

MESH SIZE FUNCTIONS FOR IMPLICIT GEOMETRIES AND PDE-BASED GRADIENT LIMITING

Per-Olof Persson

Dept. of Mathematics, Massachusetts Institute of Technology, persson@mit.edu

ABSTRACT

Mesh generation and mesh enhancement algorithms often require a mesh size function to specify the desired size of the elements. We present algorithms for automatic generation of a size function, discretized on a background grid, by using distance functions and numerical PDE solvers. The size function is adapted to the geometry, taking into account the local feature size and the boundary curvature. It also obeys a grading constraint that limits the size ratio of neighboring elements. We formulate the feature size in terms of the medial axis transform, and show how to compute it accurately from a distance function. We propose a new Gradient Limiting Equation for the mesh grading requirement, and we show how to solve it numerically with Hamilton-Jacobi solvers. We show examples of the techniques using Cartesian and unstructured background grids in 2-D and 3-D, and applications with numerical adaptation and mesh generation for images.

Keywords: mesh generation, size function, background grid, Hamilton-Jacobi, gradation control

1. INTRODUCTION

Unstructured mesh generators use varying element sizes to resolve fine features of the geometry but have a coarse grid where possible to reduce total mesh size. The element sizes can be described by a *mesh size function* $h(\mathbf{x})$ which is determined by many factors. At curved boundaries, $h(\mathbf{x})$ should be small to resolve the curvature. In regions with small local feature size (“narrow regions”), small elements have to be used to get well-shaped elements. In an adaptive solver, constraints on the mesh size are derived from an error estimator based on a numerical solution. In addition, $h(\mathbf{x})$ must satisfy any restrictions given by the user, such as specified sizes close to a point, a boundary, or a subdomain of the geometry. Finally, the ratio between the sizes of neighboring elements has to be limited, which corresponds to a constraint on the magnitude of $\nabla h(\mathbf{x})$.

In many mesh generation algorithms it is advantageous if an appropriate mesh size function $h(\mathbf{x})$ is known prior to computing the mesh. This includes the advancing front method [1], the paving method for

quadrilateral meshes [2], and smoothing-based mesh generators such as the one we proposed in [3],[4]. The popular Delaunay refinement algorithm [5], [6] typically does not need an explicit size function since good element sizing is implied from the quality bound, but higher quality meshes can be obtained with good a-priori size functions.

Many techniques have been proposed for automatic generation of mesh size functions, see [7], [8], [9]. A common solution is to represent the size function in a discretized form on a background grid and obtain the actual values of $h(\mathbf{x})$ by interpolation, as described in Section 2.1.

We present several new approaches for automatic generation of mesh size functions. We represent the geometry by its signed distance function (distance to the boundary). We compute the curvature and the medial axis directly from the distance function, and we propose a new skeletonization algorithm with subgrid accuracy. The gradient limiting constraint is expressed as the solution of our gradient limiting equation, a hyperbolic PDE which can be solved efficiently using fast

solvers.

2. DISCRETIZATION AND PROBLEM STATEMENT

We represent the mesh size function $h(\mathbf{x})$ approximately on a discretized grid. We store the function values at a finite set of points \mathbf{x}_i (node points) and use interpolation to approximate the function for arbitrary \mathbf{x} . These node points and their connectivities are part of the *background mesh* and below we discuss different options. We also describe briefly how to compute a discretized *signed distance function* for the geometry, which we will use in the calculation of local feature sizes in Section 4.

2.1 Background Meshes

The simplest background mesh is a Cartesian grid (Figure 1, top). The node points are located on a uniform grid, and the grid elements are rectangles in two dimensions and blocks in three dimensions. Interpolation is fast for Cartesian grids. For each point \mathbf{x} we find the enclosing rectangle and the local coordinates by a few scalar operations, and use bilinear interpolation within the rectangle.

This scheme is very simple to implement and the Cartesian grid is particularly good for implementing level set schemes and fast marching methods (see Section 2.2). However, if any part of the geometry needs small cells to be accurately resolved, the entire grid has to be refined. This combined with the fact that the number of node points grows quadratically with the resolution (cubically in three dimensions) makes the Cartesian background grid memory consuming for complex geometries.

An alternative is to use an adapted background grid, such as an octree structure (Figure 1, center). The cells are still rectangles or blocks like in the Cartesian grid, but their sizes vary across the region. Since high resolution is only needed close to the boundary (for the distance function), this gives an asymptotic memory requirement proportional to the length of the boundary curve (or the area of the boundary surface in three dimensions). The grid can also be adapted to the mesh size function to accurately resolve parts of the domain where $h(\mathbf{x})$ has large variations. The adapted grid is conveniently stored in an octree data structure, and the cell enclosing an arbitrary point \mathbf{x} is found in a time proportional to the logarithm of the number of cells.

A third possibility is to discretize using an arbitrary unstructured mesh (Figure 1, bottom). This provides the freedom of using varying resolution over the domain, and the asymptotic storage requirements

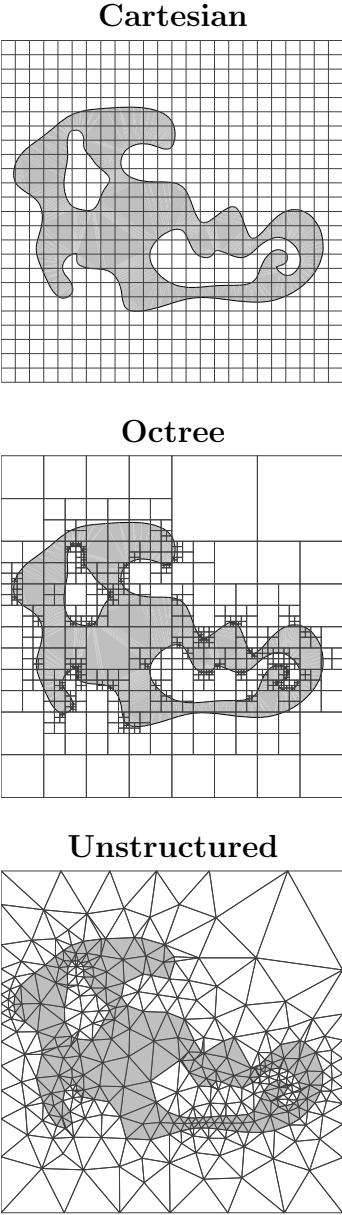


Figure 1: Background grids for discretization of the distance function and the mesh size function.

are similar to the octree grid. An additional advantage with unstructured meshes is that they can be aligned with the domain boundaries, making the distance function accurate and the curvature adaptation easier (see Section 3). An unstructured background mesh can be used to remesh an existing triangulation in order to refine, coarsen, or improve the element qualities (mesh smoothing). The unstructured background grid is also appropriate for moving meshes and numerical adaptation, where the mesh from the previ-

ous time step (or iteration) is used. Finding the triangle (or tetrahedron) enclosing an arbitrary point \mathbf{x} can still be done in logarithmic time, but the algorithm is slower and more complicated.

