**COSMASE Shortcourse Notes**
**EPF Lausanne 1996**

# *Modern Introduction to Grid Generation*

***Jochem Häuser***[c]

***Yang Xia***

***Department of Parallel Computing***
***Center of Logistics and Expert Systems***
***Salzgitter, Deutschland***


***Notes prepared for COSMASE Shortcourse in Grid Generation***
***EPF Lausanne, 23-27 September 1996***

# Contents

# List of Figures

# 1.0 Overview and Status of Modern Grid Generation

These lecture notes are the less mathematical version and an excerpt from a forthcoming book being currently written by the first author, entitled ***Grid Generation, Computational Fluid Dynamics, and Parallel Computing***, envisaged for the second half of 1997.

These notes are the continuation of courses in grid generation that were given at Mississippi State University (1996), for the International Space Course at the Technical University of Stuttgart 1995), at the Istituto per Applicacione del Calculo (IAC), Rome (1994), the International Space Course at the Technical University of Munich (1993), and at the Von Karman Institute, Brussels (1992). Material that has been prepared for presentations at the California Institute of Technology and the NASA Ames Research Center has also been partly included. However, in cooperation with P.R. Eiseman, PDC, White Pains, New York, substantial progess has been achieved in automatic blocking (***GridPro*** package). This material is presented here for the first time. To keep the number of pages for this article at a manageable size, the introduction to parallel computing and the section on interfacing multiblock grids to flow solvers had to be omitted. The reader interested in these aspects should contact the authors. Some recent publications can be found on the World Wide Web under ... In addition, a web address ***www.cle.de*** is being built where further information will be available. Furthermore, a CD-Rom containing a collection of 2D and 3D grids can be obtained (handling fee) from the authors.

With the advent of parallel computers, much more complex problems can be solved, provided computational grids of sufficient quality can be generated. The recent Workshops hely by the Aerothermodynamics Section of the European Space Agency's Technology Center (ESTEC) (1994, 1995) has proved that grid generation is one of the pacing items in CFD.

Although the emphasis is on grid generation in Aerodynamics, the codes presented can be used for any application that can be modeled by block-structured grids.

In these notes Numerical Grid Generation is presented in a way that is understandable to the scientist and engineer who is more a programmer, rather than a mathematician and whose main interest is in applications. The emphasis is on how to use and write a grid generation package. The best way to understand theory is to develop tools and to generate examples. Much emphasis is given to algorithms and data structures, which are presented from a real working code, namely the **G**rid$\star$ and **GridPro** [44] package, built by the authors. No claim is made that the packages presented here are the only or the best implementation of the concepts presented. Hence, a certain amount of low level information is presented. However, this is critical to understanding a general 2D and 3D multi-block grid generator, and is missing from other presentations on the subject.

The multi-block concept gets an additional motivation with the advent of parallel and distributed computing systems, which offer the promise of a quantum leap in computing power for Computational Fluid Dynamics (CFD). Multi-blocks may well be the key to achieve that quantum leap, as will be outlined in the chapter on grid generation on parallel computers, presenting the key issues of the parallelizing strategy, i.e. domain decomposition, scalability, and communication. A discussion of the dependence of convergence speed of an implicit flow solver on blocknumber for parallel systems wil also be given.

Since grid generation is a means to solve problems in CFD and related fields, a chapter on interfacing the final grid to the Euler or Navier-Stokes solver is provided. Although, in general, grids generated are slope continuous, higher order solvers need overlaps of 2 points in each direction. The software to generate this overlap for multi-block grids is also discussed.

First, the grid construction process is explained and demonstrated for several 2D examples. The object–oriented grid generation approach is demonstrated in 2D with a 284 block grid, viz. the "Hyperboloid Flare in the F4 windtunnel", using both the **G**rid$\star$ and **GridPro** packages. In 3D, we start with a simple mono-block grid for the double ellipsoid, explaining the 3D mapping procedure. After that, a multi-block grid for the double ellipsoid is generated to demonstrate the removal of singularities. Next, surface grid generation is demonstrated for the Hermes Space Plane, and a mono and a 7-block grid are generated. The last vehicle that is presented is the Space Shuttle Orbiter where surface grid generation is given for a 4-block grid, modeling a somewhat simplified Orbiter geometry, namely a Shuttle without body flap. In the next step, the surface grid and volume grid for a 94-block Euler mesh of a Shuttle with body flap are presented. The rationale for the choice of that topology is carefully explained. In the last step, it is shown how the 94-block Euler grid is converted to a 147-block Navier-Stokes grid, using the tools provided by **G**rid$\star$. In 3D, object–oriented grid generation in exemplied by the Electre body in the F4 windtunnel. Along with the examples, the questions of grid quality and grid adaptation are addressed.

## 2.0 Methods of Differential Geometry in Numerical Grid Generation

In the following a derivation of the most widely used formulas for nonorthogonal curvilinear coordinate systems ( Fig. 2.1) is given as needed in numerical grid generation. Variables $x^1,\ldots,x^n$ denote Cartesian coordinates and variables $\xi^1,\ldots,\xi^m$ are arbitrary curvilinear coordinates. The approach taken follows [9]. It is assumed that a one-to-one mapping $A \to M$ exists with

$$\mathbf{x}(\xi^1,\ldots,\xi^m) := \begin{pmatrix} x^1(\xi^1,\ldots,\xi^m) \\ \cdots \\ \cdots \\ x^n(\xi^1,\ldots,\xi^m) \end{pmatrix} \tag{2.1}$$

As an example, we consider the surface of a sphere where $R^m = R^2$ and $\xi^1 = \theta, -\pi/2 < \theta < \pi/2;\ \xi^2 = \psi, 0 < \psi < 2\pi$. The coordinates $x^1, x^2$ and $x^3$ correspond to the usual Cartesian coordinates $x$, $y$ and $z$. In the physical space, we have the surface of a sphere, whereas in the transformed space $\theta$ and $\psi$ form a rectangle.

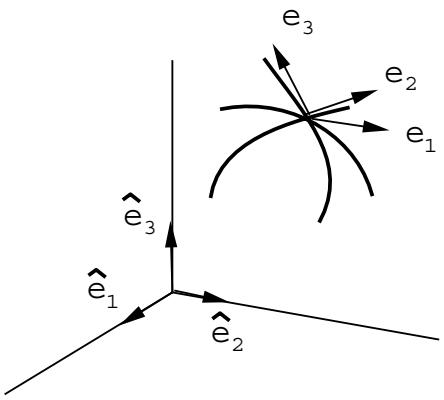


Figure 2.1: Curvilinear nonorthogonal Coordinate System. The contravariant base vectors $\mathbf{e}_i$ point in the tangential directions but are not unit vectors. Vectors $\hat{\mathbf{e}}_i$ are Cartesian unit vectors.

The tangent vectors or base vectors at a point $P \in M$ are defined by

$$\mathbf{e}_k := \frac{\partial \mathbf{x}}{\partial \xi^k};\ \ k = 1(1)m \tag{2.2}$$

The tangent vector $\mathbf{e}_k$ points in the direction of the respective coordinate line. These base vectors are called covariant base vectors. A second set of base vectors is defined by

$$\mathbf{e}^i \cdot \mathbf{e}_j = \delta^i_j \tag{2.3}$$

The $\mathbf{e}^i$ are called contravariant base vectors and are orthogonal to the respective covariant vectors for $i \neq j$. Covariant and contravariant vectors are related by the metric coefficients (see

below).
For the above example the two tangent vectors $\mathbf{e}_\theta$ and $\mathbf{e}_\psi$ are obtained by differentiating each of the functions $x(\theta,\psi), y(\theta,\psi)$ and $z(\theta,\psi)$ with respect to either $\theta$ or $\psi$.
A physical vector can either be represented by contravariant or covariant components

$$\mathbf{v} = v^i \mathbf{e}_i = v_j \mathbf{e}^j \tag{2.4}$$

where the summation convention is employed. In two dimensions, contravariant components of a vector are found by parallel projections onto the axes, whereas covariant components ar obtained by orthogonal projection.
According to its transformation behavior, a vector is called contravariant or covariant. Let $\underline{v}^i$be the components of a vector in the coordinate system described by the $x^i$ an let $v^i$ be the components in the system $\xi^i$

($\alpha$) A vector is a contravariant vector if its components transform in the same way as the coordinate differentials:

$$\begin{aligned} \mathrm{d}x^i &= \frac{\partial x^i}{\partial \xi^i}\mathrm{d}\xi^j \\ \underline{v}^i &= \frac{\partial x^i}{\partial \xi^j} v^j \end{aligned} \tag{2.5}$$

($\beta$) A vector is covariant if it transforms in the same way as the gradient of a scalar function:

$$\begin{aligned} \underline{\phi}(x^1,\ldots,x^n) &= \phi(\xi^1,\ldots,\xi^m), \\ \frac{\partial \phi}{\partial x^i} &= \frac{\partial \phi}{\partial \xi^j}\frac{\partial \xi^j}{\partial x^i} = \frac{\partial \xi^j}{\partial x^i}\frac{\partial \phi}{\partial \xi^j}, \\ \underline{v_i} &= \frac{\partial \xi^j}{\partial x^i} v_j. \end{aligned} \tag{2.6}$$

In order to measure the distance between neighboring points, the first fundamental form is introduced:

$$g_{ij} := \mathbf{e}_i \cdot \mathbf{e}_j. \tag{2.7}$$

The $g_{ij}$ are also called components of the metric tensor. The components of the inverse matrix are found from

$$g_{ij} g^{ik} = \delta^k_j. \tag{2.8}$$

The distance d$s$ between two neighboring points is given by

$$\mathrm{d}s = \sqrt{(g_{ij}\mathrm{d}\xi^i \mathrm{d}\xi^j)}. \tag{2.9}$$

In Cartesian coordinates there is no difference between covariant and contravariant components since there is no difference between covariant and contravariant base vectors. Therefore the

matrix $\hat{g}_{ij}$ (the $\hat{}$ denotes the Cartesian system) is the unit matrix.
The components of $g_{ij}$ in any other coordinate system can be directly calculated using the chain rule:

$$g_{ij} = \frac{\partial \hat{x}^k}{\partial x^i} \frac{\partial \hat{x}^l}{\partial x^j} \hat{g}_{kl}. \tag{2.10}$$

In order to find the transformation rules of derivatives of scalars, vectors and tensors, the Christoffel symbols of the first and second kinds are introduced. Suppose that coordinate $\xi^j$ is changed by an amount of $\mathrm{d}\xi^j$. This changes the base vector $\mathbf{e}_i$ by $\mathrm{d}\mathbf{e}_i$. Since $\mathrm{d}\mathbf{e}_i$ is a vector, it can be represented by the system of base vectors $\mathbf{e}_k$. Further, $\mathrm{d}\mathbf{e}_i$ is proportional to $\mathrm{d}\xi^j$. One can therefore write

$$\mathrm{d}\mathbf{e}_i = \Gamma^k_{ij} \mathrm{d}\xi^j \mathbf{e}_k, \tag{2.11}$$

where the symbols $\Gamma^k_{ij}$ are only coefficients of proportionality. These symbols are also called Christoffels symbols of the second kind. Taking the scalar product with $\mathbf{e}^k$, one obtains directly from equation 2.11

$$\Gamma^k_{ij} = \mathbf{e}_{i,j} \cdot \mathbf{e}^k, \tag{2.12}$$

where a comma denotes partial differentiation. The Christoffels symbols of the first kind are defined as

$$\Gamma_{ijk} := g_{il} \Gamma^l_{jk} \tag{2.13}$$

If the base vectors are independent of position, the Christoffels symbols vanish. They are, however, not tensor components, which follows directly from their transformation behaviour.
The relationship between the $\Gamma^k_{ij}$ and the $g_{lm}$ is found in the following way. Insertion of the definition of the metric components $g_{lm}$ in equation (2.7), into equation (2.12) and interchanging indices leads to

$$\Gamma^i_{jk} = \frac{1}{2} g^{ij} \left( \frac{\partial g_{jl}}{\partial \xi^k} + \frac{\partial g_{kl}}{\partial \xi^j} - \frac{\partial g_{kj}}{\partial \xi^l} \right). \tag{2.14}$$

If we use equation (2.2), the definition of the tangent vector, along with equation (2.12), one obtains the computational useful form

$$\Gamma^i_{jk} = \frac{\partial \xi^i}{\partial x^l} \frac{\partial^2 x^l}{\partial \xi^j \partial \xi^k}. \tag{2.15}$$

The knowledge of the grid point distribution, then, allows the numerical calculation of the Christoffels symbols. If we contract the Christoffels symbols, i.e. upper and lower indices are the same and are summed over, equation (2.14) yields

$$\Gamma^i_{ji} = \frac{1}{2} g^{im} \frac{\partial g_{mi}}{\partial \xi^j} = \frac{1}{\sqrt{g}} \frac{\partial \sqrt{g}}{\partial \xi^j} = (\ln \sqrt{g})_{,j}, \tag{2.16}$$

where $g$ is the determinant of the metric tensor, that is $\sqrt{g}$ is the Jacobian of the transformation. In two dimensions with curvilinear coordinates $\xi, \eta$ and Cartesian coordinates $x, y$, one finds

$$\begin{pmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{pmatrix} = J^{-1} \begin{pmatrix} y_\eta & -x_\eta \\ -y_\xi & x_\xi \end{pmatrix} \qquad \sqrt{g} = J = (x_\xi y_\eta - y_\xi x_\eta) \tag{2.17}$$

In curved space, partial differentiation is replaced by covariant differentiation which takes into account the fact that base vectors themselves have non-vanishing spatial derivatives. In the following, the nabla operator

$$\nabla := \mathbf{e}^i \frac{\partial}{\partial \xi^i} \tag{2.18}$$

is used. For the calculation of the cross product the Levi-Civita tensor is introduced.

$$\hat{\varepsilon}_{ijk} := \begin{cases} 1, & \text{for i} = 1,\ \text{j} = 2,\ \text{k} = 3 \text{ and all even permutations} \\ -1, & \text{for all odd permutations} \\ 0, & \text{if any two indices are the same} \end{cases}, \tag{2.19}$$

where $\hat{}$ again denotes the Cartesian coordinate system. From equation.(2.0) we know the transformation law for covariant components, and hence

$$\varepsilon_{ijk} = \frac{\partial \hat{x}^l}{\partial x^i} \frac{\partial \hat{x}^m}{\partial x^j} \frac{\partial \hat{x}^n}{\partial x^k} \hat{\varepsilon}_{lmn} = J \hat{\varepsilon}_{ijk}, \tag{2.20}$$

where $J$ is the Jacobian of the transformations. Forming the determinant from the components of equation (2.10), results in $g = J^2$, and therefore

$$\varepsilon_{ijk} = \sqrt{g} \hat{\varepsilon_{ijk}} \tag{2.21}$$

Raising the indices in equation (2.21) with the metric, one obtains

$$\varepsilon^{ijk} = \sqrt{g} \hat{\varepsilon^{ijk}} \tag{2.22}$$

The cross product of two vectors **a** and **b** in any coordinate frame is then written as

$$\mathbf{a} \times \mathbf{b} = a^i b^j \mathbf{e}_i \times \mathbf{e}_j = a^i b^j \varepsilon_{ijk} \mathbf{e}^k = \sqrt{g} \hat{\varepsilon}_{ijk} a^i b^j \mathbf{e}^k \tag{2.23}$$

It should be noted that equations (2.21) and (2.22) can also be derived by starting from the well-known relation in Cartesian coordinates

$$\hat{\mathbf{e}}_i \times \hat{\mathbf{e}}_j = \hat{\varepsilon}_{ijk} \hat{\mathbf{e}}_k \tag{2.24}$$

Insertion of

$$\hat{\mathbf{e}}_i = \frac{\partial x^l}{\partial \hat{x}^i} \mathbf{e}_l; \;\; \hat{\mathbf{e}}_j = \frac{\partial x^m}{\partial \hat{x}^j} \mathbf{e}_m; \;\; \hat{\mathbf{e}}_k = \frac{\partial x^k}{\partial \hat{x}^n} \mathbf{e}^n; \tag{2.25}$$

into equation (2.24) and multiplication of the resulting left and right sides by

$$\frac{\partial \hat{x}^i}{\partial x^p} \frac{\partial \hat{x}^j}{\partial x^r} \tag{2.26}$$

with summation over $i$ and $j$ finally leads to

$$\mathbf{e}_p \times \mathbf{e}_r = \sqrt{g} \hat{\varepsilon}_{prn} \mathbf{e}^n \tag{2.27}$$

With the above equations it is now possible to derive the transformation rules (i)-(viii) needed for the Euler or SWEs and for the grid generation equations themselves:

(i) Gradient of a scalar function $h$:

$$\nabla h = \mathbf{e}^i \frac{\partial}{\partial \xi^i} h = \mathbf{e}^i h_{,i} = g^{il} h_{,i} \mathbf{e}_l \tag{2.28}$$

(ii) Gradient of a vector field **u**

$$\nabla \mathbf{u} = \mathbf{e}^i \frac{\partial}{\partial \xi^i} (u^j \mathbf{e}_j) = \left( \frac{\partial u^j}{\partial \xi^i} + u^m \Gamma^j_{im} \right) \mathbf{e}^j \mathbf{e}_j \tag{2.29}$$

where equation (2.11) was used.

(iii) Divergence of a vector field **u**

$$\begin{aligned} \nabla \cdot \mathbf{u} &= \mathbf{e}^k \cdot \tfrac{\partial}{\partial \xi^k} (u^i \mathbf{e}_i) = \tfrac{\partial u^j}{\partial \xi^i} + \mathbf{e}^k \cdot \mathbf{e}_m u^i \Gamma^m_{ik} \\ &= \tfrac{\partial u^i}{\partial \xi^i} + u^i \Gamma^m_{im} = u^i_{,i} + \tfrac{1}{2g} g_{,j} u^i = \tfrac{1}{\sqrt{g}} (\sqrt{g} u^j)_{,j}, \end{aligned} \tag{2.30}$$

where equation (2.16) was used.

(iv) Cross product of two vectors (here base vectors $\mathbf{e}_i$, $\mathbf{e}_j$ since they are most often needed)

$$\mathbf{e}_i \times \mathbf{e}_j = \varepsilon_{ijk} e^k = \sqrt{g}\hat{\varepsilon}_{ijk}\mathbf{e}^k = J\hat{\varepsilon}_{ijk}\mathbf{e}^k \tag{2.31}$$

It should be noted that this formula dirctly gives an expression for the area $\mathrm{d}\mathbf{A}^k$ and the volume $\mathrm{d}V$.

$$\mathrm{d}\mathbf{A}^k := \mathrm{d}\mathbf{s}^i \times \mathrm{d}\mathbf{s}^j; \mathrm{d} := \mathrm{d}\mathbf{s}^k \cdot (\mathrm{d}\mathbf{s}^i \times \mathrm{d}\mathbf{s}^j) \tag{2.32}$$

where $i, j, k$ are all different. The vector $\mathrm{d}\mathbf{s}^i$ is of the form $\mathrm{d}\mathbf{s}^i := \mathbf{e}_i \mathrm{d}\xi^i$ (no summation over i). Thus we obtain

$$|\mathrm{d}\mathbf{A}^k| = J\mathrm{d}\xi^i \mathrm{d}\xi^j; \quad \mathrm{d}V = J\mathrm{d}\xi^i \mathrm{d}\xi^j \mathrm{d}\xi^k \tag{2.33}$$

(v) Delta operator applied to a scalar function $\psi$:

$$\Delta\psi = \nabla \cdot (\nabla\psi) = \mathbf{e}^k \cdot \frac{\partial}{\partial\xi^k}\left(\mathbf{e}_l g^{il}\frac{\partial\psi}{\partial\xi^i}\right) = g^{il}\psi_{,i,l} + \psi_{,i}\frac{1}{\sqrt{g}}(\sqrt{g}g^{ik})_k \tag{2.34}$$

where the components of the gradient of equation (2.28) were inserted into equation (2.30).

(vi) For transient solution areas, i.e. the solution area moves in the physical plane but is fixed in the computational domain, time derivatives in the two planes are related by

$$\left(\frac{\partial f}{\partial t}\right)_\xi = \left(\frac{\partial f}{\partial t}\right)_x + \left(\frac{\partial f}{\partial x^i}\right)_t \left(\frac{\partial x^i}{\partial t}\right)_\xi \tag{2.35}$$

where $(\partial x^i/\partial t)_\xi$ are the corresponding grid speeds, which can be calculated from two consecutive grid point distributions.

(vii) Relation between covariant and contravariant components of the metric tensor:

$$g^{ik} = \frac{\Delta^{ik}}{g} \tag{2.36}$$

where $\Delta^{ik}$ denotes cofactor $(i,k)$ of the matrix of the metric tensor. In two dimensions we have

$$g^{11} = g^{-1}g_{22}; \;\; g^{12} = g^{21} = -g^{-1}g_{12}; \;\; g^{22} = g^{-1}g_{11}; \tag{2.37}$$

(viii) Relationship between grid point distribution and Christoffels symbols, eqn.(16):

$$\Gamma^i_{jk} = \frac{\partial\xi^i}{\partial x^l}\frac{\partial^2 x^l}{\partial\xi^j\xi^k} \tag{2.38}$$

# 3.0 *Introduction and Overview of Numerical Grid Generation Techniques*

## 3.0.1 *General Remarks on Grid Generation Techniques*

CFD of today is marked by the simulation of flows past complex geometries and/or utilizing complex physics. A comparison between the impact of the numerical technique and the computational grid used, reveals that in many cases grid effects are the dominant factor on the accuracy of the flow solution. Despite this fact, the number of researchers working in grid generation is at least an order of magnitude smaller than the number of scientists active in CFD. The reason for this is most likely that publishing a paper in CFD is much more rewarding. First, a CFD paper is easier to publish since, e.g. a modification of a numerical scheme in 1D is sufficient to justify a new paper. Second, publishing a grid generation paper involves a large amount of programming (time consuming) and algorithm development as well as computer graphics to visualize the grid. In the past, algorithms and programming have been considered to be outside the engineering domain. Hence, the majority of the engineering software is still in Fortran. Data structures and Object-Oriented-Programming (OOP) are not taught in engineering courses and thus their importance is not recognized in this field. As a consequence, many codes in industry are not state of the art. Strange enough, the latest hardware is used together with the software concepts of the late fifties.

With the advent of massively parallel systems and high end graphics workstations a new level of performance in aerodynamic simulation can be achieved by the development of an integrated package, coined PAW (Parallel Aerospace Workbench). PAW would provide the aerospace engineer with a comprehensive, partly interactive package that, accepting the CAD geometry from the design office, provides modules for surface repair, followed by quasi-automatic grid generation (via a special language) and finally produces and visualizes the flow solution. Software development has to be based on standards like UNIX, ANSI C X11, Tcl/Tk, and Motif as well as on Open GL. For parallelization PVM (Parallel Virtual Machine) or MPI (Message Passing Interface) should be used (or a similar package), generating a hardware independent parallel flow solver.

## 3.0.2 *A Short History on Grid Generation*

Numerical grid generation has the dual distinction of being the youngest science in the area of numerical simulation and one of the most interesting fields of numerical research. Although conformal mappings were used in aerospace for airfoils, a general methodology for irregular 2D grids was first presented in the work of A. Winslow [1]. In 1974, a paper entitled ”Automatic numerical generation of body fitted curvilinear coordinate systems for fields containing any number of arbitrary two-dimensional bodies” appeared in the J. of Computational Physics [2], authored by Thompson, Thames, and Mastin. This paper can be considered a landmark paper, originating the field of boundary conformed grids, and making it possible to use the efficient techniques of finite differences and finite volumes for complex geometries.

The advent of high-speed computers has made flow computations past 3D aircraft and space-

craft as well as automobiles a reality. No longer are scientists and engineers restricted to wind tunnel and free flight experiments. On the numerical side, the development of high-speed computers as well as graphics workstations has allowed aerodynamicists to perform intricate calculations that were unthinkable a few decades ago. A clear understanding of many of the aspects of subsonic to hypersonic flows, not attainable by wind tunnel experiments, is one of the important results of these calculations. In addition, with the advent of massively parallel systems, a new era of CFD (Computational Fluid Dynamics) has started, which will allow realistic simulations of turbulent flows with real gas effects (high temperature) past complex 3D configurations. The purpose of these notes is to present a clear, vivid, comprehensive treatment of modern numerical grid generation, based on boundary conforming grids, with emphasis on the actual grid generation process as applied to complete geometries. Special emphasis is placed on algorithms and examples will be presented along with a discussion of data structures and advanced programming techniques, topics nearly almost always neglected in the scientific and engineering field. With the advent of modern programming languages like C and C++, a revolution in scientific programming is taking place. Engineers, used to Fortran since the late fifties, finally begin to realize that a new way of thinking is needed in software engineering, characterized by the concept of object-oriented-programming. Only with this approach it can be hoped to reduce the size and complexity of source code, to fully use available hardware, to write and maintain very large software packages and to implement them on arbitrary computer architectures. Grid generation packages like **GridPro**, and **G**rid$\star$ code [20] are based exclusively on C, X-Windows, and Open Gl, allowing full portability. Although nearly two decades have past since the work of Thompson et al., the development of generally applicable and easy to use 3D grid generation codes still presents a major challenge. In the case of structured grids it was soon recognized that branch cuts and slits or slabs did not offer sufficient geometrical flexibility. Therefore, the majority of structured grid generation codes now employs the multiblock concept, which is unstructured on the block level but results in a structured grid within a block. To determine the block topology of a complex SD (solution domain) requires a certain effort. The recent ***AZ–manager*** code (see Sec.**??**) is a major step forward in automatic topology definition.

Grid generation has to embedded in the overall solution process that is depicted in Fig. 3.1. Most important are tools for data conversion to directly use CAD data and to automatically interface a grid to the flow solver.

In the chart of Fig.3.1 the complete modelization process to obtain a flow solution is shown. It is assumed that a surface description of the vehicle is available, generally coming from the design office in a CAD format, e.g. CATIA. This information has to be processed to convert it into a format that can be read by the modules of **G**rid$\star$. Various conversion modules are available. The chart then shows the grid generation process, where each module generates output that can be read by the following modules, producing some kind of pipeline. The interfacing to the flow solver is done automatically and results can be visualized directly by the Plot3D package from NASA Ames or by any other package that supports this format, e.g. the widely known Tecplot software from AMTEC Engineering.

In practice, problems may be encountered with the grid generated and the flow solution may not be obtained, either because there is no convergence of the flow solution, or after a few iterations the solution may blow up. This depends very much on the flow physics and to a large extent on the experience of the user. These codes do not yet have reached a stage where they can be used as a black box. A case with Ma of 25 and thermo-chemical nonequilibrium Navier-Stokes equations is of course much harder to solve than an Euler problem at Ma 2. The stiffness of the equations

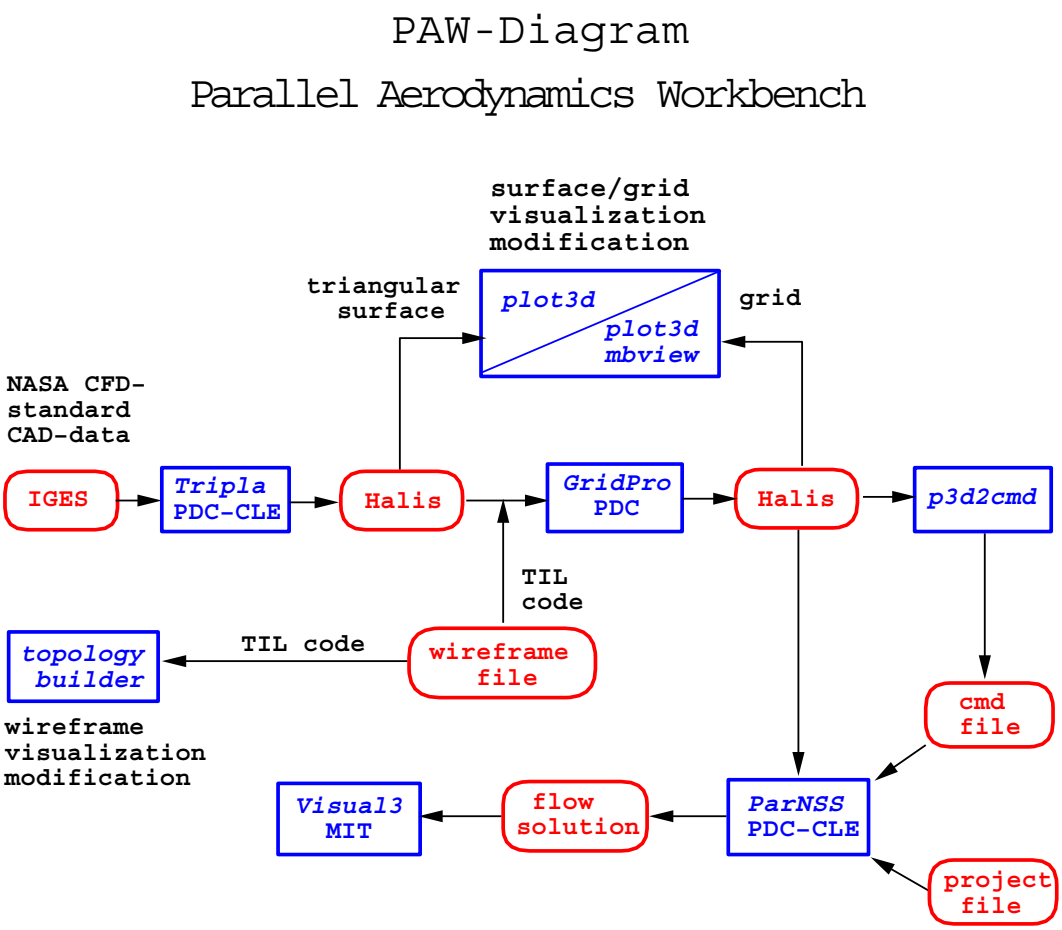


Figure 3.1: In this chart the complete modelization process starting from the surface description of the vehicle, coming from the design office, to the visualized results of a 3D flow solution is depicted. The necessary data files along with the corresponding software modules are shown.

depends on the physics and also on the grid. A Navier-Stokes grid for a 3D vehicle with a cell aspect ratio of $10^6$ in the boundary layer is not only much more difficult to generate but it is also much harder for the numerical scheme to converge to the physical solution in an efficient way. Although the emphasis of these lecture notes is on grid generation, some results for Euler and N-S calculations are presented.

### 3.0.3 *What Is a Good Grid*

First, the discussion will be restricted to multi–block grids. It is felt that any kind of geometry can be gridded by this approach, resulting in a superior grid quality when compared with unstructured grids, in particular for viscous flows. However, it should be kept in mind that grids of tens of thousands of blocks have to be generated (automatically). On the block level the grid is completely unstructured. The large number of blocks is needed because of complex topology, for instance, there may be hundreds of bodies in the flowfield, or, because the solver is run on a MIMD (Multiple Instruction Multiple Data) massively parallel system that may comprise several thousand processors.

Second, the answer is straightforward, namely that it is not possible to deciding by simply looking at a grid whether it is good or not. However, there exist **several criteria that tell when grid quality is not good**.

- ▷ high degree of skewness
- ▷ abrupt changes in grid spacing
- ▷ insufficient grid line continuity ($C^0$, $C^1$, or $C^2$)
- ▷ nonalignment of the grid with the flow
- ▷ insufficient resolution to resolve proper physical length scales
- ▷ grid topology not well suited to sufficiently cover the flow physics
- ▷ grid lacks special features needed by physical submodels
- ▷ grid is not singularity free (e.g. at stagnation point)

The first three points are mostly independent of the flow physics. Their effect is that they reduce the order of the numerical scheme, and hence the accuracy of the results may deteriorate. However, the magnitude of these effects will depend on the underlying flow physics.

The other topics are strongly coupled with the flow physics. It is known from numerical error investigations that locally 1D flow (proper grid alignmnet) will give an improved numerical accuracy, e.g. grid lines conforming to the boundary of a body will produce better accuracy. Length scales have to be properly resolved (e.g. boundary layer), otherwise the physics cannot be adequately represented, resulting in, for example, the wrong drag or skin friction. Grid topology also has an influence on the physical results. An O-type topology for a 4 element airfoil may be adequate for Euler solutions, but may be insufficient to resolve the wake resulting from viscous flow, necessitating a C-H type topology. Physical submodels may, for example, need grid line orthogonality at fixed boundaries. This is the case for some turbulence models to determine the distance from the wall. Grids having a singularity, e.g. at the nose part of an aircraft, may lead to substantially increased computing time because of a slowdown in convergence, since no information can cross the singularity.

### 3.0.4 *Aspects of Multiblock Grid Generation*

Structured grids use general curvilinear coordinates to produce a body fitted mesh. This has the advantage that boundaries can be exactly described and hence BCs can be accurately modeled. In early grid generation it was attempted to always map the physical solution domain (SD) to a single rectangle or a single box in the computational domain (CD). For multiply connected SDs, branch cuts had to be introduced, a procedure well known in complex function theory and analytic mapping of 2D domains, e.g. the Joukowski airfoil profile. However, it became soon obvious that certain grid line configurations could not be obtained. If one considers for example the 2D flow past an infinitely long cylinder, with a small enough *Re* number, it would be advantageous if the grid line distribution would be similar to the streamline pattern. A 2D grid around a circle which is mapped on a single rectangle necessarily has O-type topology, unless additional slits (double valued line or surface) or slabs (blocks are cut out of the SD) are introduced. The main advantage of the structured approach, namely that one only has to deal with a rectangle or a box, that is a code is needing only 2 or 3 ”for” loops (C language), has been lost. The conclusion is that this amount of structuredness is too rigid, and some degree of unstructuredness has to be introduced. From differential geometry the concept of an atlas consisting of a number of charts is known. The set of charts covers the atlas where charts may be overlapping. Each chart is mapped onto a single rectangle. In addition, now the connectivity of the charts has to be determined. This directly leads to the multiblock concept, which provides the necessary geometrical flexibility and the computational efficiency for the finite volume or finite difference techniques.

For a vehicle like the Space Shuttle a variety of grids can be constructed (see Sec. **??**). One can start with a simple monoblock grid that wraps around the vehicle in an O-type fashion. This always leads to a singular line, which normally occurs in the nose region. This line needs special treatment in the flow solution. It has been observed that convergence rate is reduced, although special numerical schemes have been devised to alleviate this problem. Furthermore, a singularity invariably leads to a clustering of grid points in an area where they are not needed. Hence, computing time may be increased substantially. In addition, with a monoblock mesh gridline topology is fixed. Additional requirements with regard to grid uniformity and orthogonality cannot be matched. The fourblock grid shown in Sec. **??** removes the singularity but otherwise retains the O-type structure. The number of gridpoints is reduced substantially. The grid is smooth across block boundaries. Therefore, no special discretization in the flow solver across boundaries is needed. A 94 block Euler grid for the Shuttle has been generated, which includes the body flap. The larger number of blocks is needed to get a special grid line topology, and to better represent the flow physics (see discussion in Sec.**??**).

