Mesh Generation

Mark Filipiak

Edinburgh Parallel Computing Centre

The University of Edinburgh

Version 1.0

November 1996

Table of Contents

1 Introduction

Continuous physical systems, such as the airflow around an aircraft, the stress concentration in a dam, the electric field in an integrated circuit, or the concentration of reactants in a chemical reactor, are generally modelled using partial differential equations. To perform simulations of these systems on a computer, these continuum equations need to be discretised, resulting in a finite number of points in space (and time) at which variables such as velocity, density, electric field are calculated. The usual methods of discretisation, finite differences, finite volumes and finite elements, use neighbouring points to calculate derivatives, and so there is the concept of a mesh or grid on which the computation is performed.

There are two mesh types, characterised by the connectivity of the points. Structured meshes have a regular connectivity, which means that each point has the same number of neighbours (for some grids a small number of points will have a different number of neighbours). Unstructured meshes have irregular connectivity: each point can have a different number of neighbours. Figure 1 gives an example of each type of grid. In some cases part of the grid is structured and part unstructured (e.g., in viscous flows where the boundary layer could be structured and the rest of the flow unstructured).

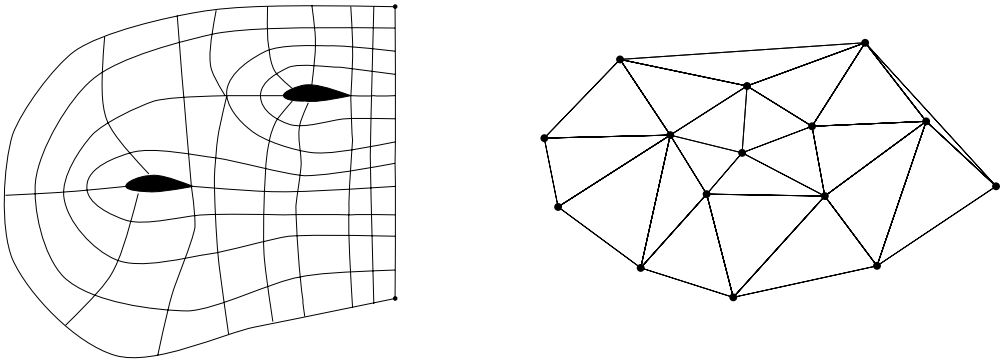

Figure 1: Structured mesh (left) and unstructured mesh (right).

In the rest of this chapter, the various discretisation methods are described, with their mesh requirements. Chapter 2 describes the methods used to generate structured meshes in simple domains (algebraic and elliptic methods) and the extension to complex domains using multiblock. Chapter 3 describes methods used to generate unstructured meshes, concentrating on the two main methods for producing triangular / tetrahedral meshes: advancing front and Delaunay triangulation. Chapter 4 gives a very brief introduction to adaptive meshing.

1.1 Discretisation and mesh type

The main discretisation methods are finite differences [8], finite volumes (which is equivalent to finite differences) [9] and finite elements [10]. To illustrate the methods, we consider the conservation form of the convection equation

$$\frac{\partial \rho}{\partial t} + \nabla \bullet (\rho \boldsymbol{U}) = S$$

where ρ is the density, $\boldsymbol{U}$ is the velocity, and S is a source term. The $\rho\boldsymbol{U}$ term is the flux of ρ.

The finite difference formulation approximates the derivatives using neighbouring points, e.g., for a regular, rectangular grid with spacing h in the x direction

$$\frac{\partial \rho}{\partial x} \approx \frac{1}{h} [\rho (x_{n+1}) - \rho (x_n)] .$$

Although irregular grids *can* be used for finite differences, regular grids are invariably used. These lead to simple difference schemes and efficient solvers (on vector machines, for example). There can be problems with the finite difference scheme at coordinate singularities for certain regular grids (e.g., spherical polar coordinates) [12]

In the finite volume formulation, the physical space is split up into small volumes V and the partial differential equations integrated over each of these volumes:

$$\frac{d}{dt}\int_V \rho d\Omega + \oint_{\partial V} (\rho \boldsymbol{U}) \bullet \boldsymbol{n} d\Gamma = \int_V S d\Omega$$

Then the variables are approximated by their average values in each volume, and the fluxes through the surfaces of each volume are approximated as function of the variables in neighbouring volumes.

Finite volume discretisation can use both regular and irregular meshes. In an irregular mesh the fluxes through the surfaces are still well defined.

The finite element method also splits the spaces up into small volumes, the ‘elements’. In each element, the variables and fluxes are approximated using weighting functions. The computational variables are the coefficients of these weighting (or ‘shape’) functions.

Finite element methods generally use irregular meshes: there is no special advantage in using regular meshes.

1.2 Mesh characteristics

For all types of meshes, there are certain characteristics that we want to control.

- The local density of points. High density gives more accuracy, but computation takes longer. This leads to adaptive meshing methods , treated very briefly in

Chapter 4.

- The smoothness of the point distribution. Large variations in grid density or shape can cause numerical diffusion or anti-diffusion and dispersion or refraction of waves. This can lead to inaccurate results or instability [12].
- The shape of the grid volumes. For instance, boundary layers in fluid flow require a grid that is very compressed normal to the flow direction. In the finite element method using triangular elements the maximum angle must be bounded strictly below π to prove convergence of the method as the element size is reduced [11].

For simple domains, the choice between regular or irregular meshes is governed mainly by the discretisation method. However, for complex domains (e.g., the inside of a hydro-electric turbine) irregular meshes are preferred to regular meshes. Irregular mesh generation (at least for triangular or tetrahedral elements) can be fully automatic and fast. Regular mesh generation requires the domain to be split up into simple blocks which are then meshed automatically. This block decomposition is at best semi-automatic and can require man-months of user effort.

2 Structured meshes

This chapter begins with a discussion of boundary-fitted grids and the discretisation of PDEs on them, then deals with the main methods for grid generation of simple domains (using algebraic or differential equation methods) and then explains the multiblock concept used for more complicated domains. References [12] and [13] are detailed expositions of structured mesh generation.

2.1 Boundary-fitted meshes

Structured meshes are characterised by regular connectivity, i.e., the points of the grid can be indexed (by 2 indices in 2D, 3 indices in 3D) and the neighbours of each point can calculated rather than looked up (e.g., the neighbours of the point (i, j) are at $(i+1, j)$, $(i-1, j)$, etc.). Meshes on a rectangular domain are trivial to generate (though come care needs to be taken in the discretisation at convex corners) and structured mesh generation techniques concentrate on meshing domains with irregular boundaries, e.g., the flow in blood vessels, or the deformation, flow and heat transfer in metal being formed in dies. Generally the meshes are generated so that they fit the boundaries, with one coordinate surface forming (part of) the boundary. This gives accurate solutions near the boundary and enables the use of fast and accurate solvers. For fluid flow these grids allow the easy application of turbulence models, which usually require the grid to be aligned with the boundary. The alternative is to use a rectangular grid which is clipped at the boundary, with local grid refinement near sharp features on the boundary ('Cartesian grids'). This will reduce the truncation order at the boundary and will require the mesh cells to be clipped at the boundary, increasing the complexity of the solver (even for a finite volume code). Cartesian grid generation is very fast and has been used for Euler aerodynamics [14]; it does not appear to be applicable to Navier-Stokes aerodynamics.

The most common method of generating boundary-fitting grids is to have one continuous grid that fits to all the boundaries. The effect is to fit a contiguous set of rectangular computational domains to a physical domain with curved boundaries (Figure 2).

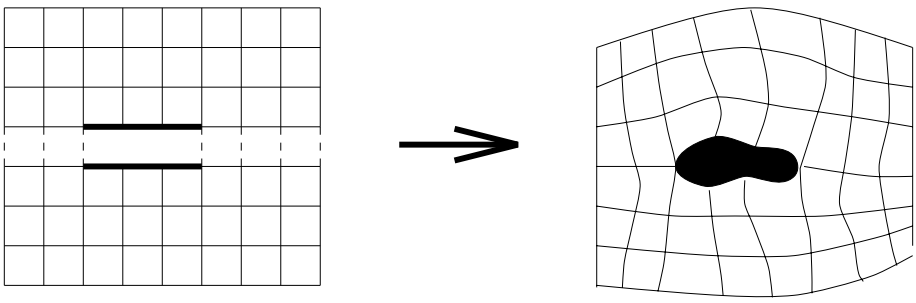

Figure 2: Two-block boundary-fitted grid

It is difficult to fit complex domains with one mapping from a rectangular computational domain without generating excessively skewed grids. To get round this prob-

lem the domain is split up into blocks and each block is gridded, with some continuity requirements at the block interfaces; this is *multiblock*. The decomposition of the domain into blocks is usually done manually using CAD techniques and is slow.

An alternative to continuous boundary-fitted grids with multiple blocks is to use a boundary fitting grid near each boundary, and simple rectangular grid in the interior, and interpolate between them. These are called *overset* or *chimera* grids (Figure 3)[15].

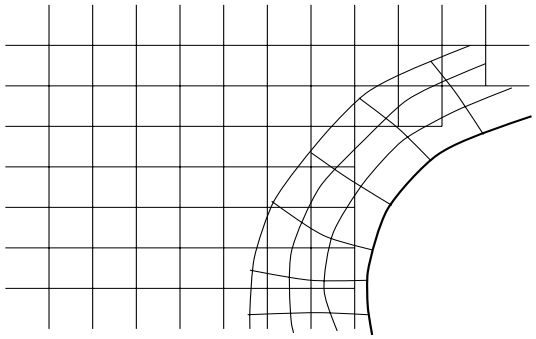

Figure 3: Overset grids

This type of grid is easier to generate than a multiblock grid since each grid is local and doesn't need to match to others. The individual grids will generally be of high quality (low distortion). However, the interpolation can be difficult, especially with more than two grids overlapping, and it adds to the solver time (increasing it by 10%-30%). The overlapping grids cannot be too different in resolution and this can cause problems with the grids required for boundary layers in viscous flow simulations.

Chimera grids are very useful for moving boundaries (e.g., helicopter blades) or multiple boundaries (e.g., particles in a fluid). Most of the grid remains fixed but the interpolation changes as the grids move with the boundaries.

Chimera grids do have certain advantages, and the recent work [16] on conservative interpolation methods have increased their usefulness. However, the bulk of structured grid generation is based on multiblock type grids, and we will concentrate on these for the rest of this chapter.

2.2 Problem solution on curved grids

Once the grid is generated, the physical problem has to be discretised and solved on this grid. It is possible to calculate finite difference or finite volume formulas using the physical grid directly, but this will reduce the truncation order of the FD/FV scheme since the grids are generally non-uniform. The preferred method is to transform the equations used to model the problem into computational space. Since the physical grid is defined as some transformation from the rectangular computational grid, this process is straightforward. The resulting equations in the computational space will include the effects of the coordinate transformation, e.g., time derivatives of moving grids. The finite differencing can be done on the transformed equations, on the computational grid. Since the computational grid is usually completely regular, high order methods can be used, and the truncation order is preserved if the resulting FD/FV equations are transformed back to the physical grid. (Flow solvers can be developed to use chimera grids, with the extra book-keeping required to interpolate between each grid.)