2.2 Initialization of the Distance Function

The signed distance function $\phi(\mathbf{x})$ for a geometry gives the shortest distance from \mathbf{x} to the boundary, with a negative sign inside the domain. The geometry boundary is given by $\phi(\mathbf{x}) = 0$, and the normal vector $\mathbf{n}(\mathbf{x}) = \nabla\phi(\mathbf{x})$. This representation is used in the *level set method* [10], where the boundary can be propagated in time by solving a Hamilton-Jacobi equation.

To initialize $\phi(\mathbf{x})$ on our background mesh, we compute the distances to the geometry boundary for the nodes in a narrow band around the boundary (typically a few node points wide). We then use the *Fast Marching Method* (Sethian [11], see also Tsitsiklis [12]) to calculate the distances at all the remaining node points. The computed values are considered “known values”, and their neighbors can be updated and inserted into a priority queue. The node with smallest unknown value is removed and its neighbors are updated and inserted into the queue. This is repeated until all node values are known, and the total computation requires $\mathcal{O}(n \log n)$ operations for n nodes.

If the geometry is given in a triangulated form, we have to compute signed distances to the triangles. For each triangle, we find a band of background grid nodes around the triangle (only a few nodes wide) and compute the distances explicitly. The sign can be computed using the normal vector, assuming the geometry is well resolved. The remaining nodes are again obtained with the fast marching method. We also mention the *closest point transform* by Mauch [13], which gives exact distance functions in the entire domain in linear time.

A general implicit function ϕ can be *reinitialized* to a distance function in several ways. Sussman et al [14] proposed integrating the reinitialization equation $\phi_t + \text{sign}(\phi)(|\nabla\phi| - 1) = 0$ for a short period of time. Another option is to explicitly compute the distances to the zero level set for nodes close to the boundary (e.g. using the approximate projections in [4]), and use the fast marching method for the rest of the domain.

2.3 The Mesh Size Function

For a given geometry, we define our mesh size function $h(\mathbf{x})$ by the following five properties. The scalar parameters K, R, G may all be functions of space, and in Section 7.3 we even allow G to be a function of the mesh size $h(\mathbf{x})$.

- 1. Curvature Adaptation** On the boundaries, we require $h(\mathbf{x}) \leq 1/K|\kappa(\mathbf{x})|$, where κ is the boundary curvature. The resolution is controlled by the parameter K which is the number of elements per radian in 2-D (it is related to the *maximum spanning angle* θ by $1/K = 2 \sin(\theta/2)$).
- 2. Local Feature Size Adaptation** Everywhere in the domain, $h(\mathbf{x}) \leq \text{lfs}(\mathbf{x})/R$. The local feature size $\text{lfs}(\mathbf{x})$ is, loosely speaking, half the width of the geometry at \mathbf{x} . The parameter R gives half the number of elements across narrow regions of the geometry.
- 3. Non-geometric Adaptation** An additional external spacing function $h_{\text{ext}}(\mathbf{x})$ might be given by an adaptive numerical solver or as a user-specified function (often at isolated points or boundaries). We then require that $h(\mathbf{x}) \leq h_{\text{ext}}(\mathbf{x})$.
- 4. Grading Limiting** The grading requirement means that the size of two neighboring elements in a mesh should not differ by more than a factor G , or $h_i \leq Gh_j$ for all neighboring elements i, j . The continuous analogue of this is that the magnitude of the gradient of the size function is limited by $|\nabla h(\mathbf{x})| \leq g$, where g depends on the interpretation of the element sizes but is approximately $G - 1$.
- 5. Optimality** In addition to the above requirements (which are all upper bounds), we require that $h(\mathbf{x})$ is as large as possible at all points.

We now show how to create a size function $h(\mathbf{x})$ according to these requirements, starting from an implicit boundary definition by its signed distance function $\phi(\mathbf{x})$, with a negative sign inside the geometry.

3. CURVATURE ADAPTATION

To resolve curved boundaries accurately, we want to impose the curvature adaptation $h(\mathbf{x}) \leq h_{\text{curv}}(\mathbf{x})$ on the boundaries, with

$$\begin{cases} h_{\text{curv}}(\mathbf{x}) = 1/K|\kappa(\mathbf{x})|, & \text{if } \phi(\mathbf{x}) = 0, \\ \infty, & \text{if } \phi(\mathbf{x}) \neq 0, \end{cases} \quad (1)$$

where $\kappa(\mathbf{x})$ is the curvature at \mathbf{x} . In three dimensions we use the maximum principal curvature in order to resolve the smallest radius of curvature.

For an unstructured background grid, where the elements are aligned with the boundaries, we simply assign values for $h(\mathbf{x})$ on the boundary nodes and set the remaining nodal values to infinity. Later on, the gradient limiting will propagate these values into the rest of the region. The boundary curvature might be

available as a closed form expression (e.g. by a CAD representation), or it can be approximated from the surface triangulation.

For an implicit boundary discretization on a Cartesian background grid we can compute the curvature from the distance function, for example in 2-D:

$$\kappa = \nabla \cdot \frac{\nabla \phi}{|\nabla \phi|} = \frac{\phi_{xx}\phi_y^2 - 2\phi_y\phi_x\phi_{xy} + \phi_{yy}\phi_x^2}{(\phi_x^2 + \phi_y^2)^{3/2}}. \quad (2)$$

In 3-D similar expressions give the mean curvature κ_H and the Gaussian curvature κ_K , from which the principal curvatures are obtained as $\kappa_{1,2} = \kappa_H \pm \sqrt{\kappa_H^2 - \kappa_K}$. On a Cartesian grid, we use standard second-order difference approximations for the derivatives.