Since multiblock grids are unstructured on the block level, information about block connectivity is needed along with the of each block. For reasons of geometrical flexibility it is mandatory that each block has its own local coordinate system. Hence blocks can be rotated with respect to each other (Sec. 3.2.3). Slope continuity across neighboring block boundaries is achieved by overlapping edges (2D) or faces (3D). For grid generation an overlap of exactly 1 row or 1 column is needed (2D). The flow solver **ParNSS** needs an overlap of 2 faces.

The solution domain is subdivided into a set of blocks or segments (in the following the words block and segment are used interchangeably). The overlap feature facilitates the construction of the flow solver substantially and allows the direct parallelization of the code on massively parallel

systems using message passing.

Each (curvilinear) block in the physical plane is mapped onto a Cartesian block in the computational plane (CP). The actual solution domain on which the governing physical equations are solved is therefore a set of connected, regular blocks in the computational plane. However, this does not mean that the solution domain in the computational plane has a regular structure, rather it may look fairly fragmented. For the parallelization the important point is that there is no nearest neighbor relation for the blocks. Therefore communication among blocks follows an irregular pattern. A parallel architecture that is based on nearest neighbor communication, e.g. lattice gauge theory problems, will not perform well for complex aerodynamic applications, simply because of the communication overhead, caused by random communication.

The grid point distribution within each block is generated by the solution of a set of three Poisson equations, one for each coordinate direction. The right hand side of the Poisson equations is used for grid point control and is determined from the specified grid point distribution on the surfaces. That means that first the right hand side is determined for each grid point on a surface and then the values of the control functions are interpolated into the interior of the solution domain, with some additional smoothing. In this context a grid point is denoted as boundary point if it lies on one of the six faces of a block in the CP. However, one has to discern between physical boundary points on fixed surfaces that can be used for computation of the right hand side of the Poisson equation and matching boundary points on overlap surfaces connecting neighboring blocks. The positions of the latter ones are not known a priori but are determined in the solution process (see above). All grid point positions on the faces of a block must be known before the Poisson equations for this block can be solved to determine the grid point positions of the interior points.

A coordinate transformation of the governing physical equations and their respective boundary conditions from the physical plane to the computational plane is also required.

## 3.1 *Equations of Numerical Grid Generation*

### 3.1.1 *Elliptic Equations for 2D Grid Generation*

In the following the elliptic PDEs used in the grid generation process will be derived, using the mathematical tools of the previous section. First, we wish to explain the motives leading to this approach. In the past, finite differences and finite elements have been used extensively to solve computational problems. The latter have been mainly used in structural mechanics. About two decades ago, Thompson et al.( see e.g. [2]) introduced the concept of a boundary fitted grid (BFG) or structureg grid (SG), that is a grid whose grid lines are aligned with the contours of the body. Clearly, such a grid has to have coordinate lines, i.e. it cannot be completely unstructured. In computational fluid dynamics (CFD) in general and in high speed flows in particular many flow situations are encountered where the flow in the vicinity of the body is aligned with the surface, i.e. there is a prevailing flow direction. This is especially true in the case of hypersonic flow because of the high kinetic energy. The use of a SG, allows the alignment of the grid in that direction, resulting in locally quasi 1D flow. Hence, numerical diffusion can be reduced, i.e. better accuracy is achieved. A BFG exactly matches curved boundaries, and for complex SDs generally will consist by a set of so called blocks, which are connected. In the approach taken by the authors, a SD may be covered by a set of hundreds or even thousands of blocks. Second, SGs can be made orthogonal at boundaries, facilitating the implementation of BCs (Boundary Condition) and also increasing the numerical accuracy at boundaries. Furthermore, orthogonality will increase the accuracy when algebraic turbulence models are employed. In the solution of the N-S (Navier-Stokes) equations, the BL (Boundary Layer) must be resolved. This demands that the grid is closely wrapped around the body to describe the physics of the BL (some 32 layers are used in general for SGs or UGs). Here some type of SG is indispensable. In addition, to describe the surface of the body a structured approach is better suited.

The resolution of the BL leads to large anisotropies in the length scales in the directions along and off the body. Since the time-step size in an explicit scheme is governed by the smallest length scale or, in the case of chemical reacting flow, by the magnitude of the chemical production terms, extremely small time steps will be necessary. This behavior is not demanded by accuracy but to retain the stability of the scheme. Thus, implicit schemes will be of advantage. In order to invert the implicit operator, factorization is generally used, resulting in two factors if LU decomposition (that is, factoring in the direction of the plus and minus eigenvalues of the Jacobians) is employed, or in three factors if the coordinate directions are used. For the unstructured approach there is no direct way to perform this type of factorization. Moreover, the use of the so-called thin layer approach, that is retaining the viscous terms only in the direction off the body, reduces the computer time by about 30 %. Since there are no coordinate lines in the UG, this simplification is not possible. A fairly complex procedure would be needed to artificially construct these lines. Moreover, the flow solver based on the UG is substantially slower than for the SG. This is due to the more complicated data structure needed for UGs. Factors of 3, and by some authors of up to 10, have been given in the literature.

In order to calculate heat loads for vehicles flying below Ma 8, turbulence models have to be used, for example, the difference in surface temperature for Sänger at cruising speed (around Ma 5) is about 500 K depending on laminar or turbulent flow calculations. Depending on the real

surface temperature encountered in flight a totally different type of vehicle has to be designed since a cooled Titanium wall cannot withstand a temperature of 1300 K for a long period (about 20 minutes). Therefore turbulent calculations are of high importance. Only a SG can provide the alignment along with the orthogonality at the boundary to accurately perform these calculations.

In general, SG provide sophisticated means both for clustering and adaptation using redistribution or local enrichment techniques. A comparison of these two approaches is given by Dannenhoffer [2] where local enrichment gives somewhat better results. However, it is much more costly to use. In many cases of practical interest, for example fine resolution of a bow shock or a canopy shock and in situations where shocks are reflected, the alignment of the grid can result in a more accurate solution than randomly filling the space with an enormously large number of smaller and smaller tetrahedrons or hexahedrons.

The highest degree of freedom of course is obtained in UGs. The majority of physical phenomena encountered in external and also in internal flows exhibit certain well ordered structures, such as a bow shock or system of reflected shocks or some type of shock-shock interaction, which can be perfectly matched by adaptive grid alignment, coupled to the solution process.It is therefore felt, however, that mesh redistribution and alignment is totally adequate for the major part of the flow situations encountered in CFD, especially in aerodynamics.

Only if a very complex wave pattern evolves due to special physical phenomena, for example, generating dozens of shock waves coming from an explosion, the UG seems to be advantageous.

In addition, the coupling of SGs with UGs is possible as has been shown by Weatherill and Shaw et al. [40]. Such a grid is called a hybrid grid.

A mixture between the boundary fitted grid approach and the completely unstructured approach is the use of multiblock grids, which has been employed in the **G**rid$\star$[27] and in the **GridPro**[35] packages. On the block level the grid is unstructured, that is the blocks itself can be considered as finite elements, but within each block the grid is structured and slope continuity is provided across block boundaries (on the coarsest level). If a block is refined locally, this feature cannot be maintained. Recently a 94 block grid (Euler) and a 147 block grid (N-S) for the Space Shuttle have been generated, demonstrating the variability of this approach. Moreover, the automatic zoning feature of **GridPro**has been used to generate a grid of several thousand blocks (see Sec. **??**).

For the derivation of the transformed grid generation equations one starts from the original Poisson equations and then the role of the dependent and independent variables is interchanged.

$$\Delta\xi = P; \quad \Delta\eta = Q \tag{3.1}$$

Using (2.34), the transformation equation of the Laplacian, and the fact that $\xi$ and $\eta$ are coordinates themselves, we find

$$\Delta\xi = g^{ik}(\xi_{,i,k} - \Gamma^j_{ik}\xi_{,j}) = -g^{ik}\mathbf{e}_{i,k}\cdot\mathbf{e}^1 \tag{3.2}$$

$$\Delta\xi = \frac{-1}{\sqrt{g}}[y_\eta(g^{11}x_{\xi\xi} + 2g^{12}x_{\xi\eta} + g^{22}x_{\eta\eta}) - x_\eta(g^{11}y_{\xi\xi} + 2g^{12}y_{\xi\eta} + g^{22}y_{\eta\eta})] = P \tag{3.3}$$

In a similar way we obtain for the $\eta$ coordinate

$$\Delta\eta = \frac{+1}{\sqrt{g}}[y_\xi(g^{11}x_{\xi\xi} + 2g^{12}x_{\xi\eta} + g^{22}x_{\eta\eta}) - x_\xi(g^{11}y_{\xi\xi} + 2g^{12}y_{\xi\eta} + g^{22}y_{\eta\eta})] = Q \tag{3.4}$$

By means of (2.37), the $g^{ij}$ can be expressed in terms of $g_{ij}$. For stability reasons in the numerical iterative solution of the above system of equations, the above equations are rewritten in the form

$$\begin{array}{l} g_{22}x_{\xi\xi} - 2g_{12}x_{\xi\eta} + g_{11}x_{\eta\eta} + g(x_\xi P + x_\eta Q) = 0 \\ g_{22}y_{\xi\xi} - 2g_{12}y_{\xi\eta} + g_{11}y_{\eta\eta} + g(y_\xi P + y_\eta Q) = 0 \end{array} \tag{3.5}$$

These equations are quasilinear, where the non-linearity appears in the expression for the metric coefficients.

For 2D, using the original functions P and Q and coordinates $x$, $y$ and $\xi$, $\eta$, one obtains:

$$\begin{array}{l} g_{22}x_{\xi\xi} - 2g_{12}x_{\xi\eta} + g_{11}x_{\eta\eta} + g(x_\xi P + x_\eta Q) = 0 \\ g_{22}y_{\xi\xi} - 2g_{12}y_{\xi\eta} + g_{11}y_{\eta\eta} + g(y_\xi P + y_\eta Q) = 0 \end{array} \tag{3.6}$$

where

$$\begin{array}{lll} g_{11} = x_\xi^2 + y_\xi^2 & g_{12} = g_{21} = x_\xi x_\eta + y_\xi y_\eta & g_{22} = x_\eta^2 + y_\eta^2 \\ g_{11} = g^{22}g & g_{12} = g_{21} = -gg^{12} = -gg^{21} & g_{22} = gg^{11} \\ g = (x_\xi y_\eta - x_\xi y_\eta)^2 & & \end{array} \tag{3.7}$$

is used. The numerical solution of Eqs.(3.6) along with specified control functions P, Q as well as proper BCs is straightforward, and a large number of schemes is available. Here a very simple approach is taken, namely the solution by Successive-Over-Relaxation (SOR). The second derivatives are described in the form

$$\begin{array}{l} (x_{\xi\xi})_{i,j} = x_{i+1,j} - 2x_{i,j} + x_{i-1,j} \\ (x_{\eta\eta})_{i,j} = x_{i,j+1} - 2x_{i,j} + x_{i,j-1} \\ (x_{\xi\eta})_{i,j} = 1/4(x_{i+1,j+1} - x_{i-1,j+1} - x_{i+1,j-1} + x_{i-1,j-1}) \end{array} \tag{3.8}$$

Solving the first of Eqs.(3.6) for $x_{i,j}$ yields the following scheme

$$
\begin{aligned}
x_{i,j} = 1/2(\alpha_{i,j}+\gamma_{i,j})^{-1} \quad \{ \quad & \alpha_{i,j}(x_{i+1,j}-x_{i-1,j})- \\
& \beta_{i,j}(x_{i+1,j+1}-x_{i-1,j+1}-x_{i+1,j-1}+x_{i-1,j-1})+ \\
& \gamma_{i,j}(x_{i,j+1}-x_{i,j-1})+ \\
& 1/2J^2_{i,j}P_{i,j}(x_{i+1,j}-x_{i-1,j})+ \\
& 1/2J^2_{i,j}Q_{i,j}(x_{i,j+1}-x_{i,j-1})\Big\}
\end{aligned}
\tag{3.9}
$$

where the notation $J^2 = g$, $\alpha_{i,j} = g_{22}$, $2\beta_{i,j} = g_{12}$, and $\gamma_{i,j} = g_{11}$ was used. Overrelaxation is achieved by computing the new values from

$$
x_{new} = x_{old} + \omega(x - x_{old}) \quad 1 \leq \omega < 2 \tag{3.10}
$$

The solution process for a multiblock grid is achieved by updating the boundaries of each block, i.e. by receiving the proper data from neighboring blocks and by sending overlapping data to neighboring blocks. Then one iteration step is performed using these boundary data, iterating on all interior points. After that, those newly iterated points which are part of the overlap are used to update the boundary points of neighboring blocks and the whole cycle starts again, until a certain number of iterations has been performed or until the change in the solution of two successive iterations is smaller than a specified bound.

### 3.1.2 *Elliptic Equations for Surface Grid Generation*

A curved surface can be defined by the triple $\begin{pmatrix} x \\ y \\ z \end{pmatrix}(u,v)$,
where u and v parametrize the surface. Defining a new coordinate system on such a surface only means another way of parametrization of the surface, i.e. a transformation from $(u,v)$ to $(\xi,\eta)$, which is a mapping from $R^2 \rightarrow R^3$. To obtain the new coordinates Poisson equations are used. The derivation for the surface grid generation equations is similar as in the 2D or 3D case. Using the $\nabla$ operator of equation 2.18, the surface grid generation equations take the form

$$
\begin{aligned}
\Delta\xi &= g_{ik}\left(\xi_{,i,k} - \Gamma^j_{ik}\xi_{,j}\right) = P \\
\Delta\eta &= g_{ik}\left(\eta_{,i,k} - \Gamma^j_{ik}\eta_{,j}\right) = Q
\end{aligned}
\tag{3.11}
$$

In this case the derivatives are with respect to $u$ and $v$, and not with respect to $\xi$ and $\eta$. Therefore, the metric is given by the transformation of the parameter space $(u,v)$ to physical space $(x,y,z)$, i.e $e_i = \partial_i(x,y,z)$ with $\partial_1 = \partial_u$ and $\partial_2 = \partial_v$. To obtain the familiar form of the equations, i.e. $(u,v)$ values are the boundary points in the $(\xi,\eta)$ grids in the computational plane (CP), the dependent and independent variables in Eqs. 3.11 need to be interchanged. Introducing the matrix $\mathbf{M}$ of the contravariant row vectors from the transformation $(u,v) \rightarrow (\xi,\eta)$,

$$
\mathbf{M} = \begin{pmatrix} \xi_u & \eta_u \\ \xi_v & \eta_v \end{pmatrix} \tag{3.12}
$$

and the inverse matrix, $M^{-1}$, formed by the covariant columnvectors,

$$\mathbf{M^{-1}} = \begin{pmatrix} u_\xi & u_\eta \\ v_\xi & v_\eta \end{pmatrix}, \tag{3.13}$$

along with the definition of $\mathbf{S} := (\xi, \eta)^T$ the surface grid equations Eqs. 3.11 take the following form:

$$g^{ik}\mathbf{S}_{,i,k} - \mathbf{M}\begin{pmatrix} g^{ik}\Gamma^1_{ik} \\ g^{ik}\Gamma^2_{ik} \end{pmatrix} = \begin{pmatrix} P \\ Q \end{pmatrix}. \tag{3.14}$$

From this folllows

$$\mathbf{M^{-1}}\begin{pmatrix} g^{ik}\xi_{,i,k} \\ g^{ik}\eta_{,i,k} \end{pmatrix} - \mathbf{M^{-1}}\begin{pmatrix} P \\ Q \end{pmatrix} = \begin{pmatrix} g^{ik}\Gamma^1_{ik} \\ g^{ik}\Gamma^2_{ik} \end{pmatrix} \tag{3.15}$$

The surface metric induces, in comparison with the plain 2D case, additional terms that are independent of $\xi, \eta$ and act as some kind of control functions for the grid line distribution. To determine the final form of the equations the role of the independent and the dependent variables in the second derivatives $\xi_{,i,k}, \eta_{,i,k}$ have to be interchanged. This is done as follows. The matrix of covariant row vectors $\mathbf{M}$ can be written in the form

$$\mathbf{M} = \begin{pmatrix} \xi_u & \xi_v \\ \eta_u & \eta_v \end{pmatrix} = \frac{\mathbf{1}}{\mathbf{J}}\begin{pmatrix} v_\eta & -u_\eta \\ -v_\eta & u_\xi \end{pmatrix} \tag{3.16}$$

Hence, $\xi_{uu} = (\xi_u)_u = \left(\frac{v_\eta}{J}\right)_u$ where $\partial_u = \xi_u\partial_\xi + \eta_u\partial_\eta$. This results in the following second derivatives

$$\xi_{uu} = \frac{1}{J^2}\left\{-\xi_u\left(v_\eta^2 u_{\xi\xi} - 2v_\xi v_\eta u_{\xi\eta} + v_\xi^2 u_{\eta\eta}\right) - \xi_v\left(v_\eta^2 v_{\xi\xi} - 2v_\xi v_\eta v_{\xi\eta} + v_\xi^2 v_{\eta\eta}\right)\right\} \tag{3.17}$$

$$\begin{aligned}\xi_{vv} = \tfrac{1}{J^2}\{ \quad & -\xi_u\left(u_\eta^2 u_{\xi\xi} - 2u_\xi u_\eta u_{\xi\eta} + u_\xi^2 u_{\eta\eta}\right) - \\ & \xi_v\left(u_\eta^2 v_{\xi\xi} - 2u_\xi u_\eta v_{\xi\eta} + u_\xi^2 v_{\eta\eta}\,)\right\}\end{aligned} \tag{3.18}$$

$$\begin{aligned}\xi_{uv} = \tfrac{1}{J^2}\{ \quad & \xi_u\left(u_\eta v_\eta u_{\xi\xi} - \left(u_\eta v_\xi + v_\eta u_\xi\right)u_{\xi\eta} + u_\xi v_\xi u_{\eta\eta}\right) + \\ & \xi_v\left(u_\eta v_\eta v_{\xi\xi} - \left(u_\eta v_\xi + v_\eta u_\xi\right)v_{\xi\eta} + u_\xi v_\xi v_{\eta\eta}\right)\ \}\end{aligned} \tag{3.19}$$

$$\begin{aligned}\eta_{uu} = \tfrac{1}{J^2}\{ \quad & -\eta_u\left(v_\eta^2 u_{\xi\xi} - 2v_\xi v_\eta u_{\xi\eta} + v_\xi^2 u_{\eta\eta}\right) - \\ & \eta_v\left(v_\eta^2 v_{\xi\xi} - 2v_\xi v_\eta v_{\xi\eta} + v_\xi^2 v_{\eta\eta}\right)\ \}\end{aligned} \tag{3.20}$$

The quantities $\eta_{vv}$ and $\eta_{uv}$ are transformed similar. With the definition of $\mathbf{S}$ stated above the formulas can be summarized as:

$$\mathbf{S}_{uu} = -\frac{1}{J^2}\mathbf{M}\begin{pmatrix} v_\eta^2 u_{\xi\xi} - 2v_\xi v_\eta u_{\xi\eta} + v_\xi^2 u_{\eta\eta} \\ v_\eta^2 v_{\xi\xi} - 2v_\xi v_\eta v_{\xi\eta} + v_\xi^2 v_{\eta\eta} \end{pmatrix} =: -\frac{1}{J^2}\mathbf{Ma} \tag{3.21}$$

$$\mathbf{S}_{uv} = \frac{1}{J^2}\mathbf{M}\begin{pmatrix} u_\eta v_\eta u_{\xi\xi} - \left(u_\eta v_\xi + v_\eta u_\xi\right) u_{\xi\eta} + u_\xi v_\xi u_{\eta\eta} \\ u_\eta v_\eta v_{\xi\xi} - \left(u_\eta v_\xi + v_\eta u_\xi\right) v_{\xi\eta} + u_\xi v_\xi v_{\eta\eta} \end{pmatrix} =: -\frac{1}{J^2}\mathbf{Mb} \tag{3.22}$$

$$\mathbf{S}_{vv} = -\frac{1}{J^2}\mathbf{M}\begin{pmatrix} u_\eta^2 u_{\xi\xi} - 2u_\xi u_\eta u_{\xi\eta} + u_\xi^2 u_{\eta\eta} \\ u_\eta^2 v_{\xi\xi} - 2u_\xi u_\eta v_{\xi\eta} + u_\xi^2 v_{\eta\eta} \end{pmatrix} =: -\frac{1}{J^2}\mathbf{Mc} \tag{3.23}$$

Sustituting the above formulas for $\mathbf{S}_{,i,j}$ into Eq. 3.15 yields

$$\frac{1}{J^2}\left(g^{11}\mathbf{a} + 2g^{12}\mathbf{b} + g^{22}\mathbf{c}\right) + \mathbf{M}^{-1}\begin{pmatrix} P \\ Q \end{pmatrix} = -\begin{pmatrix} g^{ik}\Gamma^1_{ik} \\ g^{ik}\Gamma^2_{ik} \end{pmatrix} \tag{3.24}$$

Sorting the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ with respect to the second derivatives of $\mathbf{r} = (u, v)$, then one obtains

$$\frac{1}{J^2}\left(\alpha\mathbf{r}_{\xi\xi} - 2\beta\mathbf{r}_{\xi\eta} + \gamma\mathbf{r}_{\eta\eta}\right) + \mathbf{M}^{-1}\begin{pmatrix} P \\ Q \end{pmatrix} = -\begin{pmatrix} g^{ik}\Gamma^1_{ik} \\ g^{ik}\Gamma^2_{ik} \end{pmatrix} \tag{3.25}$$

where

$$\begin{aligned} \alpha &= g^{11}v_\eta^2 - 2g^{12}u_\eta v_\eta + g^{22}u_\eta^2 \\ \beta &= g^{11}v_\xi v_\eta - g^{12}(u_\xi v_\eta + v_\xi u_\eta) + g^{22}u_\xi u_\eta \\ \gamma &= g^{11}v_\xi^2 - 2g^{12}u_\xi v_\xi + g^{22}u_\xi^2 \end{aligned} \tag{3.26}$$

$\mathbf{M}^{-1}$ has been defined previously. In the numerical solution of Eq. 3.26 that uses simple SOR, only terms $\mathbf{r}_{\xi\xi}$ and $\mathbf{r}_{\eta\eta}$ contain values $u_{i,j}$ and $v_{i,j}$ for which the surface equations have to be solved for. One has to note that the metric is also a function of variables $u, v$. In the case of planes the equations reduce to the well known 2-dimensional grid generation equations.

For the numerical calculation of the surface Christoffel symbols the following formulas are used. Given a function $f(u, v)$, derivatives $f_u$ , $f_v$, $f_{uv}$, $f_{uu}$ and $f_{vv}$ are needed. For surfaces in 3D space one has

$$\begin{aligned} e_1 &= (x_u, y_u, z_v) \\ e_2 &= (x_v, y_v, z_v) \end{aligned} \tag{3.27}$$

The numerical procedure for the calculation of the metric is as follows. First, all derivatives $x_u, x_v, x_{uu}, x_{uv}, x_{vv}$ etc. have to be calculated. After that, the 5 quantities $g^{ij}$ and $g^{ij}\Gamma^1_{ij}$ have to be stored at each grid point. These quantities and not the coordinate values $(x, y, z)$ should be interpolated in the generation of the surface grid.

Using the relations of Sec. 2.0

$$\begin{array}{l} g_{11} = x_u^2 + y_u^2 + z_u^2 \\ g_{12} = x_u x_v + y_u y_v + z_u z_v \\ g_{22} = x_v^2 + y_v^2 + z_v^2 \end{array} \tag{3.28}$$

the determinant of the metric tensor is

$$g = g_{11} g_{22} - g_{12}^2 \tag{3.29}$$

and the contravariant metric coefficients are :

$$\begin{array}{l} g^{11} = g_{22}/g \\ g^{12} = -g_{12}/g \\ g^{22} = g_{11}/g \end{array} \tag{3.30}$$

From this the Christoffel symbols can be determined.

$$\begin{array}{l} \Gamma^1_{ik} = e^1 \cdot e_{i,k} \\ \Gamma^2_{ik} = e^2 \cdot e_{i,k} \end{array} \tag{3.31}$$

$$\begin{array}{l} e^1 = g^{1k} e_k = g^{11} e_1 + g^{12} e_2 = \left(+g_{22} e_1 - g_{12} e_2\right)/g \\ e^2 = g^{2k} e_k = g^{21} e_1 + g^{22} e_2 = \left(-g_{12} e_1 + g_{11} e_2\right)/g \end{array} \tag{3.32}$$

$$\begin{array}{l} \Gamma^1_{11} = e^1 \cdot e_{1,1} = e^1_x x_{uu} + e^1_y y_{uu} + e^1_z z_{uu} \\ \Gamma^1_{12} = e^1 \cdot e_{1,2} = e^1_x x_{uv} + e^1_y y_{uv} + e^1_z z_{uv} \\ \Gamma^1_{22} = e^1 \cdot e_{2,2} = e^1_x x_{vv} + e^1_y y_{vv} + e^1_z z_{vv} \end{array} \tag{3.33}$$

Resulting in

$$\begin{array}{rl} g^{ij}\Gamma^1_{ij} = & e^1_x \left(g^{11} x_{uu} + 2g^{12} x_{uv} + g^{22} x_{vv}\right) + \\ & e^1_y \left(g^{11} y_{uu} + 2g^{12} y_{uv} + g^{22} y_{vv}\right) + \\ & e^1_z \left(g^{11} z_{uu} + 2g^{12} z_{uv} + g^{22} z_{vv}\right) \end{array} \tag{3.34}$$

An analog result follows for $g^{ij}\Gamma^2_{ij}$. All terms have a common factor $1/g$.

### 3.1.3 *Elliptic Equations for 3D Grid Generation*

The following general coordinate transformation from the cartesian coordinate system, denoted by coordinates $(x,y,z)$, to the CD, denoted by coordinates (ξ, η, ζ), is considered. The one to one transformation is given by (except for a finite (small) number of singularities):

$$\begin{array}{ll} x = x(\xi,\eta,\zeta); & \xi = \xi(x,y,z) \\ y = y(\xi,\eta,\zeta); & \eta = \eta(x,y,z) \\ z = z(\xi,\eta,\zeta); & \zeta = \zeta(x,y,z) \end{array} \tag{3.35}$$

Since there is a one-to-one correspondence between grid points of the SD and CD (except for singular points or singular lines), indices i, j, and k can be used to indicate the grid point position. The grid in the CP is uniform with grid spacings $\Delta\xi = \Delta\eta = \Delta\zeta = 1$. Similar as in 2D, a set of Poisson equations is used to determine the positions of $(\xi, \eta_j, \zeta_k)$ in the SD as functions of $x,y,z$. In addition, proper Boundary Conditions (BC) have to be specified. Normally, Dirichlet BCs are used, prescribing the points on the surface, but for adaptation purposes von Neumann BCs are sometimes used, allowing the points to move on the surface, in order to produce an orthogonal grid in the first layer of grid points off the surface. The Poisson equations for the grid generation read:

$$\begin{array}{l} \xi_{xx} + \xi_{yy} + \xi_{zz} = P \\ \eta_{xx} + \eta_{yy} + \eta_{zz} = Q \\ \zeta_{xx} + \zeta_{yy} + \zeta_{zz} = R \end{array} \tag{3.36}$$

where $P,Q,R$ are so-called control functions that depend on ξ, η and ζ.

However, this set of equations is not solved on the complex SD, instead it is transformed to the CD. The elliptic type of Eqs.(3.36) is not altered, but since ξ, η and ζ are coordinates themselves, the equations become nonlinear. In more compact notation the elliptic generation equations read.

$$\Delta\xi = P; \;\; \Delta\eta = Q; \;\; \Delta\zeta = R \tag{3.37}$$

where planes of constant ξ, η or ζ form the boundaries and x, y, z are now the dependent variables. Equations 3.37 are solved in the computational plane using Eq.2.18. The x, y, z coordinate values describing the surfaces which form the boundary of the physical solution domain, are now used as boundary conditions to solve the transformed equations below. To perform the transformation the Δ-operator in general curvilinear coordinates is used:

$$\Delta\Psi = g^{ik}\left(\Psi_{,i,k} - \Gamma^{j}_{ik}\Psi_{,j}\right) \tag{3.38}$$

where Ψ is a scalar function. The determinant of the metric tensor, $g$, is given by the transformation from (x,y,z) to $(\xi,\eta,\zeta)$. The co -and contravariant base vectors are given by $e_i = \partial_i(x,y,z)$ ; $e^i = \partial^i(\xi,\eta,\zeta)$ with $\partial_1 = \partial_\xi, \partial^1 = \partial_x$ etc.

$$g^{ij} = e^i \cdot e^j \; ; \; \Gamma^i_{jk} = e^i \cdot e_{j,k} \quad (\cdot \text{ denotes scalarproduct}) \tag{3.39}$$

If $\xi$, $\eta$ or $\zeta$ is inserted for $\Psi$, the second derivatives are always zero and $\Psi, i = 1$ only for j=1 if variable $\xi$ is considered etc. Eqs. 3.37 then become

$$-g^{ik}\Gamma^1_{ik} = P \; ; \quad -g^{ik}\Gamma^2_{ik} = Q \; ; \quad -g^{ik}\Gamma^3_{ik} = R \tag{3.40}$$

or in vector notation

$$-E \cdot \begin{pmatrix} g^{11}x_{\xi\xi} + 2g^{12}x_{\xi\eta} + \dots \\ g^{11}y_{\xi\xi} + 2g^{12}y_{\xi\eta} + \dots \\ g^{11}z_{\xi\xi} + 2g^{12}z_{\xi\eta} + \dots \end{pmatrix} = \begin{pmatrix} P \\ Q \\ R \end{pmatrix} \tag{3.41}$$

where $E = \begin{pmatrix} e^1 \\ e^2 \\ e^3 \end{pmatrix}$ is the matrix consisting of the contravariant row vectors. The inverse matrix is $F = (e_1, e_2, e_3)$ consisting of the covariant column vectors. Thus $E^{-1} = F$ and finally the 3d grid generations are in the form

$$\begin{pmatrix} g^{11}x_{\xi\xi} + \dots \\ g^{22}y_{\xi\xi} + \dots \\ g^{33}z_{\xi\xi} + \dots \end{pmatrix} = -F \cdot \begin{pmatrix} P \\ Q \\ R \end{pmatrix} \tag{3.42}$$

## 3.2 *Grid Generation Concepts*

### 3.2.1 *Computational Aspects of Multiblock Grids*

As has been discussed previously, Boundary Fitted Grids (BFG) have to have coordinate lines, i.e. they cannot be completely unstructured. In CFD in general, and in high speed flows in particular, many situations are encountered for which the flow in the vicinity of the body is aligned with the surface, i.e. there is a prevailing flow direction. This is especially true in the case of hypersonic flow because of the high kinetic energy. The use of a structured grid (SG), allows the alignment of the grid, resulting in locally 1D flow. Hence, numerical diffusion can be reduced, i.e. better accuracy is achieved. A BFG exactly matches curved boundaries, and for complex SDs will consist of a set of blocks. In the present approach, a SD may be covered by a set of hundreds or thousands of blocks. Second, SGs can be made orthogonal at boundaries and almost orthogonal within the SD, facilitating the implementation of BCs (Boundary Condition) and also increasing the numerical. Numerical accuracy will increase further when algebraic turbulence models are employed using an almost orthogonal mesh. In the solution of the N-S (Navier-Stokes) equations, the BL (Boundary Layer) must be resolved. This demands that the grid is closely wrapped around the body to describe the physics of the BL (some 32 layers are used in general for SGs). Here some type of SG is indispensable.

In addition, to describe the surface of the body a structured approach is better suited. The resolution of the BL leads to large anisotropies in the length scales in the directions along and off the body. Since the time-step size in an explicit scheme is governed by the smallest length scale or, in the case of chemical reacting flow, by the magnitude of the chemical production terms, extremely small time steps will be necessary. This behavior is not demanded by accuracy, but to retain the stability of the scheme. Thus, implicit schemes will be of advantage. In order to invert the implicit operator, a SG produces a regular matrix, and thus makes it easier to use a sophisticated impicit scheme. Moreover, the use of the so-called thin layer approach, that is retaining the viscous terms only in the direction off the body, reduces computer time by about 30 %. Since there are no coordinate lines in the UG, this simplification is not directly possible. A fairly complex procedure would be needed to artificially construct these lines. Moreover, the flow solver based on the UG approach is substantially slower than for SGs. This is due to the more complicated data structure needed for UGs. Factors of 3, and by some authors of up to 10, have been given in the literature.

An important point for the accuracy of the solution is the capability of grid point clustering and solution adaptation. In general, SGs provide sophisticated means both for clustering and adaptation using redistribution or local enrichment techniques. A comparison of these two approaches is given by Dannenhoffer [2] where local enrichment gives somewhat better results. However, it is much more costly to use. In many cases of practical interest, for example fine resolution of a bow shock or a canopy shock and in situations where shocks are reflected, the alignment of the grid can result in a more accurate solution than randomly filling the space with an enormously large number of smaller and smaller tetrahedrons or hexahedrons. The highest degree of freedom of course is obtained in UGs. It is therefore felt, however, that mesh redistribution and alignment is totally adequate for the major part of the flow situations encountered in external flows, especially in aerodynamics.

The large majority of physical phenomena encountered in external and internal flows exhibit certain well ordered structures, such as a bow shock, or system of reflected shocks or some type of shock-shock interaction, which can be perfectly matched by adaptive grid alignment, coupled to the flow solution.

Only if a very complex wave patterns evolve due to special physical phenomena, for example, generating dozens of shock waves, the UG has advantages.

In addition, the coupling of SGs with UGs is possible as has been shown by Shaw et al. []. Such a grid is called a hybrid grid. A mixture between the boundary fitted grid approach and the unstructured approach is the use of multiblock grids, which is followed in **G**rid⋆[]. On the block level the grid is completely unstructured but in each block the grid is structured and slope continuity is provided across block boundaries.

### 3.2.2 *Description of the Standard-Cube*

The following description is used for both the grid generator and the flow solver, that is the same command file is used. All computations are done on a standard–cube in the computational plane (CP) as shown in Fig. 3.2. The coordinate directions in the CP are denoted by $\xi$, $\eta$ and $\zeta$ and block dimensions are given by $I, J$, and $K$, respectively.

