equal has been used.

A.3.4 Cross Product

The cross product in 3D Cartesian coordinates is:

$$\mathbf{x} \times \mathbf{y} = (x_2 y_3 - x_3 y_2)\hat{\mathbf{e}}_1 + (x_3 y_1 - x_1 y_3)\hat{\mathbf{e}}_2 + (x_1 y_2 - x_2 y_1)\hat{\mathbf{e}}_3 \tag{1.81}$$

The above formula then remains valid in orthogonal coordinates if the components are calculated in the normalized basis.

To construct the cross product in orthogonal coordinates with covariant or contravariant bases we again must simply normalize the basis vectors, for example:

$$\mathbf{x} \times \mathbf{y} = \sum x^i \mathbf{e}_i \times \sum y^i \mathbf{e}_i = \sum x^i h_i \hat{\mathbf{e}}_i \times \sum y^i h_i \hat{\mathbf{e}}_i \tag{1.82}$$

which, written expanded out,

$$\mathbf{x} \times \mathbf{y} = (x^2 y^3 - x^3 y^2)\frac{h_2 h_3}{h_1}\mathbf{e}_1 + (x^3 y^1 - x^1 y^3)\frac{h_1 h_3}{h_2}\mathbf{e}_2 + (x^1 y^2 - x^2 y^1)\frac{h_1 h_2}{h_3}\mathbf{e}_3 \tag{1.83}$$

Terse notation for the cross product, which simplifies generalization to non-orthogonal coordinates and higher dimensions, is possible with the Levi-Civita tensor, which will have components other than zeros and ones if the scale factors are not all equal to one.

A.3.5 Differentiation

Looking at an infinitesimal displacement from some point, it's apparent that

$$d\mathbf{r} = \sum \frac{\partial \mathbf{r}}{\partial q^i}\, dq^i = \sum \mathbf{e}_i\, dq^i \tag{1.84}$$

By definition, the gradient of a function must satisfy (this definition remains true if f is any tensor)

$$df = \nabla f \cdot d\mathbf{r} \quad \Rightarrow \quad df = \nabla f \cdot \sum \mathbf{e}_i\, dq^i \tag{1.85}$$

It follows then that del operator must be:

$$\nabla = \sum \mathbf{e}^i \frac{\partial}{\partial q^i} \tag{1.86}$$

and this happens to remain true in general curvilinear coordinates. Quantities like the

gradient and Laplacian follow through proper application of this operator.

Line element Tangent vector to coordinate curve q_i:

$$d\boldsymbol{\ell} = h_i \hat{\mathbf{e}}_i = \frac{\partial \mathbf{r}}{\partial q_i} \tag{1.87}$$

Infinitesimal length

$$d\ell = \sqrt{d\mathbf{r} \cdot d\mathbf{r}} = \sqrt{h_1^2\, dq_1^2 + h_2^2\, dq_2^2 + h_3^2\, dq_3^2} \tag{1.88}$$

Surface element Normal to coordinate surface $q_k = constant$:

$$\begin{aligned} d\mathbf{S} &= (h_i q_i \hat{\mathbf{e}}_i) \times (h_j q_j \hat{\mathbf{e}}_j) \\ &= h_i h_j q_i q_j \left(\frac{\partial \mathbf{r}}{\partial q_i} \times \frac{\partial \mathbf{r}}{\partial q_j} \right) \\ &= h_i h_j q_i q_j \hat{\mathbf{e}}_k \end{aligned} \tag{1.89}$$

Infinitesimal surface:

$$dS_k = h_i h_j\, dq^i\, dq^j \tag{1.90}$$

Volume element Infinitesimal volume:

$$\begin{aligned} dV &= |(h_1\, dq_1 \hat{\mathbf{e}}_1) \cdot (h_2\, dq_2 \hat{\mathbf{e}}_2) \times (h_3\, dq_3 \hat{\mathbf{e}}_3)| \\ &= |\hat{\mathbf{e}}_1 \cdot \hat{\mathbf{e}}_2 \times \hat{\mathbf{e}}_3| h_1 h_2 h_3\, dq_1\, dq_2\, dq_3 \\ &= J\, dq_1\, dq_2\, dq_3 \\ &= h_1 h_2 h_3\, dq_1\, dq_2\, dq_3 \end{aligned} \tag{1.91}$$

where

$$J = \left| \frac{\partial \mathbf{r}}{\partial q_1} \cdot \left(\frac{\partial \mathbf{r}}{\partial q_2} \times \frac{\partial \mathbf{r}}{\partial q_3} \right) \right| = \left| \frac{\partial (x, y, z)}{\partial (q_1, q_2, q_3)} \right| = h_1 h_2 h_3 \tag{1.92}$$

is the Jacobian determinant, which has the geometric interpretation of the deformation in volume from the infinitesimal cube $dx\, dy\, dz$ to the infinitesimal curved volume in the orthogonal coordinates.

A.3.6 Differential Operators

$$\nabla\phi = \frac{\hat{\mathbf{e}}_1}{h_1}\frac{\partial\phi}{\partial q^1} + \frac{\hat{\mathbf{e}}_2}{h_2}\frac{\partial\phi}{\partial q^2} + \frac{\hat{\mathbf{e}}_3}{h_3}\frac{\partial\phi}{\partial q^3} \tag{1.93}$$

$$\nabla\cdot\mathbf{F} = \frac{1}{h_1h_2h_3}\left[\frac{\partial}{\partial q^1}\left(F_1h_2h_3\right) + \frac{\partial}{\partial q^2}\left(F_2h_3h_1\right) + \frac{\partial}{\partial q^3}\left(F_3h_1h_2\right)\right] \tag{1.94}$$

$$\begin{aligned}\nabla\times\mathbf{F} = {}& \frac{\hat{\mathbf{e}}_1}{h_2h_3}\left[\frac{\partial}{\partial q^2}\left(h_3F_3\right) - \frac{\partial}{\partial q^3}\left(h_2F_2\right)\right] + \frac{\hat{\mathbf{e}}_2}{h_3h_1}\left[\frac{\partial}{\partial q^3}\left(h_1F_1\right) - \frac{\partial}{\partial q^1}\left(h_3F_3\right)\right] \\ & + \frac{\hat{\mathbf{e}}_3}{h_1h_2}\left[\frac{\partial}{\partial q^1}\left(h_2F_2\right) - \frac{\partial}{\partial q^2}\left(h_1F_1\right)\right] = \frac{1}{h_1h_2h_3}\begin{vmatrix} h_1\hat{\mathbf{e}}_1 & h_2\hat{\mathbf{e}}_2 & h_3\hat{\mathbf{e}}_3 \\ \dfrac{\partial}{\partial q^1} & \dfrac{\partial}{\partial q^2} & \dfrac{\partial}{\partial q^3} \\ h_1F_1 & h_2F_2 & h_3F_3 \end{vmatrix}\end{aligned} \tag{1.95}$$

$$\nabla^2\phi = \frac{1}{h_1h_2h_3}\left[\frac{\partial}{\partial q^1}\left(\frac{h_2h_3}{h_1}\frac{\partial\phi}{\partial q^1}\right) + \frac{\partial}{\partial q^2}\left(\frac{h_3h_1}{h_2}\frac{\partial\phi}{\partial q^2}\right) + \frac{\partial}{\partial q^3}\left(\frac{h_1h_2}{h_3}\frac{\partial\phi}{\partial q^3}\right)\right] \tag{1.96}$$

APPENDIX B

GEOMETRY AND OBJECT ORIENTED PROGRAMMING

B.1 INTRODUCTION

This Appendix discusses the topics of representing the geometrical shapes into programming structures that can be stored in the computer memory using the techniques of Object Oriented Programming (OOP)

The Computer Language used here is the C# language. The C# language is a microsoft language that takes its origin from C/C++ Languages but also borrows the concepts of Virtual Machine Compilation, and Memory Garbage Collection, from other languages like Java.

When compiling a C# source code, the source code is compiled into an intermediate language (IL). The intermediate language is low-level representation that looks like the assembly language. It is a special language that all C# code is compiled to. When the program is compiled, the high-level code is translated into a series of evaluation stack instructions.

OBJECT-ORIENTATION is a set of tools and methods that enable software engineers to build reliable, user friendly, maintainable, well documented, reusable software systems that fulfills the requirements of its users. It is claimed that object-orientation provides software developers with new mind tools to use in solving a wide variety of problems. Object-orientation provides a new view of computation. A software system is seen as a community of objects that cooperate with with each other by passing messages in solving a problem.

B.2 OBJECT ORIENTATION CONCEPTS

An object-oriented programming language provides support for the following object-oriented concepts:

- Objects
- Classes.
- Inheritance.
- Polymorphism.
- Operator overloading.

we shall discuss in these concepts in details for their close relevance to the present work.

