Proceedings of ALGORITMY 2000
Conference on Scientific Computing, pp. 399–405

GRID GENERATION AND GEOMETRY DESCRIPTIOM WITH COG

ILJA SCHMELZER*

Abstract.
The aim of this paper is to describe the basic concepts of the grid generator COG. The most interesting aspects of this grid generation concept are the geometry description and the combination of anisotropic local refinement with an isotropic grid quality criterion – the Delaunay property.

1. Introduction. The grid generator COG [1] is a general purpose grid generator. It allows to create two- and three-dimensional simplicial grids for complex geometries.

The geometry description interface allows to define geometries of arbitrary complexity in a simple way. Implicit definitions of regions by conditions like $f(x) < 0$ for arbitrary functions $f(x)$, boolean operations, pixmaps, grids from previous steps and so on are possible and may be combined, other ideas can be easily implemented.

A necessary consequence is that the geometry description is weak in the sense that information which is often part of a geometry description is not available to the grid generator in this interface. This includes information like the number of regions and boundaries or any explicit description of the boundaries. This restricts possible grid generation algorithms.

COG uses octree-based techniques to solve this problem. These techniques allow local and anisotropic grids. This is important for the reduction of the number of grid nodes, which is a critical problem in 3D grid generation. The resulting grids are Delaunay grids.

At first, we give an introduction into the principles and concepts of the geometry description named “cogeometry”. Then, we consider the grid generation algorithms and some concepts related with anisotropic refinement.

2. Cogeometry. The abbreviation COG stands for “cogeometry”, which is itself an abbreviation for “contravariant geometry description”. The reason is that the geometry description is the most innovative part of COG, a part which makes the whole package different from many other grid generators.

If we want to use grid generation so model some real objects, we have, at first, only the object in reality. But the grid generator needs a description of this object – its geometry, its parts, their materials, boundary conditions and so on – given in a well-defined interface. We have designed such an interface, and we think that this interface has a lot of advantages with usual interfaces for geometry description. Especially it is easy to describe complex geometries in this inface.

2.1. Co- and contravariant objects. To explain our concept for the description of geometries let’s consider at first the terminus “contravariant”. It has been taken from a domain of pure mathematics – category theory. The basic idea of category theory is to formalize the notion of “natural”. Mathematicians like beautiful, natural objects, and they have observed that many of these natural objects have common formal properties, properties related with maps. This observation allows to define two classes of natural things – covariant and contravariant objects. Assume we

*WIAS Berlin

have a map $f : X \to Y$ between two objects (sets, spaces, manifolds or whatever else). Then there may be objects on X which have a natural image. That means, for every object o on X we have a well-defined image $f(o)$ on Y. Such objects are called covariant. Examples are points $x \in X$, curves $\gamma : \mathbb{R} \to X$, densities $d\mu$, homology groups $H_i(x)$. For other objects, there may be a natural preimage. That means, for every object o on Y there exists a unique preimage $f^{-1}(o)$ on X. These objects are called contravariant. Examples are functions $g : X \to \mathbb{R}$, differential forms, cohomology groups $H^i(X)$.

These images and preimages have to fulfill some natural axioms we don't want to list here. What we need in the following is only a general, philosophical idea – it is always useful to look if such natural image and preimage operations exist. And it is always better to prefer objects which have such image or preimage operations – because these are the "natural" objects.

2.2. Contravariant nature of geometry. Now, we want to apply this philosophy to geometry description. A geometry is, roughly speaking, a subdivision of some space X into different parts – regions. But this rough definition is already sufficient to understand that the natural operation is not the image, but the preimage. Indeed, if we have a subdivision of Y into parts and a rather general map $f : X \to Y$ there is a natural way to define such a similar subdivision of X: $x \in X$ is part of the preimage region $f^{-1}(A)$ if $f(x) \in Y$ is part of the region A. But there is no similar image operation. If we have a subdivision of X, there is no natural way to define such a subdivision of Y. There may be points in Y which are not part of the image of X, and other points may be part of the image of different regions. Thus, we conclude that a geometry is a contravariant object.

Now, let's consider the usual way to define a geometry. We start with vertices, then we have edges between them and surfaces. All these objects are described as maps into the space. For polygons and splines we have the same general scheme – edges are functions $f : I \to X$, faces are functions $f : I \otimes I \to X$. But these objects are not contravariant – instead, they are covariant. Thus, they are not natural for the description of a geometry, which is contravariant.