Although the order of FD/FV methods can be preserved, there are other effects of curvilinear grids on the accuracy of the solution.

- Grid size variation gives numerical diffusion (or anti-diffusion leading to instability). The effect is worst with small grid size and large solution gradients.
- Non-orthogonality, or skewness, of the (physical) grid increases the coefficient of the truncation error, but doesn't change the order. Skewness of up to $45°$ are acceptable in the centre of the grid, but one-sided differences are used at boundaries, where orthogonality should be maintained. These limits on skewness are the main reason that multiblock methods are used.
- In a multiblock scheme, the corner junctions between blocks are usually points with non-standard connectivity (e.g., in 2D, 5 grid lines from a point rather than 4) and these need special treatment (e.g., for finite volume discretisation these points need to be discretised in physical space).

2.3 Boundary fitting grids on a single block

For simple domains, a single grid can be used without leading to excessive skewness. The computational space is a rectangle or cuboid, or at least has a rectangular boundary, with a regular, rectangular grid. We then need to define a 1-1 mapping from the computational space to the physical space. The methods currently used are

- algebraic grid generation, where the grid is interpolated from the physical boundaries,
- methods where a PDE for the grid point coordinates is solved,
- variational grid generation, where some functional, e.g., smoothness, is maximised / minimised (variational grid generation is not treated in this report, see [13] for details),
- other methods, (not treated in this report), e.g., conformal mapping [12], [13].

2.4 Algebraic grid generation: interpolation

The method used in the algebraic method of grid generation is *transfinite interpolation* (*TFI*).

2.4.1 Transfinite interpolation (TFI)

Working in 2D (the extension to 3D is straightforward), take the computational domain as the unit square $[0, 1] \times [0, 1]$ with coordinates s and t, and the physical domain as some region with coordinates x and y. To generate the grid in the physical space, we would create a grid in the unit square and map this into the physical domain. There are two requirements for this mapping (Figure 4):

- It must be 1-1, and
- the boundaries of the computational space must map into the boundaries of the physical space,

but otherwise the mapping can be arbitrary, although it may not produce a good grid.

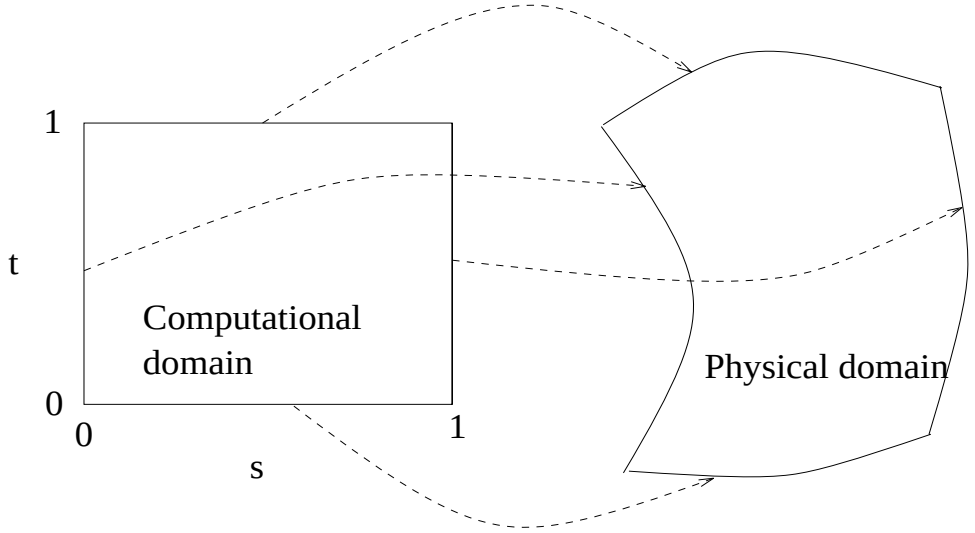

Figure 4: Mapping of boundaries

TFI is one kind of mapping, where the physical coordinates, treated as a function of the computational coordinates, are interpolated from their values on the boundaries of the computational domain. Since the data is given at a non-denumerable number of points, the interpolation is called transfinite.

The 2D interpolation is constructed as a linear combination of 2 1D interpolations (also called *projections*) and their product. First define the blending functions ϕ_0, ϕ_1 and θ_0, θ_1 that will be used to interpolate in each direction. ϕ_0 interpolates values from the $s = 0$ boundary into the domain, ϕ_1 interpolates values from the $s = 1$ boundary into the domain, and similarly for θ_0, θ_1 in the t direction. The requirements on $\phi_0(s)$ and $\phi_1(s)$ are

$$\phi_0(0) = 1, \phi_0(1) = 0$$
$$\phi_1(0) = 0, \phi_1(1) = 1$$

and similarly for θ_0, θ_1 in the t direction. The simplest blending function is linear, giving linear interpolation

$$\phi_0(s) = 1 - s$$
$$\phi_1(s) = s$$

The 1D transfinite interpolations (projections) are formed as follows (for the x coordinate)

$$P_s[x](s,t) = \phi_0(s)\,x(0,t) + \phi_1(s)\,x(1,t)$$
$$P_t[x](s,t) = \theta_0(t)\,x(s,0) + \theta_1(t)\,x(s,1)$$

and the product projection is

$$P_s P_t [x]\ (s,t) = \phi_0(s)\,\theta_0(t)\,x(0,0) + \phi_1(s)\,\theta_0(t)\,x(1,0) + \phi_0(s)\,\theta_1(t)\,x(0,1) + \phi_1(s)\,\theta_1(t)\,x(1,1)$$

which is the finite interpolant for the values of x at the four corners of the computational domain. The 2D transfinite interpolation is then

$$(P_s \oplus P_t)\ [x] = P_s [x] + P_t [x] - P_s P_t [x]$$

which interpolates to the entire boundary. To form the grid in the physical space, this interpolant is used to map the points of a regular grid in the computational space (Figure 5).

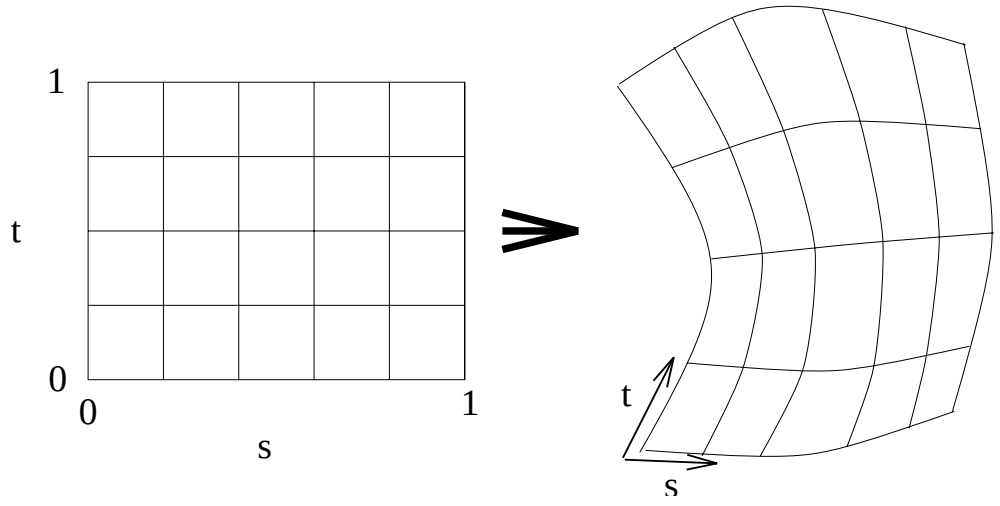

Figure 5: Transfinite interpolation

It is possible to extend the TFI to so that it interpolates to several coordinate lines, not just the boundaries. This is useful to control the grid density and also may be required to ensure a 1-1 mapping (i.e, a valid grid). The blending functions would then be, for example, cubic splines. It is also possible to match normal derivatives, e.g. dx/ds, at the boundaries. This allows grid slope continuity between the blocks of a multiblock grid.

There are some problems with TFI. The mapping will propagate boundary singularities (corners) into the interior of the domain, which is not ideal for fluid flow simula-

tions. A more serious problem is that if the mapping is not 1-1 then overspill occurs (Figure 6).

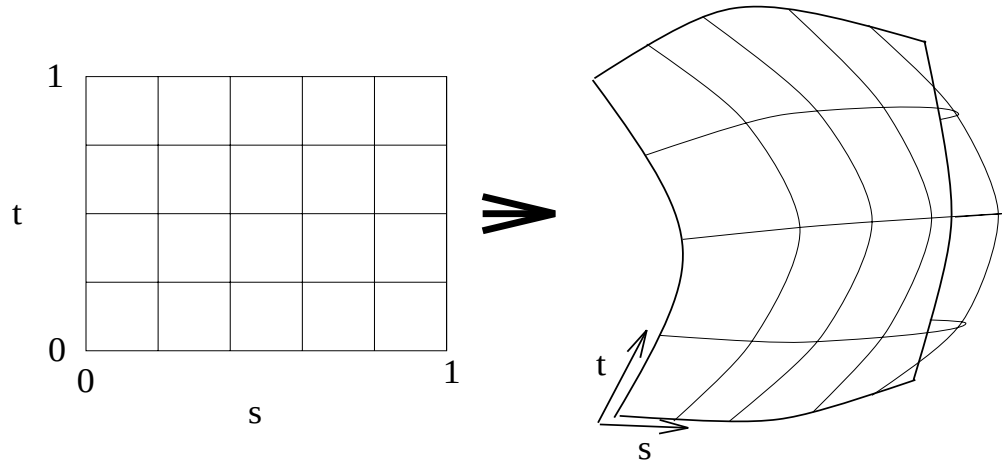

Figure 6: Overspill

This can be corrected be re-parameterising the boundaries, or adding constraint lines (surfaces in 3D) inside the domain.

However, TFI grid generation is very fast and is the only competitive method for 3D. It is possible to use an PDE based generator to refine a grid produced using TFI.

2.5 PDE grid generation

The idea behind these methods is to define PDEs for the physical space coordinates in terms of the computational space coordinates, and then solve these equations on a grid in computational space to create a grid in physical space. The main type of PDE used is elliptic, which is suitable for domains with closed boundaries (for unbounded domains, a fictitious boundary at large distances is used). The alternatives are hyperbolic and parabolic equations, which are used for unbounded domains.

2.5.1 Elliptic grid generation

These methods start with an PDE for the computational space coordinates ξ^1 and ξ^2 in terms of the physical space coordinates x^1 and x^2 (the extension to 3D is straightforward). The simplest example is the Laplace equation

$$\nabla^2 \xi^i = 0$$

Parts of the physical boundary, corresponding to coordinate lines, are then 'equipotential surfaces' (Figure 7).