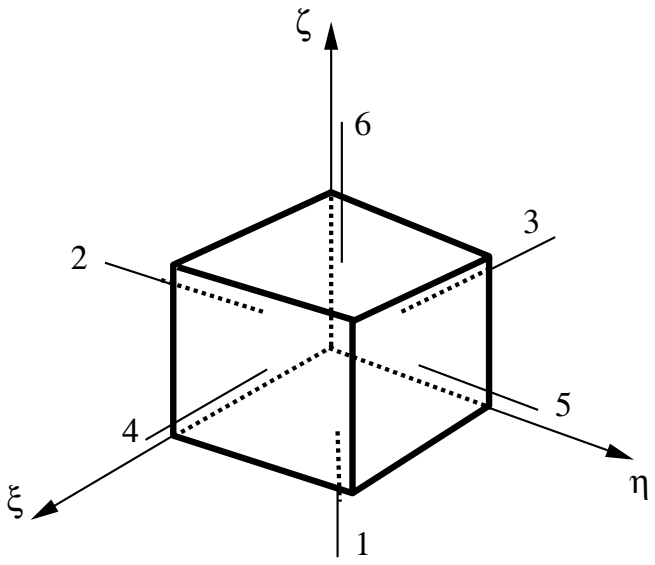


Figure 3.2: Standard cube in CP (computational plane). Each cube has its own local coordinate system. The grid is uniform in the CP.

In the CP, each each cube has its own right-handed coordinate system $(\xi, \eta, \zeta)$, where the $\xi$ direction goes from left to right, the $\eta$ direction from front to back and the $\zeta$ direction from bottom to top. The origin of this coordinate system in the lower left front–corner (Fig. 3.2). The coordinate values itself are given by the proper grid point indices $i$, $j$, $k$ in the $\xi$, $\eta$ and $\zeta$ directions, respectively. That means that the values reach from 1 to I in the $\xi$ direction, from 1 to J in the $\eta$

direction, and from 1 to K in the $\zeta$ direction. Each grid point represents an integer coordinate value in the computational plane.

A standard box has six faces, which will be numbered in the same way as a dye. Face 1 is the $\xi - \eta$ plane with a $\zeta$–value of 0, face 6 is the $\xi - \eta$ plane with a $\zeta$–value of $K - 1$. In Fig.3.2 this corresponds to the bottom and top faces. Face 2 is the front plane and face 5 the back plane, which means face 2 is defined as the $\xi - \zeta$ plane with a $\eta$ value of 0 and face 5 is the $\xi - \zeta$ plane with a $\eta$ value $J - 1$. Faces 3 and 4 then denote the left and right right boundaries of the standard box in Fig.3.2. Both are $\eta - \zeta$ planes, having $\xi$–values 0 and $I - 1$, respectively.

A simple notation of planes within a block can be achieved by specifying the normal direction along with the proper coordinate value in that direction. For example, face 2 will be uniquely defined by describing it as a J ($\eta$) plane with a j value 1, i.e. by the pair (J, 1) where the first value is the direction of the normal vector and the second value is the plane index . Thus, face 4 is defined by the pair $(I, I-1)$. This notation is also required in the visualization module.

Grid points are stored in such away that the I direction is treated first, followed by the J and K directions, respectively. This implies that K planes are stored in sequence. In the following the matching of blocks is outlined. First, it is shown how the orientation of a face of a block is determined. Second, rules are given how to describe the matching of faces between neighboring blocks. This means the determination of the proper orientation values between the two neighboring faces. The input description for the **G**rid$\star$ control (topology) file as used in **G**rid$\star$ is shown.

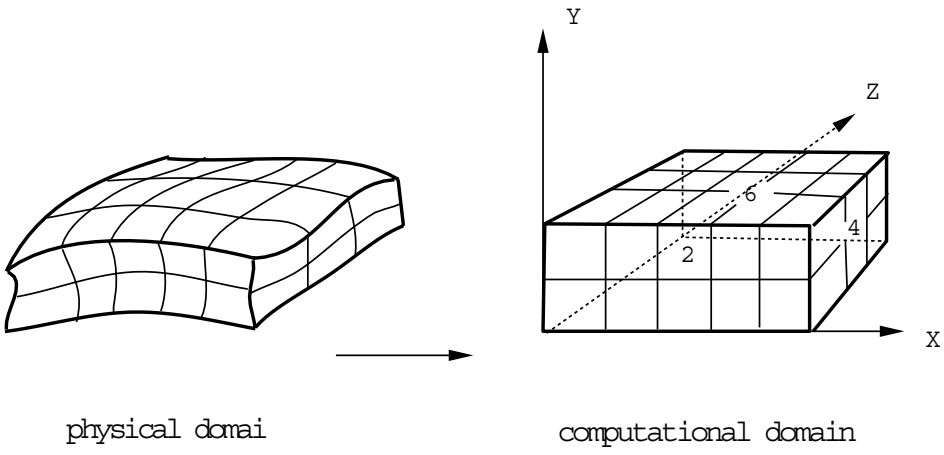


Figure 3.3: Mapping of a block from SD to CP. Arrows indicate orientation of faces, which are numbered in the following way: 1 bottom, 2 front, 3 left, 4 right, 5 back, 6 top. The rule is that plane $\zeta = 1$ corresponds to 1, plane $\eta = 1$ to 2 and plane $\xi = 1$ to 3.

To determine the orientation of a face, arrows are drawn in the direction of increasing coordinate values. The rule is that the lower coordinate varies first and thereby the orientation is uniquely determined. The orientation of faces between neighboring blocks is determined as follows, see Fig. 3.5. Suppose blocks 1 and 2 are oriented as shown. Each block has its own coordinate system (right handed). For example, orientation of block 2 is obtained by rotation of $3/2$ about the $\eta$–axis – rotations are positive in a counter clockwise sense – and a subsequent rotation of $3/2\pi$ about the new $\zeta$–axis. Thus faces 4 and 6 are matching with the orientations as shown, determined from the rules

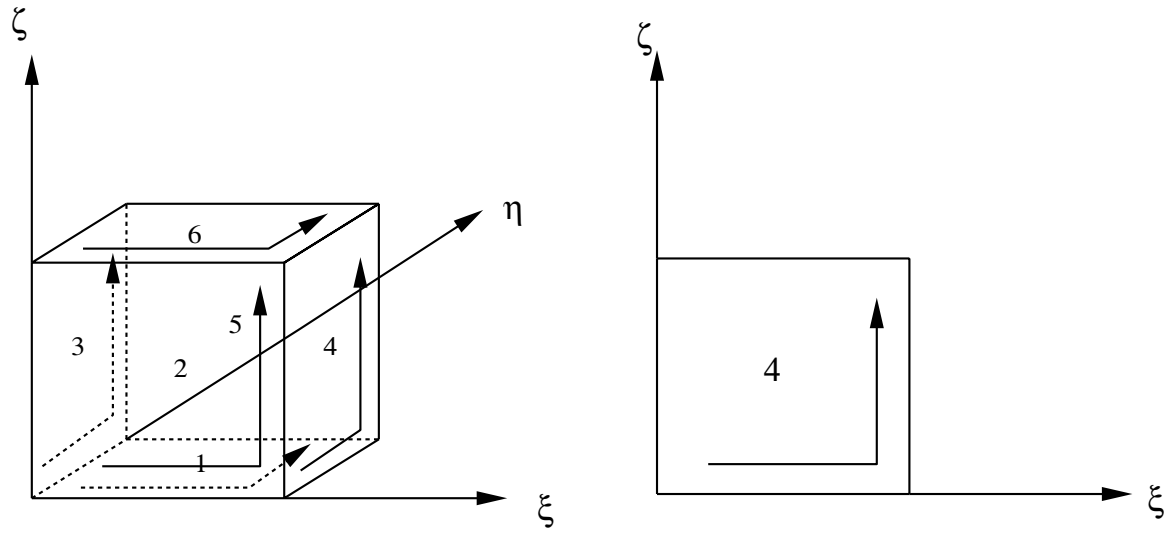


Figure 3.4: Orientation of faces. Coordinates ξ, η, ζ are numbered 1, 2, 3 where coordinates with lower numbers are stored first.

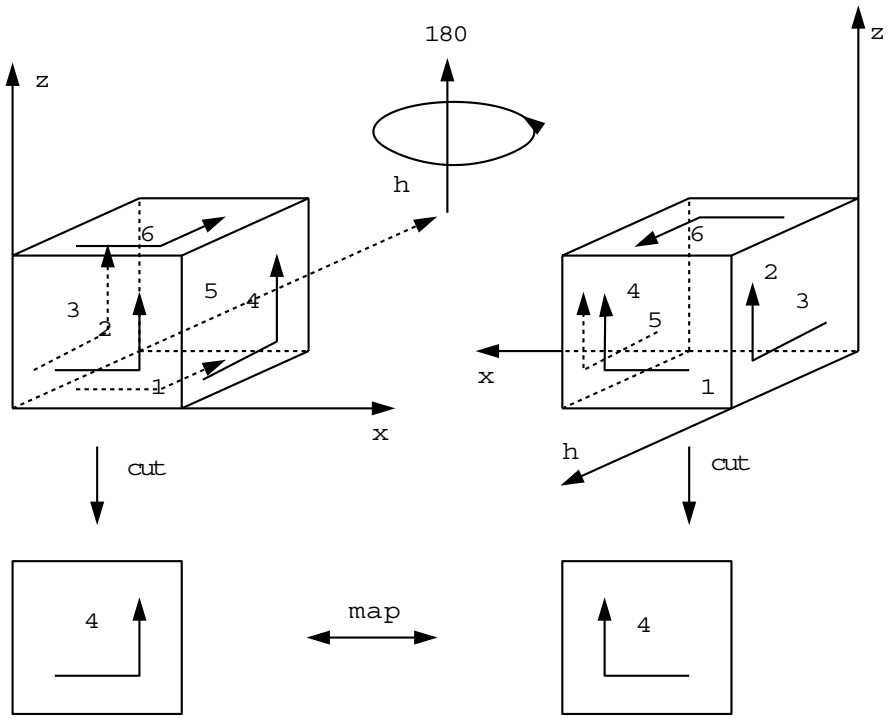


Figure 3.5: Determination of orientation of faces between neighboring blocks.

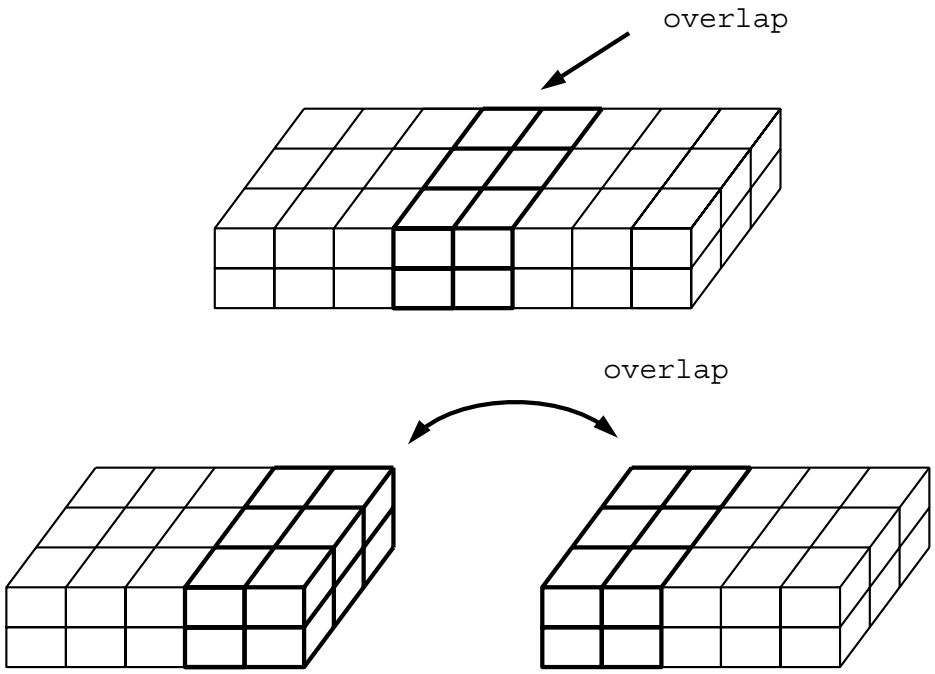


Figure 3.6: The figure shows the overlap of two neighboring blocks.

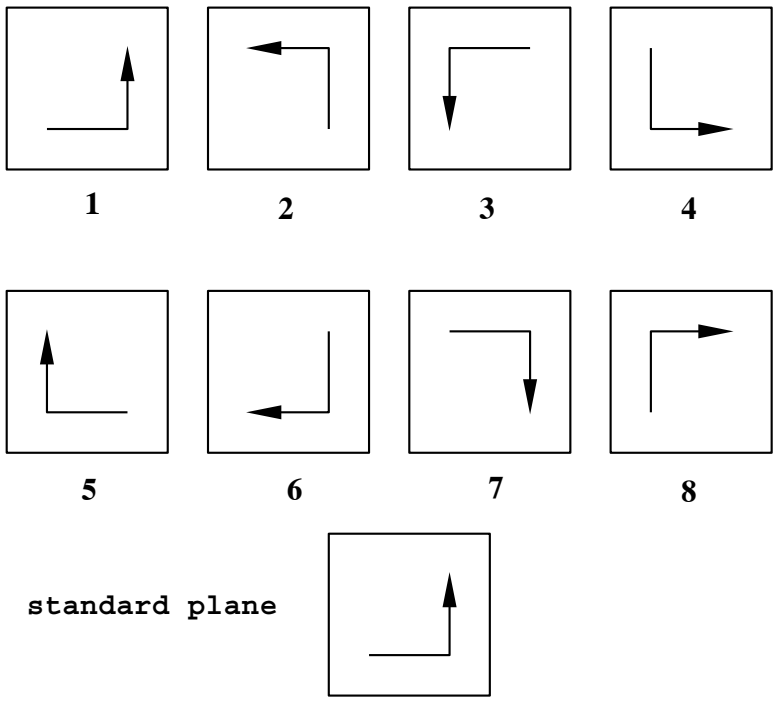


Figure 3.7: The 8 possible orientations of neighboring faces are shown. Cases 1 to 4 are obtained by successive rotations e.g. 0, $\frac{1}{2}\pi$, $\pi$ and $\frac{3}{2}\pi$. The same situation holds for cases 5 to 8.

shown in the previous figure. All cases in group 1 can be obtained by rotating about an angle of 0, $1/2\pi$ or $3/2\pi$. The same is valid for elements in group 2. The code automatically recognizes if the orientation between two faces needs a mirroring. Thus cases 1 and case 7 in Fig. 3.7 are obtained by rotating case 1 by $\pi/2$ and then do the mirroring. Thus, we have determined the input needed for Fig. 3.5, namely how to match face 4 of block 1 to face 6 of block 2. Rotations are denoted by integer 0, 1, 2, and 3 marking the corresponding orientations. However, if control faces are specified, that is, the point distribution on the face of a certain block, that will be used to calculate the control functions for the current face of the block under consideration, values 1 to 8 have to be provided. This is clear, because the code cannot anticipate in which orientation the distribution is going to be used. The control information for a 3D case therefore contains one of the 8 orientation numbers.

### 3.2.3 *The Grid⋆ Grid Generation Toolbox*

In the following the two codes developed by the authors **G**rid⋆ and **GridPro**will be presented. In addition, all grids presented have been generated by these tools. Other codes can be found in the references.

**G**rid⋆ is a general grid generation package for multiblock 2D, surface, and 3D solution domains that may be of arbitrary shape. **G**rid⋆ is a collection of ANSI Croutines and is modeled following the Unix toolbox concept. Modules for surface grid generation, 3D grid generation as well as for adaptation are provided, including grid enrichment. An algebraic-elliptic generation technique is used with convergence acceleration. Its name is derived from using the star as a wildcard, which stands for the collection of the modules that make up **G**rid⋆. Grid visualization is based on X-Windows and Motif. All major workstations are supported, including Unix PCs. Therefore, the package can be implemented on nearly any type of computer, ranging from a PC to a Cray. Grid generation is very fast and efficient, which is shown by the fact that several large have been generated and visualized on a PC. The package comprises interactive tools for surface grid generation and for grid enrichment, i.e. there are very efficient means for grid point clustering, e.g., to obtain a Navier-Stokes grid from an Euler grid. For the Hermes grid generation an Euler grid of 150,000 points was converted into a Navier-Stokes grid of 300,000 points in a few minutes on a PC. **G**rid⋆ can be used in every area where structured grids are needed, and provides special features, such as grid generation on surfaces.

**G**rid⋆ is a collection of C routines, fairly small in size, e.g. the 3D multiblock grid generation module, **G**rid⋆, consists of about 1500 lines in C. Modules are build from reusable subroutines and can be used in arbitrary order, each designed for a special task.

The grid generation modules **G**rid⋆, **G**rid⋆, and **G**rid⋆ use a combination of algebraic and elliptic grid generation methods, where elliptic techniques are mainly used for smoothing purposes. The grid visualization module **Xivis** is based on X11, allowing this system to be run on virtually any type of computer, from a PC under Linux, Solaris, or Unix station to an SGI Power Challenge etc. To speed up convergence, 3D blocks can be interactively cut out of the SD. Grid point distribution on the block faces is frozen, while points in the interior are iterated. This technique was successfully used when **G**rid⋆ generated the 150,000 points Euler grid for the Hermes space plane: 20 iterations with SOR were used for the complete spacecraft, while a block of about 8,000

points was cut out around the winglets and iterated 100 times to improve grid quality and then was inserted back. In **G**rid⋆ dynamic storage allocation is used, so the user never has to specify any array dimensions. Special data types are used, which allow a much more compact programming. All graphics functions are running under X-Windows so the code is fully portable.

Grid input consists of a control file, describing the connectivity and orientation with respect to neighboring blocks. Control functions for grid adaptation can also be specifid. Each block has its own local coordinate system. All grids are slope continuous across block boundaries. Hence block boundaries are not visible. Grid description is described by a set of keywords used in the control files and the coordinates files. The widely used Plot3d (NASA Ames) grid file format is supported.

### 3.2.4 *Input for Grid Generation in 2D and 3D*

The **G**rid⋆ package expects geometrical objects as input. In order to identify the object type, the actual data must be proceeded by an identifier. The following object types can be specified in **G**rid⋆ input- and output files:
**line2d**, **line3d**, **plane2d**, **plane3d**, **vol3d**, **cntrl2d**, **cntrl3d**.
File formats can be either in ASCII or binary, which is automatically detected by the input routines. One additional command is **file** filename, which redirects the input to file filename until an end-of-file is encountered. After that, reading is resumed by the calling file, which in general is the control (command) file. The **file** command can be nested. Furthermore, there are three internal object types, that means, object types, which are used only inside a module, namely **error**, **end**, **digit**.

The meaning and usage of these objects is as follows. Lines (curves) are needed as boundaries of planes and surfaces, which are embedded in 2- or 3-dimensional space. That is, each point is given by a pair (x,y) or by a triple (x,y,z). The same is true for surfaces. In case of a plane, object **plane2d**, two coordinate values are needed while for a surface in 3D, object **plane3d**), three coordinate values are required. A volume is part of 3-dimensional space, therefore the object name **vol3d**. For the description of the matching of neighboring blocks in 2D or 3D control structures **cntrl2d** or **cntrl3d** are needed. This concerns the command files of modules **G**rid⋆, **G**rid⋆, and **G**rid⋆. The above mentioned **file** command is a feature which makes the input more readable, especially for the large input files needed in **G**rid⋆. Instead of placing an object of type **plane3d** or **vol3d** directly after the control information of a block, which in general comprises a large number of coordinate values to describe the geometry of the boundary surface or the volume grid, the user can reference to a file containing that object. It is recommended to write the control information in one file and to create as many separate objects of type **plane2d**, **plane3d** or **vol3d** as there are blocks. Then a file reference is made to the respective file containing the plane or volume data. Volume data are used in a restart case. If a multiblock grid with a larger number of blocks (see Sec. **??**) is to be generated, the use of a separate file for the boundary data of each block is the only possible way. For the parallel grid generator this feature has to be used, because it would be completely impractical if each node read the entire input.

#### 3.2.4.1 *Rectangle Grid Example*

In this section we perform the simple task of generating a monoblock grid for a rectangle. However, this simple example allows us to introduce most of the concepts needed for advanced grid generation. In Sec. 3.2.4.2 we generate a 6 block grid for a diamond shaped body. In order to generate a grid two types of informa tion have to be provided, namely the block connectivity information and the grid point distribution on the physical boundary of the solution domain (SD).

First the so called control file or command file (extenion .cmd) has to be specified. The command file describes the blocking of the solution domain. It also contains the references to those files that contain the surface grid. Since this is the first example, a 2D problem is considered only. Therefore, the surface grid simply consists of the grid point distribution on the four edges (sides) of the rectangle. The grid point distribution can be generated interactively, using the tools (see Chapter Grid Generation Tools) **Poli** and **Spline2D**. In this example it is assumed that the command (connectivity) information is stored in the file **rect.cmd** and the geometry data (edge description) is stored in **rect.lin**. The first line of the command file contains the keyword **cntrl2D** that simply denotes a 2D SD. The **1** in next line denotes the block number, followed by the dimensions of the block ($\xi, \eta$ directions)**11 10**. It should be noted that two different coordinate systems are used. In **Physical Space** (also denoted as the **Physical Plane**) coordinates are described by $(x, y, z)$. For complex geometries, curvilinear coordinates $(\xi, \eta, \zeta)$ are needed whose directions are conforming to the shape of the configuration. The elliptic grid generation equations as well as the flow equations are solved in the transformed or **Computational Space (Plane)** (see Sec ...). In this plane coordinate directions are denoted by **I**, **J**, and **K**. In **G**rid$\star$ the following rule applies (respectively in three dimensions):

**Rule: In specifying the dimensions of a block the first number denotes the number of grid points in the $\xi\,(I)$ direction (number of columns), the second one gives the number of points in the $\eta\,(J)$ direction (number of rows).**

A second input rule determines the orientation of the edges with respect to the axes of the coordinate system. If a grid comprises several blocks, each block can have its own local coordinate system and these systems can be rotated with respect to each other. In 2D the rotation is determined automatically by the code. The four edges of a block are numbered 1, 2, 3, and 4. However, to enable the code to perform the automatic detection of rotated blocks, the following rule must be obeyed (for three dimensions see later chapter):

**Rule: With respect to the coordinate axes of the block the numbering of the edges has to be such that: edge 1 is east, edge 2 is north, edge 3 is west, and edge 4 points south.**

In what follows, a description of the simple format of the command file is given, sufficient for the rectangle example. The complete command file format is described in section **??**. The next four lines describe the edges of the rectangle.

The first integer gives the number of the edge, followed by the type of the side. A type 0 or 1 describes a fixed boundary, a type 2 or 3 indicates a matching boundary, i.e. an interblock boundary. In this example, there are only fixed edges, hence sides are either of type 0 or 1. A side of type 1 is used for interpolating an initial grid. For each point on this edge the corresponding

point on the opposite edge is known. If this edge is of type matching, the corresponding opposite side in the neighboring block is investigated. If this edge is fixed, a linear interpolation between the two corresponding fixed edges is performed. If the edge is matching, the algorithm continues through all blocks until a fixed side is encountered. Since the interpolation is linear, the quality of the initial grid depends on the choice of the edge where the interpolation process starts from. To obtain a good initial grid, several edges can be used. Moreover, an interpolated grid may be partly overwritten in this process, allowing complete freedom. The interpolated grid speeds up the grid generation process by a factor of 5 - 10, making it possible to generate grids with hundreds of thousands of points on a workstation. The next two integers are needed to describe the block connectivity, specifying the block and edge number of the neighboring block. If an edge is not neighbored, these numbers are 0. Integers 5 and 6 describe the so called control functions (see next Chapter) used for grid point clustering. The last line, **file rect.lin** is a reference to the geometry file that contains the coordinate values of the 4 edges of the rectangle in exactly the same order as the edges occur in the command file. For example, it would have been possible to describe edge 2 first and the edge 1 in the command file. In that case, however, the sequence of the edge coordinates in file **file rect.lin** must be consistent with the order of the edges in **file rect.cmd**.

```
    \ cntrl2d
1
11 10
1 1 0 0 0 0
2 0 0 0 0 0
3 0 0 0 0 0
4 0 0 0 0 0
file rect.lin
```

In file **file rect.lin** the first line indicates that data are of type **line 2d**, that is are part of a 2D curve. The next value gives the number of coordinate pairs (x, y) that form this line. It should be noted that all physiacl coordinates are given with respect to a Cartesian coordinate system that holds for all blocks.

#### 3.2.4.2 *Diamond Shape 6 Block Grid Example*

To illustrate the multiblock concept, a simple 6 block grid for a diamond shape is constructed, see Fig. 3.8. The control file information for the grid of Fig. 3.8 is shown in Fig. 3.10. The meaning of the control information is explained below. Since the example is 2D, the first line of this file starts with **cntrl2d**. In Fig. 3.11 the corresponding coordinate values are specified for all fixed (physical) boundaries. The proper type of these boundaries is therefore **line2d**.

In the following a complete description of the topology of multiblock grids is given. Having read this information, the reader should refer back to the example in Fig. 3.10.

Each data file begins with a line indicating the object type. In 3D, this line has the form **object type**
**for example**

```
\line2d                 \line2d
10                      10
10 0                    0 0
10 1                    0 1
10 2                    0 2
10 3                    0 3
10 4                    0 4
10 5                    0 5
10 6                    0 6
10 7                    0 7
10 9                    0 9
10 10                   0 10

\line2d                 \line2d
11                      11
0 10                    0 0
1 10                    1 0
2 10                    2 0
3 10                    3 0
4 10                    4 0
5 10                    5 0
6 10                    6 0
7 10                    7 0
8 10                    8 0
9 10                    9 0
10 10                   10 0
```

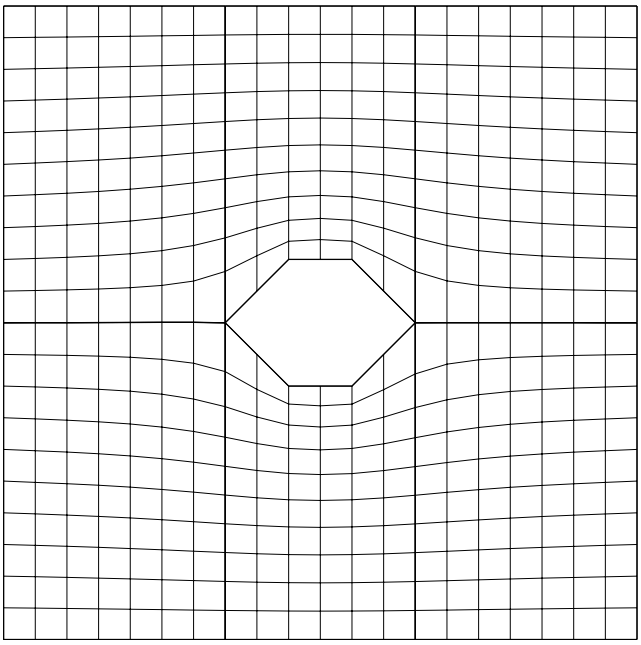

Figure 3.8: A 6 block grid for diamond shaped body. This type of grid line configuration cannot be obtained by a mono-block grid.

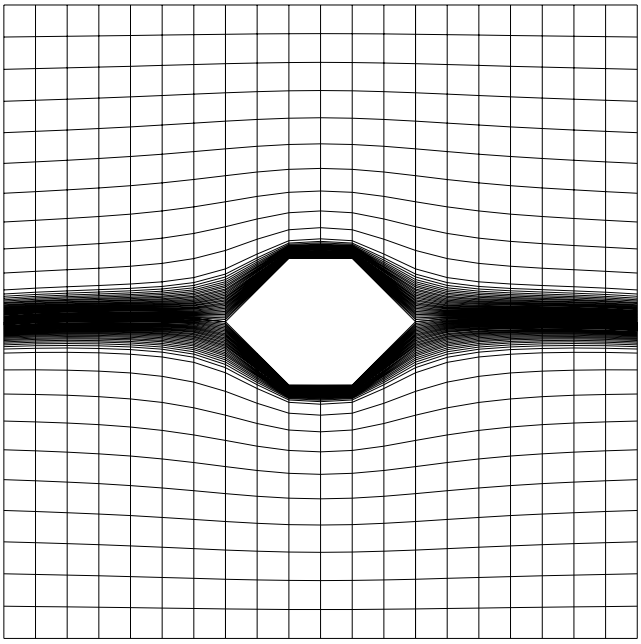

Figure 3.9: Grid line can also be clustered to match the physics of the flow; e.g. resolving a boundary layer.

```
\cntrl2d
1
8 11
1   2   3   3   0   0
2   3   2   4   0   0
3   1   0   0   0   0
4   1   0   0   0   0
2
8 11
1   3   4   3   0   0
2   0   0   0   0   0
3   1   0   0   0   0
4   2   1   2   0   0
3
7 11
1   2   5   3   0   0
2   0   0   0   0   0
3   3   1   1   0   0
4   1   0   0   0   0
4
7 11
1   2   6   3   0   0
2   0   0   0   0   0
3   2   2   1   0   0
4   1   0   0   0   0
5
8 11
1   0   0   0   0   0
2   2   6   4   0   0
3   3   3   1   0   0
4   1   0   0   0   0
6
8 11
1   0   0   0   0   0
2   0   0   0   0   0
3   3   4   1   0   0
4   3   5   2   0   0
\file diamond.lin
```

Figure 3.10: Control information for the 6 block diamond grid. This command file is also used by the parallel flow solver. *\file diamond* contains the actual coordinate values.

```
\line2d          \line2d          \line2d
11               7                11
-1 -1.0          -0.3  0            1 -1
-1 -0.9          -0.2 -0.1          1 -0.9
-1 -0.8          -0.1 -0.2          1 -0.8
-1 -0.7           0.0 -0.2          1 -0.7
-1 -0.6           0.1 -0.2          1 -0.6
-1 -0.5           0.2 -0.1          1 -0.5
-1 -0.4           0.3  0            1 -0.4
-1 -0.3           \line2d           1 -0.3
-1 -0.2           7                 1 -0.2
-1 -0.1           -0.3 -1            1 -0.1
-1  0.0           -0.2 -1            1  0
\line2d            -0.1 -1           \line2d
8                0   -1             8
-1.0 -1            0.1 -1            0.3 -1
-0.9 -1            0.2 -1            0.4 -1
-0.8 -1            0.3 -1            0.5 -1
-0.7 -1            \line2d           0.6 -1
-0.6 -1            7                 0.7 -1
-0.5 -1            -0.3 1            0.8 -1
-0.4 -1            -0.2 1            0.9 -1
-0.3 -1            -0.1 1            1   -1
\line2d            0   1             \line2d
8                0.1 1              11
-1.0 1             0.2 1             1 0
-0.9 1             0.3 1             1 0.1
-0.8 1             \line2d           1 0.2
-0.7 1             7                 1 0.3
-0.6 1             -0.3  0           1 0.4
-0.5 1             -0.2 0.1          1 0.5
-0.4 1             -0.1 0.2          1 0.6
-0.3 1              0.0 0.2          1 0.7
\line2d            0.1 0.2           1 0.8
11                 0.2 0.1           1 0.9
-1 0               0.3 0             1 1
-1 0.1                               \line2d
-1 0.2                               8
-1 0.3                               0.3 1
-1 0.4                               0.4 1
-1 0.5                               0.5 1
-1 0.6                               0.6 1
-1 0.7                               0.7 1
-1 0.8                               0.8 1
-1 0.9                               0.9 1
-1 1.0                               1   1
```

Figure 3.11: Coordinate values of the fixed (physical) boundaries of the 6 block diamond grid.

**\cntrl2d**

After this control line, object specific information is expected. One can write any number of objects (in this example **\cntrl3d**) after this control line as needed. The object specification is valid until the next control line is encountered or if the end of the current input file is read. All control lines, which can not be identified are converted to the internal object type error. In control files, command information and geometrical information of objects may be mixed, .e.g., one can start with the control information of type **\cntrl3d** to specify the block. After that, a line may be given

**plane3d**

to tell **G**rid⋆ that surface information for this block follows or a line

**file filename**

could be given to indicate that surface data for the next block are stored in file with name filename.

To be more specific, the input format for the different object types will now be explained in detail.

In short notation an object of type **\line2d**has the following form:

**line2d**
I
x(1) y(1)
::
x(I) y(I)

Object **\line2d** indicates the boundary line of a 2D SD. The value of I specifies the number of data points, followed by I pairs of x and y values, denoting the boundary points.

Objects plane2d and plane3d are of the following form.

**plane2d**
I J
x(1,1) y(1,1)
::
x(I,1) y(I,1)
x(1,2) y(1,2)
::
x(I,J) y(I,J)

**plane3d**
I J
x(1,1) y(1,1) z(1,1)
::
::
x(I,J) y(I,J) z(I,J)

In **vol3d** the I-index is running fastest, followed by the J and K indices, so the K planes (I x J coordinates each) are stored consecutively.

```
 vol3d
I J K
x(1,1,1) y(1,1,1) z(1,1,1)
::
x(I,1,1) y(I,1,1) z(I,1,1)
x(1,2,1) y(1,2,1) z(1,2,1)
::
x(I,J,1) y(I,J,1) z(I,J,1)
x(1,1,2) y(1,1,2) z(1,1,2)
::
x(I,J,K) y(I,J,K) z(I,J,K)
```

The control information for 2d reads

```
 cntrl2d
nos
I J
s1 st nb ns cb cs
s2 st nb ns cb cs
s3 st nb ns cb cs
s4 st nb ns cb cs
```

where *nos* is the block number and I, J are the number of grid points in the respective directions. The next four lines describe the four edges of the block. *s*1 to *s*4 denote the side-number where 1 is east, 2 north, 3 west, and 4 south. *st* is the side-type. 0 means fixed side, 1 is a fixed side which is used for computing the initial algebraic grid, before the Poisson equations are used. A side of type 2 is a matching side (overlap). In this case, the corresponding values for *nb* and *ns* have to be given where *nb* is the number of the neighboring block and *ns* the number of the matching side of this block. If *st* is 0 or 1, these values should be set to zero. *cb* and *ct* denote the control block and control side from which the distribution of boundary points is to be taken to calculate the controlfunctions for the right hand side of the Poisson equations. Hence, these values should reference a fixed side, which must have the same number of grid points as the current side. It is not necessary to set *s*1 to 1 etc. The edge control information can be in any order. The only restriction is that the same order is used when the boundary data are read. The 2D code allows for orthogonal grids at a fixed boundary (type 5). The distance of the first layer of grid points of a fixed boundary (type 6), and a combination of distance control and orthogonality (type 7) can also be chosen. The prescribed distance values needed for side type 6 and 7 are specified at the end of the input control file. A similar format is used for the control information in 3D.

```
 cntrl3d
nos
I J K
s1 st nb ns nr cb cs cr
s2 st nb ns nr cb cs cr
s3 st nb ns nr cb cs cr
s4 st nb ns nr cb cs cr
s5 st nb ns nr cb cs cr
s6 st nb ns nr cb cs cr
```

Again, *nos* denotes the block number, I, J and K are the dimensions in x,y and z-direction, re-

spectively. Each block has 6 sides, so for each side there is one line with side-specific information. *s*1 to *s*6 are the side numbers as described for the standard block. *st* again is the side type, where a 1 denotes the side used for initialization. In addition to the neighboring block *nb* and the neighboring side *ns*, rotation value nr is necessary. The same is true for the control side information *cb*, *cs*, and *cr*. If no clustering is desired, these values are 0. Variables must be in the following range. Only integer values can be assumed.

