3D Anisotropic Grid Generation with Intersection-Based Geometry Interface

Ilja Schmelzer *
IAAS,
Mohrenstr. 39
D-10117 Berlin

November 29, 1993

Abstract

In this paper we present a new interface for geometry description. This interface is based on four intersection routines. It allows to use geometry description without explicit boundary description, for example using characteristic functions and boolean operations. This seams to be useful especially for time dependent geometries. We show how to create and change this geometry description, and how to use this interface for grid generation. We discuss grid quality criteria and possibilities to create and manage anisotropic grids. At last we describe an algorithm for 3D grid generation based on this interface which was implemented by the author. This algorithm combines an octree-like refinement process, shift of nodes to the boundary and Delaunay grid generation.

1 Introduction

Usually a grid generator requires the full information about the boundary grid, especially a list of all boundary vertices, boundary lines and boundary

*This research was supported in part by the Institute for Mathemetics and it's Applications with funds provided by the National Science Foundation.

faces and a description of each of them. Such a boundary description can be created, for example, by standard CAD systems. But there are situations where it is easier or more natural to define a geometry in another way, for example using a characteristic function which is greater zero inside and later zero outside the region. Then we have no list of boundary parts.

We define here an alternative interface for the geometry description which makes it possible to use such incomplete geometry descriptions. We describe how the interface can be created and manipulated for different types of input data. Our geometry interface is reduced compared with the standard CAD geometry descriptions. For example, we have no lists of regions, boundary faces, edges and vertices. Therefore topological errors cannot be excluded in general. We discuss possibilities to avoid such errors.

Another problem of grid generation is *anisotropic refinement*. Most of grid generators for unstructured grids allows local refinement, but create only isotropically refined grids. But especially for 3D problems anisotropic refinement can essentially reduce the number of nodes in the grid. We discuss some problems connected with anisotropic grids, especially grid quality measures. Unfortunately, many usual techniques for isotropical grids cannot be used for anisotropic grid generation.

At last we describe an algorithm for 3D anisotropic grid generation based on our alternative geometry interface. The algorithm starts with an octree-like refinement process. Then nodes near the boundary will be shifted to the boundary. Then we create the Delaunay grid for the resulting point set. At last, some corrections near the boundary are necessary.

The algorithm is implemented in ANSI C. The speed is approximately linear, with 3-5 ms per node CPU time on a VAX 4000/90 workstation.

2 problems of usual geometry description concepts

For a specific, fixed geometrical configuration a geometry description can be generated by usual CAD systems. But such CAD systems cannot handle a geometry changing in time. This may be caused by mechanical deformation or a chemical reaction on a surface. Then the geometry description has to be changed automatically at every time step. The available input data may

be very different. There may be a (scalar or direction dependent) reaction rate on a boundary grid, a concentration in a region, a velocity field and so on. This usually has to be done by the application programmer, because only he knows the velocities, reaction rates and so on which are necessary to compute the new boundary position.

At first let's consider the process of changing the "standard" geometry description. This geometry description contains:

- a list of all geometrical objects, that means regions, boundary faces, boundary lines and boundary vertices.
- a set of coincidence and neighbourhood relations between the geometrical objects.
- a description of every boundary object. This will be usually a discretization of this object (boundary grid for a face, polygon for an edge, coordinates of a vertex). In CAD systems also other possibilities (spline approximations or analytical definitions) are usual, but we don't consider them here.

Assume we have computed some velocity field on the boundary grid. Then we shift the boundary grid. It is easy to see that we obtain a lot of problems:

caustics: Also if the initial grid and the velocity field are smooth, the shifted grid often will not be smooth. There may be caustics. A typical caustic in 2D consists of two "turning points" and one self-intersection. But there may be also other types of caustics. In 3D caustics are very complex global objects. Instead of the "turning points" in 2D we have now lines with difficult global behaviour. It is necessary to detect caustics and to eliminate them. But already in 2D this is a difficult task. Often only caustics with self-intersections will be detected, but already in the case of non-constant isotropic reaction rate there may be caustics without self-intersections.

intersections: There may be also other intersections of the shifted boundary grid, may be with other boundary grids, but also self-intersections. These intersections also have to be detected, but they cannot be eliminated, but require a change of the topology. This requires a very accurate classification of the intersection, because every failure leads to an

obsolete data structure or global errors. Especially we have to consider rounding errors. In 2D we have to find an intersection point, in 3D we have a global intersection line which may intersect other boundary lines or caustics. The intersecting parts are obviously in the same space region, but usually "far away" in the data structure of the boundary grid. To make the test fast enough we have to use search tree structures.

We see, that changing the standard geometry description is a very difficult. So, the usual method is to make small time steps to avoid caustics and not to consider topological changes.