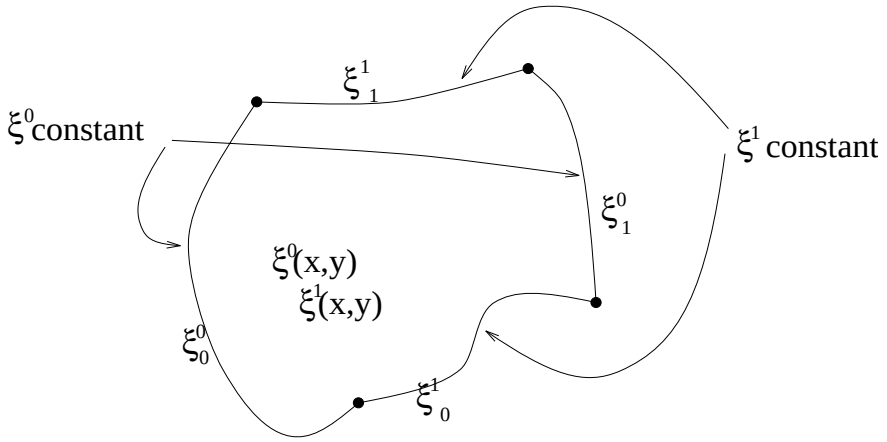

Figure 7: Elliptic grid generation boundaries

In 2D, the extremum principle guarantees that the mapping is 1-1, and the Laplace operator is smoothing so that boundary slope discontinuities are not propagated into the interior.

In order to solve the equation in its current form, you would already need a grid in the physical domain, so the equation is transformed so that x^i are the dependent variables, and ξ^i are the independent variables

Using the equation for $\nabla^2\xi^k$ in the relation

$$\nabla^2\vec{r} = g^{ij}\vec{r}_{\xi^i\xi^j} + \left(\nabla^2\xi^k\right)\vec{r}_{\xi^k} = 0$$

gives

$$g^{ij}\vec{r}_{\xi^i\xi^j} = 0$$

and g^{ij} (the metric tensor) can be expressed in terms of $g_{ij} = \vec{r}_{\xi^i} \bullet \vec{r}_{\xi^j}$ (see [12] for details).

This quasi-linear equation can then be solved in the rectangular ξ^i domain, with the location of the points on the physical boundaries as the boundary data at the points on the computational boundaries. Any standard solver can be used, and an algebraically generated grid can be used as the initial guess to ensure / increase convergence.

Grids based on the Laplace equation are not very flexible. The coordinate lines tend to be equally spaced, except near convex or concave boundaries, and it is not possible to control the grid line slope at the boundaries (since this is a 2nd order equation).

To control the grid spacing and slope, we can add a source term (*control function)*, giving Poisson's equation

$$\nabla^2\xi^i = P^i$$

The effect of the control function is shown in Figure 8

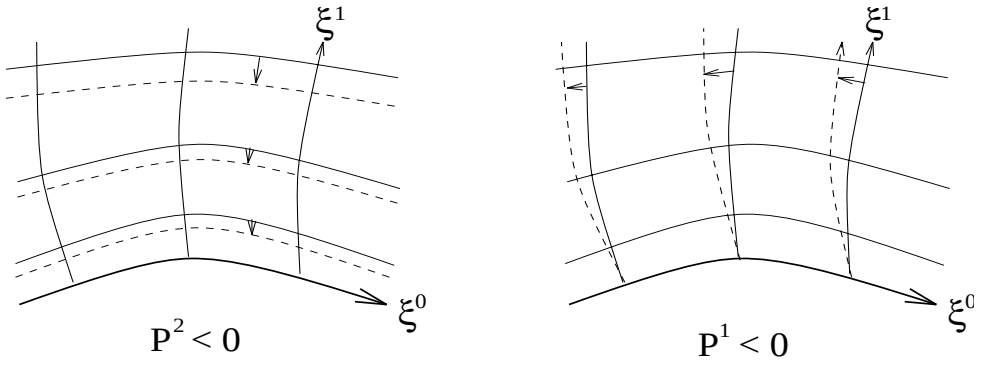

Figure 8: Effect of control functions

The metric tensor is factored out of the source terms to give, in general

$$\nabla^2 \xi^i = g^{jk} P_{jk}^{\ \ i}$$

The transformed equations now become

$$g^{ij}\left(\vec{r}_{\xi^i \xi^j} + P_{ij}^k \vec{r}_{\xi^k} \right) = 0$$

Usually only one-dimensional stretching in each direction is required; then the source term is $g^{ii} P_{ii}^i$ (no sum). In theory, the source terms are defined in terms of the physical coordinates, but in practice they are defined in terms of the computational coordinates, so they will cause attraction to/repulsion from grid points rather than fixed physical points. This makes the direct (manual) use of control functions rather difficult and they tend to be used indirectly to achieve a variety of effects

- Because of the strong smoothing, the coordinate lines will tend to be equally spaced in the interior of the domain. To get the boundary point spacing to propagate into the interior, source terms can be derived from boundary data to give the desired spacing, and then interpolated (using transfinite interpolation)
- The point spacing is fixed on the boundary, and since the grid generation equations are 2nd order, it is not possible to control the coordinate line slope at the boundary as well. One possible solution is to use the biharmonic equation rather than Laplace's equation, but the preferred method is to use control functions. In Steger and Sorenson's method [12] the control functions are iteratively adjusted as the elliptic equation is solved, and the grid line slope at the boundary, the grid point spacing along the boundary, and the spacing of the first parallel grid line from the boundary, can all be controlled. This can be used to give a grid that is orthogonal at the boundary (e.g., for boundary layers). It can also be used to give slope and spacing continuity for patched grids (for embedded grids in multiblock, but see later for a better but slower method of matching at block boundaries).

2.6 Implementation in 3D

In 3D we have a cuboid as the computational domain and the generation process proceeds in the order: edges, surfaces, volume. First we choose the computational grid. The gridding of the edges is calculated using a 1D Poisson equation. Then the bound-

ary surface grids are generated. This is similar to 2D grid generation, but the surface curvature needs to be taken into account (surfaces are usually defined in the CAD system as a function of 2D parameter space, e.g., contiguous bicubic splines patches, and simple 2D grid generation can be performed in this space). Finally, the 3D grid generations can be done, using the surface grids as data.

There are some implementation difficulties with elliptic grid generation

- The transformed equations are non-linear, so using a good initial guess (from an algebraic grid generator) will speed up convergence.
- There can be slow convergence (or failure) for large control functions, sharp corners, or highly stretched grids.
- The method is really too slow for 3D grids, where algebraic techniques are generally used.

2.7 Other methods

2.7.1 Hyperbolic grid generation

Instead of using an elliptic PDE to generate the grid, hyperbolic equations have been used, e.g. in 2D

$$x_{\xi^0} x_{\xi^1} + y_{\xi^0} y_{\xi^1} = 0$$
$$x_{\xi^0} y_{\xi^1} - x_{\xi^1} y_{\xi^0} = V$$

where the first equation enforces orthogonality and the second controls the cell area. These methods are also available in 3D. They have certain characteristics

- they produce orthogonal grids,
- they don't smooth boundary discontinuities,
- they are applicable to exterior domains (e.g., the flow around an aerofoil). The solution for the grid coordinates is marched out from the boundary,
- they can fail for concave boundary surfaces (as the solution proceeds, gridlines overlap and cell volumes become negative),
- they are fast, taking about the same time as one iteration of an elliptic grid generator.

2.7.2 Parabolic grid generation

Parabolic methods have been used for grid generation, combining the speed of the hyperbolic generator (marching) with increased smoothing, but these methods do not seem to be widely used.

2.8 Multiblock

In theory, complex goemetries can be mapped to one rectangular (here take rectangular to apply to 2D and 3D) region (perhaps not simply-connected) but this will lead to unacceptable distortion of the grid cells. In practice, the physical region is broken up into pieces that each have a (relatively) simple mapping from a rectangular grid. These pieces (*blocks*) are fitted together (*multiblock*) with some degree of grid continuity at their interfaces, ranging from none (even the grid point coordinates differ) to

complete (looks like a single grid with no slope or spacing discontinuities). So the grid generation process splits into two parts, the decomposition of the physical domain into blocks and the gridding of each block. The decomposition process has not yet been fully automated, and requires considerable user interaction (e.g., choosing block edges to align with object edges) to produce good meshes. The meshing of the blocks can proceed automatically, using one of the methods discussed above.

Before looking at multiple blocks, we look at some extensions to a simple rectangular (physical) block that can be used to fit objects and then can be inserted into the multi-block scheme as a single block.

2.8.1 C, O, H grids

C, O, and H grids, so named because of their approximate shape, are slight extensions of the simple 'rectangular' block. The computational blocks are usually simple rectangles, but corners in the computational space need not correspond to corners in the physical space and *vice versa* (Figure 9). At these degenerate points care needs to be taken with the difference formulation.

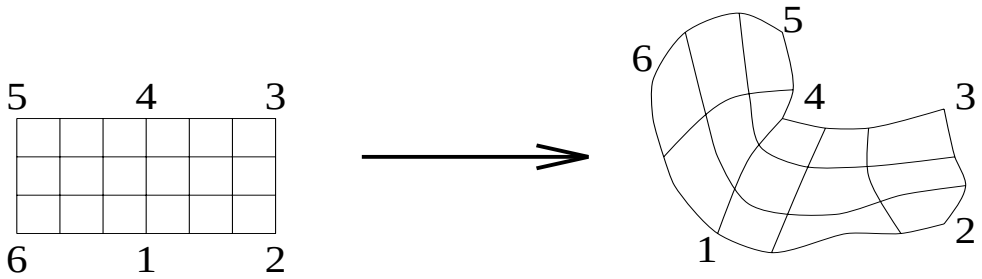

Figure 9: Degenerate points

The computational block can be multiply connected (Figure 10),

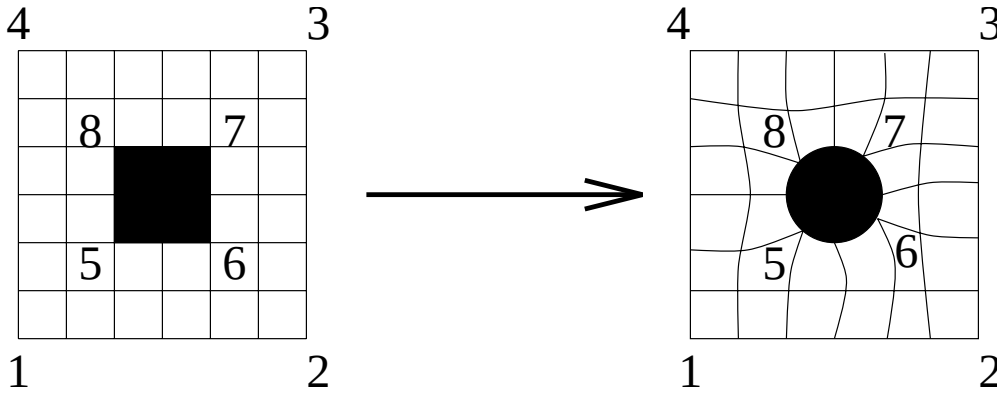

Figure 10: Multiply-connected block

but cuts can be made to transform them to simply-connected blocks, mappable to a rectangle, and this leads to the C, O and H grids (in fact the H grid type includes this multiply-connected block).