These difference approximations give us accurate curvatures at the node points, and we could compute mesh sizes directly according to (1) on the nodes close to the boundary, and set the remaining interior and exterior nodes to infinity. However, since in general the nodes are not located on the boundary, we get a poor approximation of the true, continuous, curvature requirement (1). Below we show how to modify the calculations to include a correction for node points not aligned with the boundaries.

In two dimensions, suppose we calculate a curvature κ_{ij} at the grid point \mathbf{x}_{ij} . This point is generally not located on the boundary, but a distance $|\phi_{ij}|$ away. If we set $h_{\text{curv}}(\mathbf{x}_{ij}) = 1/(K|\kappa_{ij}|)$ we introduce two sources of errors:

- We use the curvature at \mathbf{x}_{ij} instead of at the boundary. We can compensate for this by adding ϕ_{ij} to the radius of curvature:

$$\kappa_{\text{bound}} = \frac{1}{\frac{1}{\kappa_{ij}} + \phi_{ij}} = \frac{\kappa_{ij}}{1 + \kappa_{ij}\phi_{ij}} \quad (3)$$

Note that we keep the signs on κ and ϕ . If, for example, $\phi > 0$ and $\kappa > 0$, we should increase the radius of curvature. This expression is exact for circles, including the limiting case of zero curvature (a straight line).

- Even if we use the corrected curvature κ_{bound} , we impose our h_{curv} at the grid point \mathbf{x}_{ij} instead of at the boundary. However, the grid point will be affected indirectly by the gradient limiting, and we can get a better estimate of the correct h by adding $g|\phi_{ij}|$. Interpolation of the absolute function is inaccurate, and again we keep the sign of ϕ and subtract $g\phi_{ij}$ (that is, we add the distance inside the region and subtract it outside).

Putting this together, we get the following definition

of h_{curv} in terms of the grid spacing Δx :

$$h_{\text{curv}}(\mathbf{x}_{ij}) = \begin{cases} \left| \frac{1 + \kappa_{ij}\phi_{ij}}{K\kappa_{ij}} \right| - g\phi_{ij}, & |\phi_{ij}| \leq 2\Delta x, \\ \infty, & |\phi_{ij}| > 2\Delta x. \end{cases} \quad (4)$$

This will limit the edge sizes in a narrow band around the boundaries, but it will not have any effect in the interior of the region. A similar expression can be used in three dimensions, where the curvature is replaced by maximum principal curvature as before, and the correction makes the expression exact for spheres and planes.

4. FEATURE SIZE ADAPTATION

For feature size adaptation, we want to impose the condition $h(\mathbf{x}) \leq h_{\text{lfs}}(\mathbf{x})$ everywhere inside our domain, where

$$\begin{cases} h_{\text{lfs}}(\mathbf{x}) = \text{lfs}(\mathbf{x})/R, & \text{if } \phi(\mathbf{x}) \leq 0, \\ \infty, & \text{if } \phi(\mathbf{x}) > 0. \end{cases} \quad (5)$$

The *local feature size* $\text{lfs}(\mathbf{x})$ is a measure of the distance between nearby boundaries. It is defined by Ruppert [5] as “the larger distance from \mathbf{x} to the closest two non-adjacent polytopes [of the boundary]”. For our implicit boundary definitions, there is no clear notion of adjacent polytopes, and we use instead the similar definition (inspired by the definition for surface meshes in [15]) that the local feature size at a boundary point \mathbf{x} is equal to the smallest distance between \mathbf{x} and the medial axis. The medial axis is the set of interior points that have equal distance to two or more points on the boundary.

For geometries with sharp corners that consist of separate boundary sections, we exclude medial axis points that have equal distance to two neighboring boundaries. The feature size should not be larger than an edge or face connecting sharp corners. Our medial axis based method will in some cases not detect this since it is a result of having sharp corners and not because of the actual boundary. However, this effect on the feature size is local and it is easily incorporated by explicit constraints on $h(\mathbf{x})$ along the edge or the face (possibly with a correction $-g\phi$ like in the curvature adaptation).

The definition of local feature size can be extended to the entire domain in many ways. We simply add the distance function for the domain boundary to the distance functions for the medial axis, to obtain our definition:

$$\text{lfs}_{\text{MA}}(\mathbf{x}) = |\phi(\mathbf{x})| + |\phi_{\text{MA}}(\mathbf{x})|, \quad (6)$$

where $\phi(\mathbf{x})$ is the distance function for the domain and $\phi_{\text{MA}}(\mathbf{x})$ is the distance to its medial axis (MA).

The distances $\phi_{MA}(\mathbf{x})$ are always positive, but we take its absolute value to emphasize that we always add positive distances.

The expression (6) obviously reduces to the definition in [15] at boundary points \mathbf{x} , since then $\phi(\mathbf{x}) = 0$. For a narrow region with parallel boundaries, $\text{lfs}(\mathbf{x})$ is exactly half the width of the region, and a value of $R = 1$ would resolve the region with two elements.

To compute the local feature size according to (6), we have to compute the *medial axis transform* $\phi_{MA}(\mathbf{x})$ in addition to the given distance function $\phi(\mathbf{x})$. If we know the location of the medial axis we can use the techniques described in Section 2.2, for example explicit distance calculations near the medial axis and the fast marching method for the remaining nodes. The identification of the medial axis is often referred to as *skeletonization*, and a large number of algorithms have been proposed. Many of them, including the original Grassfire algorithm by Blum [16], are based on explicit representations of the geometry. Kimmel et al [17] described an algorithm for finding the medial axis from a distance function in two dimensions, by segmenting the boundary curve with respect to curvature extrema. Siddiqi et al [18] used a divergence based formulation combined with a *thinning* process to guarantee a correct topology. Telea and Wijk [19] showed how to use the fast marching method for skeletonization and centerline extraction.

Although in principle we could use any existing algorithm for skeletonization using distance functions, we have developed a new method mainly because our requirements are different than those in other applications. Maintaining the correct topology is not a high priority for us, since we do not use the skeleton topology (and if we did, we could combine our algorithm with thinning, as in [18]). This means that small “holes” in the skeleton will only cause a minor perturbation of the local feature size. However, an incorrect detection of the skeleton close to the boundary is worse, since our definition (6) would set the feature size to a very small value close to that point.