B.2.1 Objects

In object-oriented programming we create software objects that model real world items. The objects are modeled after those items in that they have state and behavior. A software object maintains its state in one or more variables . A variable is an item of data named by an identifier. The software object implements its behavior with methods. A method is a function associated with an object, thus in summary "an object is a software bundle of variables and related methods".

An object is also known as an instance. For example Karuna's bicycle is an instance of a bicycle. It refers to a particular bicycle. Sandile Zuma is an instance of a Student.

The variables of an object are formally known as instance variables because they contain the state for a particular object or instance. In a running program, there may be many instances of an object. For example there may be many Student objects. Each of these objects will have their own instance variables and each object may have different values stored in their instance variables. For e.g. each Student object will have a different number stored in its StudentNumber variable.

B.2.2 Classes

In object-oriented programs, it's possible to have many objects of the same kind that share common characteristics: rectangles, circles, lines, ..., etc. A class is a software blueprint for objects and is used to manufacture or create more objects.

The class declares the instance variables necessary to contain the state of every object. It would also declare and provide implementations for the instance methods necessary to operate on the state of the object. Thus a class may be defined as "The blueprint that defines the variables and the methods common to all objects of a certain kind".

Class may also be considered as a factory for constructing objects. The non-static parts of the class specify, or describe, what variables and methods the objects will contain. This is part of the explanation of how objects differ from classes: Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Encapsulation Object diagrams show that an object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. Packaging an object's variables within the protective custody of its methods is called encapsulation.

Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two benefits to software developers:

- Modularity: The source code for an object can be written and maintained in- dependently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.
- Information-hiding: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and meth- ods that can be changed at any time without affecting other objects that depend on it.

Messages Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B

There are three parts of a message: The three parts for the message Console.WriteLine("Hello World"); are:

- The object to which the message is addressed (Console)
- The name of the method to perform (WriteLine)
- Any parameters needed by the method ("Hello World!")

B.2.3 Inheritance

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

Example of inheritance is the case of a moving objects like cars, and planes. while cars are moving on the ground, planes are flying in the sky. The move method is considered to be an abstraction for these types of objects, however the move implementation differes if it is associated with cars and or planes. Inheritance permits the programmer to inherit the behaviour of the base object into the new inherited object, but with the ability to change what it really do while executing the same behaviour.

Another famous example is the speak behaviour of objects. While all creatures can speak, their language can differ from voice to movement of body. The programmer then should make a base class called Creature with Speak method to define the behaviour, then inherting any creature from this super class to make a sub class (derived class) that has the same named behaviour "Speak" but with different implementation of what is really done.

This existing class is called the base class , and the new class is referred to as the derived class. (Other programming languages, such as Java, refer to the base class as the superclass a nd the derived class as the subclass .) A derived class represents a more specialized group of objects. Typically, a derived class contains behaviors inherited from its base class plus additional behaviors. A derived class can also customize behaviors inherited from the base class. A direct base class is the base class from which a derived class explicitly inherits. An indirect base class i s inherited from two or more levels up the class hierarchy.

B.2.4 Polymorphism

Polymorphism enables the programmer to "program in the general" rather than "program in the specific." In particular, polymorphism enables the writing of programs that process objects that share the same superclass in a class hierarchy as if they are all objects of the superclass; this can simplify programming.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes Fish, Frog and Bird represent the three types of animals under investigation. Imagine that each of these classes extends superclass Animal, which contains a method move and maintains an animal's current location as x-y coordinates. Each subclass implements method move. Our program maintains an array of references to objects of the various Animal subclasses. To simulate the animals' movements, the program sends each object the same message once per second namely, move. However, each specific type of Animal responds to a move message in a unique way a Fish might swim one meter, a Frog might jump 2 km and a Bird might fly 3 meters. The program issues the same message (i.e., move) to each animal object generically, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement. Relying on each object to know how to "do the right thing" (i.e., do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism. The same message (in this case, move) sent to a variety of objects has "many forms" of results, hence the term polymorphism.

With polymorphism, we can design and implement systems that are easily extensible—new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that the programmer adds to the hierarchy. For example, if we extend class Animal to create class Tortoise (which might respond to a move message by crawling one inch), we need to write only the Tortoise class and the part of the simulation that instantiates a Tortoise object. The portions of the simulation that process each Animal generically can remain the same.

B.2.5 Operator Overloading

In object-oriented programming, operator overloading—less commonly known as operator ad hoc polymorphism—is a specific case of polymorphism, where different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

Operator overloading is used because it allows the developer to program using notation closer to the target domain[1] and allows user-defined types a similar level of syntactic support as types built into the language. It is common, for example, in scientific computing, where it allows computational representations of mathematical objects to be manipulated

with the same syntax as on paper.

B.3 GEOMETRICAL AND MATHEMATICAL TYPES

The algorithm developed for this thesis has gone through careful design process to ease the programming phase. This section discusses the invented types that was made specially in the software source code.

Geometrical Computational Types are special user defined types that helps in the 3D calculations through the entire life of the running program. When implementing these types, the programmer usually is trying to mimic the mathematical behaviour of these types into the program code for making it easy to write the rest of the program. For example when making a vector class that contains three elements of x, y, z the implementor should implement the mathematical opetrations of this type to simply use one line of code as following:

```
Vector v1 = new Vector(3,2,5);
Vector v2 = new Vector(5,3,1);
Vector total = v1 + v2;
```

The Addition '+' operation is implemented through the vector type itself, through what is called Operator overloading. Writing code with this approach is preferred and readable through the entire program listing, and it is a more convenient way than implementing the same code as following:

```
Vector v1 = new Vector(3,2,5);
Vector v2 = new Vector(5,3,1);
Vector total = new Vector(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z)
```

By implementing such mathematical behaviour in the geomtrical computation types, the concept of abstraction and re-use of object oriented programming is achieved through the entire program.

This section will go through a set of geometrical types that have the most influence effect on writing this thesis algorithms. These types can be summarized into pure mathematical types, and geometrical mathematical types:

- General Vector
- Plane
- Quadrilateral
- Angle
- Quaternion
- Matrix

B.3.1 General Vector

General Vector type is an abstracted concept that hold the information of Euclidean vector in any coordinate system available throughout the software. The x, y, and z component names has been replaced with X1, X2, and X3 names to generalize the concept of vector depending on the current coordinate system.

If the coordinate system is Cartesian system, then the X1, X2, and X3 components are correspoding to x, y, and z components; However if the coordinate system is Spherical system, then X1, X2, and X3 components are corresponding to r, θ, and ϕ components.

The following list that is taken directly from the source code of the software, illustrate the implementation of this behaviour:

```
public struct GeneralVector : IEquatable<GeneralVector>
{
 public double X1;
 public double X2;
 public double X3;
 public ICoordinateSystem CoordinateSystem
 {
  get
  {
   if (_CoordinateSystem == null)
    _CoordinateSystem = CoordinateSystems.CartesianCoordinates;
    return _CoordinateSystem;
   }
   set
   {
    if(_CoordinateSystem==null)
    _CoordinateSystem = CoordinateSystems.CartesianCoordinates;
    var x = _CoordinateSystem.x(X1, X2, X3);
    var y = _CoordinateSystem.y(X1, X2, X3);
    var z = _CoordinateSystem.z(X1, X2, X3);

    _CoordinateSystem = value;
    X1 = _CoordinateSystem.x1(x, y, z);
    X2 = _CoordinateSystem.x2(x, y, z);
    X3 = _CoordinateSystem.x3(x, y, z);
   }
 }
}
```

The software supports the three famous coordinate systems Cartesian (which is the default Coordinate System), Cylindrical, and Spherical Coordinates. Setting the property of CoordinateSystem shown in the General Vector type automatically converts the X1, X2, and X3 values into the corresponding values of this coordinate system.

The coordinate system propery type is of the ICoordinateSystem interface, which is any Coordinate System class should implement; The following list is for the Spherical coordinate

system Class implementation:

```
public class SphericalSystem : ICoordinateSystem
{
  public double x1(double x, double y, double z)
  {
    var r = Math.Sqrt(Math.Pow(x, 2) + Math.Pow(y, 2) + Math.Pow(z, 2));
    return r;
  }

  public double x2(double x, double y, double z)
  {
    var r = x1(x,y,z);
    var theta = Math.Acos(z / r);
    return theta;
  }

  public double x3(double x, double y, double z)
  {
    var phi = Math.Atan2(y, x);
    return phi;
  }

  public double x(double x1, double x2, double x3)
  {
    var x = x1 * Math.Sin(x2) * Math.Cos(x3);
    return x;
  }

  public double y(double x1, double x2, double x3)
  {
    var y = x1 * Math.Sin(x2) * Math.Sin(x3);
    return y;
  }

  public double z(double x1, double x2, double x3)
  {
    var z = x1 * Math.Cos(x2);
    return z;
  }
}
```

Each ICoordinateSystem implemented class should implement the conversion between its coordinate system and the Cartesian system to help the GeneralVector in converting its coordinates data according to its current system.The general vector type play a cruical role in the foundation of the software source code. It is the heart of any point stored in the memory throughout the program life time.