The O grid just a polar grid (Figure 11)

Figure 11: O type grid

The C grid (Figure 12) was developed for aerofoils, with rounded leading edges and sharp trailing edges. It gives a much better grid (and more accurate solutions) than an H type grid about an aerofoil.

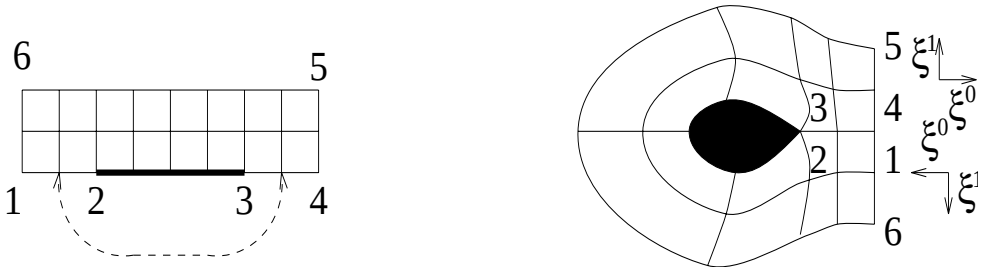

Figure 12: C type grid

The H grid is a Cartesian-type grid, but allowing a multiply-connected region as shown before, or with the interior slab reduced to a slit (Figure 13).

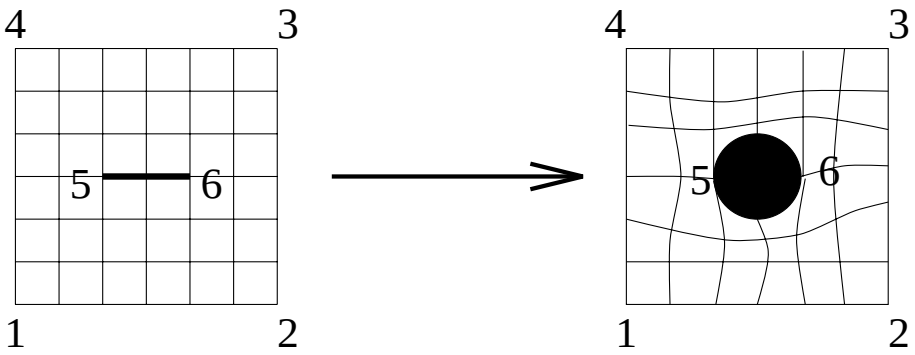

Figure 13: H type grid

Across the cut, the coordinate directions or species can change (see Figure 12). This will mean that special cases will have to be processed in the difference formulation of the problem (and for the grid generation if a PDE method is used). This complication can be avoided by using a extra layer of points, a *halo*, to hold copies of the connecting points from the other side of the cut (Figure 14). This is almost the same as using haloes in domain decomposition for parallel processing, but in Figure 14 the 'boundary' is along a line/surface of grid points. In a finite volume formulation the data

points are centred on the grid cells, and in this case the haloes are exactly the same as in domain decomposition.

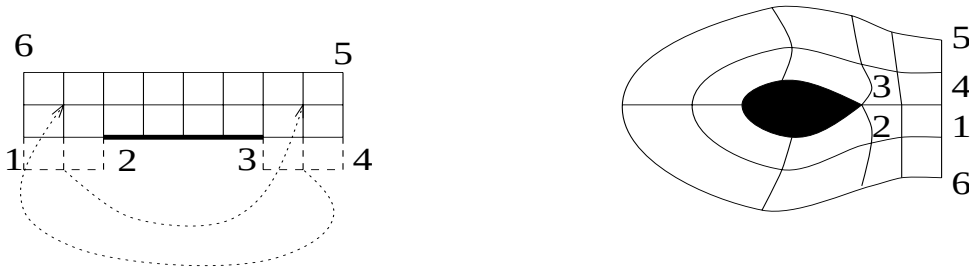

Figure 14: Halo points

Once the haloes are set up, the same difference formulas can be used in the interior and on the cuts.

2.8.2 Multiblock

Using C, O and H grids in the component blocks, we can now create composite grids: *multiblock*.

The first step is to segment the physical region into sub-regions each bounded by 4 (curved) sides in 2D or 6 surfaces in 3D. This step requires considerable user interaction to generate a good decomposition of the domain (e.g., with block edges aligned with sharp features).

The second, automatic, step is to generate grids in each sub-region. The overall grid is formed by joining these sub-grids together. The degree of continuity of the grid lines at the boundaries between the sub-regions can be anything from complete continuity of all derivatives to no continuity, not even the grid points on the boundary. Evidently, the less continuity, the more interpolation / approximation will have to be done in the solver when processing sub-region boundaries, and there will be some loss of accuracy in the difference formulations. If algebraic methods are used in each block, grid line continuity is ensured by generating the common edges and then surfaces (in 3D) and then volumes, each step using the boundary points generated by the previous one. It is also possible to apply TFI to grid line slopes, and this allows slope / spacing continuity to be ensured. If PDE (elliptic) methods are used, the equations can be solved in the whole domain, using the haloes to transfer physical coordinate information from one block to its neighbours. This gives complete continuity of the grid coordinates and the region boundaries will also adjust. Solving over the whole domain is very time consuming, so the alternative is to solve in each block and use Steger and Sorenson type control functions to ensure slope / spacing continuity.

Figure 15 shows a simple multiblock example, with the computational grid points that map to a single physical grid point identified.

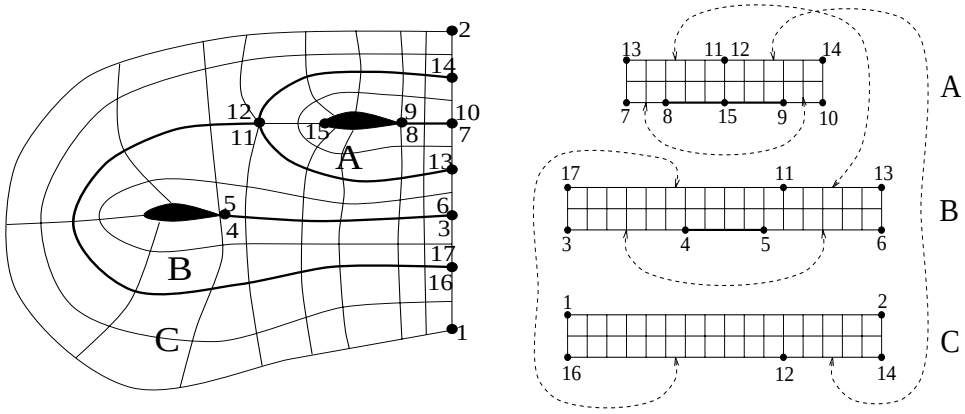

Figure 15: Example of multiblock: 3 C-type grids

There are some disadvantages with fully boundary-fitted multiblock grids:

- The blocking requires a great deal of user effort, this can be 1 man-month for complex configurations. Several different physical configurations may be investigated in the design phase of a product, and each may need a different block structure
- Changes of geometry in one block can cause changes to many other blocks
- Change in the grid point distribution in one block, e.g. adding points near a sharp feature on an object, will generally cause changes in other blocks if grid point continuity is to be maintained.
- Requiring grid point continuity makes it difficult to increase resolution in one block without (unnecessarily) increasing the resolution in other block

2.9 Summary

The main method of structured grid generation is multiblock with TFI or elliptic generation in the blocks. The regularity of structured grids allows fast solvers to be used, vectorisation / parallelisation is easy and cache use is efficient. There is a large user effort in constructing the multiblock decomposition and the tendency for complex configurations is now to use unstructured meshes, which can be automatically generated, even though the solvers are slower.

3 Unstructured Meshes

Unstructured meshes have been developed mainly for the finite element method. There is a large range of possible shapes for finite elements: tetrahedra, prisms, blocks, and there can be arbitrary connectivity, leading to unstructured meshes. However, the only shapes that can be used to generate meshes *fully* automatically are triangles in 2D and tetrahedra in 3D. Although block elements are desirable, it is much harder to generate grids of block elements automatically (and there will be prisms or tetrahedra required in some regions). The bulk of this chapter considers only triangular / tetrahedral mesh generation. Reference [17] is a detailed exposition of unstructured mesh generation.

3.1 Mesh requirements for the Finite Element Method

The Finite Element Method (FEM) has certain requirements of a mesh.

- The mesh must be valid, i.e. no holes, no self-intersection, no faces joined at two or more edges, etc. This is an obvious requirement, but many mesh generation schemes require (and their implementations contain) a large amount of checking for these conditions.
- The mesh must conform to the boundary of the domain. Again this is an obvious requirement, but again some schemes (e.g., a straightforward Delaunay triangulation) will not satisfy this, and the generation scheme will need to check and correct any edges / faces that intersect the boundary.
- The density of the mesh must be controllable, to allow trade-off between accuracy and solution time.
- The grid density will vary depending on local accuracy requirements, but this variation must be smooth to reduce or eliminate numerical diffusion / refraction effects.
- There are some requirements on the shape of elements. In general, the elements should as 'equiangular' as possible, e.g. equilateral triangles, regular tetrahedra. Highly distorted elements, e.g. long, thin triangles, squashed tetrahedra, can lead to numerical stability problems caused by round-off errors. In addition, the convergence properties of FEM are proved in the limit of small element size only if the maximum angle in an element is bounded away from π [11]. These requirements are modified for boundary layers, where highly stretched elements are desired and the FEM formulation allows for them. Even in this case, the min-max-angle property is required.

Triangles / tetrahedra can fit irregular boundaries and allow a progressive change of element size without excessive distortion, and so are well-suited for mesh generation for FEM. In addition, there are fully-automatic methods for generating triangular / tetrahedral meshes. However, linear tetrahedra are not that good for FEM (they are too 'stiff') and a high density of elements is needed to give acceptable results; this leads to

increased solver time. Quadratic quadrilateral / hexahedral elements are much better, but it is difficult to automatically generate all-hexahedral meshes (even allowing transition elements such as prisms). Quadratic tetrahedra have as good FEM properties as quadratic hexahedra and do allow automatic mesh generation.

3.2 Mesh generation methods

There are several methods for generation for unstructured meshes discussed in this chapter

- Decomposition and mapping, which is essentially multiblock with TFI
- Overlaying a regular grid or quadtree / octree grid and warping elements to fit the boundaries
- Advancing front
- Delaunay triangulation
- Less common methods, hybrid structured / unstructured grids, mesh smoothing

The most popular methods are Delaunay triangulation and advancing front.