So it seams natural to look for alternatives, which allow big time steps and topological changes. An example of such an alternative is a "characteristic function" of a region which is positive in the region and negative outside. Often it is easier to define such a characteristic function for the new region (for example a minimal distance to a boundary) as to compute the new boundary grid. Sometimes such a characteristic function may be natural for the given physical problem (some critical concentration).

The natural generalization for the case of more than two different regions is the "region-function" — a function defining the region a point with given coordinates lies in. In some sense this seams to be the minimal geometry description — if this function cannot be defined, the geometry is not correctly defined.

On the other hand, using this minimal interface we also obtain a lot of problems:

- the problem of finding the correct boundary now has the grid generator. Obviously, using this geometry description you also cannot avoid topological errors, if your grid is not fine enough to detect very small regions and so on.

- For time dependent geometry it is necessary to find the position of the boundary with small error. Otherwise, especially for small time steps, the error in the velocity of the boundary becomes very big. If we have only the region-function, we need an iteration process to compute the boundary position — not an ideal solution.

- It is not possible to transfer in this way boundary data — for example different boundary types, boundary charge distributions and so on.

- special boundary lines cannot be detected — edges and vertices usually will be "rounded" by the grid generator.

3 definition of the interface

The interface we consider here is based on the second concept — a function defining the region for given coordinates. But we have modified the concept so that it is possible to transfer more information through the interface.

In some sense this interface can be considered as the maximal interface that can be approximated by the minimal interface. So, if you have a geometry defined by the minimal interface, you can use it as the input of the grid generator. But if you have more detailed information about the geometry, you have the possibility to transfer some of this information through the interface or to use faster and more accurate algorithms for the interface.

Since in this concept it is possible to use the minimal interface as input, the main problem of this interface — the possibility of topological errors — cannot be completely solved. But we obtain new possibilities to avoid such errors.

So the starting point of the interface we want to define is the region-function:

```
int Region(float x[DIM]);
```

returning a descriptor of the region the point with the coordinates x lies in. This descriptor in principle may be a pointer to some data, but we prefer to use an integer value enumerating the regions.

3.1 the node data type

The first question we have to consider is the allocation and computation of other nodal data (function values) which are necessary in the process of grid generation. For example, assume, that some function values are necessary for refinement criteria. Than it is necessary to compute and store them in the process of grid generation. The storage allocation has to be done by the grid generator ("knowing" the current node number), but the data have to be computed by the geometry description ("knowing" the data on the old grid).

So, node functions can (and have to) be interpolated in the process of grid generation, and the natural way is to use the region-function to interpolate these data. Then they have to be transferred to the refinement criteria. The grid generator only has to allocate storage for the data and to transfer the data. We have realized this using an abstract node data type containing all information about the node:

```
typedef struct{ float x[DIM], region dom, ... } Node;
```

So the region-function now has the form

```
int Region(Node *node);
```

3.2 data for the grid search algorithm

Another change in the interface was done to allow a fast implementation of the region-function for a geometry defined by a grid (for example the grid of the previous time step).

Let us consider now algorithms finding a point with given coordinates in a grid. There are two interesting algorithms for this search:

- The quadtree/octree method. A special search data structure (called quadtree in 2D, octree in 3D) has to be created to search elements near the given point. For every node we have $O(\log n)$ operation for a grid with n nodes.
- The neighbourhood search. Beginning with some start element we test if the point is inside the element. If this test fails because the nodes lies behind a side of the element we go to the neighbour element at this side.

The efficiency of the neighbourhood search depends of the start element we use. It seams the best to start with the element containing the nearest point we have found before. Then the efficiency of the algorithm depends on the point order. As an example consider a regular refined grid with the same density for the old and the new grid. Then the nodes of the highest refinement level usually have a direct neighbour of a lower level which was created before. The "distance of one neighbour" needs a fixed time (This will be true also for local refined grids if the density of the old and the

new grid are nearly the same). For lower level nodes we have a two times bigger distance to the nearest node created before. For a regular, isotropically refined rectangluar grid we can explicitely compute this time. For an 1D grid we have $n/2$ finest level nodes with distance 1, $n/4$ nodes with distance 2, $n/8$ nodes with distance 4 and so on. The result is an $n \log n$ time behaviour for the full grid. For a 2D or 3D grid we can have a better behaviour if we use a good refinement order. Then we will have (for 2D) $3n/4$ nodes of the finest level with distance 1, $3n/16$ nodes of the first coarser level with distance 2 and so on with resulting linear behaviour.

That means, we have a chance to get a linear behaviour using the neighbourhood search algorithm with a good start element. But to realize this we have to change the interface definition to organize the transfer of the information about the nearest previously created node (which is available only for the grid generator) and the element it was found in (which is an object of the old grid and hidden from the grid generator by the region-function). To do this we include the pointer to the data of the nearest previously found point into the region-function call:

```
int Region(Node *nnew, Node *nold);
```

Into the Node data structure we include the descriptor of the element where we have found the node (usually an integer number). The region-function now has to write the result of the search into the Node structure and can read it from nearest. For the first node nold will be NULL.