We also need a higher accuracy of the computed medial axis location. Applications in image processing and computer graphics often work on a pixel level, and having a higher level of detail is referred to as *subgrid accuracy*. A final desired requirement is to have a minimum number of user parameters, since the algorithm should work in an automated way. Other algorithms typically use fixed parameters to eliminate incorrect skeleton points close to curved regions. We use the curvature to determine if candidate points should be accepted based on only one parameter specifying the smallest resolved curvature.

Our method uses a Cartesian grid, but should be easy

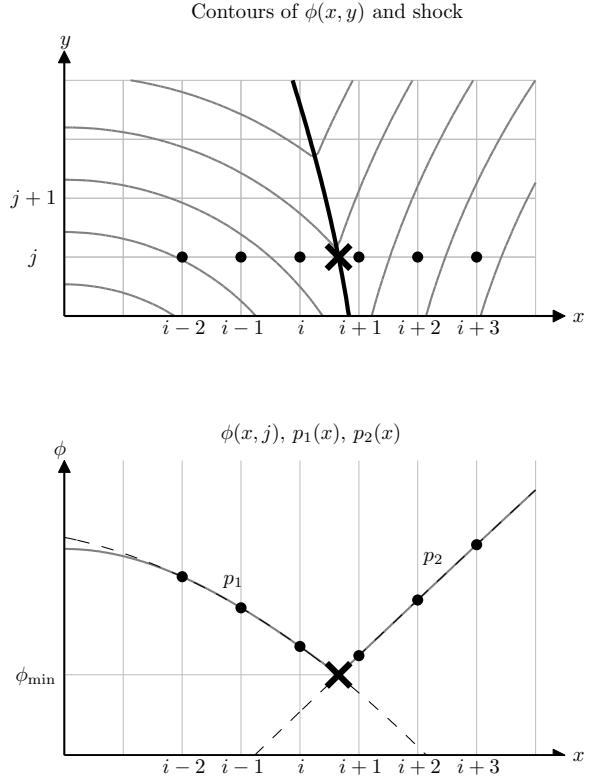


Figure 2: Detection of shock in the distance function $\phi(x, y)$ along the edge $(i, j), (i + 1, j)$. The location of the shock is given by the crossing of the two parabolas $p_1(x)$ and $p_2(x)$.

to extend to other background meshes. For all edges in the computational grid, we fit polynomials to the distance function at each side of the edge, and detect if they cross somewhere along the edge (Figure 2). Such a crossing becomes a candidate for a new skeleton point and we apply several tests, more or less heuristic, to determine if the point should be accepted.

The complete algorithm is shown in Table 1. We scale the domain to have unit spacing, and for each edge we consider the interval $s \in [-2, 3]$ where $s \in [0, 1]$ corresponds to the edge. Next we fit quadratic polynomials p_1 and p_2 to the values of the distance function at the two sides of the edge, and compute their crossings. Our tests to determine if a crossing should be considered a skeleton point are summarized below:

- There should be exactly one root s_0 along the edge $s \in [0, 1]$.
- The derivative of p_2 should be strictly greater than the derivative of p_1 in $s \in [-2, 3]$ (it is sufficient to check the endpoints, since the derivatives are linear)

Algorithm 1 - Skeletonization using Distance Function

Description: Detect medial axis
Input: Grid \mathbf{x}_{ijk} , dist. func. ϕ_{ijk} , parameters $\gamma, \kappa_{\text{tol}}$
Output: Crossings \mathbf{p}_i and neighboring distances ϕ_{MA}

Normalize \mathbf{x}_{ijk} and ϕ_{ijk} to have unit grid spacing
 Approximate $\nabla\phi_{ijk}$ with one-sided finite differences
 Approximate max. principal curvature κ_{ijk} from ϕ_{ijk}
for all consecutive six nodes $\mathbf{x}_{i-2:i+3,j,k}$
 Define $\phi_1, \dots, \phi_6 = \phi_{i-2,j,k}, \dots, \phi_{i+3,j,k}$
 Fit parabolas $p_1(s)$ and $p_2(s)$ to the data points

$$(s, \phi) = (-2, \phi_1), (-1, \phi_2), (0, \phi_3), \quad \text{and}$$

$$(s, \phi) = (1, \phi_4), (2, \phi_5), (3, \phi_6)$$

Find real roots of $\Delta p(s) = p_2(s) - p_1(s)$
if one root s_0 in $[0, 1]$ **and** $d\Delta p/ds > 0$ in $[-2, 3]$
 Let $\kappa_1 = \kappa_{i-1,j,k}$ and $\kappa_2 = \kappa_{i+2,j,k}$
 Compute dot product α between fronts

$$\alpha = \nabla\phi_{i,j,k} \cdot \nabla\phi_{i+1,j,k}$$

if $\alpha < 1 - \gamma^2 \max(\kappa_1^2, \kappa_2^2, \kappa_{\text{tol}}^2)/2$:
 Accept $\mathbf{p} = \mathbf{x}_{ijk} + e_1 h s_0$ as a MA point
 Compute medial axis normal:

$$\mathbf{n} = (n_x, n_y, n_z) = \frac{\nabla\phi_{i,j,k} - \nabla\phi_{i+1,j,k}}{\|\nabla\phi_{i,j,k} - \nabla\phi_{i+1,j,k}\|}$$

Compute neighboring distances

$$\phi_{\text{MA},1} = |n_x h s_0|$$

$$\phi_{\text{MA},2} = |n_x h (1 - s_0)|$$

end if
end if
end for
 In each interval keep only p_i with largest $d\Delta p/ds(s_0)$
 Repeat for consecutive nodes in y - and z -direction

Table 1: The algorithm for detecting the medial axis in a discretized distance function and computing the distances to neighboring nodes.

- The dot product α between the two propagation directions should be smaller than a tolerance, which depends on the curvatures of the two fronts (see below).
- We reject the point if another crossing is detected within the interval $[-2, 3]$ with a larger derivative difference $dp_2/ds - dp_1/ds$ at the crossing s_0 .