3.2.1 Decomposition and mapping

This was the earliest method developed, and is still used in several commercial FEM packages. The domain is decomposed interactively into regions, as in multiblock for structured grid generation. Then the individual regions are meshed using some mapping method. This can be transfinite interpolation, but a simpler method based on the shape functions used to define *isoparametric* finite elements has been used extensively [18]. The isoparametric interpolation is a point interpolation (i.e., a finite interpolation) and is illustrated here in 2D using a quadratic mapping (parabolic quadrilateral). In this case the points to be interpolated are at the corners and midpoints of the edges of the curvilinear quadrilateral (Figure 16) and the interpolation formulas are

$$x = \sum_{i=1}^{8} N_i x_i \qquad y = \sum_{i=1}^{8} N_i y_i \qquad z = \sum_{i=1}^{8} N_i z_i$$

where

$$N_1 = \frac{1}{4}(1-\xi)\,(1-\eta)\,(-\xi-\eta-1) \qquad N_2 = \frac{1}{2}(1-\xi)\left(1-\eta^2\right)$$

$$N_3 = \frac{1}{4}(1-\xi)\,(1+\eta)\,(-\xi+\eta-1) \qquad N_4 = \frac{1}{2}\left(1-\xi^2\right)(1+\eta)$$

$$N_5 = \frac{1}{4}(1+\xi)\,(1+\eta)\,(\xi+\eta-1) \qquad N_6 = \frac{1}{2}(1+\xi)\left(1-\eta^2\right)$$

$$N_7 = \frac{1}{4}(1+\xi)\,(1-\eta)\,(\xi-\eta-1) \qquad N_8 = \frac{1}{2}\left(1-\xi^2\right)(1-\eta)$$

Other interpolation formulas (cubic, quartic, etc.,) can be used, and the method extends to 3D. The quadrilateral can be degenerated to a triangle and edges can be identified as in C and O type grids.

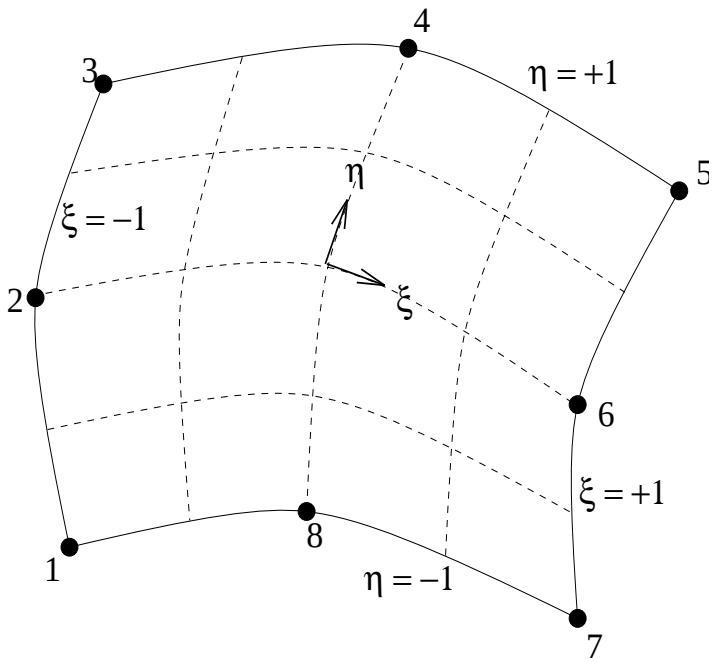

Figure 16: Isoparametric meshing

3.2.2 Grid based methods

In these methods a triangular or rectangular grid is overlaid on the domain, extending beyond the boundaries. The grid is then cropped to the domain. Finally the edge points are moved (*warped*) to the boundary and, if necessary, quadrilaterals are split into triangles (Figure 17). This method is fast, but gives poor quality grids at the boundaries, which is usually where high grid quality is required (for boundary layers, stress concentrations, etc.).

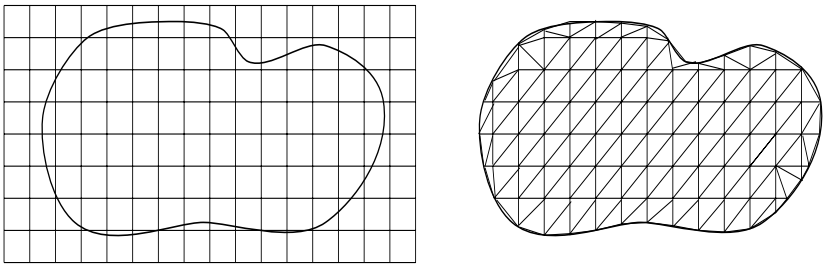

Figure 17: Overlaid grid, cropped, warped to boundary and split into triangles

A development of this method is to use a quadtree (Figure 18) in 2D [19] or octree in 3D [20] as the basic grid. This leads to similar problems with the grid at the boundaries. The cropping to the boundary can often be done by the CAD solid modelling

package used to create the object being meshed. This method (in 3D) is used in some commercial packages.

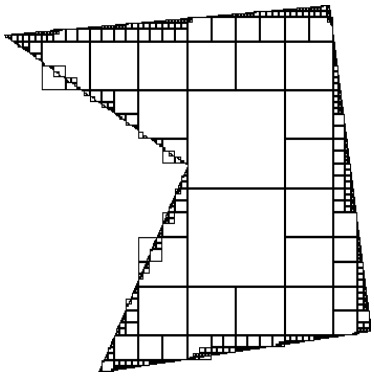

Figure 18: Quadtree grid, before triangulation and warping

3.2.3 Advancing front

This method is based on a simple idea (Figure 19)

- Discretise the boundary (e.g., in 2D fit it with polygon(s)). This is the initial *front*.
- Add triangles / tetrahedra into the domain, with (at least) one edge / face on the front. At each step this will update the front.
- When the front is empty (e.g., in 2D no polygon segments left) the mesh generation is complete. This requires that the domain be bounded, but for unbounded domain the front can be advanced until it is at some large distance from the object.

Although the idea is simple the algorithmic details of this method are complex. The method will be illustrated in 2D [21], with comments on the differences for the 3D

case [22]. In general the advancing front method in 3D is much more sensitive to geometric tolerances and numerical errors (especially for stretched grids) than in 2D.

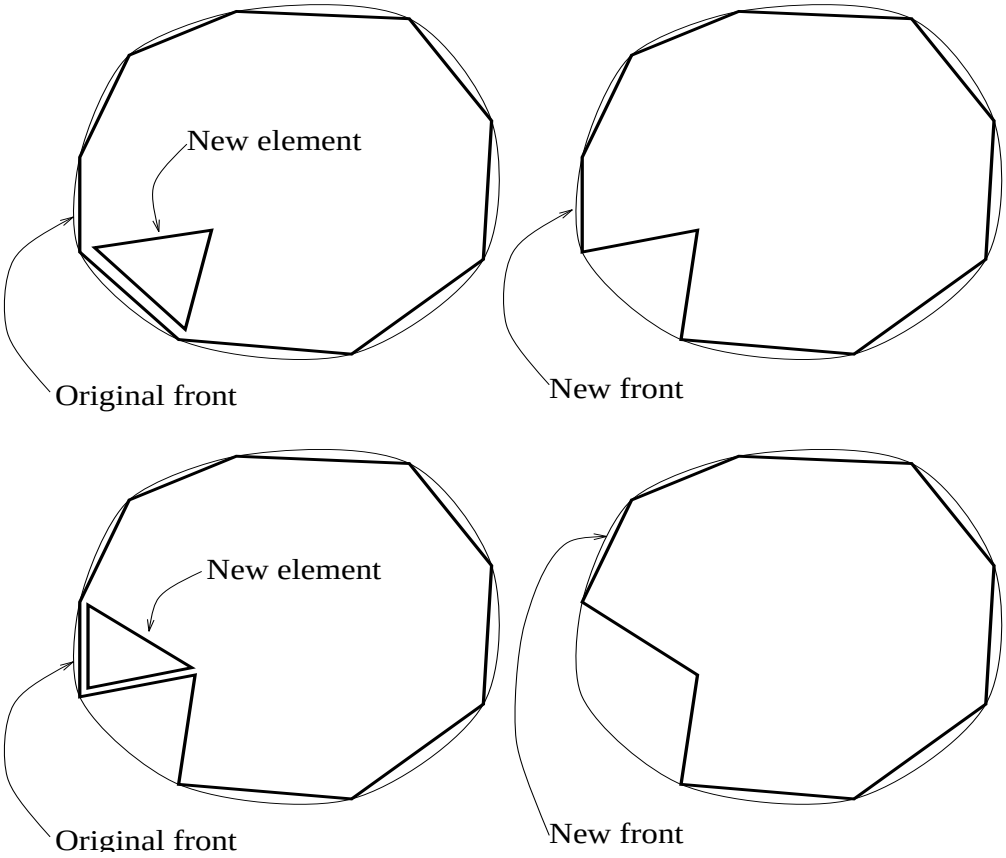

Figure 19: Advancing front

The initial data for the method are the domain boundary, and some mesh parameters (density, stretching) defined over the domain.

The boundary is made up of closed loops of parameterised curves / surfaces patches (e.g., generated by the CAD system used to design the object / domain being simulated).

The mesh parameters are the average node spacing δ and any directional stretching factors. The parameters are usually defined on a background grid and the spatial distribution of mesh parameters is given by (linear) interpolation from values specified at the nodes. The background grid can be one large triangle / tetrahedron enclosing the whole domain (e.g. for constant mesh size), or a coarse grid created interactively, or a grid already used for computation (this is the case for adaptive meshing, using the inverse of the local error to specify the new local grid size). It is possible to control the grid size from the boundary discretisation, but this can reduce robustness.

The boundary discretisation quality is important, especially in 3D, where a poor triangulation of the boundary surface can cause very poor 3D meshes. The boundary discretisation can be set up in physical space, or the parameter space of the curved segments.

In physical space the parameterisation is effectively the arc length. First compute the length l of each segment (usually the CAD package can provide this information) and interpolate the mesh parameters onto each segment, at some large number of sampling points. Then calculate the number of sides N_s required along each segment as

$$A_s = \int_0^l \frac{1}{\delta} dl \qquad N_s = \text{nint}(A_s)$$

here assuming no stretching. Node positions are at arc lengths l_i where

$$\frac{N_s}{A_s} \int_0^{l_i} \frac{1}{\delta} dl$$

takes integer values.

A more general solution is to transform the mesh parameters from physical space to the CAD parameter space, creating a mesh in the parameter space using formulae similar to those used above, and then transforming back to get the points on the boundary.

In 3D the boundary surfaces themselves need to be triangulated to form the initial front in 3D. Each boundary surface will be represented in the CAD system used to create the geometry as a mapping from the plane with some parameterisation (u, v), either as a concatenation of surface patches (Bezier, Coons, etc) or a a cubic spline surface with a rectangular grid of control points. Even in the case of solid geometry CAD models a surface representation can be generated. This background surface will intersect other boundary surfaces to give the edges of the volume (represented by cubic splines, say). Then the 3D curved surface triangulation is achieved by doing a 2D triangulation (see below) in the (u, v) parameter space, and transforming to the 3D surface. The edges of the 3D surface can be discretised using the arc length, as in the 2D case, and the generated points transformed to the (u, v) plane. The mesh parameters also have to be transformed from 3D to the 2D parameter space to control the mesh generation in the parameter space.