3.3 the position of boundary faces

Consider now the problem of exact definition of the boundary position. For time dependent problems we need the boundary position with high accuracy. That's why it is not enough to define only the region for regular grid nodes and to consider grid nodes near the boundary as the boundary nodes. If we have found an edge with different region numbers at the ends we have to compute the intersection of this edge with a boundary.

In principle it is possible to do this using the bisection algorithm and the region-function. Depending of the accuracy we need we have a fixed number of region-function calls. The number of edges intersecting boundaries is small compared with the number of nodes especially if the grid is fine, so this is time consuming compared with other parts of the algorithm only for

coarse grids. But usually there are also better algorithms possible depending of the realization of the geometry description. Especially we can get more information about the intersection — special boundary types, function values defined on the boundary and so on. Therefore it is useful to include a special function into the interface defining the intersection with boundary faces:

```
int Face(Node *nint, Node *n1, Node *n2);
```

Input are the node data for the ends of the edge n_1 and n_2. The output data are parts of the Node structure of the "intersection node" n_{int}, especially the coordinates and the boundary face identifier. n_{int} is defined as the first intersection with a boundary with the line from n_1 to n_2.

We have a default implementation using the bisection algorithm with iterative calls of the region-function. So it is not necessary to implement this function. But this implementation is not ideal. For example, we may not find the first intersection point if there are many, we obtain only an approximation and cannot transfer boundary data. So for some interesting input data we have special implementations:

- For the search in a grid there is an analogon of the neighbourhood search — the neighbourhood search along the edge. We find here the exact position, really the first intersection with the boundary face, the algorithm will be faster, and we have a possibility to interpolate non-trivial boundary data from the grid.

- For a change of the geometry with a characteristic function we use linear interpolation to find the boundary face intersection — the zero value of the characteristic function. This algorithm will be usually faster and allows higher accuracy, especially if the function is approximately linear. Therefore we recommend to use characteristic functions with nearly linear behaviour near the border.

3.4 the position of boundary lines

The same problem as for the boundary face position we have also for the position of boundary lines. For many applications a good approximation for the position of edges is necessary. It is possible to compute the position of an boundary line by an iteration process of calls of the region- and face-function, but for special geometry data there are algorithms which will be faster, more

accurate and allow the transfer of other data. Therefore it seams useful to include another function defining the position of boundary lines into the interface:

```
int Line( Node *nint, Node *nface, Node *n1, Node *n2, Node
*n3);
```

The input data are the three nodes of a triangle $n_1, ..., n_3$ and an intersection n_{face} of the line from n_1 to n_2 with a boundary face. This intersection is assumed to be the result of a previous call of the face- or line-function. There are two possible results:

- There may be other intersections of this boundary face with the border of the triangle. Than n_{int} is the first of these intersections if we go along the intersection curve of the boundary face with the triangle.
- There may be no other intersections of this boundary face with the border of the triangle. That means, the intersection curve of the border with the triangle ends inside the triangle. Than n_{int} is this end — the intersection of a boundary line with the triangle.

We also have implemented a default algorithm (an analogon of the bisection algorithm for the boundary faces) using calls of the region- and the face-function (So using a fast face-function makes also this algorithm faster). In the following we describe this algorithm:

At first we "go around" the triangle from the given intersection n_{face} to the next intersections with a boundary using face-calls. If we find an intersection with the same face number we return this intersection. This is the analogon of "comparing the sign" of the bisection algorithm. Here we may return an incorrect result if there are different intersections. If the triangle is smaller than the necessary accuracy we return the middle point of the triangle. Else we divide the triangle into four parts and call recursively the line-function for the small triangle containing the given intersection n_{face}. If the output is an intersection of the border with the outside or an inner intersection with an boundary line we return this intersection. Else we go to the neighbour and call the line-function for this neighbour with the output of the previous call as input. It is possible to get an infinite cycle (since the line-function can make errors). So we have to break down this loop after a given number of steps. Than we return the last (inner) border intersection

we have found as an inner intersection with an edge. That means we also can get erroneous boundary lines dividing the border into different parts using this algorithm.

Because of the possible errors of this algorithm it seams useful to have special implementations, especially for grids, there it is possible to find the exact solution.

3.5 the position of boundary vertices

The same has to be done to define the position of boundary vertices. We have included the following call into the interface:

```
int Vertex (Node *nint, Node *nline, Node *n1, Node *n2,
Node *n3, Node *n4);
```

Input are the four nodes of a tetrahedron and an intersection n_{line} of the first side (n_1, n_2, n_3, n_4) with an edge. Output n_{int} are the data of the next intersection of this boundary line with the border of the tetrahedron or the vertex — the end of the boundary line — inside the tetrahedron. An equivalent default implementation calling region-, face- and line-function is available. A special (fast and exact) realization for a grid is also possible (but at the current state not implemented).