The dot product α is evaluated from one-sided difference approximations of $\nabla\phi$. This is compared to the expected dot product between two front from a cir-

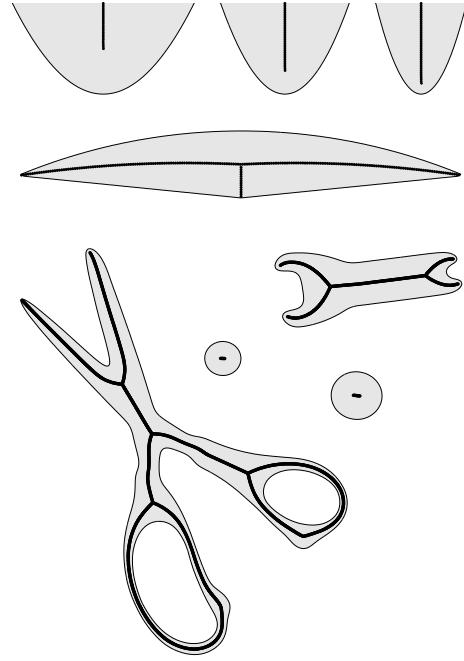


Figure 3: Examples of medial axis calculations for some planar geometries.

cle of radius $1/|\kappa|$, where κ is the largest curvature at the two points. With one unit separation between the points and an angle θ between the fronts, this dot product is

$$\cos \theta = 1 - 2 \sin^2(\theta/2) = 1 - 2(|\kappa|/2)^2 = 1 - \kappa^2/2 \quad (7)$$

We reject the point if the actual dot product α is larger than this for any of the curvatures κ_1, κ_2 at the two sides of the edge or the given tolerance κ_{tol} . We calculate κ using difference approximations, and to avoid the shock we evaluate it one grid point away from the edge. To compensate for this we include a tolerance γ in the computed curvatures.

If the point is accepted as a medial axis point, we obtain the normal of the medial axis by subtracting the two gradients. The distance from the medial axis to the two neighboring points are then $|n_x h s_0|$ and $|n_x h (1 - s_0)|$. These are used as boundary conditions when solving for $\phi_{\text{MA}}(\mathbf{x})$ in the entire domain using the fast marching method.

Some examples of medial axis detections are shown in Figure 3. Note how the three parabolas (top right) are handled correctly with the curvature dependent tolerances.

5. GRADIENT LIMITING

An important requirement on the size function is that the ratio of neighboring element sizes in the generated mesh is less than a given value G . This corresponds to a limit on the gradient $|\nabla h(\mathbf{x})| \leq g$ where $g \approx G - 1$. Note that we need a *linear* increase in the size function to obtain a *geometric* progression of the element sizes. To see this, consider a simple one-dimensional problem with mesh size specified at a boundary point $h(x_0) = h_0$. Generate a mesh in an advancing front-like manner by the algorithm $x_{i+1} = x_i + h(x_i)$. With a linear size function of the form $h(x) = h_0 + g(x - x_0)$ we get a constant ratio between neighboring elements:

$$\begin{aligned}\frac{h(x_{i+1})}{h(x_i)} &= \frac{h(x_i + h(x_i))}{h(x_i)} = \frac{h_0 + g(x_i + h(x_i) - x_0)}{h(x_i)} \\ &= \frac{h(x_i) + gh(x_i)}{h(x_i)} = 1 + g.\end{aligned}$$

In some simple cases, this linear increase can be built into the size function explicitly. For example, a “point-source” size constraint $h(\mathbf{y}) = h_0$ in a convex domain can be extended as $h(\mathbf{x}) = h_0 + g|\mathbf{x} - \mathbf{y}|$, and similarly for other shapes such as edges. For more complex boundary curves, local feature sizes, user constraints, etc, such an explicit formulation is difficult to create and expensive to evaluate. It is also harder to extend this method to non-convex domains (such as the example in Figure 6), or to non-constant g (Figures 14 and 15).

One way to limit the gradients of a discretized size function is to iterate over the edges of the background mesh and update the size function locally for neighboring nodes [20]. When the iterations converge, the solution satisfies $|\nabla h(\mathbf{x})| \leq g$ only approximately, in a way that depends on the mesh. Another method is to build a balanced octree, and let the size function be related to the size of the octree cells [21]. This data structure is used in the quadtree meshing algorithm [22], and the balancing guarantees a limited variation in element sizes, by a maximum factor of two between neighboring cells. However, when used as a size function for other meshing algorithms it provides an approximate discrete solution to the original problem, and it is hard to generalize the method to arbitrary gradients g or different background meshes.

We present a new technique to handle the gradient limiting problem, by a continuous formulation of the process as a Hamilton-Jacobi equation. Since the mesh size function is defined as a continuous function of \mathbf{x} , it is natural to formulate the gradient limiting as a PDE with solution $h(\mathbf{x})$ independently of the actual background mesh. We can see many benefits in doing this:

- The analytical solution is exactly the optimal gra-

dient limited size function $h(\mathbf{x})$ that we want, as shown by Theorem 5.1. The only errors come from the numerical discretization, which can be controlled and reduced using known solution techniques for hyperbolic PDEs.

- By relying on existing well-developed Hamilton-Jacobi solvers we can generalize the algorithm in a straightforward way to
 - Cartesian grids, octree grids, or fully unstructured meshes
 - Higher order discretizations
 - Space and solution dependent g
 - Regions embedded in higher-dimensional spaces, for example surface meshes in 3-D.
- We can compute the solution in $\mathcal{O}(n \log n)$ time using a modified fast marching method.

5.1 The Gradient Limiting Equation

We now consider how to limit the magnitude of the gradients of a function $h_0(\mathbf{x})$, to obtain a new *gradient limited* function $h(\mathbf{x})$ satisfying $|\nabla h(\mathbf{x})| \leq g$ everywhere. We require that $h(\mathbf{x}) \leq h_0(\mathbf{x})$, and at every \mathbf{x} we want h to be as large as possible. We claim that $h(\mathbf{x})$ is the steady-state solution to the following *Gradient Limiting Equation*:

$$\frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, g), \quad (8)$$

with initial condition

$$h(\mathbf{x}, t=0) = h_0(\mathbf{x}). \quad (9)$$

When $|\nabla h| \leq g$, (8) gives that $\partial h / \partial t = 0$, and h will not change with time. When $|\nabla h| > g$, the equation will enforce $|\nabla h| = g$ (locally), and the positive sign multiplying $|\nabla h|$ ensures that information propagates in the direction of increasing values. At steady-state we have that $|\nabla h| = \min(|\nabla h|, g)$, which is the same as $|\nabla h| \leq g$.