Once the boundary is discretised we have our initial front and element generation can proceed. We describe one particular algorithm due to Peraire *et al.* [21]

Choose a front side for the base of the new element, usually the smallest side, call it AB (Figure 20). This requires a search through the segments on the front, which can be speeded up by using suitable data structures (heap list ordered by segment length) although there is then an overhead in maintaining these structures. In the 3D case the size of the front can easily be 10,000 triangles and in this case (and generally, see below) suitable data structures are essential for acceptable performance. Also in 3D, choosing the smallest segment may not be robust. A better choice is the smallest value of |(desired area based on δ) - (actual area of segment)|.

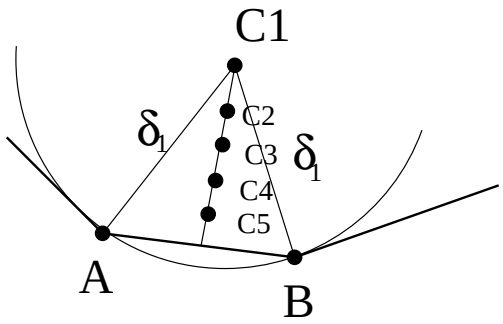

Figure 20: New element, initial choice

Interpolate the mesh parameters to midpoint of AB. Again this requires a search, this time of the background mesh. An *alternating digital tree (ADT)* [23] or region

quadtree/octree [24] can be used for the elements of the background grid, allowing proximity searches in $O(\log N)$ time.

Then find the position of the 'ideal' point C1 as shown in Figure 20, with δ_1 given by

$$\delta_1 = \begin{cases} 0.55AB & \delta < 0.55AB \\ \delta & 0.55AB < \delta < 2AB \\ 2AB & 2AB < \delta \end{cases}$$

These inequalities ensure that no elements with excessive distortion are produced; the coefficients are empirical. In 3D, δ_1 is the distance from the nodes of the segment (triangle) and average side length is used instead of AB. To prevent very small tetrahedra being formed from small or distorted triangles, a lower limit of $\delta/2 \le \delta_1$ is used.

Now, this new element could lead to poor quality triangles being generated in later steps (Figure 21), so the all nodes on the current front which are within radius δ_1 of C1 have to be found. Using an ADT or quadtree/octree for the nodes/sides of the front will speed up the search to $O(\log N)$ (compared with $O(\sqrt{N})$ in 2D).

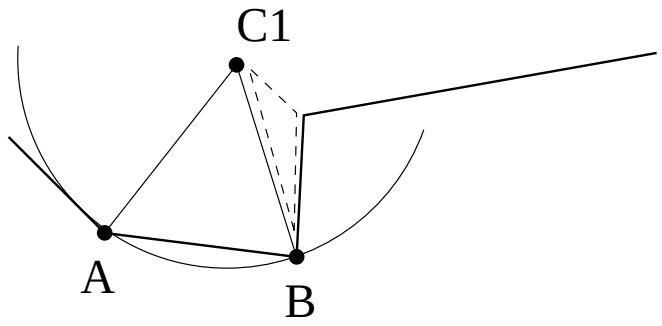

Figure 21: Poor choice of a new element will lead to distorted elements being produced later.

Other values for this search radius (e.g. $5AB$) have been used to increase robustness. In 3D, the search radius should be small for efficiency, but this doesn't appear to affect the quality or robustness (if you can't get a good tetrahedron form points close-by, you're unlikely to get a better one from points far away).

Order these nodes by distance from C1, and add C1, C2, C3, C4, C5 (see Figure 20) at the end of the list. In 3D, the list ordering strongly affects tetrahedron quality, and a suitable choice is to order by distance from centroid of the chosen segment (triangle) of the front. The points C2,..., C5 are to ensure you can form *some* triangle. In 3D, there is a new node approximately every 5 tetrahedra, and the optimal node C1 is usually OK, so only a few alternatives (C2,..., C5) are needed.

The required point is then the first one (N_i) for which the triangle ABN_i has positive area (i.e., is on the correct side of the front) and the sides AN_i and BN_i do not cut any side of the current front. Checking for side-side intersection requires another search through the front (using an ADT or quadtree/octree for the front will speed this up). In 3D the actual checking for each side can be very time consuming. The check for intersection is: 2 faces don't intersect if no side of either face intersects the other face, and all side / face combinations need to be tested. Typically there are about 40 close faces to be checked and using a simple intersection check can lead to 80% of the total grid generation time being used in the checking phase. Checking bounding box/tetra-

hedron intersection first to reduce the number of faces requiring the full side/face checking can reduce this time to 25%.

A variation of the method [25][26] inserts points (on an approximately regular grid) in the domain first, then advances the front using these points only. The points are well spaced to give good triangles. This reduces the searching required and removes almost all intersection checking.

The element generation can fail if there are large variations in mesh size, and the search radius is too small (see Figure 22), hence the need for C2,..., C5.

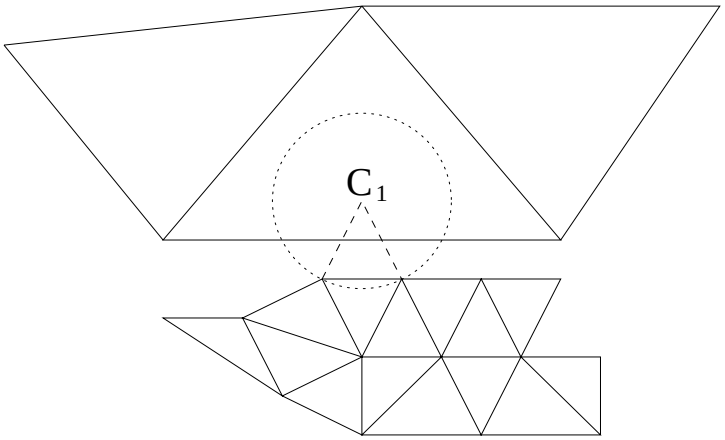

Figure 22: Element generation failure (side intersection) because of too small a value of search radius.

Once a valid element has been found, record the new element and update the front by inserting or deleting sides. The algorithm ends when the front is empty.

In 3D, clusters of ill-shaped segments can cause element generation to fail (locally) and the solution is to keep a 'temporarily rejected segments' queue. Many of the rejected segments are deleted as the front moves while they are in the rejected queue. Unfortunately, if these segments are at the last stage of the mesh generation, then there may be no good segments to use. In this case, the front needs to be back-tracked by erasing some elements/segments and the generation restarted with looser geometry tolerances.

Performance

Since this is a greedy triangulation the only active data is on the front, and the storage requirements in 2D are $O(\sqrt{N})$ and the time complexity is at most $O(N\sqrt{N})$. Most of the time is spent in searching and suitable data-structures can reduce this. Two suitable structures are the *alternating digital tree* (*ADT*) and the (region) quadtree/octree as discussed above. Using these data structures reduces the time complexity to $O(N\log N)$. In 3D the time complexity could be $O(N\log N)$ as well, but in practice this is more like $O(N^{1.2}\log N)$, because the octree used for searching the front for 'nearby' points is often unbalanced. If the front advances randomly, the octree will be well balanced, but if it advances sequentially, the octree will become unbalanced. In general the performance will depend on the desired grid size, stretching, and domain shape.

Summary

In the advancing front method nodes and elements are added simultaneously, with smooth mesh point placement. The method is boundary conforming (the initial front is the boundary). It can fail when there are large variations in grid spacing. The need to search for nearby nodes and edges/faces and the complex intersection checking

mean that the advancing front method is slow. In addition, it is better to add points rather than elements (2 triangles per point in 2D, 5 tetrahedra per point in 3D, on average). The Delaunay triangulation method treated in the next section adds points, has less searching to perform, and has much simpler geometry checking. It is much faster than the advancing front method and is the preferred method in 3D.

3.2.4 Delaunay triangulation

The Delaunay triangulation of a set of points has a well developed theory [27] (here triangulation includes 'tetrahedralisation' in 3D). The techniques used to generate the triangulation can obviously be used to generate unstructured meshes.

Given a set of discrete points in the plane, a Voronoi polygon about point P is the region that is closer to, or as close to, P as any other point. The Voronoi or Dirichlet tesselation is made up of these Voronoi polygons and the Delaunay triangulation is the dual of the Dirichlet tesselation. Figure 23 illustrates these concepts.

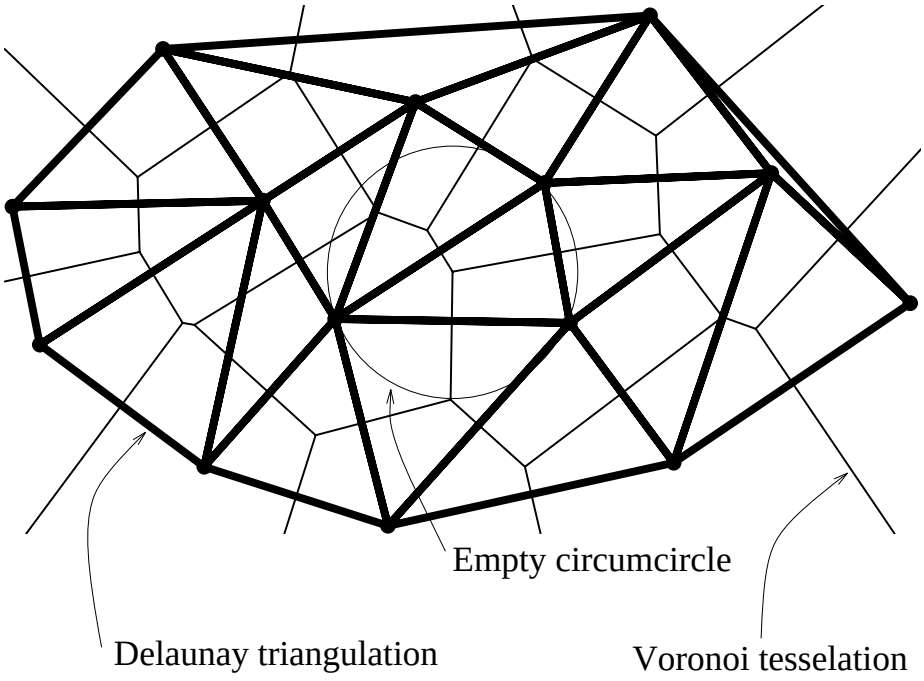

Figure 23: Delaunay triangulation and its dual, the Voronoi or Dirichlet tesselation.

The Delaunay triangulation has two properties that are useful in mesh generation:

- No point is contained in the circumcircle (see Figure 23) of any triangle. This 'empty circumcircle' property is used in several Delaunay triangulation algorithms. This property holds in all dimensions; in 3D 'circumcircle of any triangle' is replaced with 'circumsphere of any tetrahedron'.
- In 2D only, of all triangulations, the Delaunay triangulation maximises the minimum angle for all triangular elements. Note that what is required for good quality finite elements is to minimise the maximum angle, but in practical mesh generators (which generate the points as well as the triangulation) the elements generated are of good quality. Unfortunately, this max-min angle property is lost in 3D (and higher dimensions), where very poor quality elements (*slivers*, Figure

24) can be formed.