3.6 degenerate cases

For all of the previous function we have defined their behaviour only for the case of "general position". But what we have to do in degenerate cases?

There aren't any special output conventions for degenerate cases. If the answer is not unique, you have to give one of the possible answers. For example if the input point of the region-function lies on a boundary, every region containing this boundary can be used. But you have to be sure that you obtain in this way a consistent picture. So if you have returned different region numbers for two points, you must be able to return a boundary face intersection for this edge. Considering such degenerate cases and problems caused by rounding errors is the most difficult part of the implementation of such an interface.

3.7 coincidence functions

Using this algorithm we cannot avoid topological errors, if the grid density defined by the refinement criteria is to small to detect all regions. But it is possible to do very much to avoid such errors testing the coincidence of different geometrical objects. Assume, for example, we have a very thin layer which intersects a grid edge. Using the face-function we have found one of the two intersections of the face of the layer with the edge. How can we find the other? If we have some test of the coincidence of the boundary face number and the region numbers of the ends of the grid edge we can easily detect that there is a problem and use further refinement to solve it. There will be a lot of other possible errors which can be detected using such coincidence information. So it seams useful to include such tests into the interface:

```
int TestRF(int region, int face);

int TestRL(int region, int line);

int TestRV(int region, int vertex);

int TestFF(int face, int face);

int TestFL(int face, int line);

int TestFV(int face, int vertex);

int TestLL(int line, int line);

int TestLV(int line, int vertex);

int TestVV(int vertex, int vertex);
```

with the possible answers 0 - no, 1 - yes, 2 - unknown.

In the case of the minimal interface we have a priori no boundary face numbers. But for every boundary face point we have found by the bisection algorithm we know the two (different) region numbers of the nearest points we have considered. We can use these region numbers to create a default boundary face number. Using this number we can get a nontrivial implementation of the TestRF- and TestFF-functions.

Remark that for a boundary face there must not be defined a unique left and right region.

3.8 making recursive calls possible

We have considered possibilities to describe a geometry, for example given by a grid or by a characteristic function. But for time-dependent problems we need a possibility to change a given geometry. This also can be done using a characteristic function for the part which has to be changed. Assume that we have defined such a function. To compute the result of the region-function we have to compute the result of the function and the result of the old region-function. So we can have recursive calls of the region-function. There may be recursive calls also for the other functions. It seams not good to transfer the parameters of the geometry using global variables. Therefore we include an abstract pointer to "user data" as the first parameter into each function call.

3.9 conclusion

So we have defined a new interface for the geometry description. It consists of the region-function, three boundary intersection functions and nine test functions.

In our implementation we have included also service functions, for example a free-function to release the storage occupied by the parameters if the geometry is no longer necessary:

```
void Free(void *user-data);
```

Other useful functions may be save and load (currently not implemented).

There is a lot of information usually available for the grid generator which is not included into the interface and therefore not available for our grid generator:

- the number and lists of identifiers of the regions, faces, lines and vertices.
- explicit discretizations of the faces and lines, coordinates of the vertices.
- other derived information, for example curvature and so on.

4 realizations of the interface

Let's consider now possibilities to realize this interface for different input data. Some of these realizations we have already considered in the previous considerations:

- We have considered the approximation of the interface if only a region-function is given.
- We have considered the case of a complete grid. In this case the neighbourhood search algorithm seams to be the best way to realize the region function. There are also fast and exact realizations of the three intersection functions.
- We have considered the usage of characteristic functions, especially as a method to change a given geometry.
- Characteristic functions can also be used to subdivide boundary faces and lines. This method can be used to avoid the "rounding off" of lines and corners, because after the subdivision of a face or edge the exact position of the edge or corner dividing the face resp. edge can be computed by the grid generator using the edge resp. corner function.
- A trivial geometry (only one region, no boundary) can be used as a start for further changes.
- It is easy to define "boolean operations" (union, intersection) not only for regions, but also for boundary faces and lines.
- It is possible to use graphical input data (pixel maps) to define a region-function.
- For time-dependent problems the following algorithm can be used: Start with the grid of the previous time step. Then compute the characteristic function (or necessary data for the computation of this function) on this grid. Then use the region-function for a geometry defined by a grid to interpolate the characteristic function for the nodes of the new grid. At last make the change-operation using this characteristic function.

To guarantee that this algorithm works the old region-function will be called in the change-operation before the characteristic function will be evaluated.

- It is also possible to realize this interface for the "standard" geometry descriptions.

  If the geometry is described by a complex boundary grid fast search algorithms have to be used to make the realization efficient. For example, search trees (quadtree/octree) can be created. Another possibility is to create the Delaunay grid containing the boundary nodes and use the neighbourhood search algorithm described before.

We see, that it is possible to realize the interface for all usual types of input data for a geometry description. With this interface it is possible to create even infinite geometries (for example Julia sets)!