For the special case of a convex domain in \mathbb{R}^n and constant g , we can derive an analytical expression for the solution to (8), showing that it is indeed the optimal solution:

Theorem 5.1. *Let $\Omega \subset \mathbb{R}^n$ be a bounded convex domain, and $I = (0, T)$ a given time interval. The steady-state solution $h(\mathbf{x}) = \lim_{T \rightarrow \infty} h(\mathbf{x}, T)$ to*

$$\begin{cases} \frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, g) & (\mathbf{x}, t) \in \Omega \times I \\ h(\mathbf{x}, t)|_{t=0} = h_0(\mathbf{x}) & \mathbf{x} \in \Omega \end{cases} \quad (10)$$

is

$$h(\mathbf{x}) = \min_y (h_0(y) + g|\mathbf{x} - y|). \quad (11)$$

Proof. The Hopf-Lax theorem [23] states that the solution to the Hamilton-Jacobi equation $\frac{du}{dt} + F(\nabla u) = 0$ with initial condition $u(x, 0) = u_0(x)$ and convex $F(w)$ is given by

$$u(x, t) = \min_y [u_0(y) + tF^*((x - y)/t)], \quad (12)$$

where $F^*(u) = \max_w (wu - F(w))$ is the conjugate function of F .

For our equation (10), rewrite as $\frac{\partial h}{\partial t} + F(\nabla h) = 0$, with $F(w) = |w| - \min(|w|, g)$. The conjugate function is

$$\begin{aligned} F^*(u) &= \max_w (wu - F(w)) \\ &= \max_w (wu - |w| + \min(|w|, g)) \\ &= \begin{cases} g|u|, & \text{if } |u| < 1, \\ +\infty & \text{if } |u| \geq 1. \end{cases} \end{aligned} \quad (13)$$

Using (12), we get

$$\begin{aligned} h(x, t) &= \min_y [h_0(y) + tF^*((x - y)/t)] \\ &= \min_{\substack{y \\ |x-y|\leq t}} (h_0(y) + g|x - y|). \end{aligned} \quad (14)$$

Let $t \rightarrow \infty$ to get the steady-state solution to (10):

$$h(x) = \min_y (h_0(y) + g|x - y|). \quad (15)$$

□

Note that the solution (11) is composed of infinitely many point-source solutions as described before. We could in principle define an algorithm based on (11) for computing h from a given h_0 (both discretized). Such an algorithm would be trivial to implement, but its computational complexity would be proportional to the square of the number of node points. Instead, we solve (10) using efficient Hamilton-Jacobi solvers.

The gradient limiting is illustrated by a one dimensional example in Figure 4, where (10) is solved using different values of g and a simple scalar function as initial condition. Note how the large gradients are reduced exactly the amount needed, without affecting regions far away from them. This is very different from traditional smoothing, which affects all data and gives excessive perturbation of the original function $h_0(x)$. Our solution is not necessarily smooth, but it is continuous and $|\nabla h| \leq g$ everywhere.

5.2 Implementation

One advantage with the continuous formulation of the problem is that a large variety of solvers can be used almost as black-boxes. This includes solvers for structured and unstructured grids, higher-order methods, and specialized fast solvers.

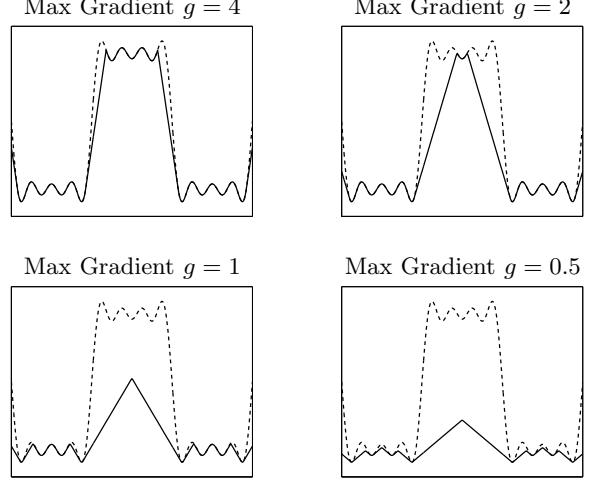


Figure 4: Illustration of gradient limiting by $\frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, g)$. The dashed lines are the initial conditions h_0 and the solid lines are the gradient limited steady-state solutions h for different parameter values g .

On a Cartesian background grid, the equation (8) can be solved with just a few lines of code using the following iteration:

$$h_{ijk}^{n+1} = h_{ijk}^n + \Delta t (\min(\nabla_{ijk}^+, g) - \nabla_{ijk}^+) \quad (16)$$

where

$$\begin{aligned} \nabla_{ijk}^+ &= [\max(D^{-x} h_{ijk}^n, 0)^2 + \min(D^{+x} h_{ijk}^n, 0)^2 + \\ &\quad \max(D^{-y} h_{ijk}^n, 0)^2 + \min(D^{+y} h_{ijk}^n, 0)^2 + \\ &\quad \max(D^{-z} h_{ijk}^n, 0)^2 + \min(D^{+z} h_{ijk}^n, 0)]^{1/2} \end{aligned} \quad (17)$$

Here, D^{-x} is the backward difference operator in the x -direction, D^{+x} the forward difference operator, etc. The iterations are initialized by $h^0 = h_0$, and we iterate until the updates $\Delta h(x)$ are smaller than a given tolerance. The Δt parameter is chosen to satisfy the CFL-condition, we use $\Delta t = \Delta x/2$. The boundaries of the grid do not need any special treatment since all characteristics point outward.

The iteration (16) converges relatively fast, although the number of iterations grows with the problem size so the total computational complexity is superlinear. Nevertheless, the simplicity makes this a good choice in many situations. If a good initial guess is available, this time-stepping technique might even be superior to other methods. This is the case for problems with moving boundaries, where the size function from the last mesh is likely to be close to the new size function, or in numerical adaptivity, when the original size function already has relatively small gradients because of numerical properties of the underlying PDE.