- *s*1..*s*6: [1,6] → side number
- *st*: [0,3] → side type
- *nb*: [1,N] (N is total number of blocks) → neighboring block
- *ns*: [1,6] → neighboring side of block *nb*
- *nr*: [0,3] → rotation needed to orient current side to neighboring side
- *cb*: [1,N] → block number for control information
- *cs*: [1,6] → side number for control information
- *cr*: [1,8] → orientation for control information

For a fixed side control information can be used from other sides (faces) or from other blocks and the orientation of these sides (faces) has to be specified (*cr*). In the 3D case a value between 1 and 8 must be given. In the 2D case, there are only 2 possibilities, namely the sides are either parallel (+1) or antiparallel (-1), which is detected by the code. If control functions are specified only on one side, and no control functions are given at the opposite fixed side, the control values at the opposite boundary are assumed to be zero. The control values within a block are determined by linear interpolation.

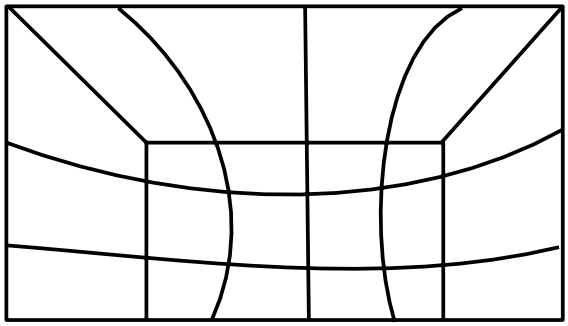

Figure 3.12: So called clamp technique to localize grid line distribution. The real power of this

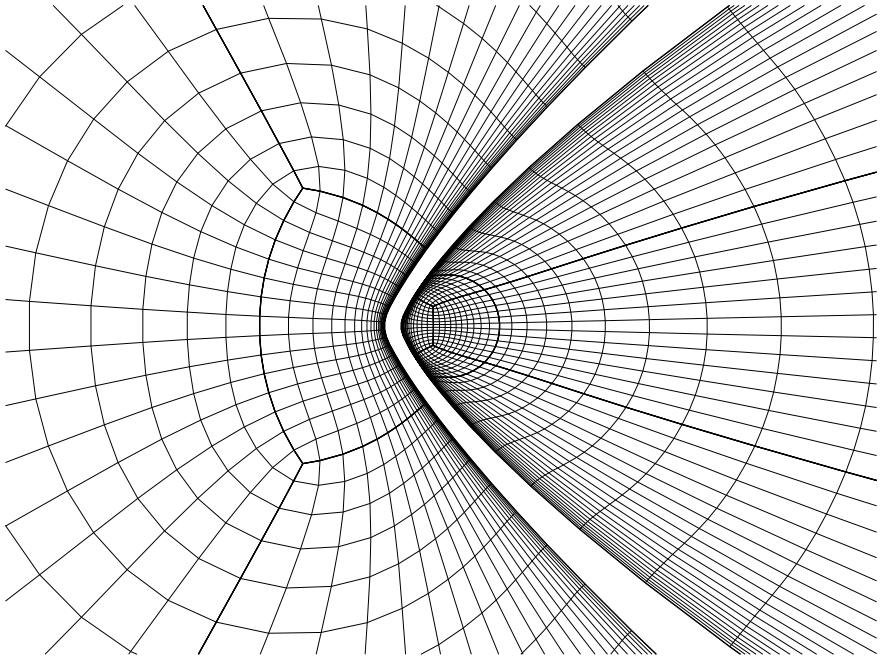

Figure 3.13: Clamp 1 at hyperboloid flare.

## 3.3 *Local Grid Clustering Using Clamp Technique*

Along fixed walls a large number of grid lines is required in order to simulate boundary layers. In the remaining solution domain this requirement only causes memory overhead and reduces the convergence speed. It is therefore mandatory to localize the grid line distribution. To this end the clamp clip technique is used, see Fig. 3.12. The principle of it is to build a closed block system connected to the physical boundary. The number of grid lines can be controlled within the block. If the grid is refined along the physical boundary, a refinement at the outer boundary is also obtained. Using clamp clips, the grid lines can be closed in clamp blocks. The local grid refinement can be achieved without influencing the far field grid, see Figures 3.13 and 3.14.

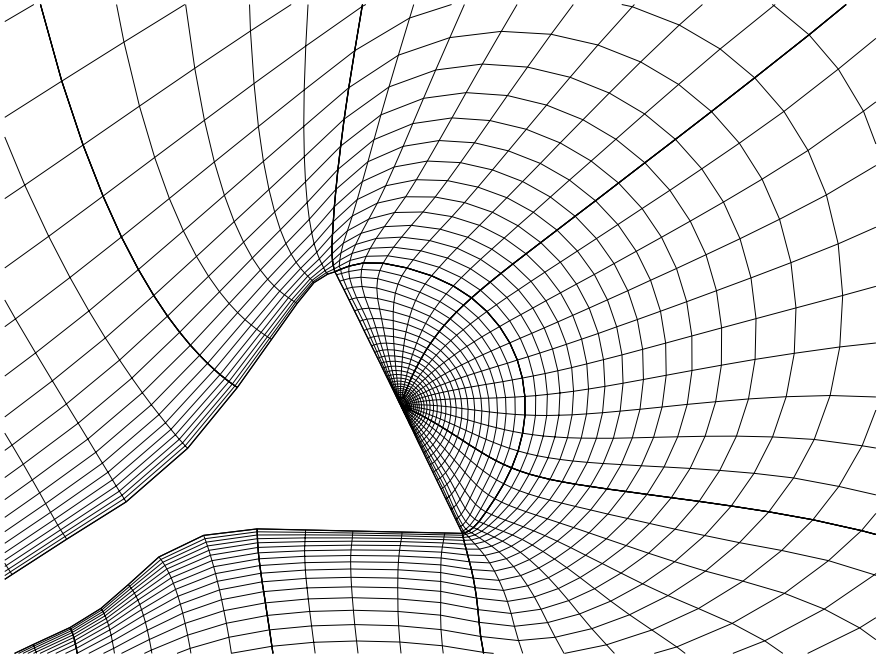

Figure 3.14: Clamp 2 at hyperboloid flare.

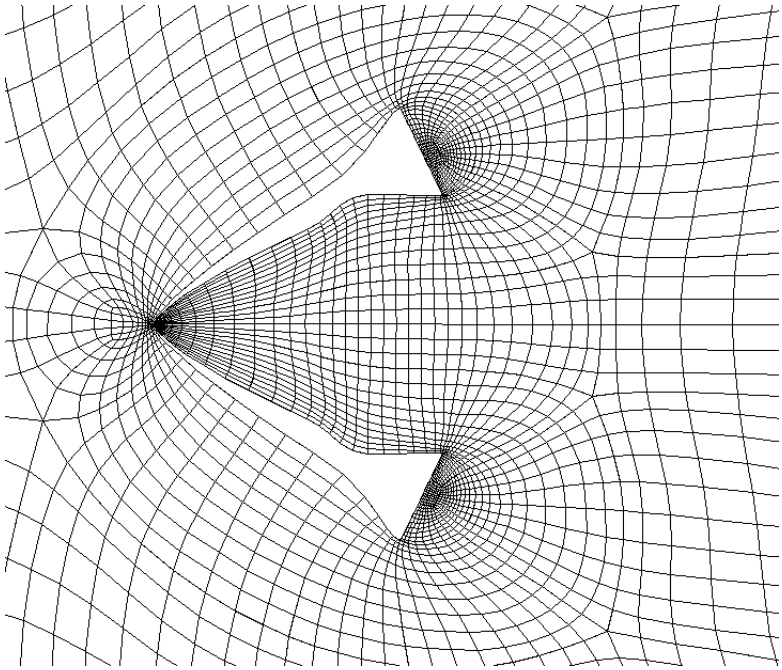

Figure 3.15: A conventional topology for a hyperboloid flare in a windtunnel. Although the topology allows a refinement of the grid from an Euler to a N–S grid, this refinement extends into the far field and thus causes a substantial computational overhead. Numbers denote block numbers, doted lines are block boundaries, solid lines are grid lines.

### 3.3.1 *Hyperboloid Flare in F4 Windtunnel Grid*

In Fig. 3.15 a conventional topology is shown to capture the boundary layers both for the windtunnel walls and the hyperboloid flare. The first disadvantage is that all gridlines are extending into the farfield, generating a huge amount of finite volumes in areas that don't need a fine resolution. Second, identification of single objects is not possible.

As a more complex example of the above strategy a "hyperboloid flare in the F4 windtunnel" grid is presented. One of the major constraints in the generation of this grid was to contain the high degree of grid line clustering around the hyperboloid flare close to the body. To this end the "clamp" technique described in Fig. 3.12 was used. Introducing clamp clips to locations near physical boundaries keeps the grid lines local. Fig. 3.16 shows the local topology of the hyperboloid flare.

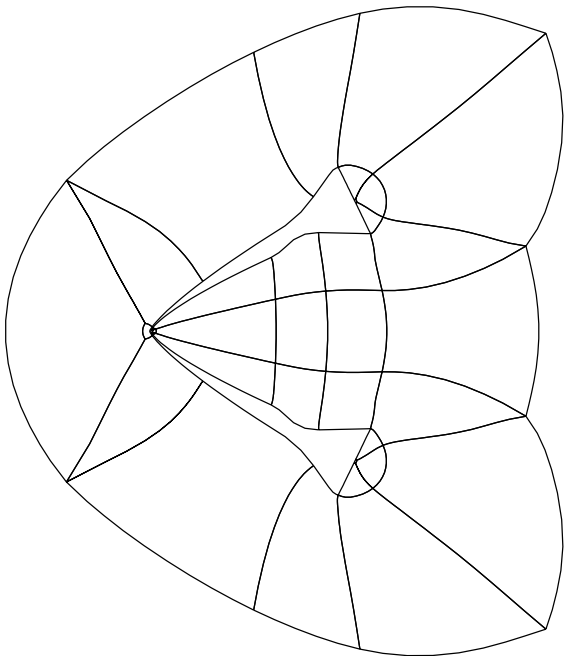

Figure 3.16: Topology of 36 block hyperboloid flare. This topology is one part of the topology of 284 block grid, see Fig. 3.18.

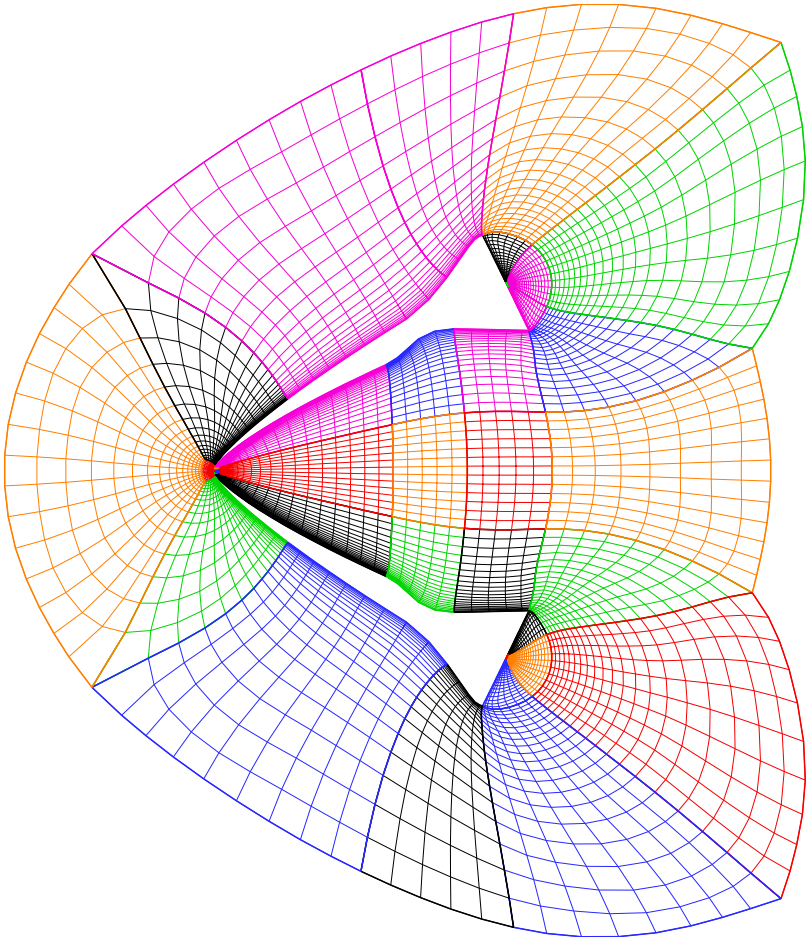

Figure 3.17: 36 block grid for hyperboloid flare.

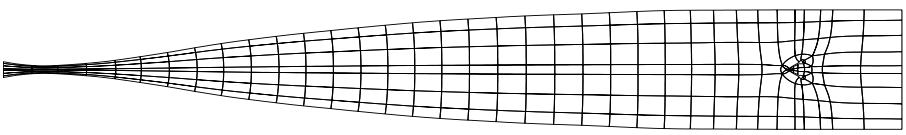

Figure 3.18: Topology of 284 block grid for hyperboloid flare in windtunnel.

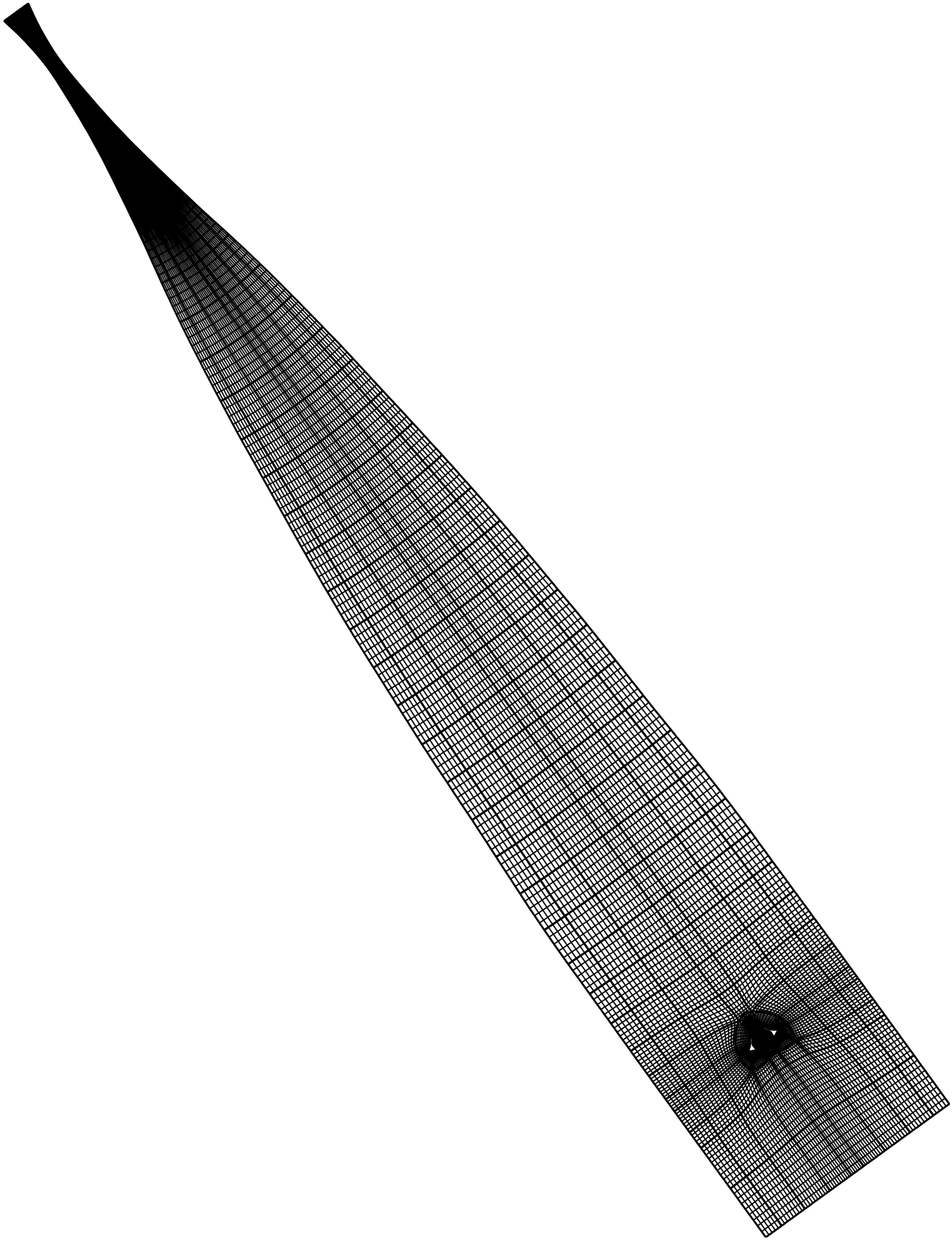

Figure 3.19: 284 block grid for hyperboloid flare in windtunnel.

## 4.0 *Grid Adaptation Techniques*

For the adaptation of a grid, knowledge about the flow solution is needed to obtain the desired grid resolution in all regions that exhibit large variations. Adaptive grid methods monitor the solution and adjust the grid dynamically, concentrating grid points in areas of larger solution variation. In many flows of interest, these regions have a regular structure so that SGs are well suited for adaptation. The following discussion is restricted to grid point movement.

Common to all adaptation techniques is the specification of a weight function, obtained from the solution features of the flow, denoted by $M_1, M_2, .., M_n$. The $M_i$ are nonnegative functions. Generally, the weight function $w$ is a linear combination of the $M_i$

$$w = 1 + c_1 M_1 + c_2 M_2 + \ldots + c_n M_n \tag{4.1}$$

For example the following expressions could be chosen as weight functions:
$w = 1 + \beta \mid \frac{\partial \Phi}{\partial \xi} \mid$, using the gradient or $w = 1 + \beta \mid \kappa \mid$ where $\kappa$ denotes the curvature of the $\Phi$. $\Phi$ is the so called monitor surface (explained below) and could be constructed, e.g. from the Ma number distribution.

The basic formula for the redistribution of grid points is the so called equidistribution statement for the weight function $w$ that, written as first order differential equation, takes the form

$$w\, s_\xi = c \tag{4.2}$$

or in differential form

$$w\, ds = c d\xi \tag{4.3}$$

where where $c = constant$ or $c = c(\xi)$, $ds$ is arc length, and $\xi$ denotes the coordinate of the computational domain. Assuming a constant c, the first order differential equation of Eq. 4.2 can be differentiated again, leading to the second order differential equation

$$(w\, s_\xi)_\xi = 0 \tag{4.4}$$

Eqs. 4.2 and 4.4 can be interpreted in several ways, giving rise to various adaptation techniques. The first adaptation technique presented is concerned with the usage of PDEs. When employing elliptic PDEs for grid generation, the weight function can be used to calculate the so called control functions $P(\xi, \eta)$ and $Q(\xi, \eta)$ (2D) that form the rhs of the elliptic PDEs, which are then called Poisson equations. The effect of $P$ and $Q$ can be described using an example from electrostatics. If there are no charges in the SD, the electrostatic potential is obtained by solving Laplace's equation. Introducing charges into the SD, that is the rhs of the Laplace equation is different from

zero (Poisson equation), concentrates the equipotential lines in the vicinity of the charges (negative) or repels the potential lines (positive charge). If the areas where to cluster the grid are known a priori, the boundary point distribution could be specified accordingly and $P$ and $Q$ could be computed from this distribution, in order to achieve the same clustering in the interior of the SD as on the physical boundaries. Without controlfunctions, the Laplace operator would equidistribute the grid points in the interior.

The second technique which is widely used is of algebraic nature, obtained by interpreting Eq. 4.2 in a discrete way. A monitor surface (MS) (monitor curve in 1D) is positioned above the physical SD, e.g., if the SD is a 2D region in the $(x,y)$ plane, the MS, denoted by $\Phi(x,y)$, is a curved surface in 3D space that is directly above the 2D region. The MS is a piecewise linear (smooth) surface. It is constructed by the user; for example, it can be the linear combination of one or more of the variables that determine the physics of the problem. To capture a shock, Ma number could be used, combined with the pressure, $p$, to cluster points in the boundary layer. The amount of clustering is given by the weight function, which may depend on the gradient or the curvature of the monitor surface. Examples of monitor surfaces will be given in the next section. Adaptation is performed by operating on a surface grid lying on the MS. The basic features of the adaptation algorithm are given below where it is assumed that a grid already exists in the physical SD, and a flow solution is available.

Regardless of the dimension, algebraic adaptation is always done on a curve by curve basis.

- ▷ the initial surface grid is obtained by projecting (lifting) the grid points of the physical domain up to the monitor surface.
- ▷ surface grid points are subsequently repositioned on the MS by equally distributing the weight function, $w$, based on the arc length of the coordinate curves.
- ▷ after that, the new surface grid on the MS is projected back down to the physical domain to yield an improved grid that more accurately represents the flow physics and thus reduces the numerical error.

The two techniques mentioned above are derived from the same principle, namely the equilibrium statement (see below) that can be interpreted as a differential equation, leading to a Poisson equation or an integral form, resulting in an algebraic approach. The two techniques are described in more detail in the subsequent sections.

### 4.0.1 *Adaptation by Controlfunctions*

Let us reconsider the grid generation equations

$$\Delta\xi = P; \quad \Delta\eta = Q \tag{4.5}$$

Interchanging the role of dependent and independent variables leads to

$$\begin{aligned} g_{22}x_{\xi\xi} - 2g_{12}x_{\xi\eta} + g_{11}x_{\eta\eta} + g(x_\xi P + x_\eta Q) = 0 \\ g_{22}y_{\xi\xi} - 2g_{12}y_{\xi\eta} + g_{11}y_{\eta\eta} + g(y_\xi P + y_\eta Q) = 0 \end{aligned} \tag{4.6}$$

These equations are quasilinear, where the non-linearity appears in the expression for the metric coefficients.

If we set the control functions P and Q to zero, we obtain the transformed Laplace equation. The control functions can be used to adapt the grid lines to better represent certain solution features or to match the distribution of the boundary points in the interior. The first process is a dynamic one. The latter is performed only once, before the computation is started. In principle, however, there is no difference between these two approaches. To visualize the distribution of the grid lines generated by Laplace's equation, one can assume that each grid point is coupled to its four nearest neighbors by springs having equal spring constants. The equilibrium of such a spring system gives the distribution of the coordinate points. If the distribution of boundary points is equidistant, one observes the following fact: A convex boundary repels grid lines, whereas a concave boundary attracts grid lines. Very often, however, the opposite behavior is desirable. For example, when a storm surge model is used for the calculation of the water level in a bay (convex area), high resolution near the shore is required or, in aerodynamics, the BL for a convex surface might have to be modeled.

One of the simplest ways to achieve the desired clustering could be to change the distribution of the boundary points representing the physics, i.e. to choose a dense distribution in the direction off the convex surface to capture the BL. Since the Laplacian produces an equidistribution in the interior, the effects of the boundary grid point distribution would be smoothed out within a few grid points and therefore would not be felt in the interior. For that reason, a Poisson equation is needed and control functions P and Q have to be determined.

Since a boundary is either a $\xi$- or an $\eta$-line (in 2D), all partial derivatives in the other direction vanish. Let us assume that $\xi$ only depends on $x$ and $\eta$ only depends on $y$. If the boundary is not a straight line, $x$ and $y$ must be replaced by arc length $s$. If we determine $P$ from a boundary formed by an $\eta$-line (i.e. $\xi$ varies) and $Q$ from a boundary which is formed from a $\xi$-line, we find from equations

$$g^{11}x_{\xi\xi} = -x_\xi P; \quad g^{22}y_{\eta\eta} = -y_\eta Q; \tag{4.7}$$

Also, since $x_{\xi\xi}$, $x_\xi$ as well as $y_{\eta\eta}$, $y_\eta$ can be calculated from the boundary point distribution, equations can be solved for $P$ and $Q$, yielding

$$P := -g^{11}\frac{x_{\xi\xi}}{x_\xi}; \quad Q := -g^{22}\frac{y_{\eta\eta}}{y_\eta} \tag{4.8}$$

Expanding now Eq. 4.4 and dividing by $w$, yields

$$s_{\xi\xi} = -\frac{w_\xi}{w}s_\xi$$

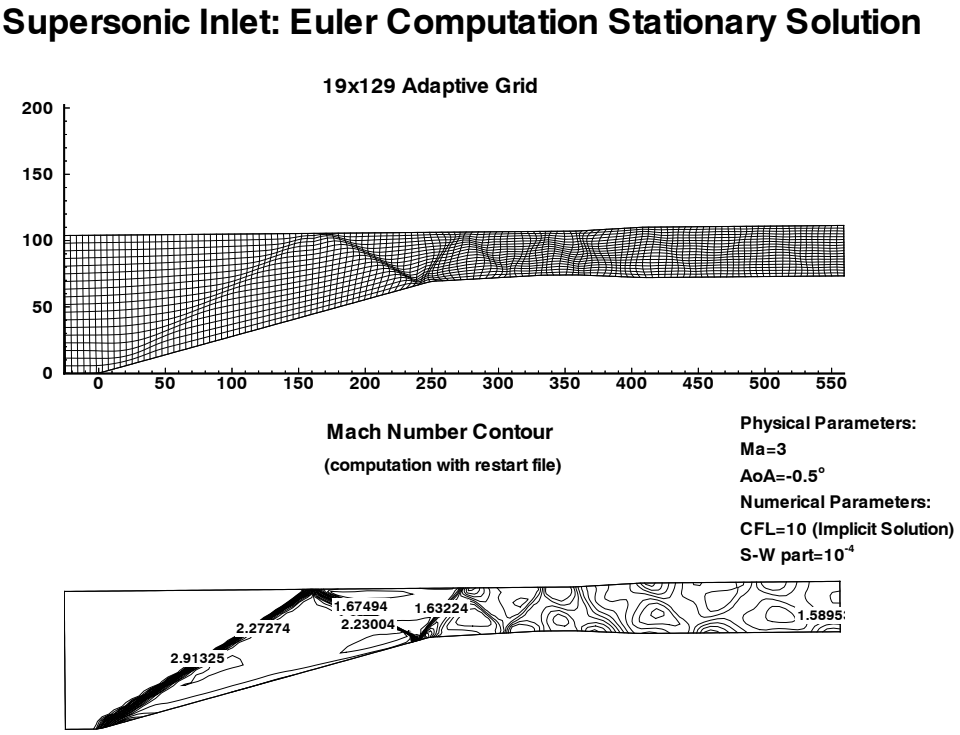


Figure 4.1: 1 block adptive grid for supersonic inlet, adapted by control functions.

Comparison with the first equation in Eqs. 4.7 leads to

$$P = g^{11}\frac{w_\xi}{w}; Q = g^{22}\frac{w_\eta}{w} \tag{4.9}$$

or when resolved for $w$

$$w = c_1\, exp[\int \frac{P}{g^{11}} d\xi],$$

where $c_1$ is the constant of integration. Using the relation for $P$ (same holds for $Q$), the weight function $w$ can be directly incorporated in the grid generation equations.

The use of these control functions guarantees that the specified boundary point distribution is obtained in the interior of the SD, i.e. this procedure yields the same grid point distribution along $\xi$- and $\eta$-lines as on the boundary. The numerical values of $P$ and $Q$ in the interior are determined by linear interpoloation between two opposite fixed sides. For curvilinear boundaries, arc length $s$ replaces $x$ and $y$.

Figs. 4.1 and 4.2 illustrate the grid adaptation by using control functions. It can be seen that the adapted grids possess a high degree of smoothness [45].

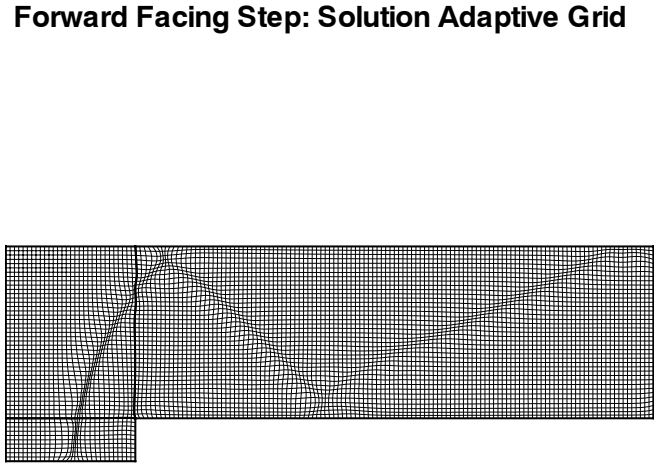


Figure 4.2: 3 block adaptive grid for forward facing step, adapted by control functions.

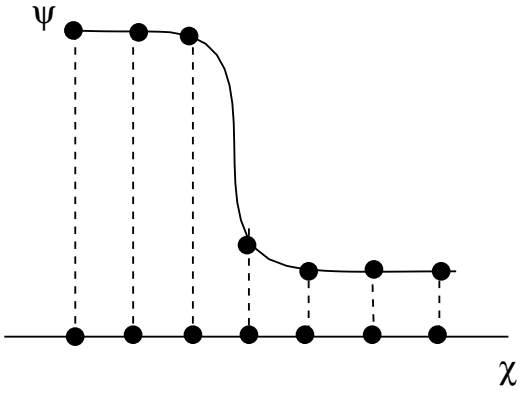


Figure 4.3: One–dimenisional monitor surface (MS). The initial grid in the physical SD is uniform. Lifting up the grid points produces the grid on the monitor surface. The variable *s* denotes arc lehgth on the MS

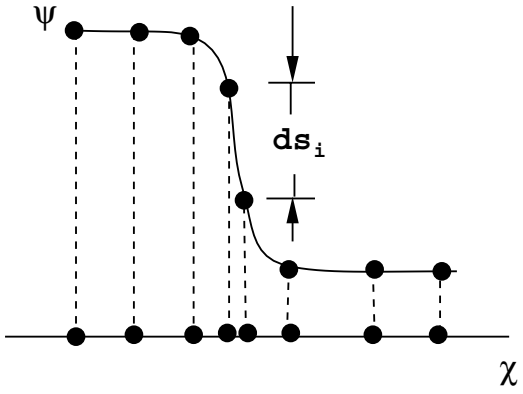


Figure 4.4: Repostioned grid on the monitor surface. Grid points are uniformly distributed on the monitor surface so that arc length spacing is constant. When projected down back to the physical solution domain, this results in a clustering according to the gradient of the monitor surface.

### 4.0.2 *Algebraic Adaptation Algorithms*

For the algebraic adaptation it is assumed that a 1D, or 2D grid has already been generated. The algorithm adapts this grid according to the specified set of weight functions that are generally obtained from the physical solution. Two algorithms will be presented that are extremely fast, so that they can be used for time dependent problems, too. The basic idea of these algorithms is taken from the papers [32], [33], and [31]. First, consider the one-dimensional problem of grid point clustering. Examples of monitor surfaces are given in the following three figures.

The following notations are used in one dimension. Their extension to higher - dimensions is straightforward.

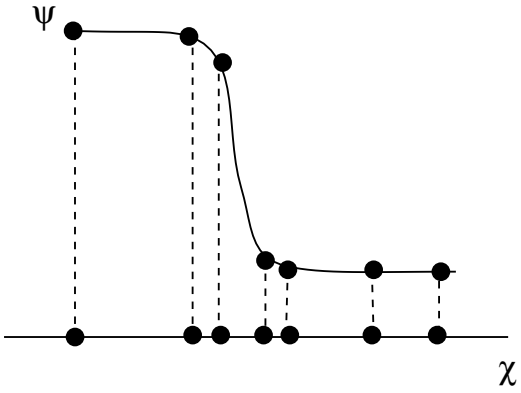


Figure 4.5: Repositioning of grid points on the monitor surface according to the magnitude of curvature, resulting in a clustering of grid points to regions where curvature $\kappa \neq 0$.

- ▷ the continuous variables in the physical SD, on the MS, and in the computational SD are given by $x, s$, and $\xi$, respectively.
- ▷ there are two sets of grid points: the initial grid points denoted by $x_i^I$, $s_i^I$ and the repositioned grid points which are given by $x_i, s_i$ where $i = 2(1)N-1$, and $N$ is the total number of grid points (or finite volumes).
- ▷ the initial and repostioned points on the monitor surface are also denoted as $\mathbf{P}_i^I$ and $\mathbf{P}_i$.
- ▷ grid points in the computational domain are denoted by $\xi_i$. In general, $\Delta\xi_i = 1$ is chosen.
- ▷ the weight function depends on arc length, i.e., $w_i = w(s_i)$. Of course, $x$ could also be chosen as the independent variable, because there is a one-to-one mapping between the grid points of the physical SD and those on the MS.
- ▷ half point values (cell faces) are averaged, e.g. $w_{i+\frac{1}{2}} := \frac{1}{2}(w_i + w_{i+1})$.

Large $w$ result in small $\Delta s_i$, i.e. there is a clustering of grid points where $w_i$ is large.

We first present the redistribution approach as described in [32] and [33], setting $c = constant$. After that, the algorithm of Hsu et al. [31] is outlined that uses the equidistribution statement with a variable $c(\xi)$. In both algorithms the original arc length of the MS and curve remain constant. The one-dimensional version of the algorithms employs the equidistribution statement in the form:

$$\int_{s_i}^{s_{i+1}} w ds = c(\xi_{i+1} - \xi_i)$$

where c is determined from the requirement of constant arc length.

**1. MS** Determine the monitor surface from the flow solution, e.g. use the Ma number distribution.

**2. Lift up** Lift the initial grid $(x_i^I)$ from the physical SD up to the MS, resulting in a grid $(s_i^I)$. If, for example, Ma is used, one could directly set $\Phi_i^I := Ma(x_i^I)$ -filtering of Ma values may

be needed - resulting in a table $(x_i^I, \Phi_i^I, s_i^I),\ i = 2(1)N-1$ where arc length is computed by piecewise linear approximation, using the Euclidean norm.

**3. Integrate** Determine the weight function ( e.g. magnitude of gradient of Ma number) and integrate the equidistribution statement from (all values are taken from the initial grid; superscript $I$ has been ommitted for notational simplicity) $s_1$ to $s$ on the MS. $c$ is obtained by integrating from $s_1$ to $s_N$

$$c = \frac{F(s_N)}{\xi_N - \xi_1}$$

where

$$F(s) := \int_{s_1}^{s} wdl$$

The values of $F(s)$ and $F(s_N)$ are related by the formula

$$F(s) = \frac{\xi - \xi_1}{\xi_N - \xi_1} F(s_N) \tag{4.10}$$

Note: In practice the integration is done numerically using the trapezoidal quadrature rule.