B.3.1.1 General Vector Properties

It is a good practice to include the known vector properties inside the type code. The General Vector has many useful geometrical information and vector calculations like :

- Coss Product
- Dot Product
- Cosine Angle with Another Vector
- Length (Magnitude)

The following list shows these methods and properties responsible of such operations:

```
public GeneralVector CrossProduct(GeneralVector gp)
{
  var a = this.ClonePoint(CoordinateSystems.CartesianCoordinates);
  var b = gp.ClonePoint(CoordinateSystems.CartesianCoordinates);
  var result = GeneralVector.Cartesian(
    a[2] * b[3] - a[3] * b[2]
    , a[3] * b[1] - a[1] * b[3]
    , a[1] * b[2] - a[2] * b[1]
    );
  result.CoordinateSystem = this.CoordinateSystem;
  return result;
}

public double DotProduct(GeneralVector gp)
{
  var a = this.ClonePoint(CoordinateSystems.CartesianCoordinates);
  var b = gp.ClonePoint(CoordinateSystems.CartesianCoordinates);
  return a[1] * b[1] + a[2] * b[2] + a[3] * b[3];
}

public double CosineAngle(GeneralVector target)
{
  return this.DotProduct(target) / this.CartesianLength * target.CartesianLength;
}

public double CartesianLength
{
  get
  {
    GeneralVector v = this;
    if (!IsCartesian)
    v = this.ClonePoint(CoordinateSystems.CartesianCoordinates);
    return Math.Sqrt(v[1] * v[1] + v[2] * v[2] + v[3] * v[3]);
  }
}
```

B.3.1.2 General Vector Overloaded Operators

To compelete the concept of the general vector, a processes called operator overloading needs to be specified in order to use the General Vector as an independent type while writing the software source code.

The important operators to be overloaded are the plus '+', minus '-', and multiplication '*' signs, and the implementation can be represented as the following source code:

```
public static GeneralVector Add(GeneralVector leftPoint, GeneralVector rightPoint)
{
  var rp = rightPoint.ClonePoint(leftPoint.CoordinateSystem);
  leftPoint.X1 += rp.X1;
  leftPoint.X2 += rp.X2;
  leftPoint.X3 += rp.X3;
  return leftPoint;
}
public static GeneralVector Subtract(GeneralVector leftPoint, GeneralVector rightPoint)
{
  var rp = rightPoint.ClonePoint(leftPoint.CoordinateSystem);
  leftPoint.X1 -= rp.X1;
  leftPoint.X2 -= rp.X2;
  leftPoint.X3 -= rp.X3;
  return leftPoint;
}
public static GeneralVector operator +(GeneralVector leftPoint, GeneralVector rightPoint
{
  return Add(leftPoint, rightPoint);
}
public static GeneralVector operator -(GeneralVector leftPoint, GeneralVector rightPoint
{
  return Subtract(leftPoint, rightPoint);
}
public static GeneralVector operator *(GeneralVector point, double scalar)
{
  var p = point;
  p.X1 *= scalar;
  p.X2 *= scalar;
  p.X3 *= scalar;
  return p;
}
public static GeneralVector operator *(double scalar, GeneralVector point)
{
  var p = point;
  p.X1 *= scalar;
  p.X2 *= scalar;
  p.X3 *= scalar;
  return p;
}
```

B.3.2 Plane

One of the foundation objects in the software source code is the Plane class. the Plane class permits you to represent 3D plane in the memory with equation of plane using the symbolic representation.

The plane equation $aX + bY + cZ + d = 0$ is stored in the Plane class as a symbolic variable.

```
public class Plane
{
  public readonly SymbolicVariable Equation;
  public readonly GeneralVector NormalVector;
  public readonly double A, B, C, D;  // plane coefficients  Ax+By
  public Plane(SymbolicVariable equation)
  {
    Equation = equation;
    var a = Equation.CoefficientOf("x");
    var b = Equation.CoefficientOf("y");
    var c = Equation.CoefficientOf("z");
    var d = Equation.CoefficientOf("");
    A = a.HasValue ? a.Value : 0;
    B = b.HasValue ? b.Value : 0;
    C = c.HasValue ? c.Value : 0;
    D = d.HasValue ? d.Value : 0;
    NormalVector = GeneralVector.Cartesian(A, B, C);
  }
}
```

The plane object can be created with two approaches:

- Point and Normal Vector from this point.
- Three Points.

```
public static Plane FromNormalAndPoint(GeneralVector normalVector, GeneralVector point)
{
  var plane = SymbolicVariable.Parse("a*(x-$x)+b*(y-$y)+c*(z-$z)");
  plane = plane.Substitute("a", normalVector.X1);
  plane = plane.Substitute("b", normalVector.X2);
  plane = plane.Substitute("c", normalVector.X3);
  plane = plane.Substitute("$x", point.X1);
  plane = plane.Substitute("$y", point.X2);
  plane = plane.Substitute("$z", point.X3);
  return new Plane(plane);
}

public static Plane FromThreePoints(GeneralVector a, GeneralVector b, GeneralVector c)
{
  GeneralVector v1 = b - a;
  GeneralVector v2 = c - b;
```

```
  GeneralVector NormalVector = v1.CrossProduct(v2);
  var plane = SymbolicVariable.Parse("a*(x-$x)+b*(y-$y)+c*(z-$z)");
  plane = plane.Substitute("a", NormalVector.X1);
  plane = plane.Substitute("b", NormalVector.X2);
  plane = plane.Substitute("c", NormalVector.X3);
  plane = plane.Substitute("$x", a.X1);
  plane = plane.Substitute("$y", a.X2);
  plane = plane.Substitute("$z", a.X3);
  return new Plane(plane);
}
```

There is also an important function that checks if certain point lies on the plane or not.

```
public bool HasPoint(GeneralVector point)
{
  // the plane equation should be equal to zero when substituting point x,y,z
  Dictionary<string, double> values = new Dictionary<string, double>();
  foreach(var symbol in Equation.InvolvedSymbols)
  {
    if (symbol.Equals("x", StringComparison.OrdinalIgnoreCase)) values.Add("x", point.X)
    if (symbol.Equals("y", StringComparison.OrdinalIgnoreCase)) values.Add("y", point.Y)
    if (symbol.Equals("z", StringComparison.OrdinalIgnoreCase)) values.Add("z", point.Z)
  }
  double result = Math.Abs(Equation.Execute(values));
  double diff = result - 0;
  if (diff < 0.00001) return true;
  // experimental difference on the basis that my unit values are in milli meter
    return false;
}
```

B.3.3 Quadrilateral

Quadrilateral type is a class that store four points to express a quadrilateral shape and also have a many useful operations for calculating the normal of shape besied the angles of each corner. The main objective of this type is to be used in memory storage of discovered cells while gridding.

```
public struct Quadrilateral
{
  public GeneralVector TopLeft;
  public GeneralVector BottomLeft;
  public GeneralVector TopRight;
  public GeneralVector BottomRight;

  private GeneralVector _QuadNormal;
  public GeneralVector TopLeft_VertexNormal;
  public GeneralVector BottomLeft_VertextNormal;
  public GeneralVector TopRight_VertexNormal;
  public GeneralVector BottomRight_VertexNormal;
```

```
public GeneralVector Normal
{
  get
  {
    if (_QuadNormal.IsZero)
    {
      // get the vectors of quad.
      var a = BottomLeft - TopLeft;

      var b = BottomRight - BottomLeft;
      var c = TopRight - TopRight;
      var d = TopLeft - TopRight;
      // first normal for a-b plane  a x b
      var n1 = a.CrossProduct(b);
      // second normal for c-d plane c x d
      var n2 = c.CrossProduct(d);
      _QuadNormal = ((n1 + n2) / 2).Normalize();
    }
    return _QuadNormal;
  }
}

public GeneralVector ClockWiseLeftVector
{
  get
  {
    return TopLeft - BottomLeft;
  }
}
public GeneralVector CounterClockWiseLeftVector
{
  get
  {
    return BottomLeft - TopLeft;
  }
}
public GeneralVector ClockWiseBottomVector
{
  get
  {
    return BottomLeft - BottomRight;
  }
}
public GeneralVector CounterClockWiseBottomVector
{
  get
  {
    return BottomRight - BottomLeft;
```

```
  }
}
public GeneralVector ClockWiseRightVector
{
  get
  {
    return BottomRight - TopRight;
  }
}
public GeneralVector CounterClockWiseRightVector
{
  get
  {
    return TopRight - BottomRight;
  }
}
public GeneralVector ClockWiseTopVector
{
  get
  {
    return TopRight - TopLeft;
  }
}
public GeneralVector CounterClockWiseTopVector
{
  get
  {
    return TopLeft - TopRight;
  }
}
public Angle TopLeftAngle
{
  get
  {
    var ctheta =
      ClockWiseTopVector.Normalize().CosineAngle(
        CounterClockWiseLeftVector.Normalize()
    );
    return Math.Acos(ctheta);
  }
}
public Angle BottomLeftAngle
{
  get
  {
    var ctheta =
      ClockWiseLeftVector.Normalize().CosineAngle(
        CounterClockWiseBottomVector.Normalize()
    );
```

```
      return Math.Acos(ctheta);
    }
  }
  public Angle BottomRightAngle
  {
    get
    {
      var ctheta =
        ClockWiseBottomVector.Normalize().CosineAngle(
          CounterClockWiseRightVector.Normalize()
        );
      return Math.Acos(ctheta);
    }
  }
  public Angle TopRightAngle
  {
    get
    {
      var ctheta =
        ClockWiseRightVector.Normalize().CosineAngle(
          CounterClockWiseTopVector.Normalize()
      );
      return Math.Acos(ctheta);
    }
  }
  public bool IsCyclic
  {
    get
    {
      Angle ss = TopRightAngle + BottomLeftAngle;
      if (ss.Degrees == 180) return true;
      return false;
    }
  }
  public bool IsRectangle
  {
    get
    {
      bool tr = TopRightAngle.Degrees == 90.0;
      bool tl = TopLeftAngle.Degrees == 90.0;
      bool bl = BottomLeftAngle.Degrees == 90.0;
      bool br = BottomRightAngle.Degrees == 90.0;
      return (tr & tl & bl & br);
    }
  }
}
```