4.1 How to use this interface

Because a lot of possible information about the geometry is not visible for the grid generator it is not possible to guarantee the topological correctness of the resulting grid. This can be easily proved considering the case of a very small enclave. Since there is no information about the existence of this enclave and there may be no grid point inside also in the finest grid we cannot even detect that we have made an error.

But there is a class of geometries there it is possible to construct the geometry without errors. These are geometries where all objects are convex:

Theorem 1 *Assume we have a geometry with a finite number of objects (regions, faces, lines and vertices) so that all these objects are closed convex sets, no interior point of an object* [1] *lies in another object of the same or higher dimension, and there is an exact implementation of the interface for this geometry. Then there is an algorithm which can reconstruct the geometry in a given bounded region exactly using only the first four functions.*

[1] Interior point for an object means interior in the subset topology of the union of all objects with the same dimension

The principal way the algorithm works is clear: In the first step we define the region number of all nodes of some grid containing the region of interest using the region-function. Then there can be defined lines with different region numbers for the end points. For these lines we use the face-function to define the first boundary face intersection. Using the inverse order for the end points we can proof that there is only one intersection. Else we make further refinement of the edge. This process stops because of the finite number of faces and the convexity (so there cannot be two intersections of the edge with the same face). Now we consider the sides. For every intersection of their border with a boundary face we use the line-function to find the continuation. If there are different inner intersections with boundary lines we use further refinement of the side. Then for every inner intersection point we subdivide the side connecting the point all boundary face nodes on the border of the side by a line. If we have two boundary face nodes of the same face on the border we also subdivide the side connecting these points by a line. At last we do the same for elements and the vertex-function. [2]

This theorem shows a way we can avoid errors: We can subdivide objects so that the resulting topology is more convex. For example, if we have a thin layer inside a region, this layer may not be found. Subdividing the region into two part (over and under the layer) makes it possible to detect the layer. Subdividing a face along a sharp edge or an edge at a corner makes it possible to find the exact position of the edge or the corner.

In general it is not possible to avoid geometrical and topological errors. But the topological errors are always local, and they can be reduced by further refinement. For many applications this may be not good enough, because also a little topological error can have global consequences. But in the applications we have in mind — the simulation of processes which may change the topology — it is clear that the topology is defined only modulo the accuracy of the simulation itself. So for these applications our concept seams to be even saver than the standard method, where every error can have global consequences.

[2] For an accurate proof we need a more accurate definition of the geometry and the interface, especially we have to consider degenerate cases in detail.

5 Grid Quality Criteria

There are a lot of different criteria to measure the quality of a grid. This seams to be natural, because there are different applications and discretizations, and a grid may be good for one but bad for another case. So for different equations and discretizations there have to be different quality measures for the grid. From this point of view it makes no sense to introduce a criterion independent of an equation or discretization.

Consider, for example, the case of the diffusion equation with nonconstant diffusion coefficient. The function and the diffusion coefficient are defined on the nodes of the grid. Now we have different possibilities for the discretization. Let's consider the following:

1. the FEM discretization (with lumping of the mass matrix).
2. the FV (finite volume or box) discretization based on the Voronoi boxes, the diffusion coefficient will be considered as constant on elements, it's value computed by averaging over the nodes of the element.
3. the same FV method, but now the diffusion coefficient will be considered as constant on the influence region of the edges and computed by averaging over the two nodes of the edge.

For the diffusion problem we have to avoid negative concentrations. So a stable discretization is necessary. In this case it is possible to find conditions so that there will be no negative concentrations (they are sufficient for the M-matrix property of the resulting discretization matrix). Consider at first the 2D case. For our three schemes we obtain the following conditions:

1. a grid without obtuse angles.
2. a grid without obtuse angles.
3. a Delaunay grid without obtuse angles opposite to a boundary edge.

So, in 2D for a given node set the Delaunay grid will be optimal for all of these methods. In the 3D case we have the follwoing picture:

1. For the FEM discretization we obtain a criterion which may be fulfilled for non-Delaunay grids and not fulfilled for the correspondent Delaunay grid.

2. For the FV method we need a Delaunay grid so that the centre of the minimal ball containing the element must be inside the element.

3. In this variant we need a Delaunay grid so that the centre of the minimal ball containing the element must be inside the region.

From point of view of grid generation it is easy to guarantee only the Delaunay property, because there is a standard algorithm to construct a Delaunay grid for a given point set. There is no algorithm known to find the optimal grid for the FEM method for a given point set in 3D. The standard method is to start with the Delaunay grid and to use local transformation and additional point inclusion. An extreme example for the difference between these criteria is the so-called sliver — a highly degenerated tetrahedron, but without obtuse angles on it's sides. Such slivers are very bad from point of view of the FEM criterion, but they may be optimal from the Delaunay criterion.

In our grid generation algorithm we create a Delaunay grid and try to avoid situations which are bad especially from point of view of the FV method, because the chance to get a good grid from this point of view is much higher. Slivers can occur in our grids, and we recommend to use FV methods instead of destroying the slivers.