2.3. Contravariant geometry description. This consideration justifies the search for a contravariant geometry description – a possibility to describe a geometry with contravariant objects. Now, a simple way to describe a geometry in a covariant way is a "region function" $r : X \to D$ where D is the discrete set of different regions. For every point $x \in X$ the function $r(x)$ returns the (identifier of the) region containing x.

The problem of this simple interface is that it does not allow to describe important information which should be part of a complete geometry description – boundary conditions. It also does not allow to find exact boundary positions – the only way to detect boundaries would be iteration. Thus, we want a better interface which allows to describe different boundary conditions. And, once we have started to consider such boundary conditions and positions of boundary faces, it would be also interesting to describe boundary lines and vertices, or, in the general n-dimensional case, boundary parts of every dimension k with $0 \leq k \leq n$.

The basic idea of the generalization to arbitrary dimension k is the following: as in the case of the region function, we use a function. The input is, instead of the point, a simplex of dimension k. The function returns an intersection of a boundary of codimension k with this simplex. But the design of the interface is not as simple as it looks now. We have to take into account some important problems:

- How do we want to use the interface for grid generation?
- Assume we have an interface definition for the first $k-1$ codimensions. Is it possible to obtain an approximation of the boundary of codimension k using some generalization of the binary intersection algorithm?

In some sense, the interface was designed together with this approximation algorithm. The advantage of such a default algorithm is obvious: we obtain the possibility to implement a geometry having only the region function.

The resulting interface differs from our first guess in some interesting points:

- The idea of *continuation*. Assume an intersection of the border of a k-simplex with a boundary of codimension $k-1$ is given. In general, the intersection of this boundary (codimension $k-1$) with the whole simplex (dimension k) is a curve – the continuation of the intersection inside the simplex. We have given one of the ends. The other end of this continuation may be another intersection with the border of the simplex. The other possibility is that it ends inside the simplex, at the boundary of the boundary of codimension $k-1$ – which is a boundary of codimension k.
- The *symmetry* of this call. Partially, this call is already symmetric – if the continuation ends on the border of the simplex, we may call it again with this intersection and should obtain the original intersection. Now, we want to have it fully symmetric, so that we can start from the intersection inside and find the continuation on the border. But there are different codimension $k-1$ boundaries which end in the given codimension k boundary. Therefore, we have to remember also the direction of a $k-1$ boundary.
- The concept of a *flag*. Now, to remember one direction is not sufficient. If we start continuation inside the k-simplex and reach the border, we have to define not only the $k-1$ boundary intersection, but also a direction of a codimension $k-2$ boundary for further continuation. And so on for lower dimensions. Therefore we need a more complex data structure known in differential geometry as a *flag*. A k-flag is defined by a point and a sequence of k tangential vectors which should be non-degenerate. And the intersection k-flag has the properties that the point lies on a boundary of codimension k and the k vectors are tangential vectors for boundaries of codimension $0, \dots, k-1$.

This leads to a beautiful geometry interface with one call in each dimension: Input is a k-simplex and or a k-flag inside the simplex, or a $k-1$-flag on its border. Output is, again, or a k-flag inside the simplex, or a $k-1$-flag on its border. The call fulfills the symmetry property: if we use the output intersection as input with the same simplex, we obtain the input intersection as output.

2.4. Default implementation. Now, for this geometry interface we can now define a default implementation for all interface functions except the first – the original region function. This default implementation generalizes the bisection algorithm for a line into arbitrary dimension. Thus, we have given a simplex and a boundary intersection flag on its border, and have to find its continuation. Let's subdivide this algorithm into two parts: subdivision into smaller simplices and what to do with the smallest simplex.

The subdivision algorithm is easy. We subdivide the simplex in a regular way, find the small simplex which contains the starting flag on its boundary. Then we search for continuation through this small simplex, using a recursive call, or, if the simplex is already small enough, the default implementation described below. If we find the continuation on an inner border, we have to continue the search in the neighbour simplex. If we find a continuation on an outer border or an intersection of higher codimension inside the small simplex, we have found what we need.

Now let's consider the smallest simplex, where the recursion breaks. We can no longer use the continuation call for dimension k, but we have all continuation calls for lower dimensions. Now, instead of looking for continuation inside, we try to find the continuation on the border of the simplex, using the continuation calls for lower dimension. And, in the case that there is no such continuation, we take the center of the simplex as the approximation of an inner intersection.