**4. Build table** From step 3 a table of values $(s_i^I, F(s_i^I)),\ i = 1(1)N$ is obtained. It should be noted that the grid $s_i^I$ is not distributed as required by the equidistribution statement and also would not lead to a uniform grid in the compuatational domain when projected back and therefore has to be repositioned.

**5. Find** $F(s_j)$ Determine the $F$ values for the repositioning of grid points. Since the left and right boundary values, denoted as $s_L$ and $s_R$, are known where $s_1 = s_1^I = s_L$, $s_N = s_N^I = s_R$ and requiring that the $\xi_j$ values have uniform grid spacing in the computational domain (as usual) with $\Delta\xi_i = 1$, one obtains from Eq. 4.11 the implicit condition for the repositioning of the $s_j$ values, namely

$$F(s_j) = \frac{j-1}{N-1} F(s_N),\quad j = 2(1)N-1 \tag{4.11}$$

It is evident that in order to obtain the new $s_j$ values, $F(s_j)$ has to be inverted.

**6. Inversion** The value $F(s_j)$ is used to search the table from step 4, returning an index $m(j)$ for which

$$F(s_m^I) < F(s_j) < F(s_{m+1}^I),\ j = 2(1)N-1$$

Linear interpolation within interval $(m, m+1)$ is utilized and yields

$$\alpha_j = \frac{F(s_j) - F(s_{m+1}^I)}{F(s_{m+1}^I) - F(s_m^I)} \tag{4.12}$$

Insertion of $\alpha_j$ directly results in

$$s_j = s_m^I + \alpha_j (s_{m+1}^I - s_m^I),\ j = 2(1)N-1 \tag{4.13}$$

**7. Repositioning** The new values in the physical domain are found by projecting back down

$$x_j = x^I_m + \alpha_j(x^I_{m+1} - x^I_m),\ \ j = 2(1)N-1$$

Using a more compact notation $P_j := (x_j, y_j)$ results in

$$\mathbf{P}_j = \mathbf{P}^I_m + \alpha_j(\mathbf{P}^I_{m+1} - \mathbf{P}^I_m),\quad j = 2(1)N-1$$

**8. Smooth** Relax the repositioned grid. Elliptic grid generation is used for a few smoothing iterations (3–4).

It should be noted that the equilibrium statement in step 3 was integrated such that the resulting integral could be solved for $\xi$ and, as a consequence, $F(s)$ had to be inverted to obtain the new arc length values. Another possibility is to divide Eq. 4.3 by $w$ so that the integration directly gives the values for $s$. In that case, however, weight function $w$ has to be determined as a function of $\xi$ and not of arc length $s$. Hence, $w(s(\xi))$ is needed, that is $s(\xi)$ has to be inverted to provide $\xi = \xi(s)$. In practical computations, the algorithm is used as presented. In Fig. 4.6 the result of an adaptation is depicted. The grid was adapted to capture a moving shock and a strong circular gradient (the details are given in [33]).

If this approach is to be applied in clustering points along a curve instead along the $x$ axis, step 7 of the above algorithm has to be replaced by steps 5 to 7 as described in the algorithm of Hsu below.

The MS algorithm can be extended to a 2D physical SD, requiring a MS in 3D space. To each gridline in the physical SD there is a corresponding curve in the MS. The curvature for this curve can be splitted into two parts, namely into normal and geodesic curvature, denoted by $\kappa_n$ and $\kappa_g$. The normal curvature is a measure of the rate at which the direction of the tangent leaves the MS while the geodesic curvature is the curvature of the curve in the MS. Since $\kappa_n$ represents changes of the MS itself, it is used in the weight function, i.e. $w = 1 + \mid \kappa_n \mid$.

The algorithm of Hsu does not use a monitor surface, but works directly on the curve $s = s(\xi)$. Adaptation in 3D is performed by independenty looping over all gridlines in directions $\xi$, $\eta$ and $\zeta$, that is there is no direct coupling of the points in a plane or within a block.

The algorithm of Hsu uses $c = c(\xi)$. The clustering of grid points is along a given curve, denoted by points $\mathbf{P}_i$.

$$\Delta s_i w_i = \lambda c_i \tag{4.14}$$

The constant $\lambda$ has been introduced to avoid an additional constraint among the $c_i$ that follows from the assumption of constant arc length. Therefore, $\lambda$ is chosen to satisfy this constraint. It should be noted that $s$ is now the arc length of the curve along which the points are to be redistributed.

Values for $c_i$ are not yet known. They are determined from the initial grid point distribution. $c_i := \Delta s^I_i;\ \ i = 2(1)N-1$

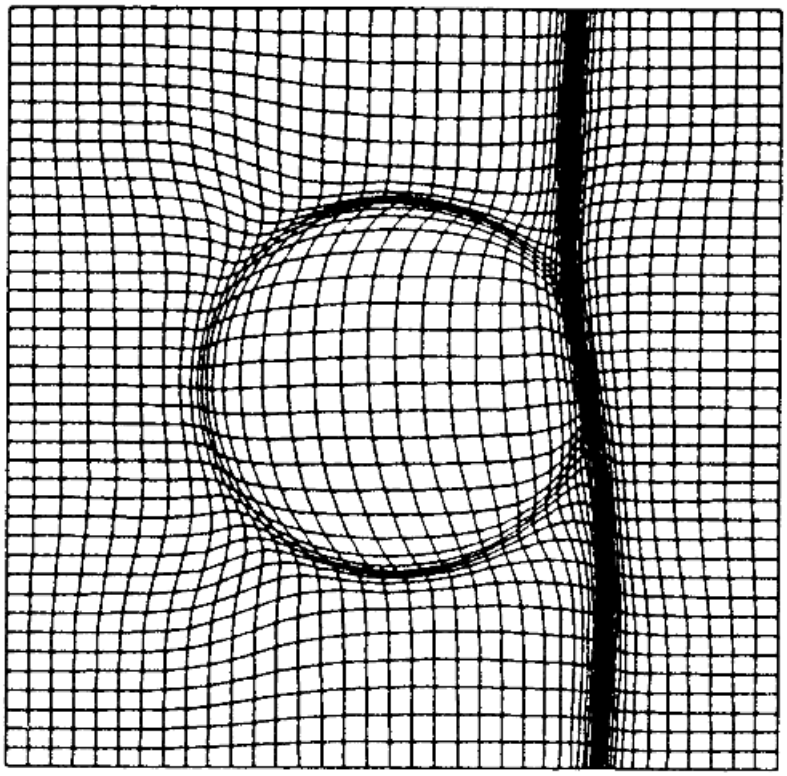

Figure 4.6: Adaptation of a grid using the monitor surface technique to capture a moving shock together with a strong circular gradient (vortex). The initial grid has uniform grid spacing.

For $w_i = 1$ this choice of the $c_i$ ensures that the grid point distribution remains unchanged, provided $\lambda$ is determined such that total arc length remains constant. From Eq. 4.14 one obtains

$$\lambda := \frac{\sum \Delta s_i^I}{\sum \frac{c_i}{w_i}} \tag{4.15}$$

Constant arc length also requires

$$\sum \Delta s_i = \sum \Delta s_i^I$$

The implementation of an adaptation algorithm based on this approach, comprises the following steps :

**1. Determine** $c_i = \Delta s_i^I$ obtained from the initial grid.

**2. Compute** $\lambda = \frac{\sum \Delta s_i^I}{\sum \frac{c_i}{w_i}}$ ; this choice of $\lambda$ guarantees that total arc length of curve $s(\xi)$ remains constant.

**3. Determine** $\Delta s_i = \lambda \frac{c_i}{w_i}$

**4. Arc length** Calculate the arc length up to point i: $s_i = \sum_{k=1}^{i} \Delta s_k$

**5. Find index** Find index $m(j)\ \varepsilon\ N$ such that: $s_m^I < s_i < s_{m+1}^I$

**6. Interpolate** Use linear interpolation between points $(x_m, y_m)$ and $(x_{m+1}, y_{m+1})$, to determine the repositioned coordinates. This is done by using $y = a_m x + b_m$ with the parameters

$$\begin{aligned} a_m &:= \frac{y_{m+1}^I - y_m^I}{x_{m+1}^I - x_m^I} \\ b_m &:= \frac{y_m^I x_{m+1}^I - y_{m+1}^I x_m^I}{x_{m+1}^I - x_m^I}. \end{aligned}$$

**7. Reposition** Next $\Delta s_i = (x_{i+1} + y_{i+1})$ the repositioned coordinates $(x_{i+1}, y_{i+1})$ have to be determined. Starting from a boundary point, the repositioned coordinate values $(x_i, y_i)$ are assumed to be known.

$$\begin{aligned} \Delta s_i = (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 = \\ (x_{i+1} - x_i)^2 + (a_m x_{i+1} + b_m - y_i)^2 \end{aligned}$$

This equation has to be solved for $x_{i+1}$. The equation has two solutions. Step 5 requires: $x_m^I < x_{i+1} < x_{m+1}^I$ and $y_m^I < y_{i+1} < y_{m+1}^I$ with $y_{i+1} = a_m x_{i+1} + b_m$

**8. Smooth** Relax the repositioned grid. Elliptic grid generation is used for a few smoothing iterations (3-4).

Notes to the algorithm:

- If the straight line in 7 has a slope of more than $\pi/4$, the role of x and y is interchanged. This can be easily checked by comparing $\mid y^I_{m+1} - y^I_m \mid$ with $\mid x^I_{m+1} - x^I_m \mid$.

- In order to have a smooth grid point distribution, the weight functions $w_i$ should be filtered, e.g. averaged. Otherwise kinks in the repositioned grid may occur.

- Adaptation works on a curve by curve basis. For 3D a loop over all coordinate directions is needed.

- The request that total arc length remains fixed, may be changed. Arc length has no physical meaning.

- The adaptation is extremely fast. It can be used for moving meshes.

- Adaptation can be performed in a step by step fashion. For example, one can adapt the grid to an oblique shock, while preserving the fine grid in the BL. Suppose that grid lines in the BL are $\xi$ lines ($\eta$ = constant). For all of these lines $w_i = 1$ is chosen. Then the fine grid in the BL remains unchanged. For adaptation of the oblique shock $\eta$ lines ($\xi$ = constant) are adapted the using pressure gradient or Ma number gradient. The final grid depicts a very sharp oblique shock and also retains the fine BL grid. No points from the BL are moved to the oblique shock region.
  On colorplate E a solution adapted grid for a monoblock cone, constructed in that way, is shown. The grid is adapted to capture both the oblique shock and the BL.

# 5.0 *A Grid Generation Meta Language*

## 5.0.1 *Topology Input Language*

With the parallel computers of today substantially more complex fluid flow problems can be tackled. Complete aircraft configurations, complex turbine geometries, or flows including combustion have been computed in industry. Consequently, geometries of high complexity are now of interest as well as very large meshes, for instance, computations of up to 30 million grid points have been performed. Clearly, grid generation codes have to be capable to handle this new class of application.

Convential grid generation techniques derived from CAD systems that interactively work on the CAD data to generate first the surface grid and then the volume grid are not useful for these large and complex grids. The user has to perform tens of thousands of mouse clicks with no or little reusability of his input. Moreover, a separation of topology and geometry is not possible. An aircraft, for example, has a certain topology, but different geometry data describe different aircraft types. The topology definition consumes a certain amount of work, since it strongly influences the resulting grid line configuration. Once the topology has been described, it can be reused for a whole class of applications. One step further would be the definition of objects that can be translated, rotated and multiplied. These features could be used to build an application specific data base that can be used by the design engineer to quickly generate the grids needed.

In the following a methodology, which comes close to the ideal situation described above, is presented. To this end a completely different grid generation approach will be presented. A compiler type grid generation language has been built, based on the ANSI–C syntax that allows the construction of objects. This grid generation process can be termed hands off grid generation. The user provides a (small) input file that describes the so called ***TIL*** code to built the wireframe model, see below, and specifies the filenames used for the geometry description of the configuration to be gridded. A variety of surface definitions can be used. The surface can be described as a set of patches (quadrilaterals) or can be given in triangular form. These surface definitions are the interface to the CAD data. In general, a preprocessor is used that accepts surface definitions following the *NASA IGES CFD* standard [**?**] and converts all surfaces into triangular surfaces. That is, internally only triangular surfaces are used. In addition, the Gridpro code allows the definition of analytic surfaces that are built in or can be described by the user in a C function type style. The user does not have to input any surface grids, that is, surface and volume grids are generated in the same run without any user interaction. This approaoch has the major advantage that it is fully reusable, portable, and that highly complex grids can be built in a step by step fashion from the bottom up, generating a hierarchy of increasingly complex grid objects, for instance see Sec. 5.0.2.

TIL allows the construction of complex grids by combining predefined objects along with operators for placement of these objects. For example, the grid around an engine could be an object (also referred to as component). Since an aircraft or spacecraft generally has more than one engine located at different positions of its structure, the basic engine object would have to be duplicated and positioned accordingly. In addition, the language should be hierarchical, allowing the construction of objects composed of other objects where, in turn, these objects may be composed of

more basic objects etc. In this way a library can be built for different technical areas, e.g. a turbomachinery library, an aircraft library or a library for automobiles. The Topology Input Language [35] has been devised with these features in mind. It denotes a major deviation from the current interactive blocking approach and offers substantial advantages in both the complexity of the grids that can be generated and the human effort needed to obtain a high quality complex grid. No claims are made that TIL is the only (or the best) implementation of the concepts discussed, but is believed that it is a major step toward a new level of performance in grid generation, in particular when used for parallel computing.

TIL has been used to generate the grids thar are presented below. The surfaces used are defined analytically or are given by a CAD system. The surface definition files are specified outside of the **GridPro** environment. The grid generator needs three different files as input. The TIL files (extension **.fra**) contain the topology definitions. The toplogy of a complex grid can be organized into **components or objects** that may be grouped. A schedule file is needed (extension **.sch**) to specify directions for the numerical scheme of the grid generator, i.e. to obtain a certain convergence level (the user should keep in mind, however, that there is nothing like a converged grid). The **GridPro** [35] package reduces the amount of information needed by an order of magnitude. The following figures show 2D and 3D grids that have been generated with the automatic zoning (blocking) approach.

The versatility and relative ease of use (comparable in difficulty in mastering LATEX) will be demonstrated by presenting several examples along with their TIL code. The examples a presented demonstrate the versatility of the approach and show the high quality of the grids generated.

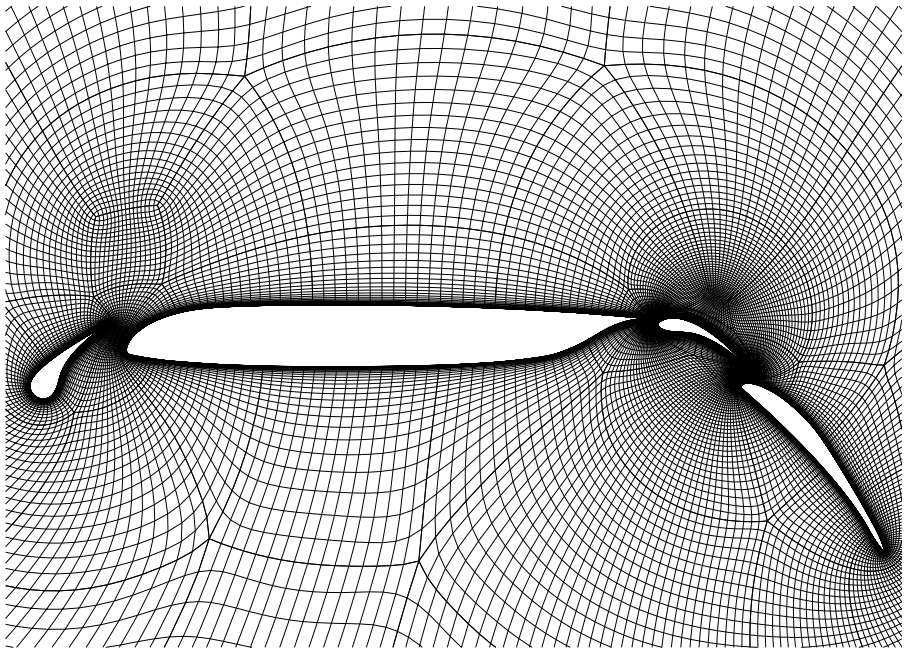

Figure 5.1: Navier-Stokes grid for a four-element airfoil, comprising 79 blocks.The first layer of grid points off the airfoil contour is spaced on the order of $10^{-6}$ based on chord length.

In order to generate a grid The following steps have to be followed in the grid generation process.

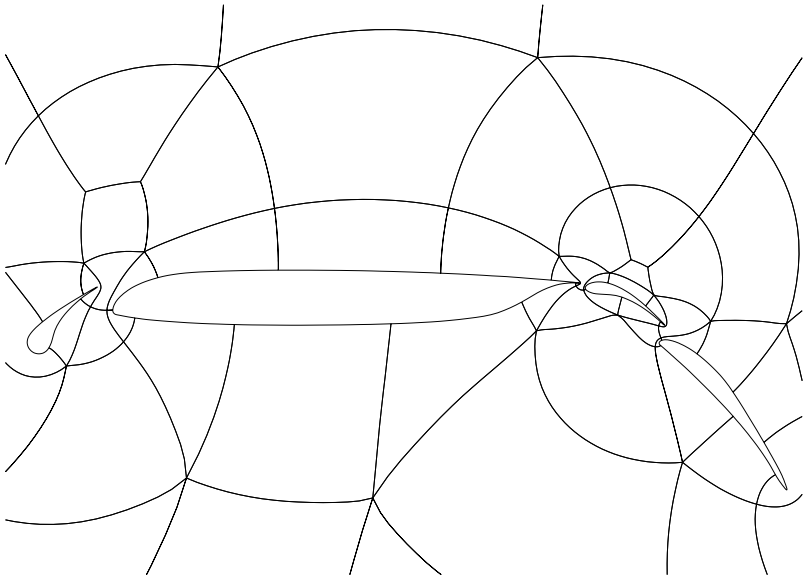

Figure 5.2: The figure shows the block structure of the four element airfoil generated by **GridPro**.

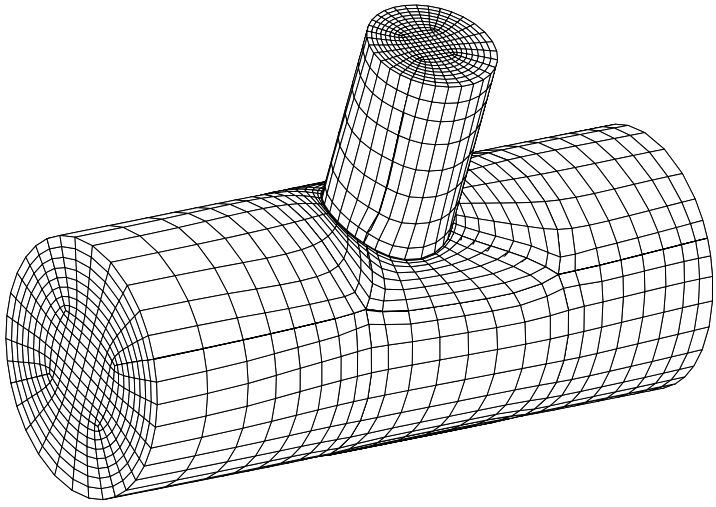

Figure 5.3: Grid for a T joint that has numerous industrial applications.

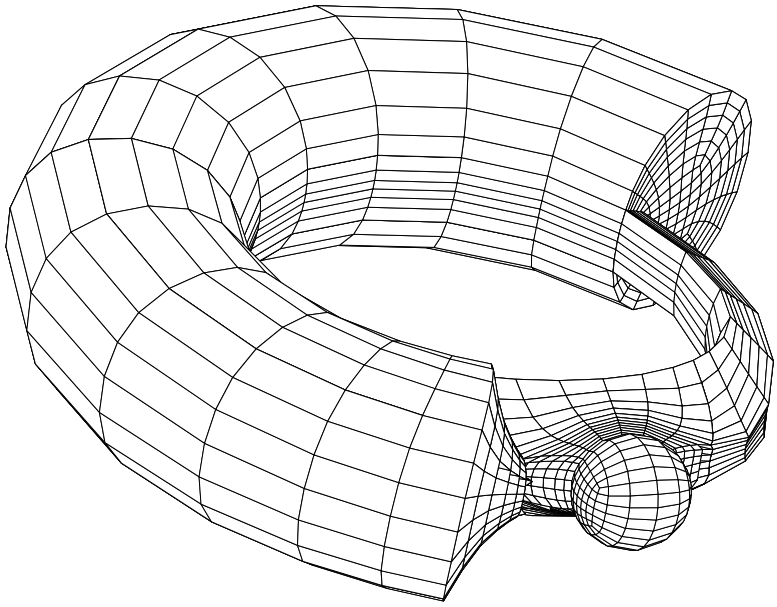

Figure 5.4: Sphere in a torus. The input for this grid is presented in the following tables.

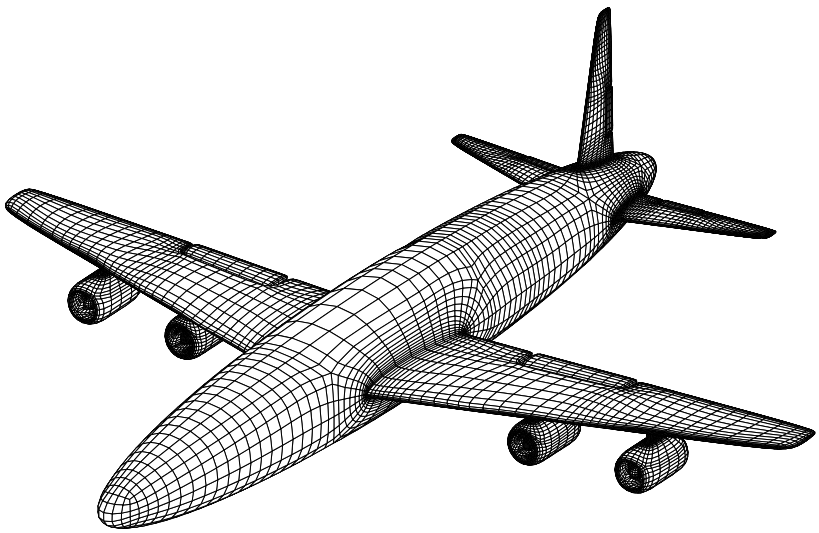

Figure 5.5: Complete 3D grid for a generic aircraft with flaps, constructed from analytical surfaces. However, the topology is exactly the same as for a real aircraft.

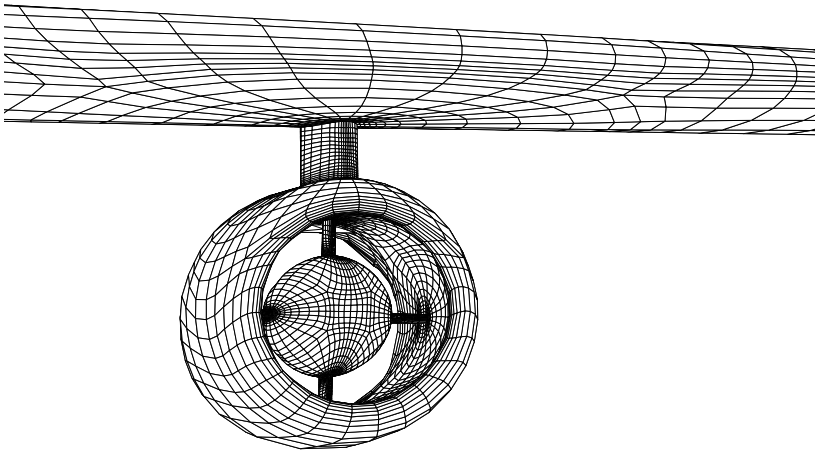

Figure 5.6: The picture shows a blowup of the engine region of the generic aircraft.Future TSTO or SSTO vehicles will exhibit a similar complex geometry, necessitating both the modelization of internal and external flows.

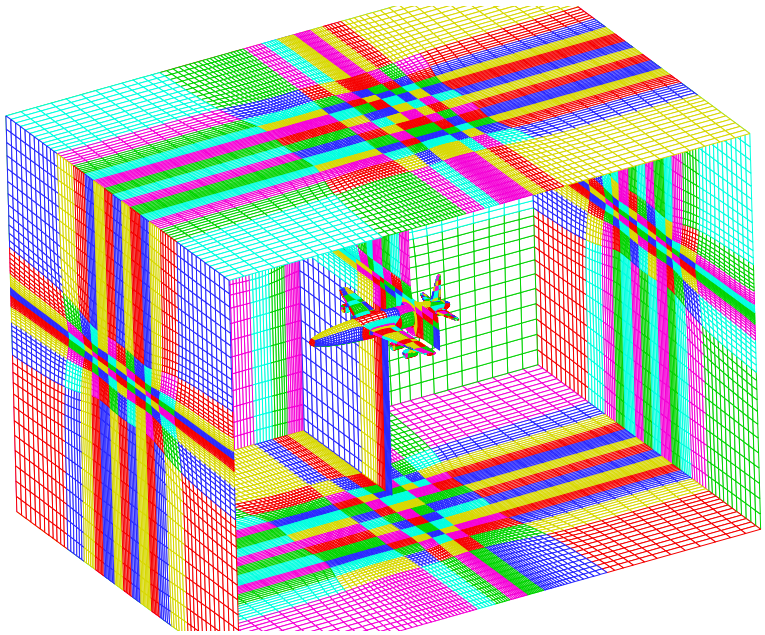

Figure 5.7:

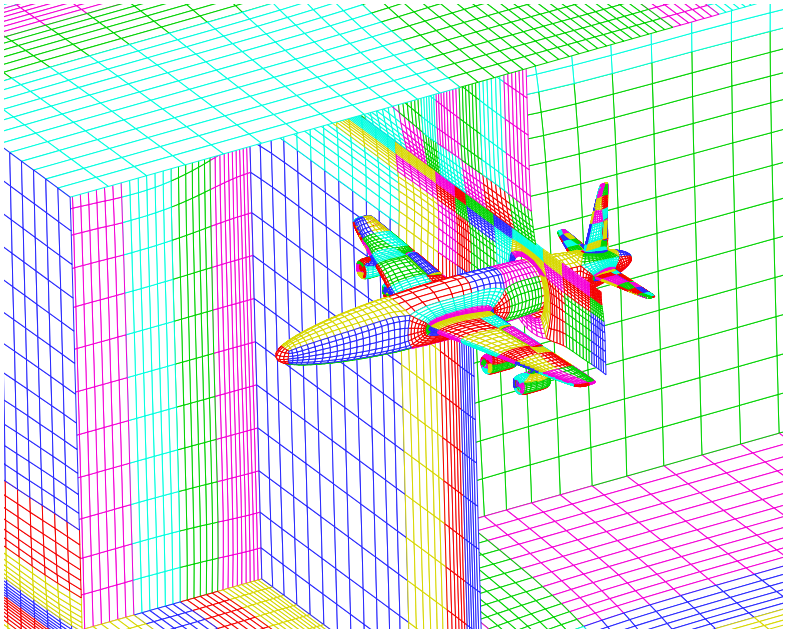

Figure 5.8:

**Step 0** In a preprocessing step, all surfaces have to be prepared, providing a certain smoothness. This procedure is called carpeting, see for example [37] and is not in **GridPro**. For the proper surface format, see below.

**Step 1** In the first step, all surfaces (curves in 2D) that form the physical boundaries of the solution domain have to be labeled, that is labels for the corners (or vertices) have to be provided. next, a list of all neighboring vertices has to be specified. The user has complete freedom in the definition of the surfaces.

**Step 2** In the second step, the user designs the block topology. This is a creative process that cannot be entirely automated. The blocking depends on the flow physics and, unless specific rules are built into the grid generator, the expertise of the aerospace engineer is needed. The choice of the topology itself leads to a clustering of grid points. Therefore, selecting the proper topology may be more effective regarding grid adaptation than the use of additional control functions. The major advantage is that TIL can be used to generate simple components first that can the be assembled to form complex grids. Thus, a component library can be built up.

In the following some general explanations about TIL are given. The concepts **function** and **component** are used interchangeably. Again, it is stressed, as a general rule, that the user has to label the surfaces (curves in 2D) that form the boundary of the solution domain (SD). How the SD is subdivided into surfaces is up to the user. Any surface is identified by its corners (or vertices), which are labeled in increasing order. Approximate corner coordinates should be specified, that is, the vertex need not be directly on the surface, but should not be positioned too far from it. The code will do the necessary projections and will also construct a grid on each surface. The labeling only serves to determine the grid topology. In addition, the user has to specify a list of neighboring (or connecting) corners that may be on other surfaces. In the following some of the TIL rules are presented that will be needed to understand the torus-sphere example, presented below.

- ▷ Any grid generated by **GridPro** is comprised by components.
- ▷ Components itself may comprise more basic components.
- ▷ The validity of component names (or numbers) is restricted to the component in which they have been defined. This also holds for variable names.
- ▷ Components in TIL are similar to functions in ANSI C or C++. A realization of a component is achieved by using a function. In the definition of a function its name is followed by the formal list of parameters, enclosed in parentheses (see e.g. Table 5.2). The component is called using the values of the actual parameter list.
- ▷ Each function contains a body in which the necessary definitions and statements occur (see, e.g. Table [5.2]).
- ▷ TIL provides special data types, namely **cIN**, **sIN**, **cOUT**, and **sOUT**. In the formal parameter list, variables of that data type can be declared, as can be seen from Tables [5.2 - 5.4]. The letters **c** and **s** stand for corner and surface, respectively while IN and OUT specify data to be imported or exported. In other words, IN indicates that variables are of input type and OUT denotes output variables. TIL is more specific than a computer language since it also specifies the direction of the data transfer.

- ▷ If a negative integer value is specified for the **In** or **OUT** data types, the corresponding **IN** or **OUT** array is initialized with 0 (for example see Table 5.3).
- ▷ In referencing corners there exist several possibilities. For example, if one specifies **cOUT(2 4..6)** an array **cOUT[1] to cOUT[4]** is allocat ed, containing the coordinates of corners labeled 2, 4, 5, and 6 in the respective component. The range parameter is given by .. . The array is allocated in the calling function (or parent function). The elements of this array can be accessed by, e.g. 1:1 or 1:3. The first integer gives the component number of the child function, the second one references the array index.
- ▷ In connecting corners i on component I and j on component J, it suffices to either specify the connectivity in I or in J, i.e. only once.
- ▷ Arrays begin with index 1. This is in contrast to ANSI C where arrays start with index 0.
- ▷ Variable names are restricted to the block in which they are defined.

The input data for the sphere in a torus example, presented in Fig. 5.4, are given in the following tables. The cross section of the torus has a radius of 0.5. The central circle of the torus is in the y-z plane (x-axis points up, y-axis goes to the right, z-axis points away from the reader) with its center at the origin and a radius of 1.5. The sphere has a radius of 0.25 and its center is at (0, 0, 1.5).

In Table 5.1 the components comprising the torus-sphere grid are read in. The **bINt()** function is the parent function to all other functions (components). The **INPUT** statement is used to read in predefined components, follwed by the component number and the component name with the associated list of the actual arguments. The first line after **BEGIN** of function **main** defines the torus surface analytically; its defintion is contained in the file **surf/torus.h**. The torus surface is labeled surface 1. The following **INPUT** statement reads in component **torus**, which consists of 4 cross sections, each defined as a component of type **tsec** (see below). In the argument list it is specified that the analaytic torus surface is an input surface to component **torus**, and that corners 1..4 of component 1 (these are the 4 inner points of a cross section forming the square that can be seen in Fig. 5.4) as well as corners 1..4 of component 2 (again a square) of component **torus** are made available to function **main** via the **cOUT** statement. It is clear from Fig. 5.4 that these corner points have to be known in component **ball** since there is a connection between these cross section corners and the corresponding corners on the **ball** surface. In addition, the same corner point on the **ball** surface is connected to its 3 neighboring corner points on the cube, forming a 4D hypercube topology. The next **INPUT** statement reads in component **ball**, labeled surface 2, and shifts its origin by (0 0 1.5). In this context the $*$ does not denote a multiplication. The argument list of **ball** contains a **cIN** data type for 8 corners. Thus, the 8 corners having been made available by the **cOUT** statement in the **torus** argument list serve as input for the **ball** component. It should be noted that **1:1..8** used in **cIN** refers to the numbering in function **main()**, while the **cOUT** data type uses the numbering of component **torus**. This distinction is important in order to address the correct variables.

In Table 5.2 the topology input for the sphere is depicted. The first line after **BEGIN** specifies a surface, numbered surface 1 of type ellipsoid (ellipse in 2D) with 3 numbers denoting the lengths of the major axes in the x,y, and z coordinate directions. The topology of the **ball** component is equivalent to a cube. Hence, the coordinates of the 8 corner points (or vertices) of that cube are connected to corners **c 1 - c 8** that belong to two cross sections of the **torus** component.

```
# Begin wire frame file #

COMPONENT bINt ()
# Define component bINt (ball_in_torus)

BEGIN
S1 -implic "surf/torus.h";
INPUT 1 torus (sIN (1),cOUT (1:1 . . 4 2:1 . . 4));
# Input a torus and export 8 corners from it.

INPUT 2 (0 0 1.5) *ball (cIN (1:1 . . 8));
# Input a ball with translation (0 0 1.5) and
# import 8 corners.

END
```

Table 5.1: Input of the toplogy files of the torus and ball (sphere) components.

```
COMPONENT ball(cIN c[1..8]) #define component ball
BEGIN

s 1 -ellip (4 4 4); #define surface by id
c 1 -0.25 0.25 -0.25 -s 1 -L c:1;
c 2 -0.25 0.25   0.25  -s 1 -L c:2 1;
c 3  0.25  0.25   0.25 -s 1 -L c:3 2;
c 4 0.25 0.25 -0.25 -s 1 -L c:4 3 1;
c 5 -0.25 -0.25 -0.25 -s 1 -L c:5 1;
c 6 -0.25 -0.25 0.25 -s 1 -L c:6 2 5;
c 7 0.25 -0.25 0.25 -s 1 -L c:7 3 6;
c 8 0.25 -0.25 -0.25 -s 1 -L c:8 4 7 5;
x b c:1 c:7 1 7;

# define corners by id, initional position, surfaces
# that the corner is on and the links (edges) to other
# defined corners that the corner has.
# exclude 2 blocks defined by corner pairs.
END
```

Table 5.2: Toplogy file for the ball component.

```
COMPONENT torus()
BEGIN
s 1 -analyt "surf/torus.h"; # Define surface
INPUT 1 (1 0 0 0 0.707 0 0 0.707 0)*
tsec(sIN (1),cIN (-8),(-8),cOUT (1..8));
INPUT 2 (1 0 0 0 -0.707 0 0 0.707 0)*
tsec(sIN (1),cIN (1:1..8),(-8),cOUT (1..8));
INPUT 3 (1 0 0 0 -0.707 0 0 -0.707 0)*
tsec(sIN (1),cIN (2:1..8),(-8),cOUT (1..8));
INPUT 4 (1 0 0 0 0.707 0 0 -0.707 0)*
tsec(sIN (1),cIN (3:1..8),(1:1..8),cOUT (1..8));
x f 1:1 3:1 1:2 3:2 1:3 3:3 1:4 3:4;
x f 1:5 3:5 1:6 3:6 1:7 3:7 1:8 3:8;
# Input tsec(torus_section) with transformation
# ((1 0 0) (0.707 0) (0 0.707 0)).
END
```

Table 5.3: Topology file for the torus component.