6 Anisotropy

Anisotropic grid generation techniques are useful to minimize the number of nodes. There are applications where anisotropic grid generation is obviously necessary because of the different length order in different directions. For example in environmental processes we have often different scales in horizontal and vertical direction. Another example are thin boundary layers in viscous fluids.

But anisotropic refinement is useful not only for such special applications. In principle, anisotropic refinement can be used every time. In the worst case there may be no reduction of the node number. But this worst case

not often appears. Usually in almost every application you have locally a "gradient direction" which has to be refined better than the orthogonal directions. The other point is that the reduction effect is much greater in 3D, because we have usually two "orthogonal" directions which must not be refined. If we obtain a factor 2 in a 2D problem, we may obtain a factor 4 in the similar 3D problem. So anisotropic techniques seams to be useful for every 3D application.

grid quality criteria for anisotropic grids: As in general, in the case of anisotropic refinement we have to consider our problem and our discretization to formulate the correct criterion. For example, consider a problem with anisotropic diffusion coefficients. Then the optimal grid may be obtained by replacing the Delaunay criterion by an anisotropic analogon — it depends on the discretization we want to use. But usually the anisotropy is not caused by the equation and discretization, but by the special situation (geometry, initial values, boundary conditions) we have to consider. So the quality measure we have to use coincide with isotropic case.

So we create a Delaunay grid and try to fulfill the additional conditions for the FV criteria also in the anisotropic situation.

6.1 problems with anisotropic Delaunay grids

There are a lot of specific problems in anisotropic Delaunay grid generation. The reason is the following *alignment property:* In a highly anisotropic Delaunay grid the nodes have to be aligned in the direction of "greatest refinement". This is easy to see. In a highly degenerated triangle the only possibility to avoid big obtuse angles is to have two nearly right angles. But this means that the shortest edge lies in the direction of high refinement. The same consideration can be made also in 3D.

This effect creates a lot of problems, because we cannot use standard methods which do not preserve the alignment property:

point insertion: inserting a point into a Delaunay grid is a standard method to solve different problems:

- local refinement
- boundary correction (divide edges going through the boundary)

- destroy slivers.

But if the new point is not "aligned" the resulting grid is very bad. We obtain elements which are too big or non-Delaunay elements.

node shifting: this method often will be used near the boundary — an inner node near the boundary can easily create bad elements near the boundary, and so it may be better to shift this node to the boundary. But here we also have to consider the alignment. So we can shift the node only in alignment direction.

grid smoothing: standard grid smoothing procedures cannot be used, because nodes may be shifted away from the "alignment curve".

We see, that our possibilities to manage an anisotropic Delaunay grid are restricted. We have to modify these methods so that they preserve the alignment or we cannot use them.

7 the algorithm

The main steps of the algorithm are

- An octree-like anisotropic refinement controlled by application dependent criteria.
- A boundary shift procedure. Intersections of the boundary with the octree will be computed, nodes will be shifted to these intersections.
- A Delaunay step computes the resulting Delaunay grid for the point set created by the previous steps.
- A boundary correction step detects and corrects incompatibilities of the resulting grid with the geometry description.

Let us consider now these steps in detail.

7.1 refinement

We start with a modification of the anisotropic quadtree/octree refinement procedure. This is a standard algorithm starting with one element (a quader in 2D, a cuboid in 3D). Then external criteria will be used to refine the elements. A tree of elements will be created. In the isotropical variant, a quader will be splitted into four and a cuboid into eight parts. In the anisotropic variant an anisotropic criterion is necessary, telling in which direction the element has to be splitted. The element will be splitted into two parts. Octree techniques are considered for example in [1], [2], the anisotropic variant in [5].

The main difference between the standard quadtree/octree method and our method is that we refine not the elements but their edges. This is a little bit more flexible. So in the standard octree method in the highest level we have at minimum four nodes, in our method there may be one. We also use a different data structure. Our structure is based on nodes. For every node we have the six (four in 2D) neighbour nodes in the orthogonal directions (minimum two of them must be defined). The cuboids play only a secondary role, and we don't use a tree structure. We have included them into our data structure only to make some search operations faster. In 2D we don't use the quaders.

The refinement criteria we use are also based not on elements, but on nodes and edges. We use the following criteria:

- a application dependent function defining the maximal length of an edge around a given point (isotropical criterion).
- a application dependent function defining if a given edge has to be refined (anisotropic criterion).
- parameter-controlled regularization criteria.

Some minimal regularization property is necessary — direct neighbour edges in the same direction must have the same or the neighbour refinement level. This is automatically guaranteed by the edge refinement procedure — a neighbour edge of a coarser level will be refined by a recursive call of the refinement procedure before the edge itself will be refined. After this refinement the other parameter-controlled regularization criteria will be

tested, and some of the neighbours with the same refinement level as the refined edge may be also refined. The time complexity of the refinement algorithm is $O(n)$.