2.5. Geometry-based functions. The definition of various functions (initial values, solutions and so on) is sometimes considered as separate from the geometry description. Sometimes this may be appropriate. But there is also a very interesting class of function which may be named geometry-based. Especially these are functions which are defined only on boundaries: surface densities, variable boundary conditions. Another important class are functions which are continuous inside the regions but have discontinuities on the boundary. A typical example is segregation: concentrations on different sides of the boundary are different.

At a first look it seems that during grid generation, we don't need functions. But some functions we need – refinement criteria. And the refinement criteria are a good example of a geometry-based function – different refinement in different regions is a reasonable choice. Thus, we have to include some management for these functions into the interface.

Fortunately, functions are contravariant too. As a consequence, these functions fit in a natural way into a contravariant geometry description. The idea of the extension of the interface is very simple – we extend only the functionality of the existing calls: all interface function have to compute the values of geometry-based functions for their output parameters. All what is necessary is to organize the storage for the function values. This allows fast prototyping: as long as we do not need such functions we do not have to implement anything. If we later implement functions, we do not have to change anything in the grid generator.

Especially interesting is the nice fit between boundary limits of discontinuous functions and flags. For discontinuous functions we do not have unique function values on the boundary points. There will be two values on boundary faces, but an unknown number on boundary edges. Instead, a boundary flag contains not only the boundary point, but also a direction into a region. This allows to define unique function values for boundary flags. This observation may be generalized for functions defined on boundary faces of codimension k with discontinuities on boundary faces of codimension $(k+1)$.

Another interesting point is the compatibility of this scheme, inclusive the definition of function values on a flag, with the default implementation algorithm. The calls of the region function in the approximation algorithm compute the function values close enough to the boundary and therefore approximate the boundary limit.

2.6. Applications. Now, once we have a default implementation for all higher order calls, all we need to define a cogeometry is the region function. This allows a fast implementation of almost every real geometry: solid modeling, boolean operations, CT pictures, pixmaps, geological profiles – to implement the region function is usually very easy. The "worst case" for the implementation is a classical geometry description by a boundary grid: in this case, we need a rather complex algorithm to define the region function.

For grid generation, another algorithm is much more important – the region function for a complete grid, for example the grid of the previous time step. For this purpose, we can use a neighbourhood search algorithm.

But, of course, the region function is not all. Often enough we have to handle boundary conditions and functions defined on the boundary. These things cannot be managed with the region function alone. We have to implement the second interface function too. This second function usually requires much more time for implementation. Nonetheless, it is usually not very hard to understand what has to be done. This is in some sense a philosophical thesis which seems impossible to prove in a strong way: once the interface itself is natural, there will be also natural algorithms to implement it at least for natural geometries.

For the higher order functions there is usually no necessity for a special implementation, because there is seldom a need for boundary conditions or functions on boundary edges and vertices.

2.7. Summary. As we have seen, for the description of a geometry and geometry -based functions the contravariant interface is a natural possibility. It allows fast prototyping strategies: for a simple prototype we have to implement only a single region function.

The interface has to be characterized as weak. Almost every imaginable geometry may be described in this way – even geometries with infinite complexity like Julia sets. A lot of information usually available (lists of regions, boundary faces and so on) is not available.

This makes it easy to define geometries, but hard to use the interface in grid generation.

3. Grid Generation. Now let's consider the algorithms used in COG for grid generation. But, at first, we have to make some remarks about our quality measures. Last not least, to create a grid is easy – the complicate thing is to create a good grid. Some grid generation techniques cannot be used simply because the information they need as input is not available. Once we have no surface grid in our geometry description, we cannot use the advancing front method to create the grid.

The basic idea is the classical octree method: we start with a cube, subdivide if necessary and define the regions containing the octree nodes. Then, we find intersections with boundary faces. For this purpose, we use the second interface function (face function) for grid lines with ends in different regions. Then we find inconsistencies on some rectangle we use the third interface function (edge function) to find the intersection of this rectangle with a boundary edge, and, last not least, use the last interface function (vertex function) to find boundary vertices in cubes.

3.1. Topology detection and convexity. The main problem is, of course, to obtain a grid with the correct topology using our weak interface. And we have to start with the remark that in principle, in the general case, it is unsolvable. Indeed, there may be an arbitrary small region inside another region. The only way to detect it would be to test a point inside this region. But we have no list of points we have to test for this purpose to find all regions. And this is not only a problem of our interface, but an intended property of the interface, because it allows to define geometries with methods where no such list is available.

A similar problem appears for other thin objects like thin channels or thin layers.