The following 3 numbers specify the initial ccordinates of the corner. These coordinate values only serve to provide an initial solution for the grid generation process. The final coordinates of the corners are computed by the program. The **-s** option followed by an integer indicates the surface that it belongs to. The link option, **-L**, lists all corner points to which the current corner is connected. The **x** command excludes pairs of corner points that otherwise would be falsely connected.

Table 5.3 gives the TIl input for the torus component. It is important to note that the **torus** component itself comprises 4 components, named **tsec** (Table 5.4). The topology of the torus is obtained by cutting it in the cross section plane. This cut results in 2 cross sections (see Fig. 5.4), while the other 2 are the left and right cross sections connecting to the sphere surfaces. Component **tsec** represents such a cross section of the torus and thus is a more basic object (again see Fig. 5.4). Each cross section contains 8 corners and its corners are either connected to 2 other cross sections ( 8 corner points for each cross section), which is the case for the cross sections at the cut, or to another cross section (8 corner points), one of the sphere surfaces (4 corner points), and to the corner points above and below the sphere of another cross section. Again, reference is made to Fig. 5.4 for visualization. Therefore, each **tsec** component needs an input of 16 corner points, which is reflected in its argument list (see Table 5.4. The 9 numers after the **INPUT** statement form a matrix, used for the transformation of the original cross section coordinates. The argument list in

The command **g** (see Table 5.4) specifies the number of grid points to be distributed along an edge. The first two variables specify the corners (vertices) of the edge. Vertices can be identified by integer values or by variable names. The third value is the number of grid points.

The surface of a torus is generated by rotating a circle about an axis that is in the plane of the circle without intersecting it. A point on the surface is determined by angles $\Theta$ and $\Phi$ that are in the y-z plane and in the cross section plane, respectively. The torus surface is given by the formula

```
COMPONENT tsec(sIN t,cIN c[1..8],C[1..8])
# Define component tsec(torus_section)
BEGIN
c 1 -0.35 1.5 0 -L c:1 C:1;
c 2 -0.35 2.7 0 -L c:2 C:2 1;
c 3 0.35 2.7 0 -L c:3 C:3 2;
c 4 0.35 1.5 0 -L c:4 C:4 3 1;
c 5 -0.5 1.0 0 -s t -L c:5 C:5 1;
c 6 -0.5 3.0 0 -s t -L c:6 C:6 2 5;
c 7 0.5 3.0 0 -s t -L c:7 C:7 3 6;
c 8 0.5 1.0 0 -s t -L c:8 C:8 4 7 5;
x f 5 7;
g 1 c:1 24 1 C:1 24;
END
```

Table 5.4: Topology file for the tsec component.

```
# Begin schedule file #
step 1: -w 100 -R sing 0.4 -r 4 43 -R adj 107 -S 100
step 100: -g 8:0 24 16:8 24 24:16 24 24:0 24 \
-R sing 0.4 -R sharp 0 - S
write -a -D 0 -f dump.tmp
write -a -D 3 -f blk.tmp
# End schedule file #
```

Table 5.5: Schedule file to specify output format.

```
#define FUNCU ya=sqrt(y*y+z*z)-1.5,1.0 - (ya*ya + x*x)*4
/* Define torus surface */
#define Ulen (-1.0 ) /* -1.0 for no period */
#define FUNCV 0.0 /* Do not care in this case */
#define Vlen (-1.0)
#define FUNCW 0.0
#define Wlen ( -1.0 )
#define FUNCZ 0.0
#define FUNCY 0.0
#define FUNCX 0.0
```

Table 5.6: Surface definition file for the sphere surface.

```
#define FUNCU (x*x + y*y + z*z)*16 - 1.0
/* Define ball surface */
#define Ulen (-1.0 ) /* -1.0 for no period */
#define FUNCV 0.0 /* Do not care in this case */
#define Vlen (-1.0)
#define FUNCW 0.0
#define Wlen ( -1.0 )
#define FUNCZ 0.0
#define FUNCY 0.0
#define FUNCX 0.0
```

Table 5.7: Surface definition file for the torus surface.

$(R = 1.5, r_c = 0.25)$

$$\mathbf{r}(\Theta,\Phi) = ([R + r_c cos\Phi]cos\Theta, [R + r_c cos\Phi]sin\Theta, r_c sin\Phi)$$

**FUNCU**, **FUNCV**, and **FUNCW** are the transformed coordinates that would be denoted as $u, v, w$ in a mathematical context, being functions of the Cartesian variables $x, y, z$. **Ulen**, **Vlen**, **Wlen** specify the interval for which the transformed variables are defined. **FUNCX**, **FUNCY**, and **FUNCZ** describe the inverse transformation.

### 5.0.2 *TIL Code for the Cassini-Huygens Configuration*

In the following we present the TIL code to generate a 3D grid for the Cassini–Huygens space probe. Cassini–Huygens is a joint NASA–ESA project and will be launched in 1997. After a flight time of 8 years, the planet Saturn will be reached and the Huygens probe will separate from the Cassini orbiter and enter the Titan atmosphere, Saturn's largest moon [**?**]. Titan is the only moon in the solar system possessing an atmosphere (mainly nitrogen). During the two hour descent measurements of the composition of the atomsphere will be performed by several sensors located at the windward side of tyhe space probe. In order to ensure that the laser sensor will function properly dust particles must not be convected to any of the lens surfaces of the optical instruments [46]. Therefore, extensive numerical simulations have been performed to investigate this problem.

In order to compute the microaerodynamics caused by the sensors the proper grid has to be generated. A sequence of grids of increasing geometrical complexity has been generated. The simplest version, comprising 6 blocks, does not contain the small sensors that are on the windward side of the probe. With increasing comlexity the number of blocks increases as well. The final grid, modeling the SSP and GCMS sensors, comprises 462 blocks. However, it is important to note that each of the more complex grids is generated by modifying the TIL code of its predecessor.

#### 5.0.2.1 *TIL Code for 6 Block Cassini–Huygens Grid*

The general approch for constructing the Cassini–Huygens grids of increasing complexity was to first produce an initial mesh for the plain space probe without any instruments which is of relatively simple shape equivalent to a box. Thus the first topology is a grid that corresponds to *a box in a box*. The refinement of the grid is achieved by adding other elements designed as different objects. The combination of all objects results in the final grid. As the first step in generating the Huygens grid, a topology for the Huygens body (without any additional elements such as SSP, GCMS and sensors) is designed, shown in Fig. 5.9. This topology describes the spherical far field and the body to be simulated. To generate this grid the following TIL code has been written. A 6 block grid is obtained, shown in Figs. 5.10 and **??**. This grid has a box–in–box structure: the outer box illustrates the far field and the interior one is the Huygens body. The corner connection to be initialized results in the wire frame topology, shown in Fig. **??**.

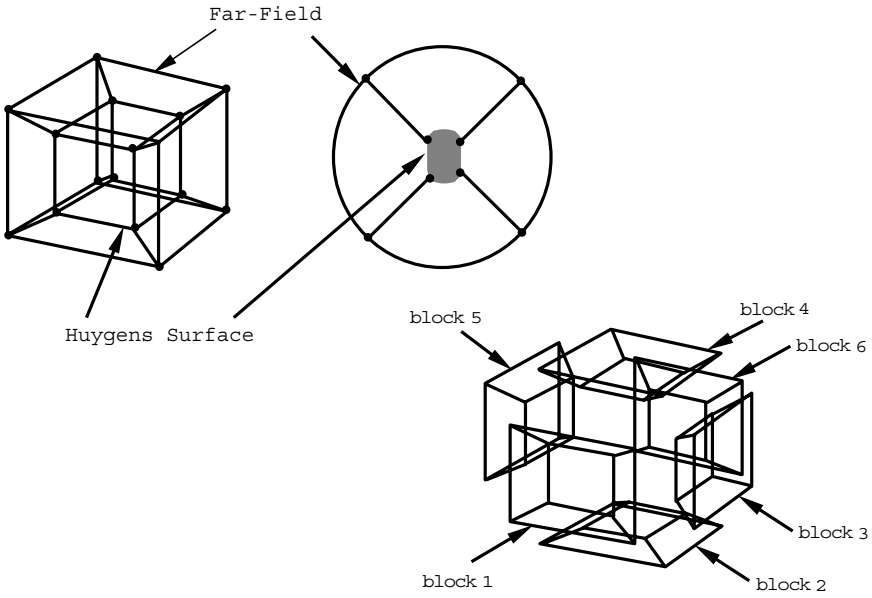


Figure 5.9: Topological design for the Huygens space probe grid. In this design all elements, such as SSP, GCMS and sensors are ignored.

**TIL Code for 6 Block Huygens Space Probe Grid:**

**SET GRIDDEN 47**
**COMPONENT Huygens()**
**BEGIN**

```
s 1 -tube "HuygensProfil.lin";
s 2 -ellip(0.00015625 0.00015625 0.00015625) -o;

  INPUT 1 (650 0 0 0 650 0 0 0 650)*
        loop(sIN (1),cIN (-4),(-4),cOUT (1..4));

  INPUT 2 (0 0 -4525)*(4525 0 0 0 4525 0 0 0 4525)*
        loop(sIN (2),cIN (1:1..4),(-4),cOUT (1..4));

  INPUT 3 (0 0 325)*(650 0 0 0 650 0 0 0 650)*
        loop(sIN (1),cIN (1:1..4),(-4),cOUT (1..4));

  INPUT 4 (0 0 4525)*(4525 0 0 0 4525 0 0 0 4525)*
        loop(sIN (2),cIN (3:1..4),(2:1..4),cOUT (1..4));
```

**END**

**COMPONENT loop(sIN s, cIN c[1..4], C[1..4])**
**BEGIN**

```
  c 1 -1 - 1 + 0 -s s -L c:1 C:1;
```

```
  c 2  1 - 1 + 0 -s s -L c:2 C:2 1;
  c 3  1 + 1 + 0 -s s -L c:3 C:3 2;
  c 4 -1 + 1 + 0 -s s -L c:4 C:4 1 3;
END
```

#### 5.0.2.2 *TIL Code for 24 Block Cassini–Huygens Grid*

By modifying the previous topology the SSP object is generated. The SSP is of cylindrical form which can be described by introducing eight corner points, shown in Fig. 5.11. The wire frame topology has different configuration than design topology. The wire frame topology has different configuration than design topology. The relationship beween corners reaim unchanged, shown in Fig. **??**. This kind of topology definition allows the subdivision of a more complex object into a set of simple ones. Adding objects into the principal body, a TIL code will be generally extended. This principle can be seen in the following TIL code.

**TIL code for 24 block Huygens**

```
COMPONENT HuygensSSP()
BEGIN

s 1 -plane(0 0 -1 162.4);
s 2 -quad "../Data/HuygensData/HuygSSP.dat";
s 3 -ellip(0.000078125 0.000078125 0.000078125) -o -t 0 0 300 -c
1000;

  INPUT 1 (0 248 162.4)*(80 0 0 0 80 0 0 0 80)*
        holeSSP(sIN (1),(2),cIN (-8),(-8),cOUT (1..8));

  INPUT 2 (0 248 0)*(80 0 0 0 80 0 0 0 80)*
        loop(sIN (-1),(2),cIN (1:5..8),(-4),cOUT (1..4));

  INPUT 3 (0 248 -50)*(40 0 0 0 40 0 0 0 40)*
        loop(sIN (-1),(-1),cIN (1:1..4),(2:1..4),cOUT (1..4));

  INPUT 4 (0 248 -12500)*(100 0 0 0 100 0 0 0 100)*
        loop(sIN (3),(-1),cIN (3:1..4),(-4),cOUT (1..4));

  INPUT 5 (0 0 -12500)*(12800 0 0 0 12800 0 0 0 12800)*
        loop(sIN (3),(-1),cIN (4:1..4),(-4),cOUT (1..4));

  INPUT 6 (0 0 -50)*(690 0 0 0 690 0 0 0 690)*
        loop(sIN (-1),(-1),cIN (5:1..4),(3:1..4),cOUT (1..4));

  INPUT 7 (650 0 0 0 650 0 0 0 650)*
        loop(sIN (2),(-1),cIN (6:1..4),(2:1..4),cOUT (1..4));
```

```
  INPUT 8 (0 0 650)*(650 0 0 0 650 0 0 0 650)*
        loop(sIN (2),(-1),cIN (7:1..4),(-4),cOUT (1..4));

  INPUT 9 (0 0 700)*(690 0 0 0 690 0 0 0 690)*
        loop(sIN (-1),(-1),cIN (8:1..4),(6:1..4),cOUT (1..4));

  INPUT 10 (0 0 13100)*(12800 0 0 0 12800 0 0 0 12800)*
        loop(sIN (3),(-1),cIN (9:1..4),(5:1..4),cOUT (1..4));

        x b 7:1 8:3;
        x f 1:5 1:7 2:1 2:3 5:1 5:3 6:1 6:3 7:1 7:3;
        g 2:1 1:5 30;
END

COMPONENT holeSSP(sIN s,s1,cIN c[1..8],C[1..8])
BEGIN
  c 1 -0.5 -0.5 0 -s s    -L c:1 C:1;
  c 2  0.5 -0.5 0 -s s    -L c:2 C:2 1;
  c 3  0.5  0.5 0 -s s    -L c:3 C:3 2;
  c 4 -0.5  0.5 0 -s s    -L c:4 C:4 1 3;
  c 5 -1.0 -1.0 0 -s s s1 -L c:5 C:5 1;
  c 6  1.0 -1.0 0 -s s s1 -L c:6 C:6 2 5;
  c 7  1.0  1.0 0 -s s s1 -L c:7 C:7 3 6;
  c 8 -1.0  1.0 0 -s s s1 -L c:8 C:8 4 5 7;
END

COMPONENT loop(sIN s,s1, cIN c[1..4], C[1..4])
BEGIN
  c 1 -1 -1 0 -s s s1 -L c:1 C:1;
  c 2  1 -1 0 -s s s1 -L c:2 C:2 1;
  c 3  1  1 0 -s s s1 -L c:3 C:3 2;
  c 4 -1  1 0 -s s s1 -L c:4 C:4 1 3;
END
```

### 5.0.3 *TIL Topology and Grid Quality*

The code, however, is not completely foolprooff and situations may occur when inconsist toplogical data is specified. GridPro tries to do some repair when information is missing or is inconsistent. In these cases, error messages are issued that quickly lead to the detection of the error. The most common blunders are described below and are depicted in Figs. 5.13 to 5.15.

Grid topology describes the connectivity among the blocks that comprise the solution domain. Instead of describing the block topology, a 3D wireframe is given, from which the block topolgy can be derived. The wireframe is specified by the user, see examples below. The wireframe itself is a collection of points and edges whose relations are to be specified. This information allows the

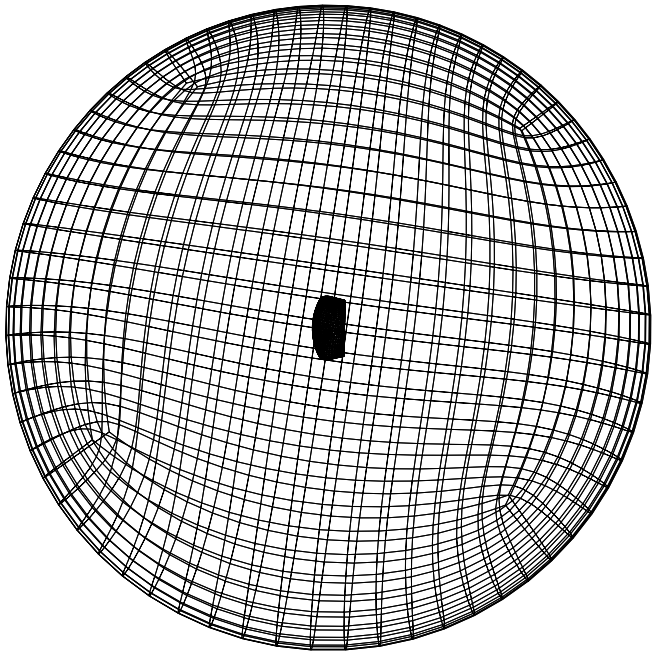

Figure 5.10: The 6 block Huygens grid is depicted. It is bounded by a large spherical far field, in which the Huygens space probe is embedded. The ratio of the far field radius and the Huygens radius is about 20.

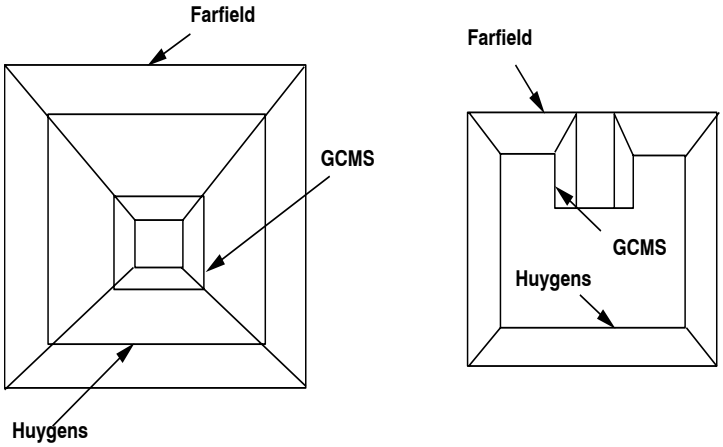


Figure 5.11: Illustration of the block topology for the 24 block Huygens grid. Left: Topology top view. The far field, the Huygens body and GCMS are clamped by quads. Right: Topology side view. Again clamps are used for the 3D topological design.

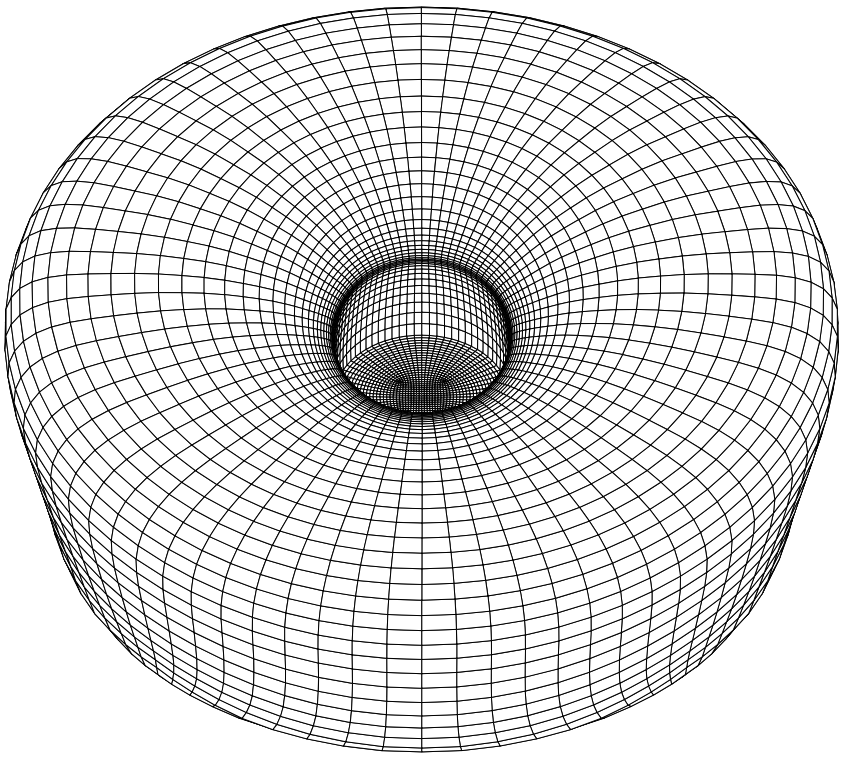

Figure 5.12: 24 block grid for Huygens space probe including SSP.

ordering of points which represent the vertices of the wireframe structure. The positions of these points must be given as spatial coordinates, and these positions are used as initial conditions for the grid generator. The accuracy of coordinate information plays an important role for the grid quality. For instance, one may generate an intersecting hyperquad with the same topology, but different initial coordinates, shown in Fig. 5.13. In Figs. 5.14 and 5.15 an example is given to demonstrate the influence of the initial wireframe coordinates on the grid quality. In the first case, there is no skewness in the configuration. Based on this initial condition, a grid without folded lines is generated, shown in Fig. 5.14. In the second case, the grid is skewed and has folded lines, shown in Fig. 5.15.

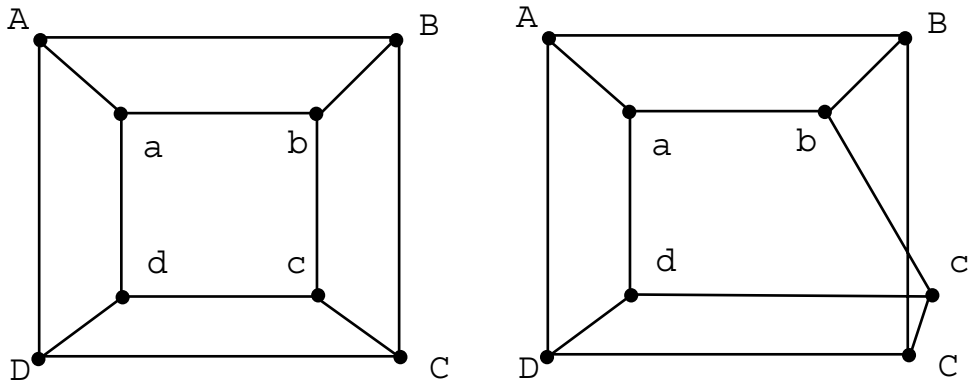


Figure 5.13: Topological description for a two–loop hyperquad is depicted. Some attention has to be given to the placement of the wireframe points. In both cases, the topology remains unchanged while wireframe corners are different. It is obvious that in the second case an overlap is encountered and grid lines will be folded.

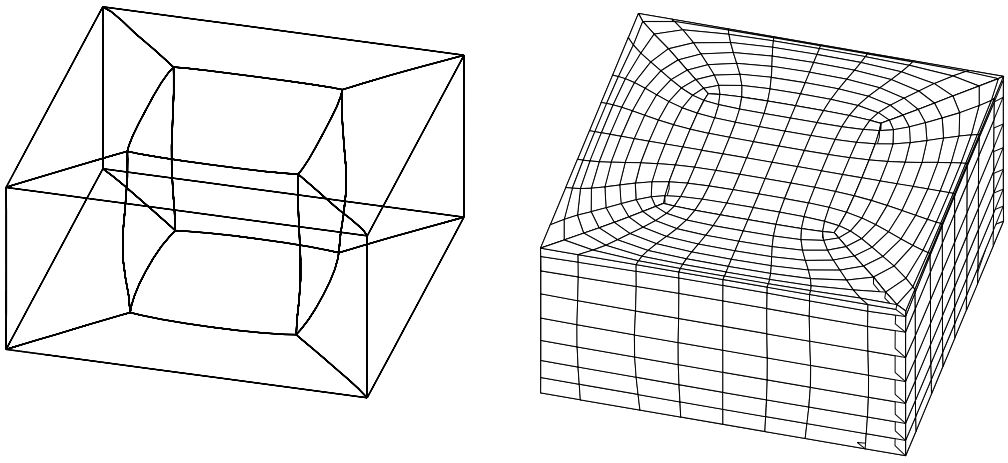

Figure 5.14: The wireframe should reflect the actual geometry. Although wireframe coordinates are not fixed, a better initial solution will lead to faster convergence and inproved grid quality.

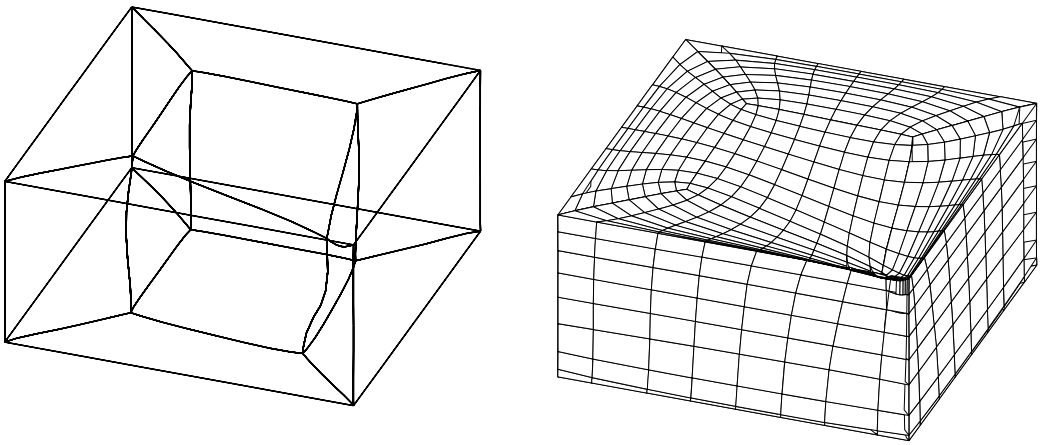

Figure 5.15: The grid quality is influenced by the initial coordinate values of the wireframe corners A distorted wireframe may cause grid skewness and folded lines.

## 6.0 *Tools for Automatic Topology Definition*

The next major step forward in automatic blocking is to provide a tool that by taking graphical input automatically produces the TIL code described in Sec. 5.0.1. **AZ–manager** that is now described, is such an interactive topology generator. In this context, topology definition means constructing a wireframe model by placing wireframe points, linking these points, and assigning them to fixed surfaces. Once the topology definition has been done, all grids are produced automatically starting GridPro from within AZ–manager and visualizing the grid while it is being computed. AZ–manager has miniCAD capabilities allowing the user to built his own geometry. This feature is particularly useful for external flows wherethe outer boundary is not of fixed shape.

The topology generation process comprises the following stages.

**Topology** The topology definition process comprises four stages:

- ▷ Generating a geometry description or using existing CAD data.
- ▷ Constructing the set of wireframe points.
- ▷ Linking corresponding wireframe points.
- ▷ Assigning wireframe points to surfaces.

In the following general rules for AZ–manager usage will be outlined, concerning mouse usage, rotating a 3D objects, and activating a surface as well as performing actions on it. It is also described how ro read in CAD data as the general means for surface description of 3D bodies. The reader should refer to this section when actively using AZ–manager, because only then the full meaning of the rules will become clear.

**Mouse Usage** All three mouse buttons are used.
**LB**: Drag to translate the object across the screen.
**MB**: Rotation (see description below).
**RM**: Drag the box that is appearing to desired size and then release mouse button. The area in the box will be enlarged.

**Rotation (Orientation of Coordinate System)** There are two different modes of rotation. On the Command Panel under topic **Rotation** two view options **world** and **eye** can be selected. Under **world** view, a rotation is about the active coordinate axis. To activate an axis, move the mouse cursor over the axis such that the rotation symbol occurs and press LB. The axis becomes a thin line, indicating that it is now the active axis, i.e. the axis of rotation. Yellow denotes the x-axis, green the y-axis, and purple the z-axis. Under **eye** view, the axis of rotation lies always in the view (screen) plane. Moving the mouse in a certain direction in the view plane, automatically and uniquely defines the axis of rotation being perpendicular to the direction of motion. It should be noted that this axis of rotation is invisible.

**Activating a Surface** Surface definition: AZ–manager has some CAD capabililty. The user can define surfaces in 2D (lines) as well as in 3D (surfaces). Surfaces are numbered consecutively,

starting from 0. Only one surface can active at a time. Only the active surface is affected by user operations.

A body or configuartion comprises a set of surfaces. The user needs to have the possibility to scroll through these surfaces and to perform certain actions on them. The active surface, on which actions are performed, is selceted by pressing ***surf*** on the command panel (the command panel is the right side of the screen, containing a multitude of commands). By using $<$ or $>$ scrolling is done and the corresponding surface is selected. Pressing the ***hand*** button, the active surface is marked by blue handles. Dragging these handles resizes the active surface.

**Setting the Display Style of Objects** There are various possibilties of displaying objects on the srceen, using the button ***stop*** of the $< STYLE >$ option on the command panel, one selects the current static display style, while button ***move*** governs the display style during the mouse movement. For the 2D examples below, one should select ***lines***. One should experiment with the options available to fully understand their meaning.

**Cutting Plane (CP)** To place wireframe points in 3D space, a plane has to be defined, termed the cutting plane. All wireframe points generated are automatically positioned on this plane. The CP can be moved by the user, i.e. translated and rotated to any desired position. Its usage is described in Example 3.

**Surface Geometry Description** In example 3 the surface of the Cassini–Huygens Space Probe is described by a set of 6 rectangular patches, given in pointwise form, that have been extracted from CAD data. Several other IGES formats are also supported. The Huygens surface data is in the file **HuygensSurf.dat**. A second file with extension *conn* is needed by GridPro, describing the connectivity of the patches. Using a conversion program, the file **HuygensSurf.dat.conn** is automatically generated. The format of this file is of no concern to the user, he just has to remember that it is needed by GridPro.

**Fit Objects on Scree** When surfaces are read in or constructed they may not fit on the screen, that is, they may not be visible, because of a distorted scaling. Press ***unzoom*** as many times as necessary to fit objects on screen. When surfaces are generated, they might have to be scaled. Turn ***hand*** on and make the surface active. The surface turns blue and then resize by dragging the surface handles.

### 6.0.1 *Example Topology 1: A Circle in a Box*

The following example constructs a 4 block grid as a 2D projection of a rod (cylinder) in a box, i.e. a doubly connected region. Gridlines wrap around the rod in an O-type fashion. However, since the outer boundary is formed by a box, special care has to be taken by the grid generator to have grid lines close to the box corners. A clustering algorithm is needed to achieve that. The user does not have to specify any parameters, the clustering is performed automatically in GridPro based on boundary curvature. The following set of pictures serve as a step by step introduction on how to actually generate the grid. Stepping through the pictures, the reader should make sure that his results are similar to those depicted in the figures. Since wireframe points are interactively placed, results will not be identical with the pictures.

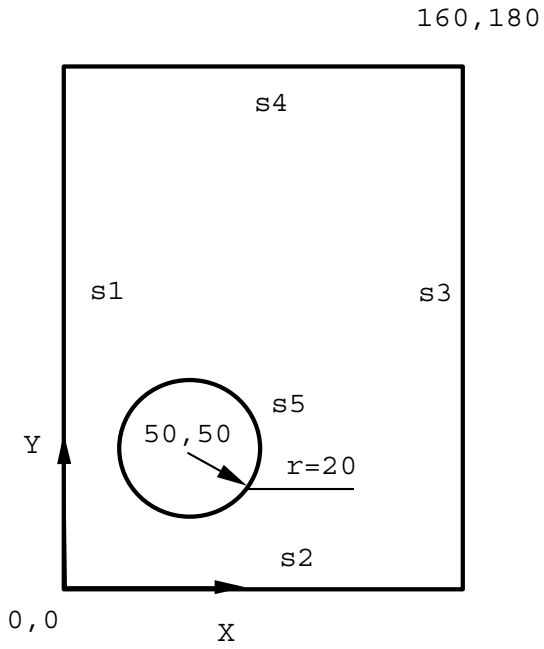


Figure 6.1: In this first example the grid for a circle in a box has to be generated. First the user has to construct the geometry (physical boundaries) as shown in the figure. Second, the blocking toplogy has to be specified.

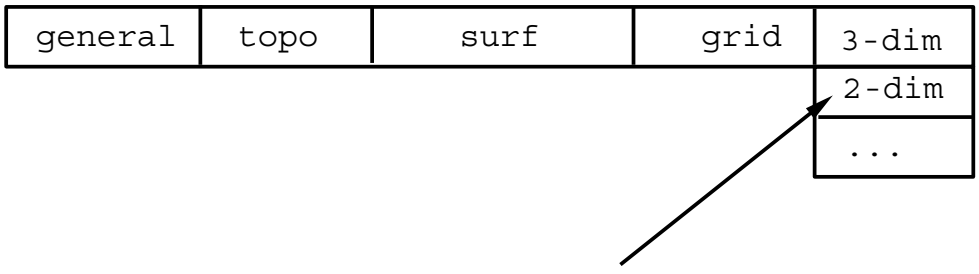


Figure 6.2: To start with the example, invoke the automatic blocking manager with command **az**. The AZ–manager window appears. Switch to **2–dim** on the menubar, as shown in figure.

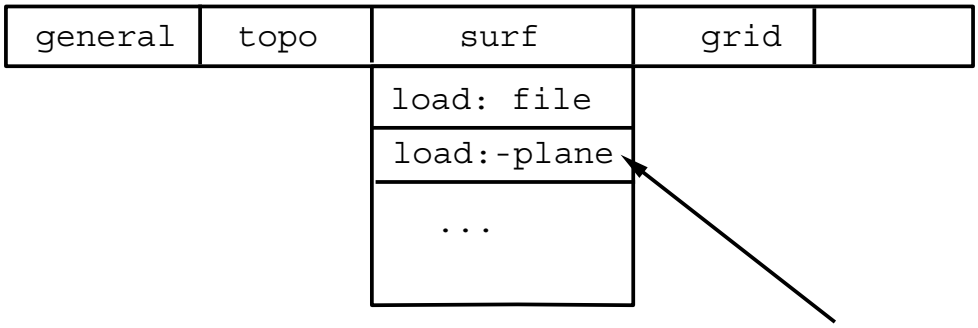


Figure 6.3: **AZ–manager** has mini CAD capability that is used to construct the boundary of the solution domain. To this end click **surf** on the menubar and select **load:–plane** from this menu.

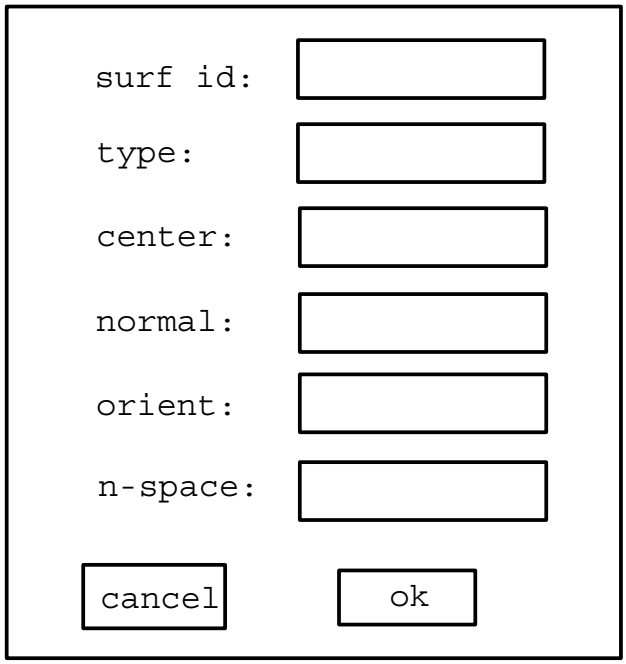


Figure 6.4: new text needed **AZ–manager** has mini CAD capability that is used to construct the boundary of the solution domain. To this end click **surf** on the menubar and select **load:–plane** from this menu.