B.3.4 Angle

Abstracting the angle from double value into its own custom type has proved to be excellent for writing the software source code. It helped a lot while debugging the code due to the ability of this type to show more information about the angle:

- Angle in Radian
- Angle in Degrees
- Revolutions count in case of angle more than 360 degree
- Absolute Angle

```
public struct Angle
{
  private readonly double _radians;
  public Angle(double radians)
  {
    _radians = radians;
  }
  public static Angle FromDegrees(double degrees)
  {
    var r = degrees * Math.PI / 180;
    return new Angle(r);
  }
  public double Radians
  {
    get
    {
      return _radians;
    }
  }
  public double Degrees
  {
    get
    {
      return _radians * 180 / Math.PI;
    }
  }

  public double Revolutions
  {
    get
    {
      return _radians / (2 * Math.PI);
    }
  }

  public Angle AbsoluteAngle
```

```
  {
    get
    {
      if (Revolutions <= 1)
      {
        if (_radians >= 0)
          return new Angle(_radians);
        else
          return new Angle(2 * Math.PI + _radians);
      }
      else
      {
        int revs = (int)Math.Floor(Revolutions);
        double g = Revolutions - revs;
        if (g >= 0) return new Angle(g);
        else return new Angle(2 * Math.PI + g);
      }
    }
  }
}
```

B.3.5 Quaternion

Quaternion can be considered as an extension to complex numbers. This special type is essential in any graphical software, because its the heart of all rotations in this software. Quaternion express a rotation; Multiplication of two rotations is another rotation.

```
public struct Quaternion : IEquatable<Quaternion>
{
  private double a;
  private double b;
  private double c;
  private double d;
  public static Quaternion operator +(Quaternion lhs, Quaternion rhs)
  {
    Quaternion result = new Quaternion(lhs);
    result.a += rhs.a;
    result.b += rhs.b;
    result.c += rhs.c;
    result.d += rhs.d;
    return result;
  }

  public static Quaternion operator -(Quaternion lhs, Quaternion rhs)
  {
    Quaternion result = new Quaternion(lhs);
    result.a -= rhs.a;
    result.b -= rhs.b;
    result.c -= rhs.c;
```

```
    result.d -= rhs.d;
    return result;
  }

  public static Quaternion operator *(Quaternion lhs, Quaternion rhs)
  {
    double ar = (rhs.a);
    double br = (rhs.b);
    double cr = (rhs.c);
    double dr = (rhs.d);
    double at = lhs.a * ar - lhs.b * br - lhs.c * cr - lhs.d * dr;
    double bt = lhs.a * br + lhs.b * ar + lhs.c * dr - lhs.d * cr;
    double ct = lhs.a * cr - lhs.b * dr + lhs.c * ar + lhs.d * br;
    double dt = lhs.a * dr + lhs.b * cr - lhs.c * br + lhs.d * ar;
    Quaternion result = new Quaternion(at, bt, ct, dt);
    return result;
  }

  public static Quaternion operator /(Quaternion lhs, Quaternion rhs)
  {
    double ar = (rhs.a);
    double br = (rhs.b);
    double cr = (rhs.c);
    double dr = (rhs.d);
    double denominator = ar * ar + br * br + cr * cr + dr * dr;
    double at = (lhs.a * ar + lhs.b * br + lhs.c * cr + lhs.d * dr) / denominator;
    double bt = (-lhs.a * br + lhs.b * ar - lhs.c * dr + lhs.d * cr) / denominator;
    double ct = (-lhs.a * cr + lhs.b * dr + lhs.c * ar - lhs.d * br) / denominator;
    double dt = (-lhs.a * dr - lhs.b * cr + lhs.c * br + lhs.d * ar) / denominator;
    Quaternion result = new Quaternion(at, bt, ct, dt);
    return result;
  }
}
```

B.3.6 Matrix

All graphical operations including 3D transformations needs a sort of matrix. The matrix type is implemented into the source code to make it easy for adding or multiplying matrices.

```
public class Matrix
{
  public int rows;
  public int cols;
  public double[,] mat;
  public Matrix L;
  public Matrix U;
  private int[] pi;
  private double detOfP = 1;
  public Matrix(int iRows, int iCols)
```

```
  {
    rows = iRows;
    cols = iCols;
    mat = new double[rows, cols];
  }
  public Boolean IsSquare()
  {
    return (rows == cols);
  }
  public double this[int iRow, int iCol]
  {
    get { return mat[iRow, iCol]; }
    set { mat[iRow, iCol] = value; }
  }
}
```

The most important matrix operation is the multiplication of matrices. Many algorithms of multiplication has been implemented for the matrix. A straight forward matrix multiplication that obeys the elementary lessons in teaching matrix multiplication can be implemented as follow:

```
public static Matrix StupidMultiply(Matrix m1, Matrix m2)
{
  if (m1.cols != m2.rows) throw new MException("Wrong␣dimensions␣of␣matrix!");
  Matrix result = ZeroMatrix(m1.rows, m2.cols);
  for (int i = 0; i < result.rows; i++)
    for (int j = 0; j < result.cols; j++)
      for (int k = 0; k < m1.cols; k++)
        result[i, j] += m1[i, k] * m2[k, j];
  return result;
}
```