Because we refine only edges in coordinate directions we obtain automatically an alignment in these directions. If one of these directions approximately coincides with the "highest refinement" direction we can obtain very anisotropic grids. For a skew highest refinement direction the resulting grid will be more isotropic. So, if the highest refinement directions are equally distributed you will have parts with isotropical refinement and parts with high anisotropic refinement. But also in this case we can obtain approximately a factor 2 in the 2D case and factor 3 in 3D for the number of nodes compared with pure isotropical refinement.

7.2 boundary shift

In the first step only inner nodes of the regions have been created. Now we have to create the boundary nodes. One possibility to do this (for example [3]) is to compute the intersections of the edges with the boundary faces and of the rectangle sides with boundary lines and the cuboids containing boundary vertices and to include these points into the grid. [3] We use the following modification of this method: instead of including a new node at the intersection we shift an existing grid node to this place, and we don't consider all of the intersections. This gives us more freedom for manipulation to obtain a good grid quality:

- The usual "point insertion" can be considered as a special combination — refinement of an edge and shift of the new node. So we don't lose anything.
- We have not so many inner nodes with low distance to the boundary. Such nodes are very dangerous from point of view of our grid quality criteria.

[3] Another posibility often used — to include the nodes of the given boundary grid into the grid — cannot be used because we have no information about this discretization in our interface. This method also fails for anisotropic grid generation because of the misalignment problem.

- Because of the alignment problem it is not good to include intersections with "long" edges into an anisotropic grid.

The algorithm in [3] starts with vertex insertion and ends with face insertion. We have to use the inverse order — we start with the faces and end with the vertices — because of our geometry interface. At first we consider edges with different regions for the end nodes. Using the face-function of the geometry description we compute the intersection with the boundary. Then we shift the nearest node to this place.

In anisotropic situations we shift only in the "short" direction to avoid misalignment. If it seams necessary to shift into other directions we prefer to make regularization refinement.

There are situations where further refinement seams to be necessary. For example, if we detect incorrect topological situations using the coincidence tests of the geometry description. Regularization refinement is also often useful. If nodes have to be shifted into a coarse refinement direction we make the grid nearly isotropical to avoid the misalignment effect.

For all newly created nodes the refinement criteria of the previous steps also have to be tested, and further refinement may be necessary. So we have to consider the question how to refine an edge with shifted ends. We use the unshifted position of the edge to compute the coordinates of the new node. It is possible that the new node has to be shifted to the boundary instead of the end of the edge.

In the second step for rectangles with irregular number of intersections of their border with boundary faces the line-function of the geometry interface will be called. If we have found an intersection of the rectangle with a boundary line a node of this rectangle will be shifted to this place. This step is also connected with possible regularization refinement.

At last the same has to be done with cuboids with incorrect intersection numbers using the corner function of the interface. If we have found a corner we shift a node to this corner. Around the corner we have to use isotropical refinement, so further regularization refinement is necessary.

7.3 Delaunay grid generation

In 2D it may be possible to construct the resulting Delaunay grid "ad hoc" starting with the given quadtree structure. But in 3D there are too many

different possibilities for refined edges and sides of a cuboid, and every nodes of the cuboid may be shifted in different directions. That's why we use a modification of the Watson algorithm [7] for the resulting point set to create the Delaunay grid. [4]

We use some of the information contained in the octree structure to make the algorithm faster. The speed of the Delaunay algorithm depends on the following things:

The point set: In the worst case, for n points we have $O(n^2)$ tetrahedra in the resulting Delaunay grid. So the procedure has maximal speed $O(n^2)$ in this case. But in our case the resulting grid will be very regular, so that we have $O(n)$ tetrahedra in the resulting grid.

The point order: If we use a bad point order, there may be very much tetrahedra we have to delete and create in every step. In our algorithm we use the order of creation. So the speed of the algorithm depends on the order of operations in the previous steps.

The search of a tetrahedron which has to be deleted. Here we use a neighbourhood search algorithm to find the tetrahedron containing the point. This search is very fast because we can find a good start using information about the refinement process. For every refined node we save one of the "father" nodes. This father node is already inserted into the Delaunay grid because of the insertion order we use. Starting from a tetrahedron containing this node we find the interesting tetrahedron very fast.

The resulting algorithm is approximately $O(n)$. In computations on a VAX 4000/90 workstation we need 2 - 3 ms for every node for the Delaunay step. For the complete grid generation we need 4 - 5 ms for every node. So in the current state it is the most time-consuming part of the algorithm.

[4] An alternative may be the method described in [2] to make the Delaunay procedure local for every cuboid. This seams to be faster than the full Delaunay algorithm. But in our method it is very difficult to connect the Delaunay grids of neighbour cuboids: The corners of the cuboids may be shifted, and so there may be a nontrivial intersection of their convex hulls.