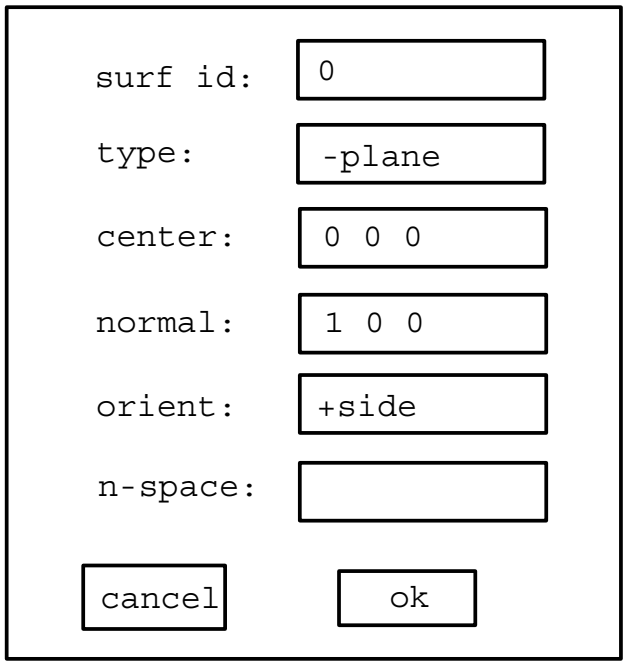


Figure 6.5: To construct the outer box, input contour data as indicated. The first surface to be generatedd, is the west side of the box. It is of type textbfplane that actually is a line in 2D.

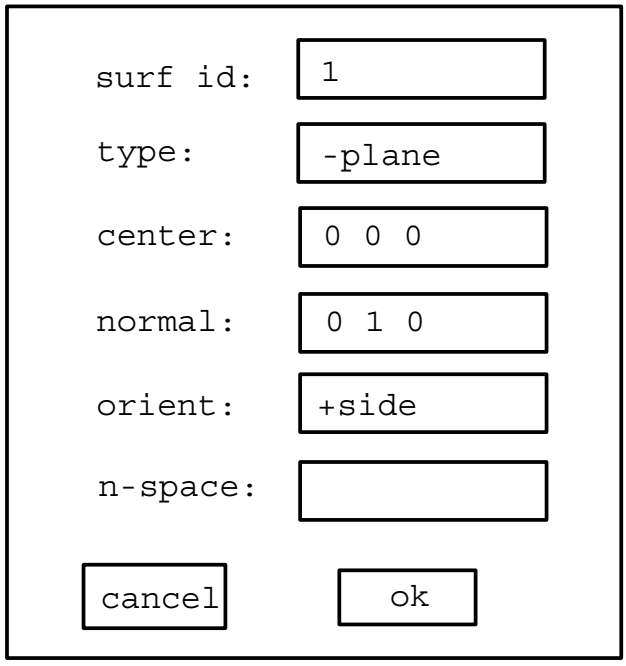


Figure 6.6: Input **plane** information as shown. The next surface is the south side of the box. Note that surfaces are consecutively numbered (surf id).

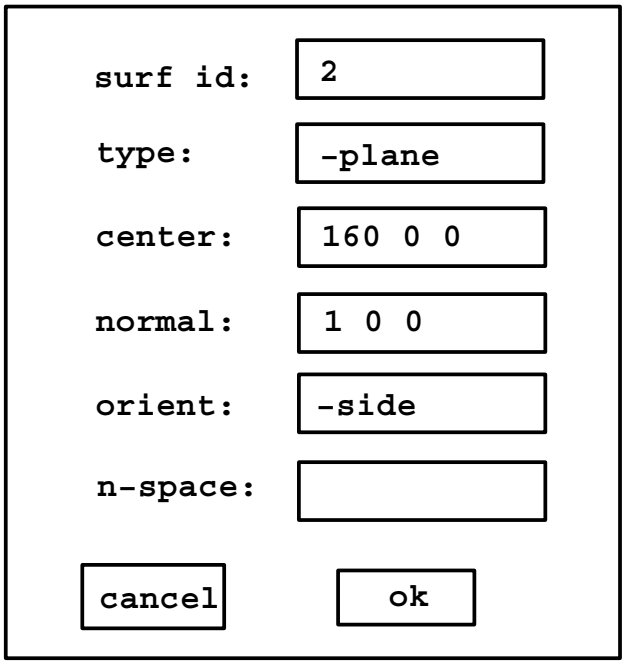


Figure 6.7: Input **plane** information as shown. This surface is the east side of the box. Note that surface orientation has to be reversed by specifying **–side**.

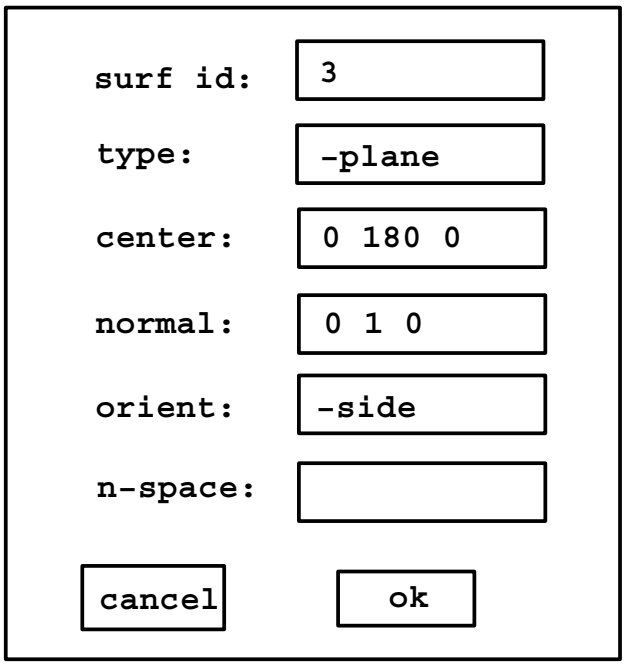


Figure 6.8: Input **plane** information as shown. This plane is the north side of the box. Note that surface orientation has to be reversed by specifying **–side**. Press **unzoom** to make a box fit on screen.

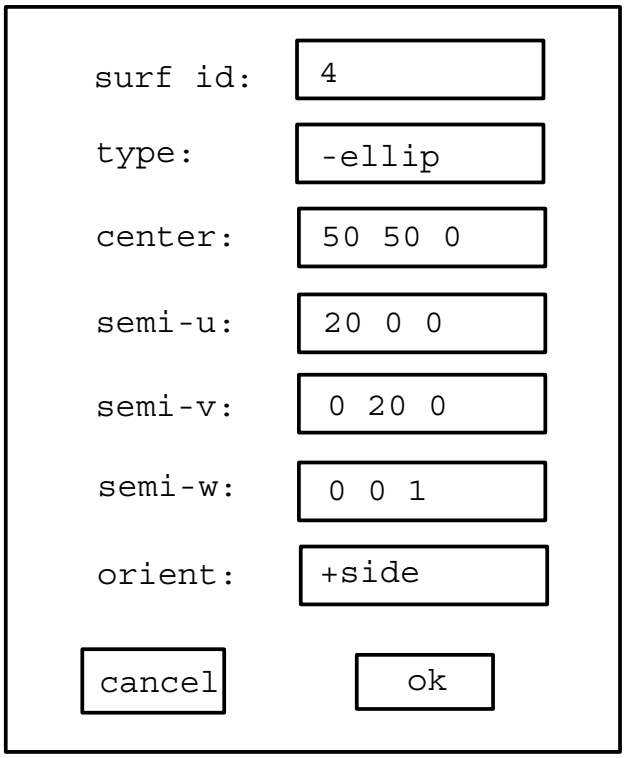


Figure 6.9: Select **load:–ellip** from the **surf** on the menubar. Input **ellip** information as shown to construct the circle.

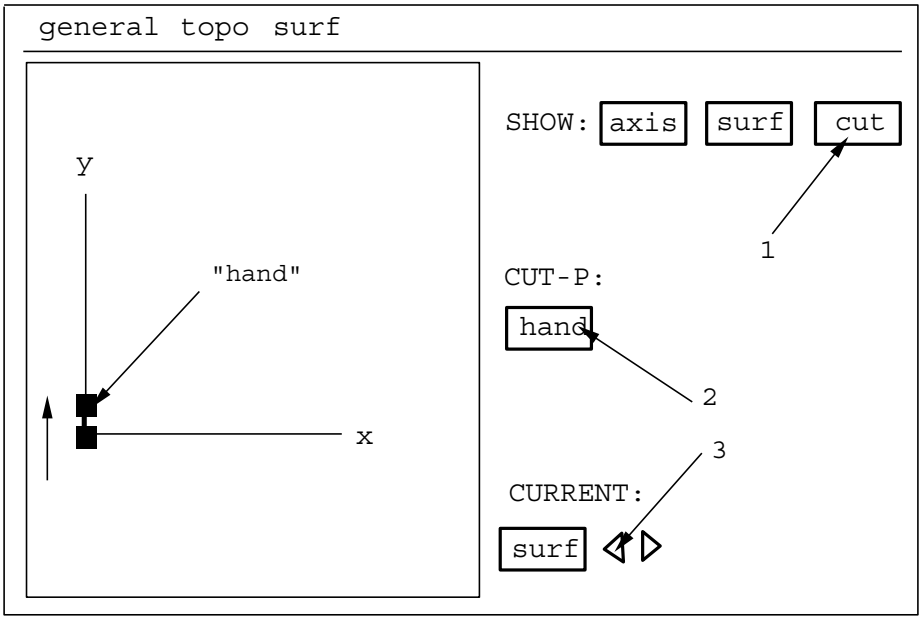


Figure 6.10: A surface may not have the desired size. In order to resize it, deselect **cut** (if it is on, i.e. the radio button is pressed) of the **SHOW** option on the Command Panel. Then activate **hand** from the **CUT–P** option on the command panel. Next, activate the proper surface (blue color) – in this example the proper edge of the box. Position the mouse curser over the **handle** and drag it in the indicated direction.

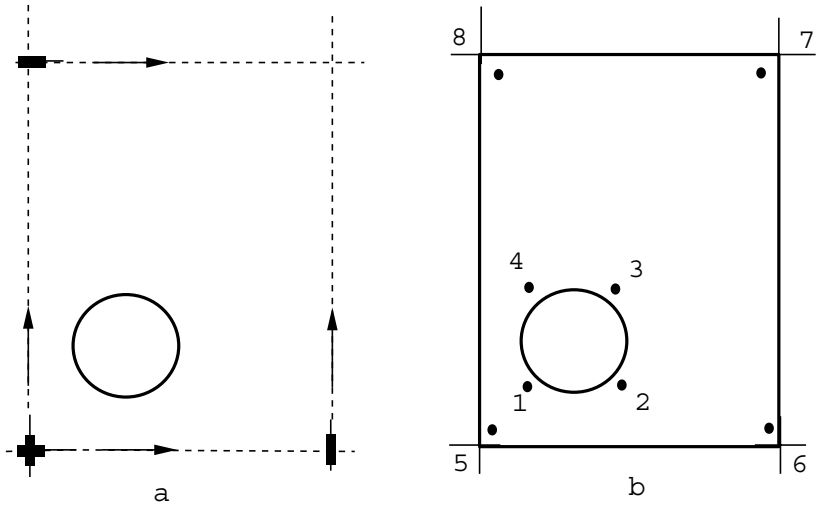


Figure 6.11: Activate the other box surfaces in the same way and drag them as shown in figure a. The dragging shoul be done such that the box contour becomes visible. The circle should be visible as well. Now wireframe points can be placed. Wireframe point don't have to ly on a surface, but they should be close to the surface to which they are going to be assigned. Wireframe points are created by pressing "c" on the keyboard and clicking at the respective positions with LB. Make sure that < **Corner Creation Mode** appears in the upper left corner of the window. Finish point positioning as depicted in figure b.

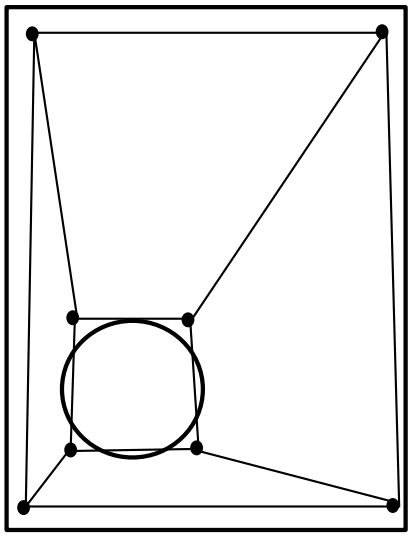

Figure 6.12: Link the corners by pressing ”e” on the keyboard and click each pair of points with LB. Make sure that < **Link Creation Mode** appears in the upper left corner of the window. After completing the linkage process, a wireframe model should like as depicted in the figure.

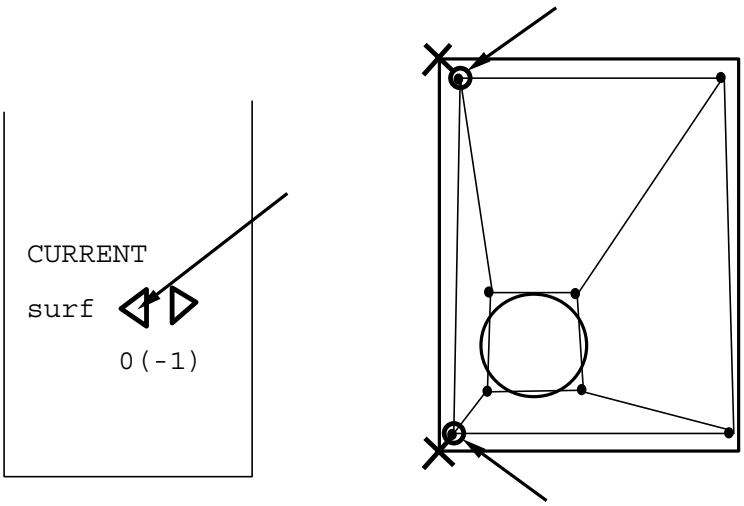


Figure 6.13: Activate surface 0. Press ”s” on the keyboard and click the two points, marked by ”X”. If points are selected, their color turns from yellow to white. That is the corners are assigned to surface 0.

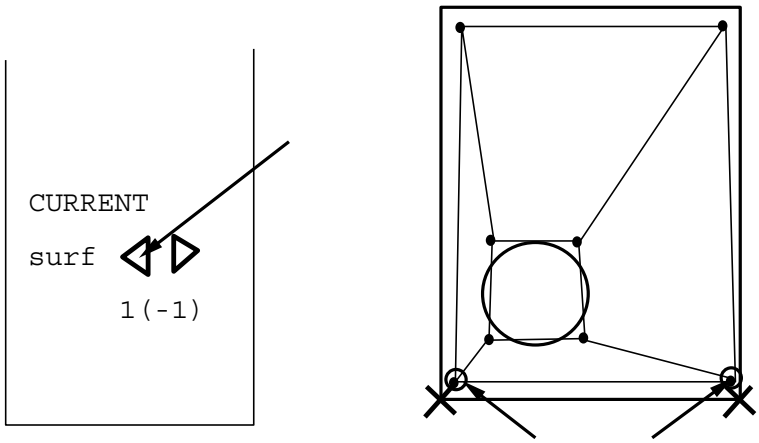


Figure 6.14: Repeat this action for surface 1, shown in figure.

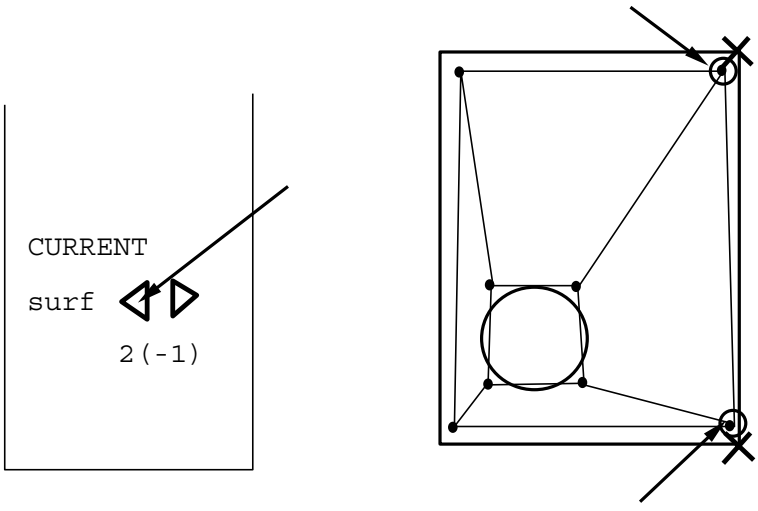


Figure 6.15: Repeat this action for surface 2, shown in figure.

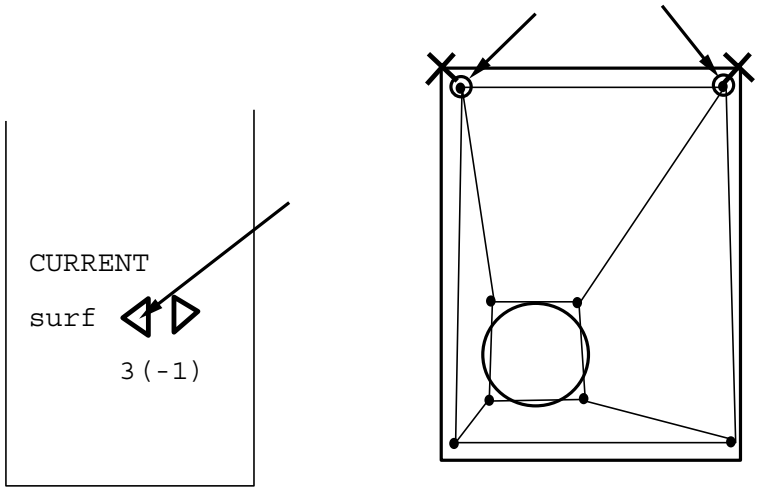


Figure 6.16: Repeat this action for surface 3, shown in figure.

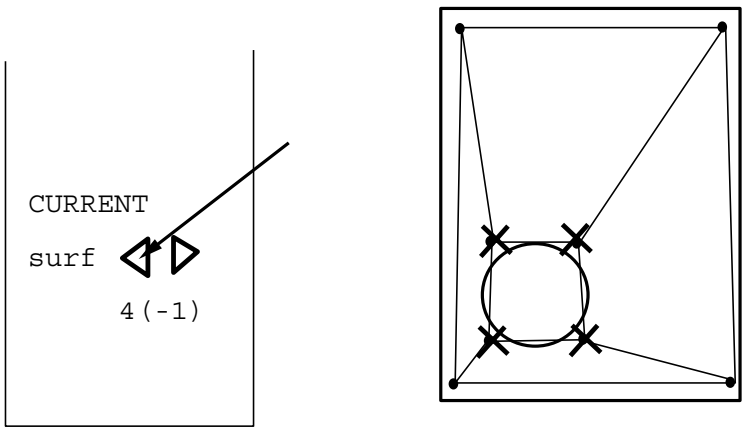


Figure 6.17: Activate surface 4 and assign the remaining 4 points to the circle.

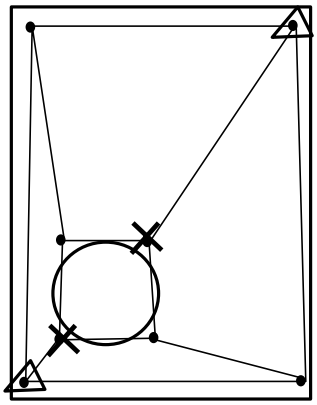

Figure 6.18: Since GridPro is 3D internally, it is necessary to explicitly remove two blocks. The first block to be revoned is formed by the 4 points which build diagnoals, as shown in figure. Press ”f” and click two points, marked by triangles. Perform the same action for points marked by crosses. A red line between the corresponding diagnoals indicates the succesful performance.

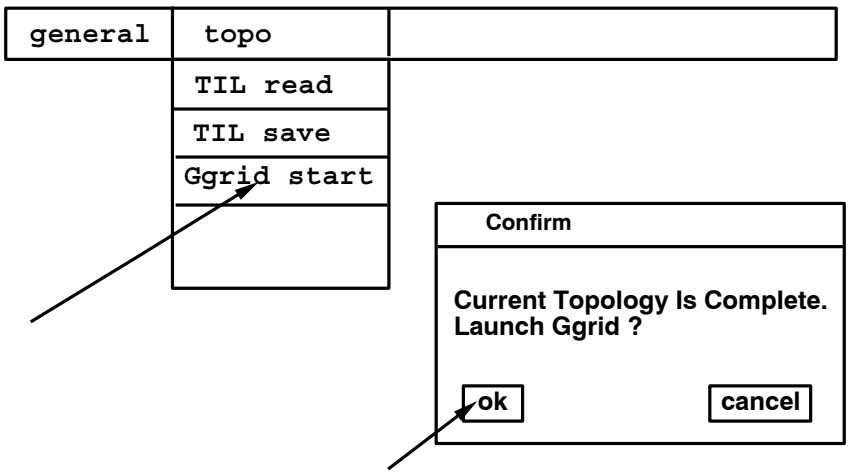


Figure 6.19: Save the topology file (TIL code) by selecting **topo** and clicking **TIL save**. Select **Ggrid start** to generate the grid.

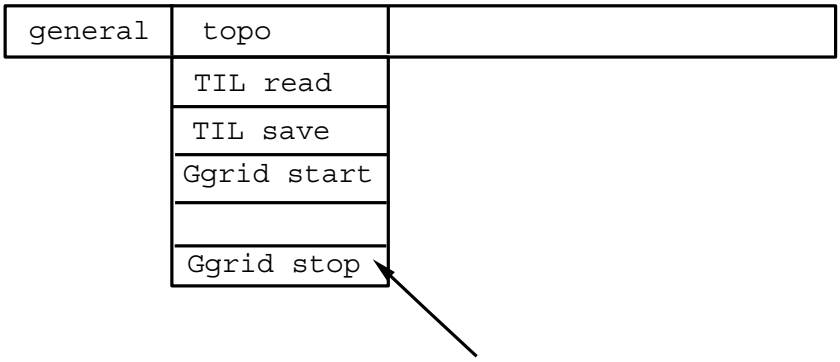


Figure 6.20: Select ***topo*** from the menubar and click ***Ggrid stop***. The generation process is now suspended. Select ***grid*** from the menubar and click ***load new***, the grid file ”blk.tmp” can be read.

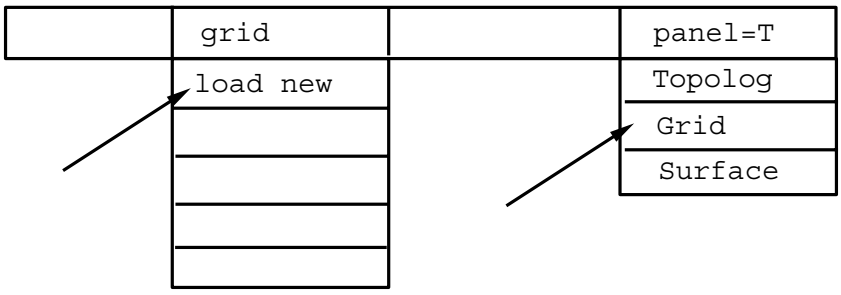


Figure 6.21: Select ***panel=T*** from the menubar and click ***Grid***. The grid appears as wireframe model. At the right side of the menu window, select ***STYLE*** and click ***lines*** under ***stop***. Click ***shell*** by ***Make Shell*** at the right low side of the menu window. The grid appears.

### 6.0.2 *Example Topology 2: Two Circles in a Box*

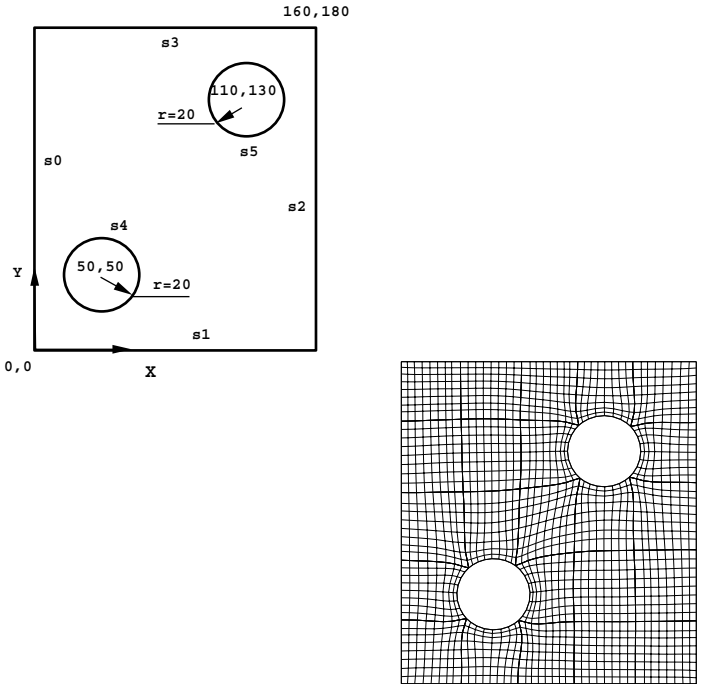


Figure 6.22: Constrct geometry in the same way as described in example 1.

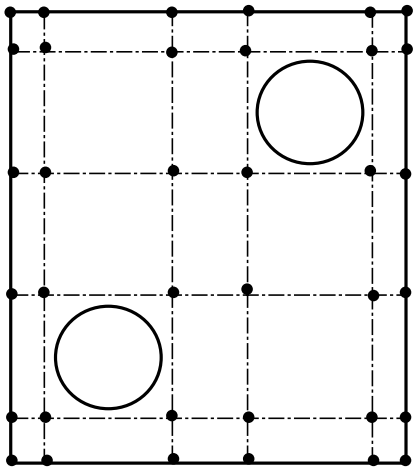

Figure 6.23: It is suggested to place the wirefrace points in a row by row fashion. Other toplogies, of course, would be possible.

```
SET DIMENSION 2
SET GRIDEN 8

COMPONENT DoubleCircle ()
BEGIN

s 1 -plane (1 0 0 0);
s 2 -plane (0 1 0 0);
s 3 -plane (-1 0 0 160);
s 4 -plane (0 -1 0 180);
s 5 -ellip (0.05 0.05 0) -t 50 50 0;
s 6 -ellip (0.05 0.05 0) -t 110 130 0;

END
```

Table 6.1: It is als possible to read the geometry in a TIL file. This TIL file only contains the geometry of the solution domain. In practice, CAD data will be available that will be used for geometry description.

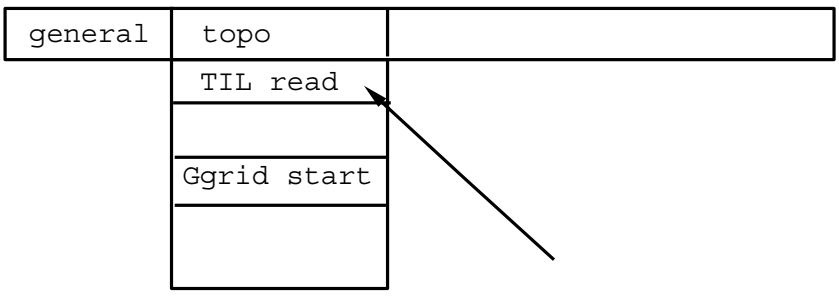


Figure 6.24: Press ***TIL read*** under item ***topo*** of the menubar to read in the geometry data.

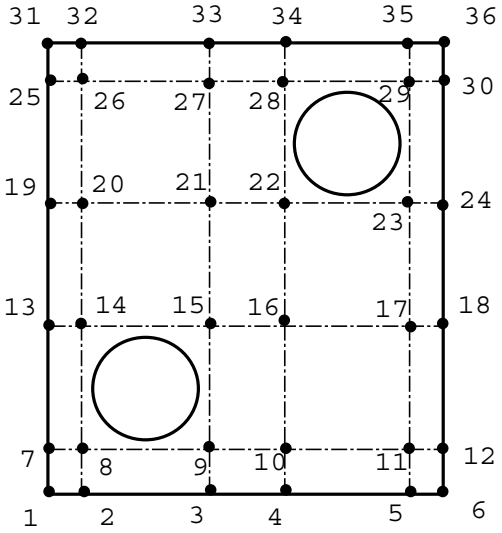


Figure 6.25: A row by row pattern results in a wireframe point numbering as shown.

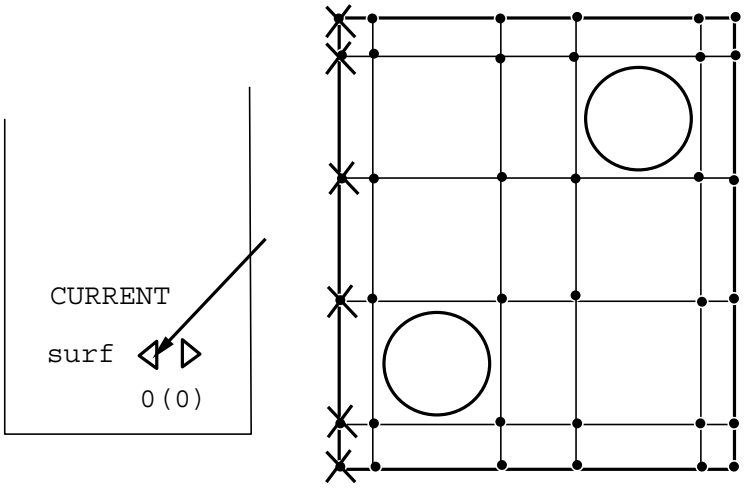


Figure 6.26: Assignment of wireframe points to surfaces is exactly as in the first example.

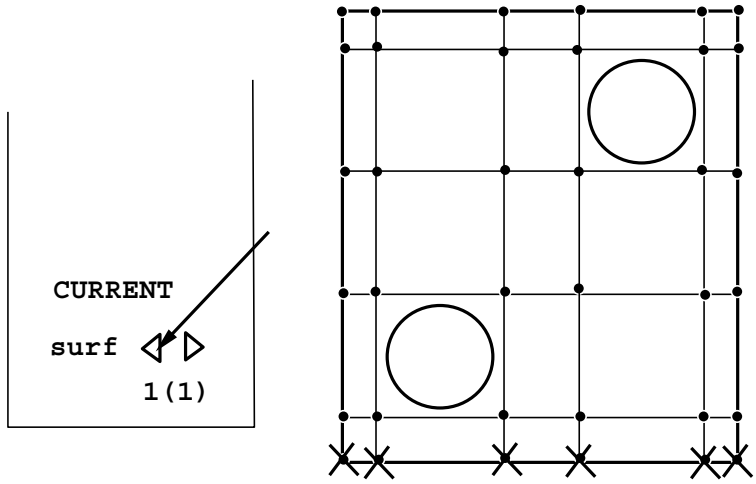


Figure 6.27: Activate surface 1 and assign wireframe points.

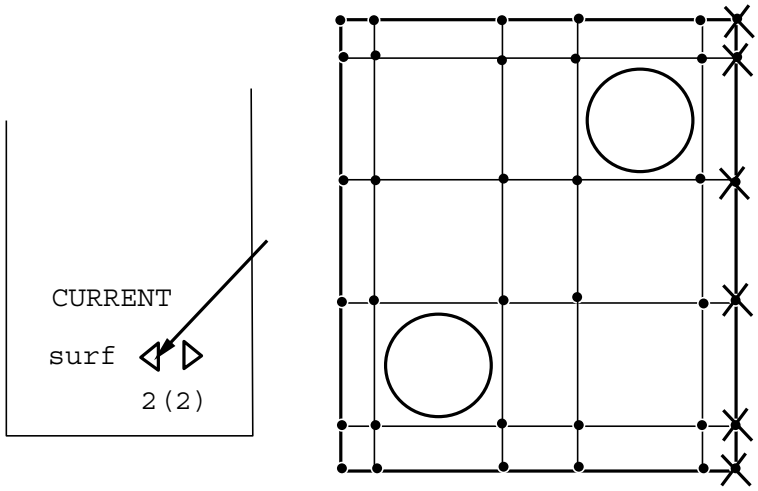


Figure 6.28: Activate surface 2 and assign wireframe points.

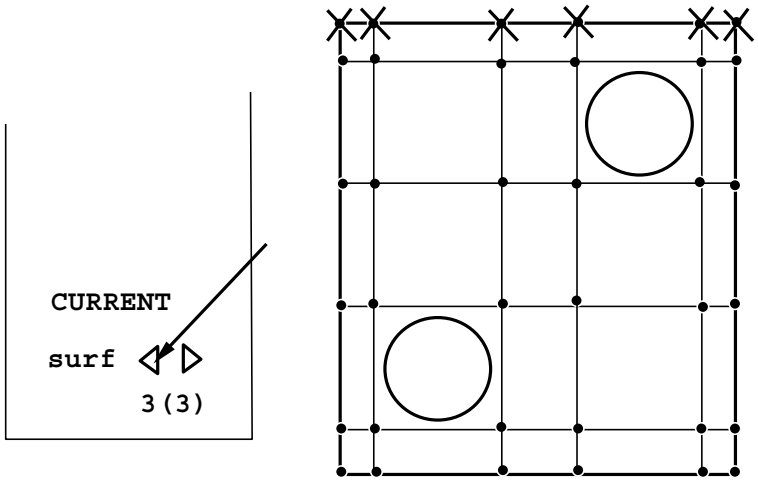


Figure 6.29: Activate surface 3 and assign wireframe points.

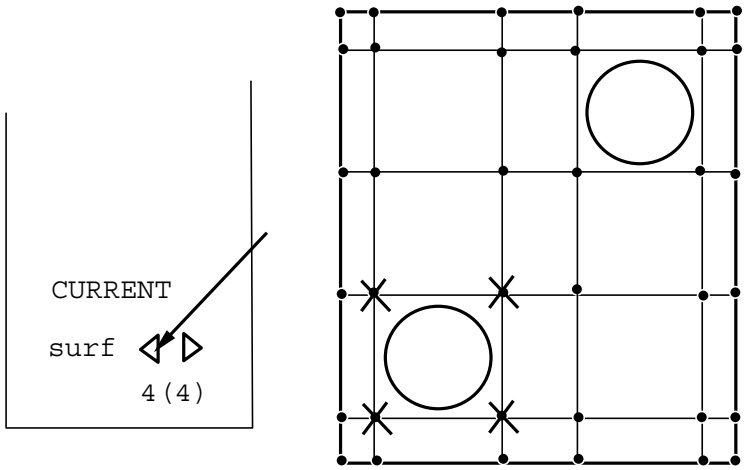


Figure 6.30: Activate surface 4 and assign wireframe points.