However the problem of this multiplication lies in its performance. The running time of this algorithm is $O(n^3)$. Another algorithm such Strassen's algorithm, devised by Volker Strassen in 1969 and often referred to as "fast matrix multiplication". It is based on a way of multiplying two 2x2 matrices which requires only 7 multiplications (instead of the usual 8), at the expense of several additional addition and subtraction operations. Applying this recursively gives an algorithm with a multiplicative cost of $O\left(n^{\log_2 7}\right) \approx O(n^{2.807})$. The source code of Strassen matrix calculation can be found in the following code:

```
private static Matrix StrassenMultiply(Matrix A, Matrix B)
{
  if (A.cols != B.rows) throw new MException("Wrong␣dimension␣of␣matrix!");
  Matrix R;
  int msize = Math.Max(Math.Max(A.rows, A.cols), Math.Max(B.rows, B.cols));
  if (msize < 32)
  {
    R = ZeroMatrix(A.rows, B.cols);
    for (int i = 0; i < R.rows; i++)
```

```
    for (int j = 0; j < R.cols; j++)
      for (int k = 0; k < A.cols; k++)
        R[i, j] += A[i, k] * B[k, j];
  return R;
}
int size = 1; int n = 0;
while (msize > size) { size *= 2; n++; };
int h = size / 2;

Matrix[,] mField = new Matrix[n, 9];
/*
*  8x8, 8x8, 8x8, ...
*  4x4, 4x4, 4x4, ...
*  2x2, 2x2, 2x2, ...
*  . . .
*/
int z;
for (int i = 0; i < n-4; i++) // rows
{
  z = (int)Math.Pow(2, n - i - 1);
  for (int j = 0; j < 9; j++)  mField[i, j] = new Matrix(z, z);
}
SafeAplusBintoC(A, 0, 0, A, h, h, mField[0, 0], h);
SafeAplusBintoC(B, 0, 0, B, h, h, mField[0, 1], h);
StrassenMultiplyRun(mField[0, 0], mField[0, 1], mField[0, 1 + 1], 1, mField)
    ;

SafeAplusBintoC(A, 0, h, A, h, h, mField[0, 0], h);
SafeACopytoC(B, 0, 0, mField[0, 1], h);
StrassenMultiplyRun(mField[0, 0], mField[0, 1], mField[0, 1 + 2], 1, mField)
    ;

SafeACopytoC(A, 0, 0, mField[0, 0], h);
SafeAminusBintoC(B, h, 0, B, h, h, mField[0, 1], h);
StrassenMultiplyRun(mField[0, 0], mField[0, 1], mField[0, 1 + 3], 1, mField)
    ;

SafeACopytoC(A, h, h, mField[0, 0], h);
SafeAminusBintoC(B, 0, h, B, 0, 0, mField[0, 1], h);
StrassenMultiplyRun(mField[0, 0], mField[0, 1], mField[0, 1 + 4], 1, mField)
    ;

SafeAplusBintoC(A, 0, 0, A, h, 0, mField[0, 0], h);
SafeACopytoC(B, h, h, mField[0, 1], h);
StrassenMultiplyRun(mField[0, 0], mField[0, 1], mField[0, 1 + 5], 1, mField)
    ;

SafeAminusBintoC(A, 0, h, A, 0, 0, mField[0, 0], h);
SafeAplusBintoC(B, 0, 0, B, h, 0, mField[0, 1], h);
```

```
  StrassenMultiplyRun(mField[0, 0], mField[0, 1], mField[0, 1 + 6], 1, mField)
      ;

  SafeAminusBintoC(A, h, 0, A, h, h, mField[0, 0], h);
  SafeAplusBintoC(B, 0, h, B, h, h, mField[0, 1], h);
  StrassenMultiplyRun(mField[0, 0], mField[0, 1], mField[0, 1 + 7], 1, mField)
      ;

  R = new Matrix(A.rows, B.cols);
  /// C11
  for (int i = 0; i < Math.Min(h, R.rows); i++) // rows
    for (int j = 0; j < Math.Min(h, R.cols); j++) // cols
      R[i, j] = mField[0, 1 + 1][i, j] + mField[0, 1 + 4][i, j] - mField[0, 1
          + 5][i, j] + mField[0,1+7][i, j];

  /// C12
  for (int i = 0; i < Math.Min(h, R.rows); i++)  // rows
    for (int j = h; j < Math.Min(2*h, R.cols); j++)  // cols
      R[i, j] = mField[0, 1 + 3][i, j - h] + mField[0, 1 + 5][i, j - h];
  /// C21
  for (int i = h; i < Math.Min(2*h, R.rows); i++)  // rows
    for (int j = 0; j < Math.Min(h, R.cols); j++)  // cols
      R[i, j] = mField[0, 1 + 2][i - h, j] + mField[0, 1 + 4][i - h, j];
  /// C22
  for (int i = h; i < Math.Min(2 * h, R.rows); i++)  // rows
    for (int j = h; j < Math.Min(2 * h, R.cols); j++)  // cols
      R[i, j] = mField[0, 1 + 1][i - h, j - h] - mField[0, 1 + 2][i - h, j - h
          ] + mField[0, 1 + 3][i - h, j - h] + mField[0, 1 + 6][i - h, j - h];

  return R;
}
```

B.4 GRAPHICAL MODELING CLASSES

This section describes the classes that is used in the visualization of the geomtric types like lines and cirlces. Objects of these classes are capapable of using the computer graphic card through the OpenGL library to display its contents on the computer screen.

OpenGL (Open Graphics Library) is a code library (first invented by Sillicon Graphics) that gives the programmer a set of functions that helps in drawing 3D graphics.

B.4.1 Space Point

Space Point Class is a 3D modeling object that can be viewed in the viewport of the software. This class holds General Vector Type named Center which contains the location of the point in the 3D space.

It is very important to distinguish between the Space Point object and its Center, because

it is valid to have many space points with the same center. However there is no two identical space points event if their Center field are the same.

The Space Point class can be abstracted to the following list

```
public class SpacePoint : Model
{
  public SpacePoint(GeneralVector position)
  {
    Center = position;
  }
}
```

B.4.2 Space Line

Space Line Class is a 3D modeling object that contains two members of Space Point Objects. This is another important data structure decesion that all modeling classes contains their sub members from others modeling classes.

```
public class SpaceLine : Model, ICurve
{
  private SpacePoint _FirstPoint;
  private SpacePoint _LastPoint;
  public SpaceLine(GeneralVector firstPoint, GeneralVector secondPoint)
  {
    _FirstPoint = new SpacePoint( firstPoint);
    _LastPoint = new SpacePoint( secondPoint);
  }
}
```

B.4.3 Line Strip

Line Strip Class is a 3D modeling object that contains an array of Space Points, and array of Space Lines.

Between every two successive Space Points, there is one Space Line Object. It is important to mention that the second space point is always existing in two Space Lines.

```
public class LineStrip : Model
{
  public List<SpacePoint> Points = new List<SpacePoint>();
  List<SpaceLine> _Lines = new List<SpaceLine>();
  public void AddPoint(SpacePoint point)
  {
    point.Owner = this;
    if (Points.Count > 0)
    {
      SpaceLine sl = new SpaceLine(Points.Last(), point);
      _Lines.Add(sl);
    }
```

```
    Points.Add(point);
    RefreshCentroid();
  }
}
```

The Line Strip class contains a method for calculaing the centroid of the polygon that the object of line strip represents.

```
public static GeneralVector CalculateCentroid(SpacePoint[] points)
{
  // http://stackoverflow.com/questions/2792443/finding-the-centroid-of-a-polygon
  GeneralVector centroid = GeneralVector.Cartesian(0, 0, 0);
  if (points == null || points.Length == 0) return centroid;
  double signedArea = 0.0;
  double x0 = 0.0;
  double y0 = 0.0;
  double x1 = 0.0;
  double y1 = 0.0;

  double a = 0.0;
  int i = 0;
  for (i = 0; i < points.Length - 1; ++i)
  {
    x0 = points[i].Center.X1;
    y0 = points[i].Center.X2;
    x1 = points[i + 1].Center.X1;
    y1 = points[i + 1].Center.X2;
    a = x0 * y1 - x1 * y0;
    signedArea += a;
    centroid.X1 += (x0 + x1) * a;
    centroid.X2 += (y0 + y1) * a;
  }
  x0 = points[i].Center.X1;
  y0 = points[i].Center.X2;
  x1 = points[0].Center.X1;
  y1 = points[0].Center.X2;
  a = x0 * y1 - x1 * y0;
  signedArea += a;
  centroid.X1 += (x0 + x1) * a;
  centroid.X2 += (y0 + y1) * a;
  signedArea *= 0.5;
  centroid.X1 /= (6 * signedArea);
  centroid.X2 /= (6 * signedArea);
  return centroid;
}
```

B.4.4 Space Bezier Curve

Space Bezier Curve Class is a 3D modeling class that represents bezier curve with four control points, the control points are of the Space Point objects. This class is also able to deduce the curver parametric equations $C(x(t), y(t), z(t))$

```
public class SpaceBezierCurve: Model, I1DTopology
{
  private List<SpacePoint> _ControlPoints = new List<SpacePoint>(4);
  public SpaceBezierCurve(SpacePoint cp1, SpacePoint cp2, SpacePoint cp3,
     SpacePoint cp4)              : base()
  {
    SurfaceMaterial = Materials.Polished_Gold;
    cp1.Owner = this;
    cp2.Owner = this;
    cp3.Owner = this;
    cp4.Owner = this;
    _ControlPoints.Add(cp1);
    _ControlPoints.Add(cp2);
    _ControlPoints.Add(cp3);
    _ControlPoints.Add(cp4);
  }

  public double Ax(int index)
  {
    if (index == 0) return _ControlPoints[0].Center.X1;
    if (index == 1)
    {
      return 3 * (_ControlPoints[1].Center.X1 - _ControlPoints[0].Center.X1);
    }
    if (index == 2)
    {
      return 3 * (_ControlPoints[2].Center.X1 - _ControlPoints[1].Center.X1) -
          Ax(1);                  }
    if (index == 3)
    {
      return (_ControlPoints[3].Center.X1 - _ControlPoints[0].Center.X1) - Ax
         (2) - Ax(1);
    }
    throw new ArgumentException("Only␣from␣0␣to␣3␣to␣get␣the␣coefficient");
  }

  public double Ay(int index)
  {
    if (index == 0) return _ControlPoints[0].Center.X2;
    if (index == 1)
    {
      return 3 * (_ControlPoints[1].Center.X2 - _ControlPoints[0].Center.X2);
    }
```

```
  if (index == 2)
  {
    return 3 * (_ControlPoints[2].Center.X2 - _ControlPoints[1].Center.X2) -
        Ay(1);
  }
  if (index == 3)
  {
    return (_ControlPoints[3].Center.X2 - _ControlPoints[0].Center.X2) - Ay
       (2) - Ay(1);
  }
  throw new ArgumentException("Only␣from␣0␣to␣3␣to␣get␣the␣coefficient");
}

public double Az(int index)
{
  if (index == 0) return _ControlPoints[0].Center.X3;
  if (index == 1)
  {
    return 3 * (_ControlPoints[1].Center.X3 - _ControlPoints[0].Center.X3);
  }
  if (index == 2)
  {
    return 3 * (_ControlPoints[2].Center.X3 - _ControlPoints[1].Center.X3) -
        Az(1);
  }
  if (index == 3)
  {
    return (_ControlPoints[3].Center.X3 - _ControlPoints[0].Center.X3) - Az
       (2) - Az(1);
  }
  throw new ArgumentException("Only␣from␣0␣to␣3␣to␣get␣the␣coefficient");
}

public string XFunction
{
  get
  {
    return Ax(3).ToString() + "*t^3+" + Ax(2).ToString() + "*t^2+" + Ax(1).
       ToString() + "*t+" + Ax(0).ToString();
  }
}

public string YFunction
{
  get
  {
    return Ay(3).ToString() + "*t^3+" + Ay(2).ToString() + "*t^2+" + Ay(1).
       ToString() + "*t+" + Ay(0).ToString();
  }
```

```
  }

  public string ZFunction
  {
    get
    {
      return Az(3).ToString() + "*t^3+" + Az(2).ToString() + "*t^2+" + Az(1).
          ToString() + "*t+" + Az(0).ToString();
    }
  }
}
```