Nonetheless, in most applications this does not present a serious problem – if the locations of the regions are known, precautions may be used so prevent this. There are two prevention strategies: first, refinement. We can require sufficient local refinement in the critical domain. For this purpose we have to know the domain where the thin regions are located, and their size.

The other method is to avoid concave geometries by artificial subdivision of regions and boundaries. Even a very thin layer may be found by our algorithm if it is located on the boundary between two other regions: the face function will be called because the ends are in different regions, and it returns the intersection of the first region with our thin layer. We search for the continuation with another call of the face function and find the other boundary of the thin layer.

In a similar way, but already for a thin channel which intersects a rectangle and the edge function, we can detect thin channels if they are located between three different regions or two different regions with different boundary conditions on the two sides of the channel. This concept also works for small regions.

In principle, what we need for the algorithm to detect the geometry correctly is that the regions, boundary faces and edges are approximately convex.

3.2. Delaunay property. Once we have clarified how to use the cogeometry in the grid generator it is useful to consider some other questions: grid quality criteria.

Here we have to note that different applications possibly need different criteria. For diffusion equations tetrahedral grids are fine, but in mechanics hexahedral grids are preferred (tetrahedra are "too stiff"). The initial point for this grid generator have been diffusion equations, and we create, therefore, tetrahedra.

But the ideal grid depends also on more subtle things, like the discretization method. It is well-known (but not widely known) that the classical FEM and FVM methods for standard diffusion which have the same quality criteria in 2D (no obtuse angles $\rightarrow$ M-matrix property) give different criteria in 3D. FEM requires non-obtuse planar angles for the tetrahedra for the M-matrix property. Instead, FVM requires the Delaunay property. These criteria are different. The most important difference are so-called slivers – very thin tetrahedra, which have a very small volume and give very small terms with correct sign in the case of finite volumes, but very large terms with wrong sign in the case of FEM.

We prefer the FVM method and, therefore, prefer the Delaunay criterion. One advantage of the Delaunay criterion is that there are simple algorithms for Delaunay grid generation for a given point set, while there are no such simple algorithms to obtain FEM grids. We suggest to consider this as an argument in favour of the FVM method.

3.3. Local and anisotropic refinement. The main purpose of local and anisotropic refinement is node economy. We do not consider here anisotropic refinement because we have in mind applications with anisotropic material coefficients. Instead, the purpose is to have less points in the resulting grid. The problem itself is assumed to be isotropic, the grid quality criterion – Delaunay – is isotropic. Nonetheless, in the typical application we have directions where not much happens and other directions – gradient directions – with fast changes. To approximate such configurations on a grid with a given accuracy we do not need an isotropic grid. Instead, an anisotropic grid can describe the same situation with the same accuracy with much less nodes.

In 3D point economy remains to be an essential point, because we need a lot of nodes in 3D computations. Moreover, strategies for point economy are more effective in 3D: if we use local instead of global refinement in 3D which allows a much greater reduction of node numbers. The same holds for anisotropic refinement: If we have high refinement only in the gradient direction, we have node economy in two other directions (instead of one in 2D). Therefore, local anisotropic refinement is an important way to reduce the number of nodes which is usually neglected.

The octree algorithm used in COG allows local anisotropic refinement in a natural way – if the gradient direction approximately coinsides with one of the grid directions.

3.4. Local coordinates. But in may applications the gradient directions are skew. The original octree algorithm in this case reduces to isotropic refinement, it gives no serious advantage.

Now, for this purpose we have developed a variant of the algorithm which works with several local coordinates. In this variant, we have to define local coordinates in some subdomain so that the important gradient direction is approximately one of the coordinate directions. The usual octree algorithm is used to compute a set of points in these local coordinates. Then the point sets for the different local coordinates are combined. Here we have to define which point set has to be used in the intersection of different local coordinates. For the resulting point set we use the Delaunay algorithm to obtain the Delaunay grid.

In the current implementations the coordinates should be orthogonal. Only in this case we obtain optimal anisotropical refinement.

3.5. Summary. The grid generator COG allows to use geometry descriptions defined in the "cogeometry" interface to create a grid. In the general case, no warranty can be given that the topology is correctly described, but some reasonable methods are available to solve these problems.

COG creates Delaunay grids, which gives good numerical properties if FVM discretization is used.

An important feature of COG is that local and anisotropic grid refinement allows to reduce the number of nodes necessary to describe a given situation.

REFERENCES

[1] COG, http://www.wias-berlin.de/cog/index.html