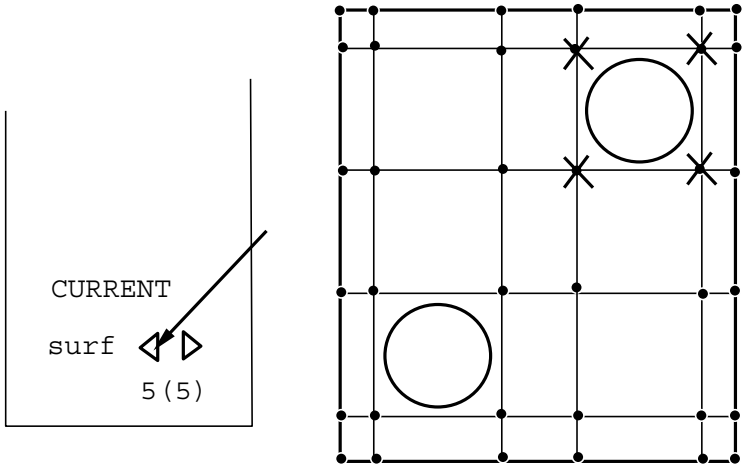


Figure 6.31: Activate surface 5 and assign wireframe points.

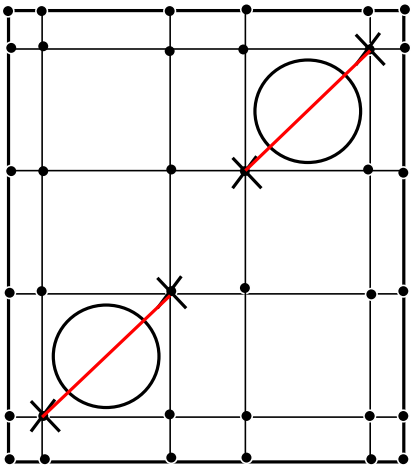

Figure 6.32: Activate surface 6 and assign wireframe points.

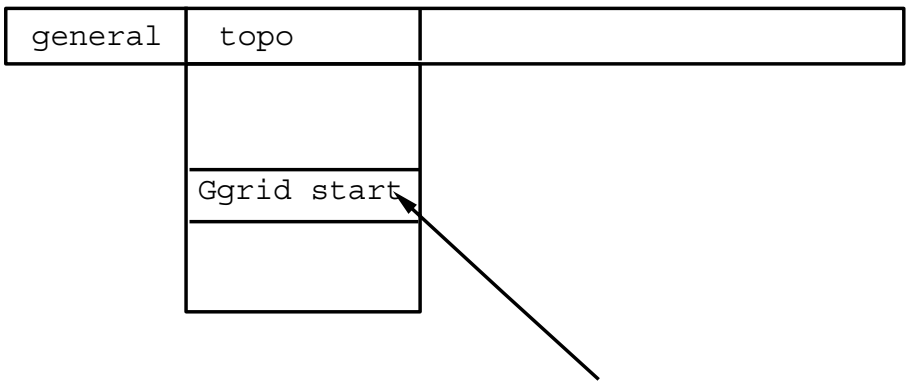


Figure 6.33: Start **Ggrid** by selecting **Ggrid start**.

```
SET DIMENSION 2
SET GRIDEN 10

COMPONENT Huygens ()
BEGIN

s 1 -linear "HuygensSurf.dat";

END
```

Table 6.2: This TIL code is read in using **topo TIL read** as in former examples). The file HuygensSurf.dat contains the Huygens surface data as described in **??**. The Huygens surface is displaced on the srceen and can be rotated as described in **??**. Select **surf** and click **load:–ellip**. Construct a circle with radius 5000.

### 6.0.3 *Example Topology 3: Cassini–Huygens Space Probe*

In this example we will do our first 3D problem and, at the same time, will use surface description data coming from a CAD system (see above).

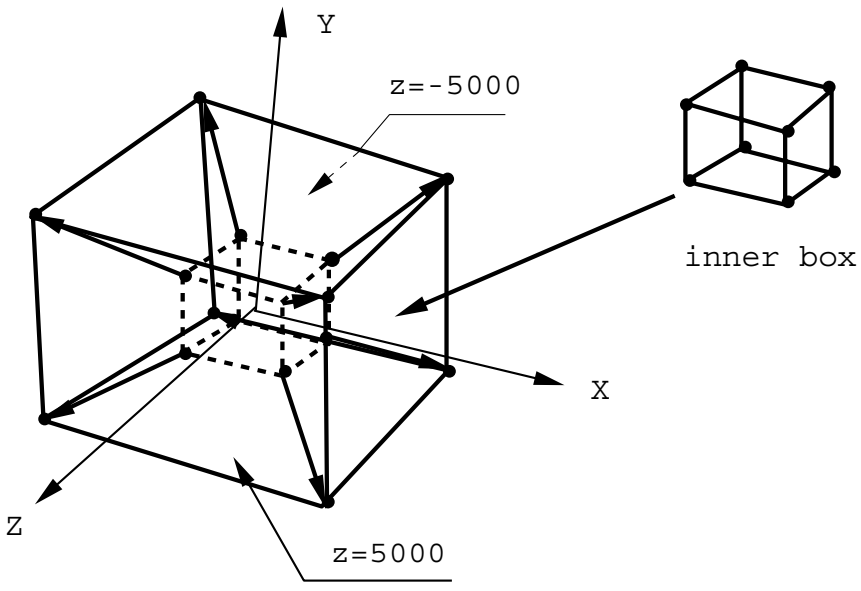


Figure 6.34: The topology to be generated for Huygens is of type box in a box. The outer 8 wireframe points should be placed on a sphere with radius 5000, the inner 8 points are located on the Huygens surface.

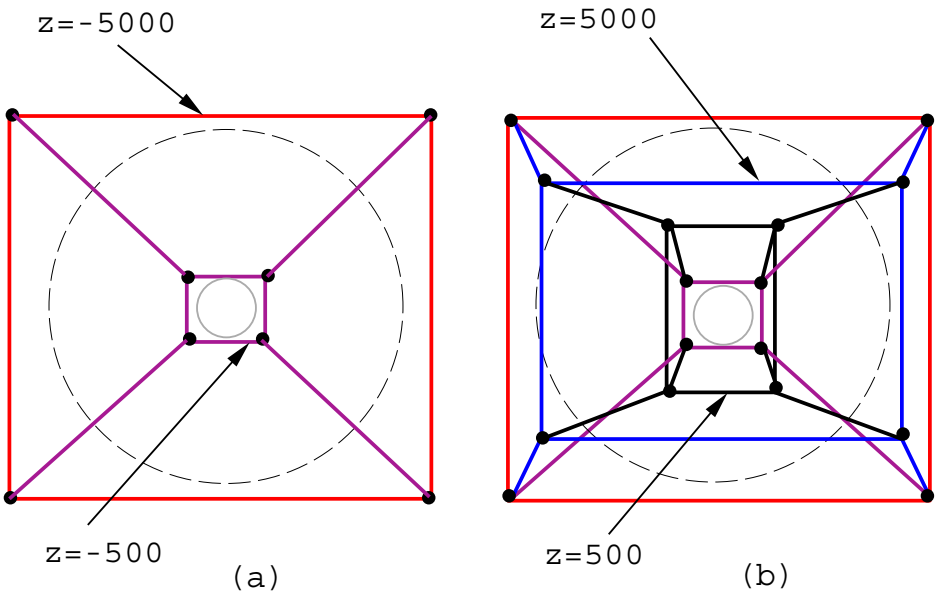


Figure 6.35: Construct wireframe model as in figure.

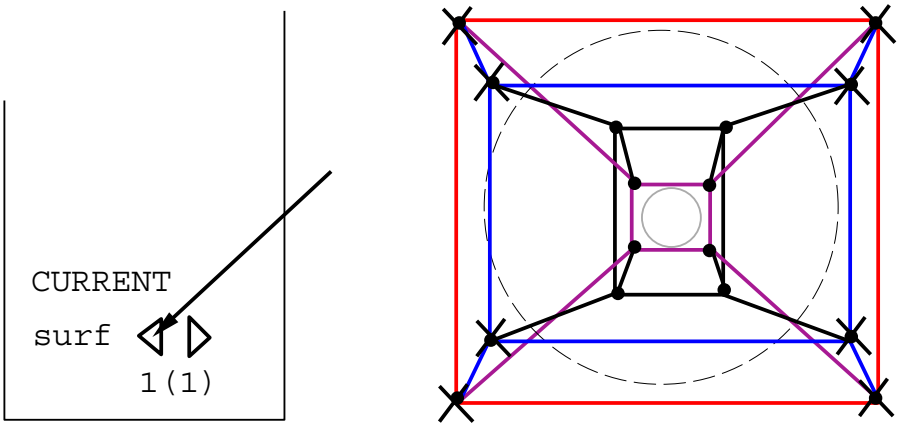


Figure 6.36: Assign surface 2 as in figure.

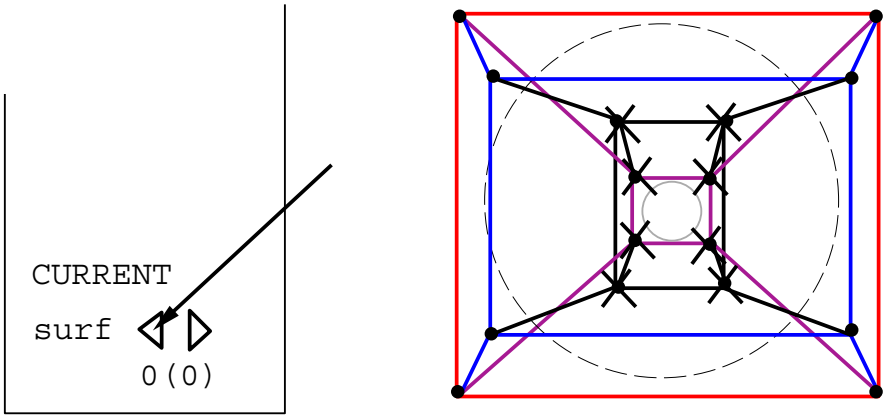


Figure 6.37: Assign surface 1 as in figure.

# 7.0 Parallelization Strategy for Complex Geometries

## 7.0.1 The Abstract Parallel Machine

The following philosphy is basic to the parallelizing approach of multiblock codes. In writing cost effective parallel software whose quality can be ensured, one has to separate the methods of the application from the details of implementation on a particular parallel architecture, with its particular processors and perhaps a vendor-specific parallel file system. This can be achieved by writing code for an abstract machine, then implementing the abstract machine on specific hardware platforms.

The abstract machine has a communication system with these methods:

- ▷ Sending locally blocking tagged messages,
- ▷ Blocking receives of messages chosen by tag or by source processor,
- ▷ A system message buffer of given fixed size,
- ▷ A collective operation, called ”reduce” in MPI for sums over all the processors, and other such operations.

In addition, if needed by a CFD code, the abstract machine has a processor with these BLAS1 vector methods:

- ▷ Adding vectors and multiplying by a scalar (saxpy/daxpy operations).
- ▷ Gather/Scatter operations on index vectors.

The abstract machine has the following file system methods:

- ▷ Collective synchronized Open of a file,
- ▷ Multiple atomic seek-and-read operations with a shared file pointer,
- ▷ Multiple write operations with a single shared file pointer.

Some parallel processing systems support these operations very efficiently. For example the vector operations on a Cray or IBM RS/6000 processor, also implemented as pipeline operations on RISC processors such as the i860. When a processor does not directly support one of the methods, it can be implemented in software.

The abstract machine specified above is well suited for unstructured or structured mesh computations, being portable to any software or hardware environment, yet taking advantage of the specific machine configuration to maximum advantage.

### 7.0.2 *General Strategy*

Parallelization is achieved by domain decomposition. Do-loop level parallelization will neither be economical nor effective.

Since for structured grids the grid topology (that is the number and size of blocks needed to obtain a certain gridline configuration) is dictated by the physics, load balancing is achieved by mapping any arbitrary number of blocks to a single processor. This demands a slightly more sophisticated message passing scheme, but leads to a much more flexible approach. One might surmise that for unstructured grids a domain decomposition algorithm will result in a load balanced application, making the mapping of several subdomains (blocks) to a single processor unnecessary. However, this may not be the case for very large grids, because a lenghty optimization stage might be required, for instance, to use simulated annealing to minimize the cost function.

The decompositions will be performed statically via pre-processing of the grids rather than dynamically during the flow solutions. This is adequate for the current problems However, future developments may include unsteady calculations on moving and/or solution adaptive grids and thus any modifications made will keep these future developments in mind while not allowing them to dominate the initial parallelization effort.

In order to minimize code maintenance only a parallel version of the code is kept. A single processor machine is simply considered as a one processor parallel machine.

All codes are implemented under PVM or MPI, but the communication libraries of Intel and Ncube are directly supported as well as the IBM parallel environment.

### 7.0.3 *Encapsulation of Message Passing*

Message passing routines are called indirectly, using a set of encapsulated routines that in turn call PVM routines.

Efficient coding: In parallel programming a number of operations, such as global reduction operations (”global sum”) are required quite frequently. Since PVM and MPI lack this type of operations they are realized by the application itself. These pieces of code are assembled in a communication library.

The parallelization of I/O can be very different with respect to the programming models (SPMD, host-node - not recommended) and I/O modes (host-only I/O, node local I/O, fast parallel I/O hardware, etc.) supported by the parallel machines. The differences can also be hidden in the interface library.

Portability: Encapsulation of message passing routines helps to reduce the effort of porting a parallel application to different message passing environments.

Source code: Encapsulation allows to keep only one source code both for sequential and par-

allel machines.

Maintenance and further development: Encapsulation keeps message passing routines local. Thus, software maintenance and further development will be facilitated.

Common message passing subset: The table below lists the basic message passing operations which are available in most of the message passing environments. These routines are functionally equivalent although the semantics of the arguments and return valves may differ in detail. Portability can be highly increased by restricting oneself to use only operations included in the common subset for implementing the interface routines.

- ▷ Since each processor of the parallel machine takes one or more blocks, there may not be enough blocks to run the problem on a massively parallel machine. We have tools to automatically split the blocks to allow the utilization of more processors.
- ▷ The blocks may have very different sizes, so that the blocks must be distributed to the processors to produce a reasonable load balance. We have tools to solve this bin-packing problem by an algorithm discussed below.

An extremely simple message-passing model is ismplemented, consisting of only *send* and *receive*. The simplicity of this model implies easy portability.

For an elementary Laplace solver on a square grid, each gridpoint takes the average of its four neighbors, requiring 5 flops, and communicates 1 floating-point number for each gridpoint on the boundary. The **G**rid⋆grid generator is a more sophisticted elliptic solver, where about 75 flops occur per internal grid point, while grid coordinates are exchanged across boundaries. Ourflow solver,**ParNSS**, in contrast does a great deal of calculation per grid point, while the amount of communication is still rather smalli.

Thus we may expect any implicit flow solverto be highly efficient in terms of communication, as shown by the timing results below. When the complexity of the physics increases, with turbulence models and chemistry, we expect the efficiency to get even better. This is why a flow solveris a viable parallel program even when running on a workstation cluster with slow communication (Ethernet).

### Parallel Machines and CFD

For the kinds of applications that we are considering in this report, we have identified four major issues concerning parallelism, whether on workstation clusters or massively parallel machines.

**Load balancing** As discussed above, the number of blocks in the grid must be equal to or larger than the number of processors. We wish to distribute the blocks to processors to produce an almost equal number of gridpoints per processor; this is equivalent to minimizing the maximum number of gridpoints per processor. We have used the following simple algorithm to achieve this.

The largest unassigned block is assigned to the processor with the smallest total number of gridpoints already assigned to it, then the next largest block, and so on until all blocks have been assigned.

Given the distribution of blocks to processors, there is a maximum achievable parallel efficiency, since the calculation proceeds at the pace of the slowest processor, i.e. the one with the maximum number of gridpoints to process. This peak efficiency is the ratio of the average to the maximum of the number of gridpoints per processor, which directly proceeds from the standard definition of parallel efficiency.

**Convergence** For convergence acceleration a block-implicit solution scheme is used, so that with a monoblock grid, the solution process is completely implicit, and when blocks are small, distant points become decoupled. Increasing the number of processors means that the number of blocks must increase, which in turn may affect the convergence properties of the solver. It should be noted [**?**] that any physical fluid has a finite information propagation speed, so that a fully implicit scheme may be neither necessary nor desirable.

**Performance** It is important to establish the maximum achievable performance of the code on the current generation of supercomputers. Work is in progress with the Intel Paragon machine and a Cray YMP.

**Scalability** Massively parallel processing is only useful for large problems. For a flow solver, we wish to determine how many processors may be effectively utilized for a given problem size – since we may not always run problems of 10 000 000 gridpoints.

### 7.0.4 *Aspects of Software Engineering: C versus Fortran*

Only a few remarks shall serve to highlight the discussion. Fortran has been the programming language of science and engineering since its inception in the late fifties. It has been updated in 1977 and recently new versions called Fortran90 (or HPF) have been devised. On the other hand, ANSI-C is a quasi de-facto standard that is available on virtually any hardware platform. In the following, the most important features of a modern high-level language are listed, mandatory to write portable, safe, robust, efficient, and reusable code that can also be maintained.

- ▷ Special Parallel Languages not needed for CFD
- ▷ Desirable features of a high level language for CFD:
  - wide availability and standardized
  - portability (sequential and parallel machines)
  - rapid prototyping (high productivity)
  - compact code

  - versatile loop and conditional commands
  - dynamic storage allocation
  - recursion

- ▷ Advanced features
  - pointers
  - user defined data types (structures)
  - classes (information hiding)
  - conditional compilation (implementation of drivers for different parallel machines)

The conclusion drawn is the following:

- ▷ F77: does not possess modern constructs
- ▷ HP Fortran and F90: some modern features available
- ▷ C: all features available, except classes ; dynamic storage allocation limited
- ▷ C++: all features available Recommendation : C++ language of choice for the 90's, but not all of its advanced features are needed for CFD

### 7.0.5 *Numerical Solution Strategy: A Tangled Web*

It is important to note that the succesful solution of the parallel flow equations can only be performed by a **Triad** numerical solution procedure. Numerical **Triad** is the concept of using **grid generation, domain decomposition**, and the **numerical solution scheme** itself. Each of the three **Triad** elements has its own unique contribution in the numerical solution process. However, in the past, these topics were considered mainly separately and their close interrelationship has not been fully recognized. In fact, it is not clear which of the three topics will have the major contribution to the accurate and efficient solution of the flow equations. While it is generally accepted that grid quality has an influence on the overall accuracy of the solution, the solution dynamic adaptation process leads to an intimate coupling of numerical scheme and adaptation process, i.e. the solution scheme is modified by this coupling as well as the grid generation process. When domain decomposition is used, it may produce a large number of independent blocks (or subdomains). Within each subdomain a block-implicit solution technique is used, leading to a decoupling of grid points. Each domain can be considered to be completely independent of its neighboring domains, **parallelism** simply being achieved by introducing **a new boundary condition**, denoted as inter-block or inter-domain boundary condition. Updating these boundary points is done by message–passing. It should be noted that exactly the same approach is used when the code is run in serial mode, except that no messages have to be sent to other processors. Instead, the updating is performed by simply linking the receive

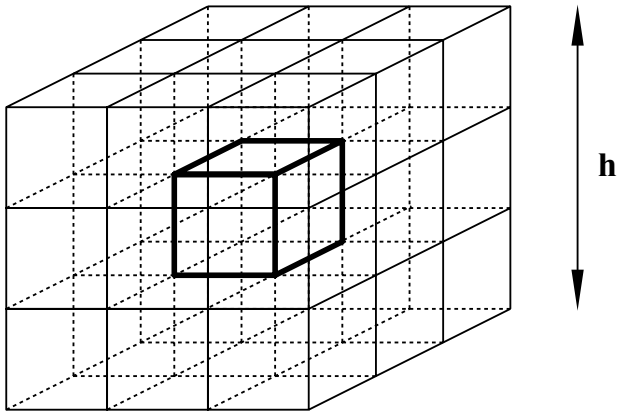


Figure 7.1: Flow variables are needed along the diagonals to compute mixed second derivatives for viscous terms. A total of 26 messages would be needed to update values along diagonals. This would lead to an unacceptable large number of messages. Instead, only block faces are updated (maximal 6 messages) and values along diagonals are approximated by a finite difference stencil.

buffer of a block to its corresponding neighboring send buffer. Hence, **parallelizing a multi-block code neither demands rewriting the code nor changing its structure**.

A major question arises in how the decomposition process affects the convergence rate of the implicit scheme. First, it should be noted that the N-S equations are not elliptic, unless the time derivative is omitted and inertia terms are neglected (Stokes equations). This only occurs in the boundary layer when a steady state has been reached or has almost been reached. However, in this case the Newton method will converge quadratically, since the initial solution is close to the final solution. The update process via boundaries therefore should be sufficient. In all other cases, the N-S equations can be considered hyperbolic. Hence, a full coupling of all points in the solution domain would be unphysical, because of the finite propagation speed, and is therefore not desirable and not needed. To retain the numerical order across block (domain) boundaries, an overlap of two points in each coordinate direction has been implemented. This guarantees that the numerical solution is independent of the block topology. The only restriction comes from the computation of flow variables along the diagonals on a face of a block (see Fig. 7.1), needed to compute the mixed derivatives the viscous terms.

It would be uneconomical to send these diagonal values by message passing. Imagine a set of 27 cubes with edge length $h/3$ assembled into a large cube of edge length $h$. The small cube in the middle is surrounded by 26 blocks that share a face, an edge, or a point with it. Thus, 26 messages would have to be sent (instead of 6 for updating the faces) to fully update the boundaries of this block. Instead, the missing information is constructed by finite difference formulas that have the same order of truncation error, but may have larger error coefficients.

To continue the discussion of convergence speed it should be remembered that for steady state computations implicit techniques converge faster than fully explicit schemes.

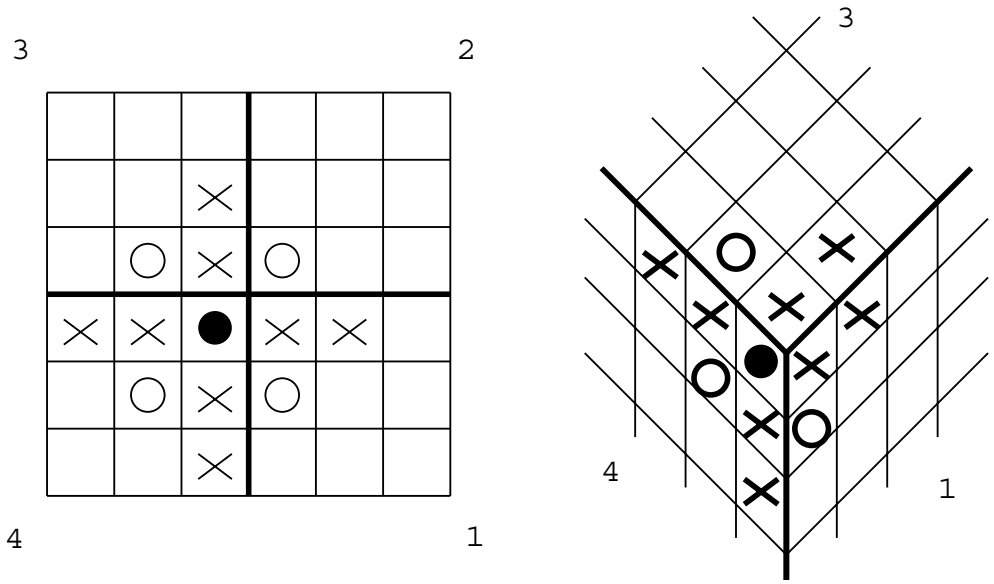


Figure 7.2: The figure shows the computational stencil. Points marked by a cross are used for inviscid flux computation. Diagonal points (circles) are needed to compute the mixed derivatives in the viscous fluxes. Care has to be taken when a face vanishes and 3 lines intersect.

The former are generally more computationally efficient, in particular for meshes with large variations in grid spacing. However, since a full coupling is not required by the physics, decomposing the solution domain should result in a convergence speed up, since the inversion of a set of small matrices is faster than the inversion of the single large matrix, although boundary values are dynamically updated. On the other hand, if the decomposition leads to a blocksize of 1 point per block, the scheme is fully explicit and hence computationally less efficient than the fully implicit scheme. Therefore, an optimal decomposition topology must exist that most likely depends on the flow physics and the type of implict solution process. So far, no theory has been developed.

However, a number of numerical experiments has been performed with the **ParNSS** code [**?**], clearly demonstrating the convergence speed up. Block numbers have been varied from 5 to 140, using an otherwise identical grid.

Second, domain decomposition may have a direct influence on the convergence speed of the numerical scheme. In this paper, the basis of the numerical solution technique is the Newton method, combined with a Conjugate-Gradient technique for convergence acceleration within a Newton-Iteration. In the preconditioning process used for the Conjugate-Gradient technique, domain decomposition may be used to decrease the condition number (ratio of largest to smallest eigenvalues) of the matrix forming the left hand side, derived from the discretized N-S equations. In other words, the eigenvalue spectrum may be compressed, because the resulting matrices are smaller. It is shown in [**?**] that this ratio is a measure of the convergence speed for generalized conjugate residual algorithms. Having smaller matrices the condition number should not increase; using physical reasoning it is concluded that in general the condition number should decrease.

From these remarks, it should be evident that only a combination of grid generation scheme, numerical solution procedure, and domain decomposition approach will result in an effective, general numerical solution strategy for the parallel N-S equations on complex geometries. Because of their mutual interaction these approaches must not be separated. Thus, the concept of numerical solution procedure is much more general than devising a single numerical scheme for discretizing the N-S equations. ***Only the implementation of this interconnectedness in a parallel solver will lead to the optimal design tool.***

# Bibliography


[1] Winslow,J.A.,1966: Numerical Solutions of the quasi-linear Poisson Equation in a Nonuniform Triangular Mesh, J.Computational Phys.2,149.

[2] Thompson, J. F., Thames, F. C., and Mastin, C. W., 1974: Automatic numerical generation of body fitted curvilinear coordinate systems for fields containing any number of arbitrary two-dimensional bodies, J. of Computational Physics, 15, 299.

[3] Häuser, J., Taylor, C., (eds.) 1986: Numerical Grid Generation in Computational Fluid Dynamics, Pineridge Press,793pp.

[4] Dannehoffer, J. F. III, 1991: A Block-Structuring Technique for General Geometries, AIAA-91-0145, 14 pp.

[5] Visich, M. et al., 1991, Advanced Interactive Grid Generation Using Rambo-4G, AIAA-91-0799, 5 pp.

[6] Takanashi, S., 1990: Large Scale Numerical Aerodynamic Simulations for Complete Aircraft Configurations, NAL TR-1073T, ISSN 0389-4010, CH6FU, Tokyo, Japan, 12 pp.

[7] Gentzsch, W., Häuser, J., 1988: Mesh Generation on Parallel Computers in Sengupta, S. et al. (Eds.), Numerical Grid Generation in Computational Fluid Dynamics, Pineridge Press, pp. 113-124.

[8] Häuser, J. et al. 1992: Parallel Computing in Aerospace Using Multiblock Grids, Part I: Application to Grid Generation, Concurrency Practice and Experience (G. Fox ed.), Vol. 4(5), pp. 357-376.

[9] Häuser, J. et al., 1986: Boundary Fitted Conformed Co-ordinate Systems for Selected Two Dimensional Fluid Flow Problems. Part I, II, J. Num. Meth. Fluids, Vol 6, pp. 507-539.

[10] Eiseman, P.R., 1990: Interactive Grid Generation with Control Points, Computing Systems in Eng., Vol 1, Nos 2-4, pp. 293-304.

[11] Thompson, J. F., 1990: General Structured Grid Generation Systems, in: Applications of Mesh Generation to Complex 3-D configurations, AGARD-CP-464, 8 pp.

[12] Steger, J.S., 1989: Technical Evaluation Report on the Fluid Dynamics Panel Specialists' Meeting on Application of Mesh Generation to Complex 3-D configurations, AGARD-AR-268

[13] Häuser, J., A. Vinckier: Recent Developments in Grid Generation in: Applications of Mesh Generation to Complex 3-D Configurations, AGARD-CP-464, 15 pp.

[14] G. Fox et al, 1988: Solving problems on Concurrent Processors, Vol. 1, Prentice Hall, 592 pp.

[15] Gustafson, J.L. et al, 1988: Development of Parallel methods for a 1024-Processor Hypercube, SIAM, J. of Scientific and Statistical Computing, Vol. 9, No. 4, July, pp. 609-638

[16] Chapman, G.T., 1988: An Overview of Hypersonic Aerothermodynamics, Communications in Applied Numerical Methods, Vol. 4, pp. 319-325.

[17] Anderson,J.; Hypersonic and High Temperature Gas Dynamics, McGraw-Hill, 1989.

[18] Jameson, A. and S. Yoon, 1987: Lower-Upper Implicit Schemes with Multiple Grids for the Euler Equations; AIAA Journal, Vol.25, No.7, pp.929-935.

[19] Häuser, J. et al.,1988: Numerical Solution of 2D and 3D Compressible Navier Stokes Equations Using Time Dependent Boundary Fitted Coordinate Systems; First International Short Course on Modern Computational Techniques in Fluid Mechanics, University of Catalonia, Spain, 21 - 24 March,50 pp.

[20] Häuser, J., Simon, H.D., 1992: Aerodynamic Simulation on Massively Parallel Systems; in Parallel Computational Fluid Dynamics '91, eds. W. Reintsch et al., Elsevier-North Holland

[21] Koppenwallner, G., 1988: Aerothermodynamik Ein Schluessel zu neuen Transportgeraeten des Luft und Raumfahrt, Z. Flugwiss. Weltraumforschung 12, pp. 6-18

[22] Häuser, J., Williams, R.D., 1991: Strategies for Parallelizing a Navier-Stokes Code on the Intel Touchstone Machines, submitted to J. Num. Methods in Fluids, 7 pp. (preprint available)

[23] Wong, H., Häuser, J., 1991: Equilibrium Solution of the Euler and Navier-Stokes Equations around a Double Ellipsoidal Shape with Mono- and Multi-Blocks Including Real Gas Effects in Aerodynamics for Space Vehicles, ed. W. Berry, pp. 453-458, ESA SP318

[24] Hornung, H., Sturtevant, B., 1990: Challenges for High-Enthalpy Gasdynamics Research During the 1990's, Caltech Graduate Aeronautical Lab., Report FM 90-1, 23 pp.

[25] Park, C., 1989: Non-Equilibrium Hypersonic Aerothermodynamics, Wiley-Interscience, New York.

[26] Coleman, R.M., 1985: INMESH: An Interactive Program for Numerical Grid Generation, DTNSRCDC-851054, 33 pp.

[27] Häuser, J., Paap, H.-G., Wong, H., and M. Spel, 1991: **G**rid$\star$: A General Multiblock Surface and Volume Grid Generation Toolbox, in Numerical Grid Generation in Computational Fluid Dynamics (eds. A. Arcilla et al.), Elsevier North-Holland, 1991.

[28] J.F. Thompson, 1990: General Structured Grid Generation Systems, in: Applications of Mesh Generation to Complex 3-D configurations, AGARD-CP-464, 8 pp.

[29] J.L. Steger, 1989: Technical Evaluation Report on the Fluid Dynamics Panel Specialists' Meeting on Application of Mesh Generation to Complex 3-D configurations, AGARD-AR-268

[30] Parthasarathy, V.N., S. Sengupta, 1990: An Adaptive Re-Gridding Scheme Using Solution Contour Information; ASME International Computers in Engineering Conference, Boston, USA, 10 pp. bibitemrambo RAMBO-4G: G.D. Widhopf et al., 1990: An Interactive General Multiblock Grid Generation and Graphics Package for Complex Multibody CFD Applications, AIAA-90-0328

[31] Hsu, CP T. , J.K Lythe, 1989: AIAA 89 - 1984

[32] Eiseman, P.R., 1987: Adaptive Grid Generation, in Computer Methods in Applied Mechanics and Engineering, pp.321-376

[33] M.J. Bockelie, P.R. Eiseman, 1990: A Time Accurate Adaptive Grid Method an the Numerical Simulation of a Shock-Vortex Interaction, NASA-2998

[34] Thompson, J.F., 1987: A General Three-Dimensional Elliptic Grid Generation System on a Composite Block Structure, in: Computer Methods in Applied Mechanics and Engineering 64, pp. 377-411

[35] Eiseman, P.R., et al., 1994: GridPro/az 3000, Users's Guide and Reference Manual, 111 pp., Program Development Corporation of Scarsdale, Inc.

[36] Häuser, J. et al., 1992: Aerothermodynamic Calculations for the Space Shuttle Orbiter, AIAA 92-2953, Nashville, July 6-8, 19 pp.

[37] Hamann, B. et al., 1993: Interactive Surface Correction Based on a Local Approximation Scheme, Finite elemnts, Grid Generation, and Geometric design, Vol. 1, No. 1, to appear November 1993.

[38] Smith, R. E. (ed.), 1992: Software Systems for Surface Modeling and grid Generation, NASA Conference Publications 3143, 517 pp.

[39] Dener, C., 1992: Development of an Interactive Grid Generation and Geometry Modeling System with Object Oriented Programming, Ph D dissertation, Vrije Universiteit Brussel, 180 pp.

[40] Weatherill, N.P., 1990: Grid Generation VKI Lecture Series 1990-06 on Numerical Grid Generation, June 1990.

[41] Deconinck, H., Barth, T. (eds.), 1992: Unstructured Grid Methods for Advection Dominated Flows, AGARD-R-787.

[42] Steinbrenner, J.P et al., 1990: Multiple Block Grid Generation in the Interactive Environment, AIAA 90-1602, 20 pp.

[43] Soni, B.K., 1994: Algebraic methods and CAGD techniques in structured grid generation, in Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, (eds. N.P. Weatherill et al.), Pineridge Press LTD, pp. 365 – 375.

[44] Eiseman, P.R., Cheng, Z., Häuser, J., 1994: Applications of multiblock grid generations with automatic zoning, in Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, (eds. N.P. Weatherill et al.), Pineridge Press LTD, pp. 123 –134.

[45] Xia, Y., 1994: Two–dimensional Solution Adaptive Grid Generation in Computational Fluid Dynamics, diploma thesis, University of Applied Sciences, Braunschweig/Wolfenbüttel.

[46] IAF–89–028: Cassin Huygens Entry and Descent Technologies, *Octpber 7–13, 1989, Malage, Spain, 40th Congress of the International Astronautical Federation*.

[47] Buuce, A. Smith: JPL Sets Acoustic Checks of Cassini Tests Model, *Aviation Week and Space Technology, August 28, 1995*.