The class has an important properties like the Length as follows

```
public double Length
{
  get
  {
    if (!length.HasValue)
    {
      // length of curve is obtained by Integration of
      //  integration (sqrt (  (dx/dt)^2 + (dy/dt)^2 + (dz/dt)^2))
      // or numerically by getting the points of the curve and getting length between
      //  every two points and summing them up.
      const double interval = 0.01;
      double Total = 0.0;
      double t = 0.0;
      GeneralVector PreviousPoint = Execute(t);
      t += interval;
      GeneralVector CurrentPoint = Execute(t);
      for (; t <= 1; t += interval)
      {
        // calculate length between current point and previous point
        // then accumulate the result.
        var l = CurrentPoint - PreviousPoint;
        Total += l.CartesianLength;
        PreviousPoint = CurrentPoint;
        CurrentPoint = Execute(t);
      }
    length = Total;
    }
    return length.Value;
  }
}
```

B.4.5 Space Circle

Space Circle Class is a 3D Modeling class that contains an array of caluculated space points based on center and radius of this circle.

```
public class SpaceCircle: Model, I1DTopology, IGriddable
{
  SpacePoint _RadiusPoint;
  private int _Segments;
  SpacePoint _Centroid;
  GeneralVector UpVector;
  List<GeneralVector> _CirclePoints;

  void CalculateCirclePoints()
  {
    _CirclePoints = new List<GeneralVector>(Segments);
    double step = (2.0 * Math.PI) / Segments;
    var rad = Radius;
    Axis axis = Axis.CustomAxis(NormalVector);
    var points = axis.Revolve(RadiusVector, 2 * Math.PI, Segments);
    // translate the points to begin from center.
    for (int i = 0; i < points.Length; i++)
      points[i]+=_Centroid.Center;     _CirclePoints.AddRange(points);
  }
}
```

B.4.6 Space Arc

Space Arc Class is a 3D Modeling class that contains an array of space points based on three points to draw the arc between them. The following list illustrate the calculations of the arc based on the input of three points.

```
public class SpaceArc : Model, I1DTopology
{
  SpacePoint _FirstPoint, _SecondPoint, _ThirdPoint;
  SpaceLine _ArcWatar;
  List<GeneralVector> _ArcPoints;
  public SpaceArc(SpacePoint firstPoint, SpacePoint thirdPoint):base()
  {
    _FirstPoint = firstPoint;
    _ThirdPoint = thirdPoint;
    _FirstPoint.Owner = this;
    _ThirdPoint.Owner = this;
    _ArcWatar = new SpaceLine(firstPoint, thirdPoint);
  }
  double CalcCenter(out GeneralVector c_center)
  {
    var pt1 = _FirstPoint.Center;
    var pt2 = _SecondPoint.Center;
    var pt3 = _ThirdPoint.Center;
    double yDelta_a = pt2.Y - pt1.Y;
    double xDelta_a = pt2.X - pt1.X;
    double yDelta_b = pt3.Y - pt2.Y;
    double xDelta_b = pt3.X - pt2.X;
```

```
    if (Math.Abs(xDelta_a) <= 0.000000001 && Math.Abs(yDelta_b) <= 0.000000001)
    {
      c_center = GeneralVector.Cartesian(0.5 * (pt2.X + pt3.X)
, 0.5 * (pt1.Y + pt2.Y), pt1.Z);
      return (pt1 - c_center).Length;
    }
    double aSlope = yDelta_a / xDelta_a;
    if (Math.Abs(aSlope - bSlope) <= 0.000000001)
    {
      NoIntersection = true;
      c_center = GeneralVector.Zero;
      return -1;
    }
    var mmX = (aSlope * bSlope * (pt1.Y - pt3.Y) + bSlope * (pt1.X + pt2.X)
        - aSlope * (pt2.X + pt3.X)) / (2 * (bSlope - aSlope));
    var mmY = -1 * (mmX - (pt1.X + pt2.X) / 2) / aSlope + (pt1.Y + pt2.Y) / 2;
    var mmZ = pt1.Z;
    c_center = GeneralVector.Cartesian(mmX, mmY, mmZ);
    return (pt1 - c_center).Length;
  }
  public void ArcCalculate()
  {
    if (_SecondPoint == null) return;
    _Radius = CalcCenter(out Center);
    if (_Radius > -1)
    {
      _Centroid.Center = Center;
      UpVector = _Centroid.Center + GeneralVector.Cartesian(0, 0, 1);
      _ArcPoints = new List<GeneralVector>(_segments);
      Line cL1 = new Line(Center, _FirstPoint.Center);
      Line rL = new Line(Center, _SecondPoint.Center);
      Line cL2 = new Line(Center, _ThirdPoint.Center);
      Line L12 = new Line(_FirstPoint.Center, _SecondPoint.Center);
      Line L23 = new Line(_SecondPoint.Center, _ThirdPoint.Center);
      Line L13 = new Line(_FirstPoint.Center, _ThirdPoint.Center);
      GeneralVector positiveVector = L12.LineVector.CrossProduct(L23.LineVector);
      var xax = GeneralVector.X_Axis;
      ReferencePointAngle = Math.Acos(cL1.LineVector.Normalize().CosineAngle(xax));
      if (cL1.LineVector.Y < 0) ReferencePointAngle = -1 * ReferencePointAngle;
      Angle A1 = Math.Acos
(cL1.LineVector.Normalize().CosineAngle(rL.LineVector.Normalize()));
      Angle A2 = Math.Acos
(rL.LineVector.Normalize().CosineAngle(cL2.LineVector.Normalize()));
      var h1 = cL1.LineVector.Normalize().CrossProduct(rL.LineVector.Normalize());
      var h2 = rL.LineVector.Normalize().CrossProduct(cL2.LineVector.Normalize());
      var hhc = h1.CosineAngle(h2);
      Angle B1 = A1;
      Angle B2 = A2;
      if (hhc < 0)
```

```
        {
          if (A1 > A2) B1 = (Angle.FromDegrees(360) - A1);
          else B2 = Angle.FromDegrees(360) - A2;
        }
        _ArcAngle = B1 + B2;
        Axis axis = Axis.LineAxis(Center, Center + GeneralVector.Z_Axis);
        // compare the positive vector with the z axis
        // (this should be the plan normal we are drawing on)
        var oryangle = positiveVector.CosineAngle(GeneralVector.Z_Axis);
        if (oryangle <= 0)
        {
          _ArcAngle = _ArcAngle * -1;
        }
        var points = axis.Revolve(_FirstPoint.Center, _ArcAngle, _segments);
        _ArcPoints.AddRange(points);
        NoIntersection = false;
      }
      else
      {
        NoIntersection = true;
      }
    }
  }
```

B.4.7 Quadrilateral Space

The Quadrilateral Space class is the corner stone type in making grids. Basically domain area is divided into partitions using quadrilateral spaces, then each space is gridded by selecting the four points of the quadrilateral space. The program will connect these spaces together and will specify the neibourhood cells.

This object has four sides which are composed of four Bezier Curves. Corners of these objects are space points, which in turn indicates that the space point is shared between two bezier curves. Moreover, when discovering the neihbour cells, the program algorithm depend on corner cells to know the other shared curves, and if they are belonging to other quadratic cells.