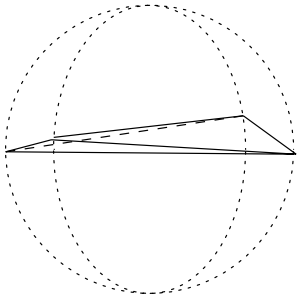

Figure 24: Sliver tetrahedron

These can be detected and they and the surrounding tetrahedra can be re-triangulated, but the resulting triangulation is no longer Delaunay.

The Delaunay triangulation is unique except for degenerate distributions of points. In 2D, 4 points on a circle (Figure 25),

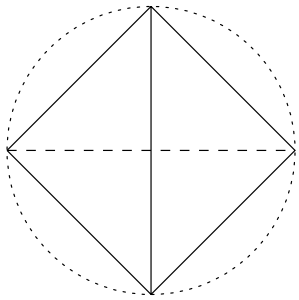

Figure 25: Degenerate points, 2D

in 3D, 5 points on a sphere. In 3D, 6 points (octahedron) on a sphere can give an invalid mesh (Figure 26)

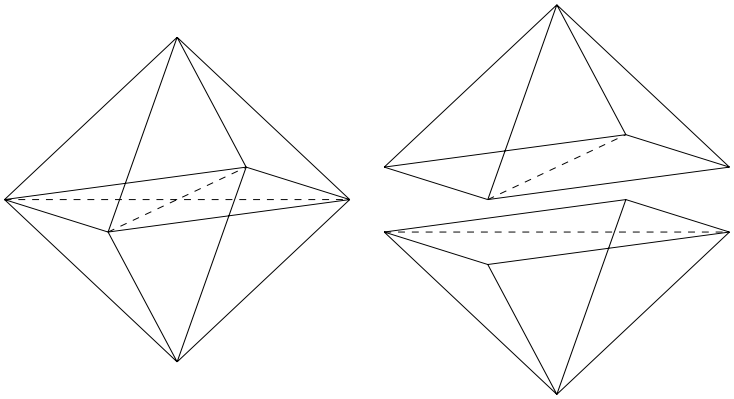

Figure 26: Degenerate points, 3D

There are many algorithms for Delaunay triangulation [27], mainly in 2D, and the asymptotic worst case complexity is $O(N\log N)$ in 2D. However, the most used practical algorithm, applicable in any dimension, is the Bowyer-Watson algorithm [28][29]. This adds points sequentially into an existing Delaunay triangulation, usually starting from a very simple triangulation (e.g., one large triangle) enclosing all the points to be triangulated. The algorithm proceeds as follows

- Add a point to the triangulation

- Find all existing triangles whose circumcircle contains the new point (Figure 27). First the triangle which contains the new point has to be found, which is a proximity search taking time $O(\log N)$ for a suitable data structure (e.g., quadtree / octree). Then the neighbours of this triangle are searched and then their neighbours, etc., until no more neighbours have the new point in their circumcircle. This takes, on average, time $O(1)$ but in the worst case the circumcircles of *all* the existing triangles contain the new point, taking time $O(N)$.
- Delete these triangles, which creates (always) a convex cavity.
- Join the new point to all the vertices on the boundary of the cavity (Figure 27).

For 3D replace 'triangle' with 'tetrahedron'.

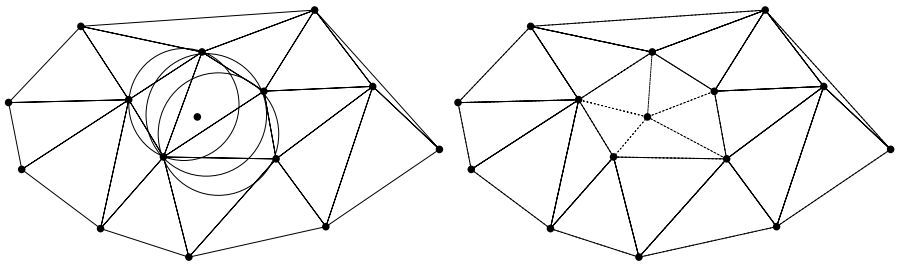

Figure 27: Bowyer-Watson triangulation: circumcircles that contain the new point, and the resulting triangulation

Performance

This has worst case complexity N^2 in 2D, but near $O(N)$ performance has been reported for mesh generation applications in both 2D and 3D. The poor worst case performance can be avoided by randomising the order of point insertion.

Point generation

Like all Delaunay triangulation methods, the Bowyer-Watson algorithm assumes that the points to be triangulated are already known, and so only addresses half of the mesh generation problem. The points can be pre-generated, say from the vertices of overlapping structured grids about component parts of the boundary, with some filtering and smoothing of the point distribution to get good triangulations. However, the most useful method is to generate the points simultaneously with the triangulation, choosing the points to improve the quality of the triangles [30]. Quality is defined by (e.g.) size of circumcircle relative to the locally required value interpolated from a background grid. This eliminates large and ill-shaped triangles, and large and (most) ill-shaped tetrahedra. The algorithm proceeds as follows

- Form an initial triangulation of a box containing the domain (if exterior domain, can use a bounding circle / sphere).
- Insert the boundary points into this triangulation using the Bowyer-Watson algorithm.
- Order the triangles by their quality, worst first
- Get the first triangle in the list and insert a new point at its *circumcentre*. This results in triangulations (eventually) with angles bounded between 30° and 120° in 2D; there is no such guarantee in 3D.
- Retriangulate using Bowyer-Watson. Since the chosen triangle will be in the set of triangles to be deleted, the 'search for containing triangle' can be skipped. However, the insertion / deletion of triangle into the list sorted by quality is $O(\log N)$ so there is no overall saving in time.

- Insert the new triangles in the quality list, if they are still not good enough.

This method has produced very efficient mesh generation code. The mesh is less smooth than the advancing front method, but the generation is about 10 times faster. There can be robustness problems, especially in the initial phase when there are highly distorted elements. A variation, which produces smoother grids, is to generate the interior points by the advancing front method [31].

Boundaries

So far, little regard has been taken of the boundaries (compare the advancing front method where the boundary is the initial front). The Delaunay construction triangulates a set of points, and does not necessarily conform to imposed boundary, i.e., fixed edges (Figure 28)

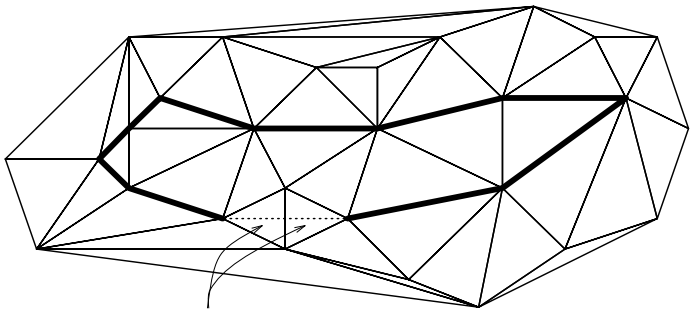

Figure 28: Delaunay triangulation not conforming to a boundary

There are two ways to force the Delaunay triangulation to include the boundary properly. In 2D is possible to define a *constrained Delaunay triangulation* [32]. In a constrained Delaunay triangulation the pre-defined edges are in the triangulation and the empty circumcircle property is modified to apply only to points that can be 'seen' from at least one node of the triangle, where the pre-defined edges are treated as opaque. The existence of the constrained Delaunay triangulation guarantees the possibility of modifying a Delaunay triangulation to include the boundary edges (via edge-swapping, Figure 29, or more extensive re-triangulation).

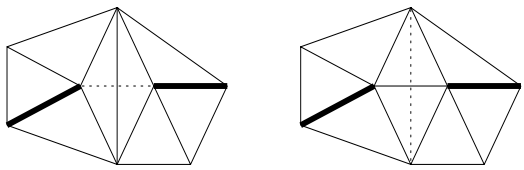

Figure 29: Edge swapping to give a constrained Delaunay triangulation

In 3D, there appears to be no concept of constrained Delaunay triangulation. It is possible to recover the boundary edges and faces by using face-edge swapping (Figure

30) or more extensive re-triangulation, but generally extra points need to be inserted. The resulting triangulation is no longer Delaunay.

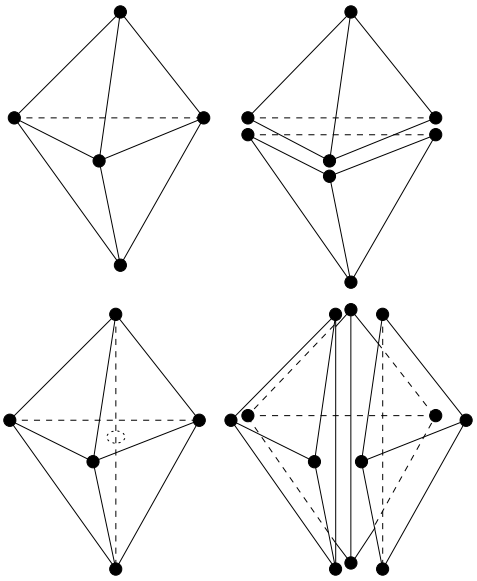

Figure 30: Face-edge swapping

The alternative to constrained triangulation is to make the Delaunay triangulation conform to the boundary curve / surface (rather than the boundary discretisation) by adding enough points on the boundary to ensure that the Delaunay triangulation will include the (small) edges on the surface. Often this leads to elements that are much smaller than required for the finite element error tolerance, but this technique applies in all dimensions.

The triangulation can be made boundary conforming either just after the boundary points have been triangulated or at the end of the Delaunay triangulation process. In the latter case the (unconstrained) Delaunay triangulation can introduce points near the eventual boundary, leading to poor quality meshes when the boundary edges / surfaces are recovered. Making the triangulation boundary conforming just after the boundary points have been triangulated leads to better quality meshes near the boundaries. In this case the boundary integrity needs to be maintained throughout the triangulation process. This is possible in 2D using constrained Delaunay triangulation, but in 3D the mesh will not be Delaunay after boundary recovery, and subsequent Delaunay-type point insertions may fail.

In 2D the boundaries can be discretised using methods similar to those used for the advancing front method. In 3D, the boundary surfaces can be (Delaunay) triangulated in the flat 2D parameter space of the surfaces, as in the advancing front method.

Summary

Delaunay triangulation is the fastest method for unstructured mesh generation but boundary conformance needs to be checked / maintained. In 3D, poor quality elements can be generated and these need to be corrected by local re-triangulation at the end of the main generation phase.

3.2.5 Other methods

A variety of other methods have been developed for unstructured mesh generation but they all seem to be still at the ‘research’ stage, with no widespread usage. The following is really just a list of references to these methods.

Paving [33]

This is the quadrilateral / hexahedral version of the advancing front method. The algorithm fills the domain by advancing layers of quadrilaterals / hexahedra. Complex collision rules are needed when fronts merge, but the method gives very good elements at the boundaries.

Whisker weaving/spatial twist continuum [34]

This is an interesting concept, but there appear to be difficulties with implementation.