The scheme (16) is first-order accurate in space, and higher accuracy can be achieved by using a second-order solver. See [10] and [24] for details.

For faster solution of (8) we use a modified version of the fast marching method (see Section 2.2). The main idea for solving our PDE (8) is based on the fact that the characteristics point in the direction of the gradient, and therefore smaller values are never affected by larger values. This means we can start by fixing the smallest value of the solution, since it will never be modified. We then update the neighbors of this node by a discretization of our PDE, and repeat the procedure. To find the smallest value efficiently we use a min-heap data structure.

During the update, we have to solve for a new h_{ijk} in $\nabla_{ijk}^+ = g$, with ∇_{ijk}^+ from (17). This expression is simplified by the fact that h_{ijk} should be larger than all previously fixed values of h , and we can solve a quadratic equation for each octant and set h_{ijk} to the minimum of these solutions.

Our fast algorithm is summarized as pseudo-code in Table 2. Compared to the original fast marching method, we begin by marking all nodes as TRIAL points, and we do not have any FAR points. The actual update involves a nonlinear right-hand side, but it always returns increasing values so the update procedure is valid. The heap is large since all elements are inserted initially, but the access time is still only $\mathcal{O}(\log n)$ for each of the n nodes in the background grid. In total, this gives a solver with computational complexity $\mathcal{O}(n \log n)$. For higher-order accuracy, the technique described in [11] can be applied.

An unstructured background grid gives a more efficient representation of the size function and higher flexibility in terms of node placement. A common choice is to use an initial Delaunay mesh, possibly with a few additional refinements. Several methods have been developed to solve Hamilton-Jacobi equations on unstructured grids, and we have implemented the positive coefficient scheme by Barth and Sethian [25]. The solver is slightly more complicated than the Cartesian variants, but the numerical schemes can essentially be used as black-boxes. A triangulated version of the fast marching method was given in [26], and in [27] the algorithm was generalized to arbitrary node locations.

One particular unstructured background grid is the octree representation, and the Cartesian methods extend naturally to this case (both the iteration and the fast solver). The values are interpolated on the boundaries between cells of different sizes. We mentioned in the introduction that octrees are commonly used to represent size functions, because of the possibility to balance the tree and thereby get a limited variation of cell sizes. Here, we propose to use the octree as

Algorithm 2 - Fast Gradient Limiting

```

Description: Solve (8) on a Cartesian grid
Input: Initial discretized  $h_0$ , grid spacing  $\Delta x$ 
Output: Discretized solution  $h$ 

Set  $h = h_0$ 
Insert all  $h_{ijk}$  in a min-heap with back pointers
while heap not empty
    Remove smallest element  $IJK$  from heap
    for neighbors  $ijk$  of  $IJK$  still in heap:
        compute upwind  $|\nabla h_{ijk}|$ 
        if  $|\nabla h_{ijk}| > g$ 
            Solve for  $h_{ijk}^{\text{new}}$  in  $\nabla_{ijk}^+ = g$  from (17)
            Set  $h_{ijk} \leftarrow \min(h_{ijk}, h_{ijk}^{\text{new}})$ 
        end if
    end for
end while

```

Table 2: The fast gradient limiting algorithm for Cartesian grids. The computational complexity is $\mathcal{O}(n \log n)$, where n is the number of nodes in the background grid.

a convenient and efficient representation, but the actual values of the size function are computed using our PDE. This gives higher flexibility, for example the possibility to use different values of g .

6. RESULTS

We are now ready to put all the pieces together and define the complete algorithm for generation of a mesh size function. The size functions from curvature and feature size are computed as described in the previous sections. The external size function $h_{\text{ext}}(\mathbf{x})$ is provided as input. Our final size function must be smaller than these at each point in space:

$$h_0(\mathbf{x}) = \min(h_{\text{curv}}(\mathbf{x}), h_{\text{lhs}}(\mathbf{x}), h_{\text{ext}}(\mathbf{x})) \quad (18)$$

Finally, we apply the gradient limiting algorithm from Section 5 on h_0 to get the mesh size function h , by solving:

$$\frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, g) \quad (19)$$

with initial condition $h(\mathbf{x}, t = 0) = h_0(\mathbf{x})$.

We now show a number of examples, with different geometries, background grids, and feature size definitions. All triangular and tetrahedral meshes are generated with the smoothing-based mesh generator for distance functions in [3], [4]. For some of the 2-D examples we have also generated meshes using an advancing front generator with similar results.

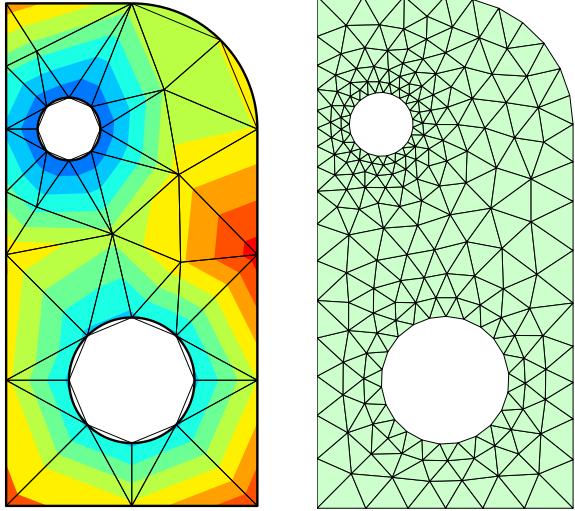


Figure 5: Example of gradient limiting with an unstructured background grid. The size function is given at the curved boundaries and computed by (8) at the remaining nodes.

6.1 Mesh Size Functions in 2-D and 3-D

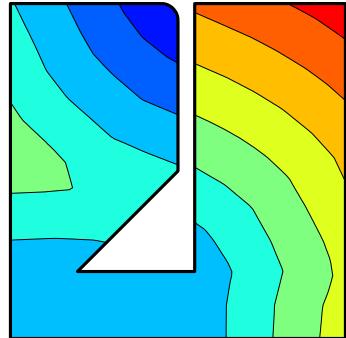
We begin with a simple example of gradient limiting in two dimensions on a triangular mesh. For the geometry in Figure 5, we set $h_0(\mathbf{x})$ proportional to the radius of curvature on the boundaries, and to ∞ in the interior. We solve our gradient limiting equation using the positive coefficient scheme to get the mesh size function in the middle plot. A sample mesh using this result is shown in the right plot.