The usage of curved sides in quadratic cell, allow the representation of any shape that can be morphed between rectangle and cirlce. The shape can be adjusted and the homotopy between corresponding sides can be calculated.

B.4.7.1 Quadrilateral Space Grid

The Quadrilateral Space Grid is an inner type that represents the lines of the grid in the Quadrilateral Space object. This object can be used to obtain the quadrilateral cells from the grid with its coordinates.

B.4.8 Quadrilateral Cell Element

This type is a Two-Dimensional representation to the quadrilateral cell, with its four space points. The type contain the mathematical representation of quadrilateral object.

B.4.9 Hexahedron Cell Element

This type is a Three-Dimensional representation that contains 8 space points, with 6 faces. The type also contains the functionality of discovering the neighbour cells on the faces by checking for the space points shared owners. This way we can write any algorithm that goes from cell to cell without breaking the continuity.

APPENDIX C

PROGRAM LISTINGS OF IMPORTANT ALGORITHMS

C.1 CURVE ALGORITHMS

C.1.1 Approximation of Circular Arc to Bezier Curve

Obtaining Tangent Equations It is known that Tangent is always perpendicular to the Radius vector

$$m = \frac{y_a - y_c}{x_a - x_c}$$

Tangent Slope as a negative reciprocal is obtained by

$$m_t = -1\left(\frac{1}{m}\right)$$

the tangents equation may be written by

$$y = mx + b$$

hence the final b constant for tangent is

$$b = y_a - m_t x_a$$

Specifiying Tangent Orientation To keep consistency with the smaller angle between two selected points we get the intersection of the two tangents

$$m_{tP_2} x + b_{P_2} = m_{tP_1} x + b_{P_1}$$

hence we obtain a value for x

this value is reused in constructing two tangents line equations with the desired orientation.

Obtaining the P_3 and P_2 Coordinates The coordinates of points are then calculated based on the testing length g

$$x_P = g \cdot \mathbf{i}$$

$$y_p = g \cdot \mathbf{j}$$

```
dv = (2 * Math.PI * Radius)/300;
while ((1 - dLR) <= 0.99)
{
  P2 = BiggerTangentLine.GetPoint_ByLength(TestLength);
  P3 = SmallerTangetLine.GetPoint_ByLength(TestLength);
  TestCurve = new Curve(P1, P2, P3, P4);

  // Get the length difference between curve center and arc center point
  dL = (ArcCenterPoint - TestCurve.Execute(0.5)).CartesianLength;
  dLR = dL / Radius;

  if (dv < 1e-3) break; // In case of too small increment value
}
```

dL should be decreased in every iteration (a special treatment for the increment value has been added to the final algorithm that permits of adjusting Δv.

C.2 GRIDDING ALGORITHMS

This section lists the important algorithms and program listings in the Gridding of Quadrilateral Shapes.

C.2.1 Homotopy Between Two Opposite Curves in Quadrilateral Shape

```
public Curve AdjustedCurveBetweenFirstAndThirdCurves(double t)
{
  // first and third curve should be in the same sense to get a correct behaviour
  if (FirstSideCurveEquation == null)
  {
    if (C0310 | C0313 | C3000 | C3300)
    {
      FirstSideCurveEquation = Sides[0].GetCurve();
```

```
    }
    else
    {
      // all othe options are reveresed
      FirstSideCurveEquation = Sides[0].GetReversedCurve();
    }
    if (C2313 | C2310 | C2033 | C2030 )
      ThirdSideCurveEquation = Sides[2].GetCurve();
    else
      ThirdSideCurveEquation = Sides[2].GetReversedCurve();
  }

  Curve c = FirstSideCurveEquation.HomotopyCurve(ThirdSideCurveEquation, t);
  // fourth curve to second curve
  // 0313 | 0013 | 2010 | 2310
  if (C0313 | C0013 | C2010 | C2310)
  {
    // -ve sense
    c[3] = Sides[1].ExecuteGeneralCurveEquation(1 - t);
  }
  else
  {
    c[3] = Sides[1].ExecuteGeneralCurveEquation(t);
  }
    // 3300 | 3303 | 2030 | 2330
  if (C3300 | C3303 | C2030 | C2330)
  {
    c[0] = Sides[3].ExecuteGeneralCurveEquation(1 - t);
  }
  else
  {
    c[0] = Sides[3].ExecuteGeneralCurveEquation(t);
  }
  return c;
}
```

C.2.2 Concentration Stretching Function

```
double GeometricStretchM(int step)
{
  /*
    length = sum of the series   ==  a*((1-r^n)/(1-r))
    r :   base     (our parameter)
    n :  number of grid lines   (given)
    calculate a
    then calculate the sum to
  */
  int n = _Cells_M_Count;
```

```
  double r = GridQuadCell.MFactor;
  if (r == 1 || r == 0) return UniformStretchM(step);
  // 1 =  a *((1-r^n)/(1-r))
  // a = 1/((1-r^n)/(1-r))
  // s[n] ..> ar^n
  double a = 1.0 / ((1.0 - Math.Pow(r, n)) / (1.0 - r));
  double total = 0.0;
  for (int i = 0; i < step; i++)
  {
    total = total + (a * Math.Pow(r, i));
  }
  return total / 1;
}

double GeometricDoubleStretchM(int step)
{
  // make the expansion on first half and contraction on second half
  int n_A = _Cells_M_Count / 2;
  // first half
  int n_B = _Cells_M_Count - n_A;
  // second half
  double r = GridQuadCell.MFactor;
  if (r == 1 || r == 0) return UniformStretchM(step);
  // 1 =  a *((1-r^n)/(1-r))
  // a = 1/((1-r^n)/(1-r))
  // s[n] ..> ar^n
  double r_A = r;
  double r_B = 1 / r;
  double a_A = 1.0 / ((1.0 - Math.Pow(r_A, n_A)) / (1.0 - r_A));
  double a_B = 1.0 / ((1.0 - Math.Pow(r_B, n_B)) / (1.0 - r_B));
  double total_A = 0.0;
  double total_B = 0.0;
  int b_i = 0;
  // different index for the second half
  for (int i = 0; i < step; i++)
  {
    if (i < n_A)
    {
      total_A = total_A + (a_A * Math.Pow(r_A, i));
    }
    else
    {
      total_B = total_B + (a_B * Math.Pow(r_B, b_i));
      b_i++;
    }
  }
  double total = total_A + total_B;
  return total / 2;
}
```

C.2.3 Senses Coding Listing

```
public bool C0310 { get { return Sides[0][3].Center.Equals(Sides[1][0].Center); } }
public bool C0313 { get { return Sides[0][3].Center.Equals(Sides[1][3].Center); } }
public bool C0013 { get { return Sides[0][0].Center.Equals(Sides[1][3].Center); } }
public bool C0010 { get { return Sides[0][0].Center.Equals(Sides[1][0].Center); } }
public bool C3000 { get { return Sides[3][0].Center.Equals(Sides[0][0].Center); } }
public bool C3300 { get { return Sides[3][3].Center.Equals(Sides[0][0].Center); } }
public bool C3303 { get { return Sides[3][3].Center.Equals(Sides[0][3].Center); } }
public bool C3003 { get { return Sides[3][0].Center.Equals(Sides[0][3].Center); } }
public bool C2313 { get { return Sides[2][3].Center.Equals(Sides[1][3].Center); } }
public bool C2310 { get { return Sides[2][3].Center.Equals(Sides[1][0].Center); } }
public bool C2010 { get { return Sides[2][0].Center.Equals(Sides[1][0].Center); } }
public bool C2013 { get { return Sides[2][0].Center.Equals(Sides[1][3].Center); } }
public bool C2033 { get { return Sides[2][0].Center.Equals(Sides[3][3].Center); } }
public bool C2030 { get { return Sides[2][0].Center.Equals(Sides[3][0].Center); } }
public bool C2333 { get { return Sides[2][3].Center.Equals(Sides[3][3].Center); } }
public bool C2330 { get { return Sides[2][3].Center.Equals(Sides[3][0].Center); } }
```

C.2.4 Neighbour Quadrilateral Shapes Discovery

```
public QuadCell GetSideCell(int i)
{
  var s = Sides[i];
  var cells = from o in s.Owners
        where o.GetType() == typeof(QuadCell) && o.Id != this.Id
select (QuadCell)o;
  var cell = cells.FirstOrDefault();
  return cell;
}
```