Hybrid methods

A number of methods combine structured meshes near the boundary and unstructured meshes elsewhere (see [31] for a review). This give the advantages of boundary-fitted meshes without the difficulties of multiblock. Related to these approaches are the advancing layer version of the advancing front method and the advancing normal method of Delaunay point insertion, both of which give highly structured meshes near the boundaries.

Other

A variety of other methods are discussed in Ho-Le's review [35].

3.2.6 Smoothing

The meshes generated by automatic methods are often improved by adjusting the point positions to give a smoother mesh. The usual method is *Laplacian smoothing*

$$\vec{X}_i^{new} = \frac{1}{n_i} \sum_{j=1}^{n_i} \vec{X}_j$$

where $\vec{X}_i^{new}$ is the new position of point i and the $\vec{X}_j$ are the coordinates of the n_i points connected to point i. Each point is moved to the barycentre of the points connected to it and after 2-5 iterations the process converges. Only minor irregularities can be smoothed, and in some cases (Figure 31) the smoothing can give negative area / volume elements which need to be corrected ('un-smoothed').

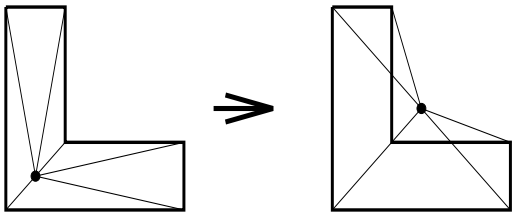

Figure 31: Failure of Laplacian smoothing

4 Adaptive Meshing

This chapter is a very brief introduction to *adaptive meshes* (see, for example, [31], [36], [37]).

4.1 Adaptive meshes

In the previous chapters we have dealt with techniques to generate structured and unstructured meshes, given some specified grid density. This grid density is chosen to give acceptable solution accuracy or acceptable solution time. It is possible to specify certain aspects of the grid at the start of the computation, e.g., high grid density near sharp features of the boundary, but the solution may also develop sharp features (e.g., shocks in compressible aerodynamics) which cannot be predicted in advance. At these sharp solution features the grid will need to be refined to give acceptable accuracy. In contrast, there can be large areas of the domain in which the solution is very smooth, where the grid can be coarsened and still lead to acceptable accuracy, with a decrease in solution time. So using a sequence of meshes that adapt to the solution as it develops in time (or by iteration for steady-state simulations) can lead to acceptably accurate results in optimum time.

There are two parts to an adaptive mesh scheme.

- Measurement of the local error and translation to refinement / coarsening. The measure of error should be based on some technique like Richardson extrapolation, but in practice the solution curvature (e.g., difference between solution and smoothed solution) or solution gradient (e.g., at shocks) are used. The required local grid density can then be calculated using the measured / required error and the local order of the discretisation method. The required distribution of grid density should equi-distribute the error.
- Re-generating the mesh with the specified grid density distribution. The method used depends on whether the problem is steady-state or time-dependent. For steady-state problems a small number of optimum adaptations (e.g., mesh regeneration) tend to be used. For time-dependent problems, a large number of adaptations are required as the solution changes. The adaptive meshing must be fast, and it must be capable of refinement and coarsening (e.g., as fronts move through the domain). The interpolation from mesh to mesh must be accurate in the time-dependent case; for steady-state problems this is less important.

  There are three methods for refinement: mesh point movement, local refinement, and re-meshing the complete domain. (An alternative to changing the grid density is to change the local order of the discretisation method, '*p*-refinement'.)

  - Mesh movement (*r*-refinement) maintains the original mesh connectivity, but moves the mesh nodes. This method can only cope with relatively small adjustments without introducing very distorted cells, but it allows refinement / coarsening and is fast. It is very suitable for structured meshes, which require a fixed connectivity, and also for time-dependent problems, where speed and the ability to refine *and* coarsen are important.

- Local refinement (*h*-refinement) sub-divides grid cells [38]. Surrounding cells will also have to have some degree of subdivision to maintain a valid mesh (e.g., in 2D, no nodes on triangle edges). This method is fast, allows coarsening by reversing the sub-division, and enables accurate interpolation between the nested meshes. It is ideal for time-dependent problems. The mesh connectivity is changed, so this method is only suitable for unstructured meshes.
- Mesh generation with new density parameters. This is the most general method, but it is slow, and there is no simple way to reverse the refinement. With Delaunay triangulators, extra points can be added to the existing triangulation to refine the mesh.

4.2 Parallel mesh generation

Adaptive mesh generation leads naturally to mesh generation in parallel: when the solution method is implemented on parallel computers, evidently the adaptive meshing needs to be implemented in parallel as well. Later revisions of this document will include an overview of parallel meshing techniques; more information can be found in [39] for structured meshes and [36], [40] for unstructured meshes.

5 References

General references: mesh generation conferences, computational geometry conferences, numerical methods journals with papers on mesh generation.

[1] R. Schneiders, Mesh Generation & Grid Generation on the Web, http://www-users.informatik.rwth-aachen.de/~roberts/meshgeneration.html

[2] Numerical grid generation in computational fluid mechanics (Editors: S. Sengupta *et al.*), Swansea, Pineridge Press, 1988 (2nd Conference)

[3] Numerical grid generation in computational fluid mechanics and related fields (Editors: A. S.-Arcilla *et al.*), New York, North-Holland, 1991 (3rd Conference, Barcelona)

[4] Numerical grid generation in computational fluid mechanics and related fields (Editors: N.P. Weatherill *et al.*), Swansea, Pineridge Press, 1994 (4th Conference, Swansea)

[5] Proceedings of the ACM Symposia in Computational Geometry

[6] International Journal for Numerical Methods in Engineering

[7] Journal of Computational Physics

Finite difference, finite volume and finite element methods

[8] EPCC, Introduction to Computational Science

[9] C.A.J. Fletcher, Computational techniques for fluid dynamics Vol. 1, 2nd ed., Berlin, Springer-Verlag, 1991

[10] Claes Johnson, Numerical solution of partial differential equations by the finite element method, Cambridge, CUP, 1987

[11] I. Babuska and A.K. Aziz, On the angle condition in the finite element method, SIAM Journal on Numerical Analysis, 13, 214-226 (1976)

Structured meshes

[12] J.F. Thompson, Z.U.A. Warsi, and C.W. Mastin, Numerical grid generation: Foundations and applications, New York, North-Holland/Elsevier, 1985

[13] P. Knupp, and S. Steinberg, Fundamentals of grid generation, Boca Raton, CRC Press, 1993. See http://math.unm.edu/~stanly/me/Fortran.shar for updated code.

[14] http://www.nas.nasa.gov/~melton/cartesian.html

[15] G. Chesshire and W.D. Henshaw, Composite overlapping meshes for the solutions of partial differential equations, Journal of Computational Physics 90, 1-64 (1990). This introduces the (free) CMPGRD grid generation program available from ftp.c3.lanl.gov

[16] G. Chesshire and W.D. Henshaw, A scheme for conservative interpolation on overlapping grids, SIAM journal on scientific and statistical computing 15(4), 819-845 (1994)

[17] P.L. George, Automatic mesh generation: application to finite element methods, Wiley, 1991

[18] O.C. Zienkiewicz and D.V. Phillips, An automatic mesh generation scheme for plane and curved surfaces by 'isoparametric' co-ordinates, International Journal for Numerical Methods in Engineering, 3, 519-528 (1971)

[19] P.L. Baehmann *et al.*, Robust, geometrically based, automatic two-dimensional mesh generation, International Journal for Numerical Methods in Engineering, 24, 1043-1048 (1987)

[20] M.A. Yerry and M.S. Shephard, Automatic three-dimensional mesh generation by the modified-octree technique, International Journal for Numerical Methods in Engineering, 20, 1965-1990 (1984)

[21] J. Peraire, M. Vahdati, K. Morgan and O.C. Zienkiewicz, Adaptive remeshing for compressible flow calculations, Journal of Computational Physics, 72, 449-466 (1987)

[22] P. Moller and P. Hansbo, On advancing front mesh generation in 3 dimensions, International Journal for Numerical Methods in Engineering, 38, 3551-3569 (1995)

[23] Javier Bonet and Jaime Peraire, An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems, International Journal for Numerical Methods in Engineering, 31, 1-17, (1991)

[24] H. Samet, The design and analysis of spatial data structures, Reading, Mass., Addison-Wesley, 1989

[25] R.D. Shaw and R.G. Pitchen, Modifications to the Suhara-Fukuda method of network generation, International Journal for Numerical Methods in Engineering, 12, 93-99 (1978)

[26] S.H. Lo, A new mesh generation scheme for arbitrary planar domains, International Journal for Numerical Methods in Engineering, 21, 1403-1426 (1985)

[27] J. O'Rourke, Computational geometry in C, Cambridge, Cambridge University Press, 1994

[28] A. Bowyer, Computing Dirichlet tessellations, The Computer Journal, 24, 162-166 (1981)

[29] D.F. Watson, Computing the *n*-dimensional Delaunay tesselation with application to Voronoi polytopes, The Computer Journal, 24, 167-172 (1981)

[30] D.G. Holmes and D.D. Snyder, The generation of unstructured triangular meshes using Delaunay triangulation, in Numerical grid generation in computational fluid mechanics (Editors: S. Sengupta *et al.*), 643-652, Swansea, Pineridge Press, 1988

[31] D.J. Mavriplis, Unstructured mesh generation and adaptivity, ICASE Report No. 95-26, NASA, 1995

[32] L.P. Chew, Constrained Delaunay triangulations, Proceedings of the 3rd Symposium on Computational Geometry, 215-222, ACM Press, 1987

[33] T.D. Blacker and M.B. Stephenson, Paving: a new approach to automated quadrilateral mesh generation, International Journal for Numerical Methods in Engineering, 32, 811-847 (1991)

[34] S. Benzley *et al.*, Hexahedral mesh generation via the dual, Proceedings of the 11th Symposium on Computational Geometry, C4-C5, Vancouver, ACM Press, 1995

[35] K. Ho-Le, Finite element mesh generation methods: a review and classification, Computer Aided Design, 20, 27-38 (1988)

[36] J.K. Wilson and B.H.V Topping, Parallel adaptive tetrahedral mesh generation, Developments in Computational Techniques for Structural Engineering (Editor: B.H.V. Topping), 403-426, Edinburgh, CIVIL-COMP Press, 1995

[37] Grid Adaptation in Computational PDEs: Theory and Applications, 1st-5th July 1996, Edinburgh, Scotland, http://www.ma.hw.ac.uk/icms/apde/

[38] M.C. Rivara, Algorithms for refining triangular grids suitable for adaptive and multigrid methods, International Journal for Numerical Methods in Engineering, 20, 745-756 (1984)

[39] J. Hauser *et al.*, Parallel computing in aerospace using multi-block grids. Pert 1: Application to grid generation, Concurrency: Practice and Experience, 4, 357-376 (1992)

[40] R. Williams, DIME: Distributed Irregular Mesh Environment, ftp://ftp.ccsf.caltech.edu/dime/, 1990