7.4 boundary correction

A well-known problem of Delaunay grid generation is that the resulting Delaunay grid is not compatible with the boundary description. For example, there may be tetrahedra in the resulting grid which intersect the boundary. In such cases a correction is necessary. The first problem is to detect such situations. There will be different types of errors. The first are obvious inconsistencies of the resulting grid, for example if the two end nodes of an edge have different region numbers. The other are correct grids with incorrect topology. With our interface it is not possible to detect the second sort of errors. This can be done only for the standard interface, for example comparing topological invariants.

If an error was detected, we have to make a correction. There are two possibilities to make this correction:

1. We can modify the grid using swapping procedures. The resulting grid will be a restricted Delaunay grid, but it is not good enough to guarantee the additional property we want to have near the boundary (no obtuse angles looking on a boundary edge). Another problem is that local grid transformation is very difficult in 3D. We have to consider a lot of different swapping procedures (instead of one "edge swapping" procedure in 2D), and there are often situations there it is not possible to use them (see for example [9]).

2. We can include a new boundary point into the element or edge intersecting the boundary. This method seams to be the best for isotropical refinement. But in anisotropic situations this method doesn't work well (alignment problem).

So we have no ideal method for our correction. Therefore the best way is to avoid such situations in the first steps (refinement and boundary shift). Unfortunately this is not easy. In some situations it is also not natural. There are situations there it seams better to ignore a small obtuse angle instead of including a lot of new nodes into the grid. So we try to avoid such situations. If they occur, we try to use the first method, and if this also fails we use the second.

7.5 further development

The resulting algorithm allows fast (approximately linear) Delaunay grid generation with local, anisotropic refinement. The resulting grid quality is good inside the regions, but not so good around the boundary. In 2D the grid generator works well, obtuse angles we obtain usually only for special situations (thin layers, triple points), and for approximately convex geometries we have usually no topological errors. In 3D the situation looks not so good. Local geometrical and topological errors occur, especially around the boundary lines. But they don't lead to incorrect grid structures or program errors. So we have a stable algorithm generating 2D and 3D grids.

In the future we have to do a lot of work in 3D to avoid errors and to get better grid quality around thin layers and boundary lines.

To reduce the number of elements in the grid we want to consider other element types (cuboids, pyramids, prisms and other).

For real time-dependent problems it seams to be too expansive to create the complete grid in every time step. So it is necessary to think about incremental techniques — grid coarsening, node shifting, doing the refinement step only once for several time steps and so on.

To get a grid generator which creates automatically good anisotropic Delaunay grids in "skew" directions (with automatical detection of this direction) seams to be an open question especially for our geometry interface.

References

[1] **P.L. George:** *Automatic Mesh Generation* Masson, Paris 1991

[2] **W.J. Schroeder, M.S. Shepard:** *A combined octree/Delaunay method for fully automatic 3-d mesh generation* Int.J.Numer.Methods Eng., 29:37-55, 1990

[3] **M.S. Shepard, M.K. Georges:** *Automatic three-dimensional mesh generation by the finite octree technique* Int.J.Numer.Methods Eng., 32:709-749, 1991

[4] **N. Hitschfeld, P. Conti, W. Fichtner:** *Grid Generation for 3D Nonplanar Semiconductor Device Structures* Simulation of Semiconduc-

tor Devices and Processes Vol. 4 edited by: W. Fichtner, D. Aemmer Hartung-Gorre Verlag Konstanz 1991

[5] **P. Conti:** *Grid Generation for Three-dimensional Semiconductor Device Simulation* Hartung-Gorre Verlag Konstanz 1991

[6] **N. Hitschfeld:** *Grid Generation for Three-Dimensional Non-Rectangular Semiconductor Devices* Hartung-Gorre Verlag Konstanz 1993

[7] **A. Bowyer:** *Computing Dirichlet tesselations* The Computer Journal vol.24 nr.2,pp.162-166 (1981)

D.F. Watson: *Computing the n-dimensional Dirichlet tesselation with Application to Voronoi Polytopes* The Computer Journal vol.24 nr.2,pp.167-172 (1981)

[8] **R.A. Dwyer:** *Higher-Dimensional Voronoi Diagrams in Linear Expected Time* Discrete Comput Geom vol.6 nr.4,pp.343-367 (1991)

K. Mehlhorn, St. Meiser, C. O'Dunlaing: *On the Construction of Abstract Voronoi Diagrams* Discrete Comput Geom vol.6 nr.3,pp.211-224 (1991)

[9] **B. Joe:** *Three-Dimensional Triangulations from Local Transformations* SIAM on Scientific and Statistical Computing vol.10,nr.4,pp.718-741 (1989)

[10] **B. Chazelle,L. Palios:** *Triangulating a Nonconvex Polytop* Discrete Comput Geom vol.5 nr.5,pp.505-526 (1990)