C.2.5 Grouping Algorithm

The grouping algorithm takes all the neihbor cells and adding them into a new instance of QuadCellGroup object. The code used to make this grouping is as follow:

```
public static QuadCellGroup DiscoverCellGroup(QuadCell qc)
{
  QuadCellGroup targetGroup = new QuadCellGroup();
  targetGroup.AddCell(qc);
  Discover_AllCellSide_On(qc, targetGroup);
  targetGroup.RefreshCenter();
  return targetGroup;
}

private static void Discover_AllCellSide_On(QuadCell qc, QuadCellGroup targetGroup)
```

```
{
  var sc1 = qc.GetSideCell(0);
  var sc2 = qc.GetSideCell(1);
  var sc3 = qc.GetSideCell(2);
  var sc4 = qc.GetSideCell(3);
  if (sc1 != null)
  {
    if (!targetGroup.Contains(sc1))
    {
      targetGroup.AddCell(sc1);
      Discover_AllCellSide_On(sc1, targetGroup);
    }
  }
  if (sc2 != null)
  {
    if (!targetGroup.Contains(sc2))
    {
      targetGroup.AddCell(sc2);
      Discover_AllCellSide_On(sc2, targetGroup);
    }
  }
  if (sc3 != null)
  {
    if (!targetGroup.Contains(sc3))
    {
      targetGroup.AddCell(sc3);
      Discover_AllCellSide_On(sc3, targetGroup);
    }
  }
  if (sc4 != null)
  {
    if (!targetGroup.Contains(sc4))
    {
      targetGroup.AddCell(sc4);
      Discover_AllCellSide_On(sc4, targetGroup);
    }
  }
}
```

The algorithm depends on a recursive discovery which test each side on the quadratic cell for another cell, if a cell is found, it is stored in the list of discovered cell (if it was not there before) and it continues like that until all cells behind this cell are consumed and the new QuadCellGroup object is returned.

The main concept of the gridding process is to divide the 2D space into complete quadratic cell partitions. This is important to have all the area covered by adjacent quadratic cells.

The first phase of gridding the geometry is to select certain points and connecting them with lines to form big areas that will serve as an independent big quadratic cells as seen in Fig. (C.1)

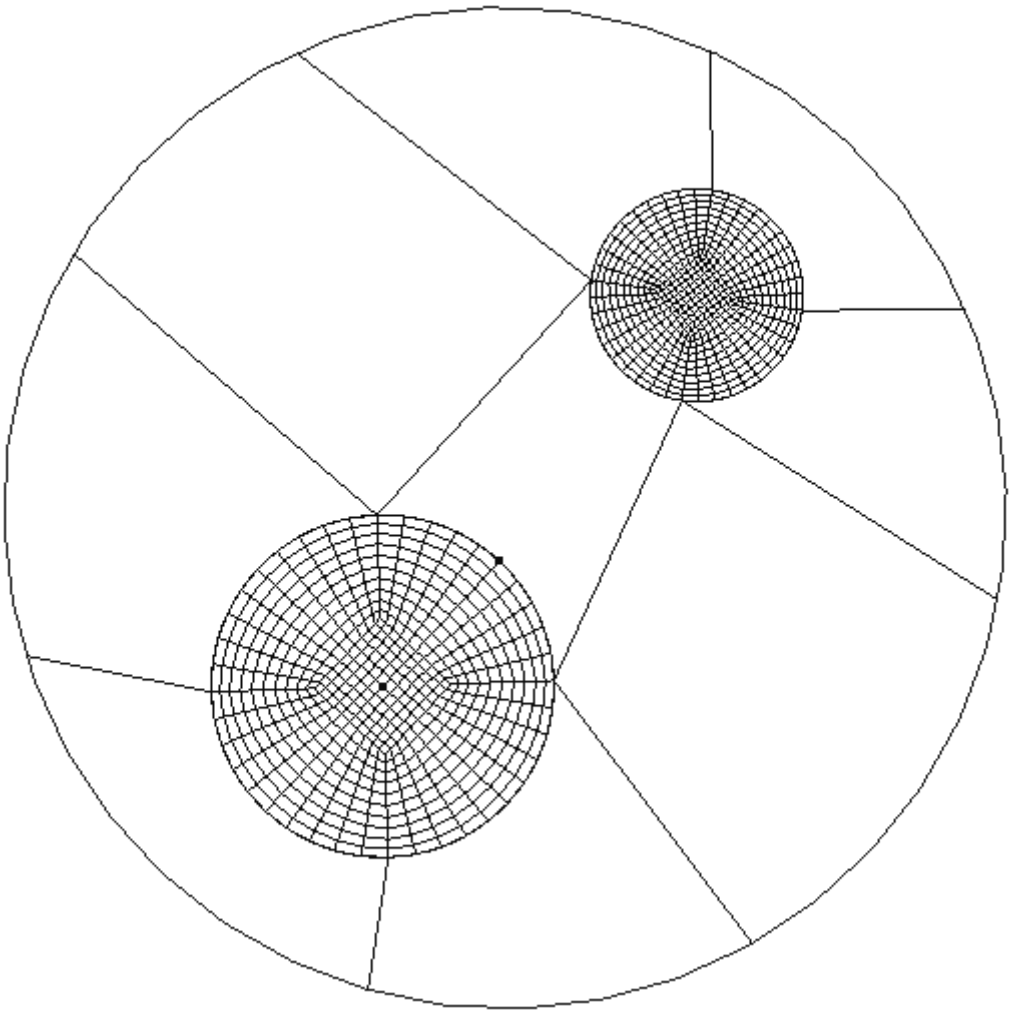


Figure C.1: Partial Gridded Geometry

After specifieng the big areas that will serve as independent quadratic shapes, the points of the connected corners will serve as the information points that connect up to four quadratic cells, and the gridding the whole cell will affect the neighbor quadratic cells with their points number on the edge. The fully gridded geometry can be seen in Fig. (C.2).

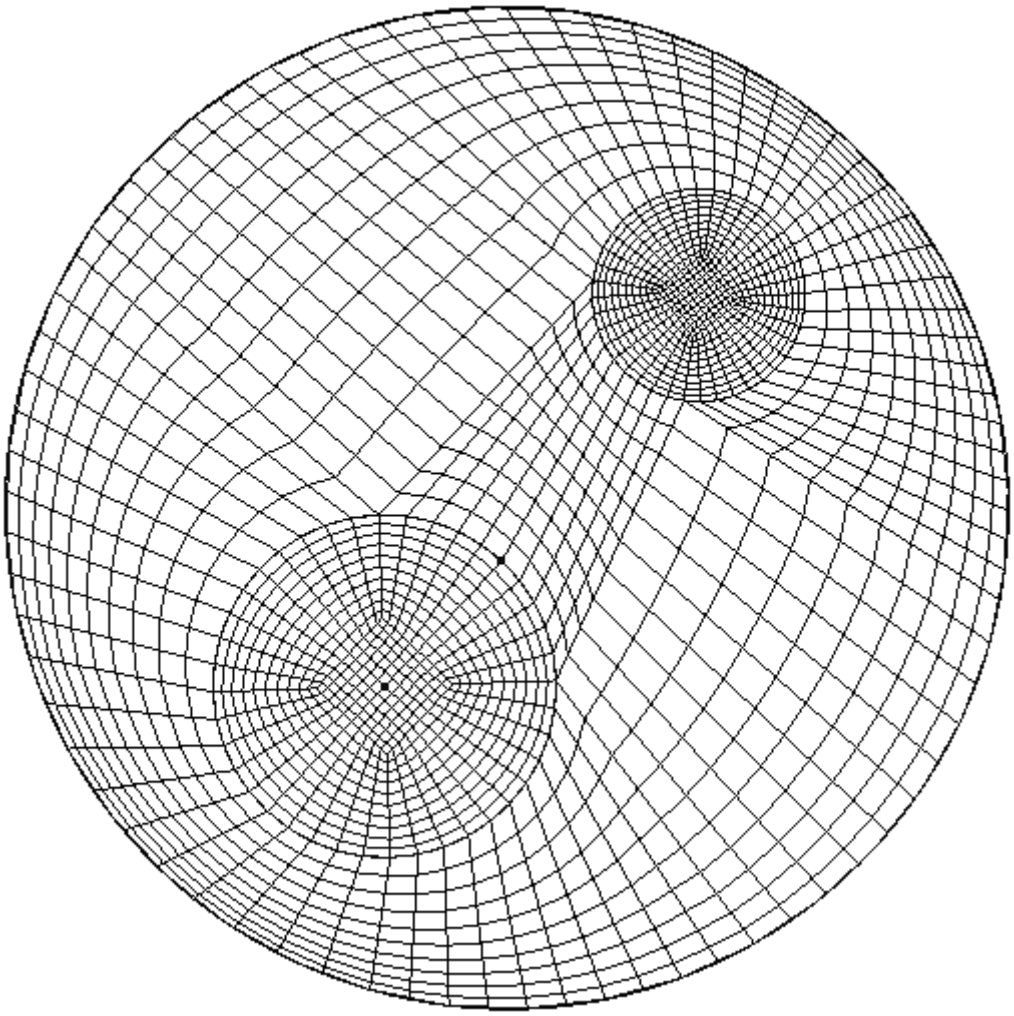


Figure C.2: Fully Gridded Geometry