This example shows that we can apply size constraints in an arbitrary manner, for example only on some of the boundary nodes. The PDE will propagate the values in an optimal way to the remaining nodes, and possibly also change the given values if they violate the grading condition. For this very simple geometry, we can indeed write the size function explicitly as

$$h(\mathbf{x}) = \min_i(h_i + g\phi_i(\mathbf{x})). \quad (20)$$

Here, ϕ_i and h_i are the distance functions and the boundary mesh size for each of the three curved boundaries. But consider, for example, a curved boundary with a non-constant curvature. The analytical expression for the size function of this boundary is non-trivial (it involves the curvature and distance function of the curve). One solution would be to put point-sources at each node of the background mesh, but the complexity of evaluating (20) grows quickly with the number of nodes. By solving our gradient limiting equation, we arrive at the same solution in an efficient and simple way.

Mesh Size Function $h(\mathbf{x})$



Mesh Based on $h(\mathbf{x})$

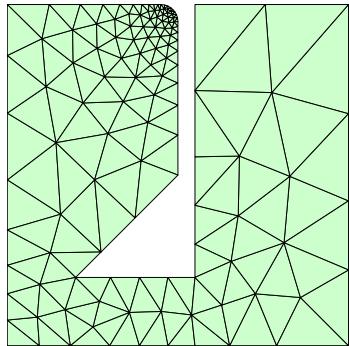
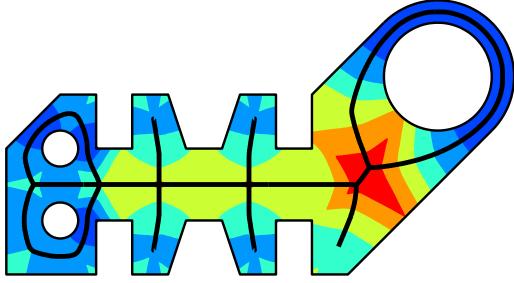


Figure 6: Another example of gradient limiting, showing that non-convex regions are handled correctly. The small sizes at the curved boundary do not affect the region at the right, since there are no connections across the narrow slit.

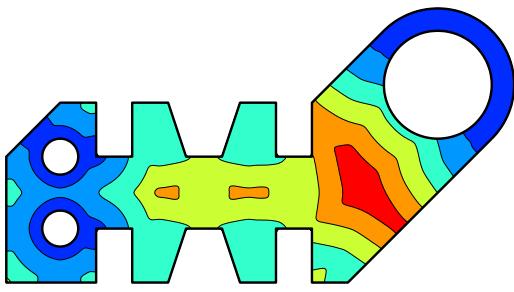
In Figure 6 we show a size function for a geometry with a narrow slit, again generated using the unstructured gradient limiting solver. The initial size function $h_0(\mathbf{x})$ is based on the local feature size and the curved boundary at the top. Note that although the regions on the two sides of the slit are close to each other, the small mesh size at the curved boundary does not influence the other region. This solution is harder to express using source expressions such as (20), where more expensive geometric search routines would have to be used.

A more complicated example is shown in Figure 7. Here, we have computed the local feature size everywhere in the interior of the geometry. We compute this using the medial axis based definition from Section 4. The result is stored on a Cartesian grid. In some regions the gradient of the local feature size is greater than g , and we use the fast gradient limiting solver in Algorithm 2 to get a well-behaved size function. We also use curvature adaptation as before. Note that this mesh size function would be very expensive

Medial Axis and Feature Size



Mesh Size Function $h(x)$



Mesh Based on $h(x)$

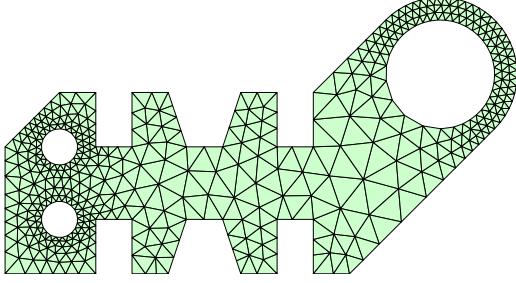


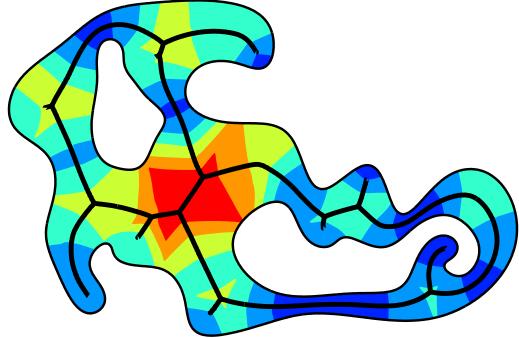
Figure 7: A mesh size function taking into account both feature size, curvature, and gradient limiting. The feature size is computed as the sum of the distance function and the distance to the medial axis.

to compute explicitly, since the feature size is defined everywhere in the domain, not just on the boundaries.

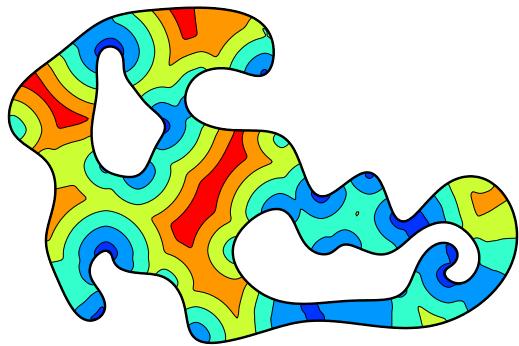
As a final example of 2-D mesh generation, we show an object with smooth boundaries in Figure 8. We use a Cartesian grid for the background grid and solve the gradient limiting equation using the fast solver. The feature size is again computed using the medial axis and the distance function, and the curvature is given by the expression with grid correction (4) since the grid is not aligned with the boundaries.

The PDE-based formulation generalizes to arbitrary dimensions, and in Figure 9 we show a 3-D example.

Medial Axis and Feature Size



Mesh Size Function $h(x)$



Mesh Based on $h(x)$

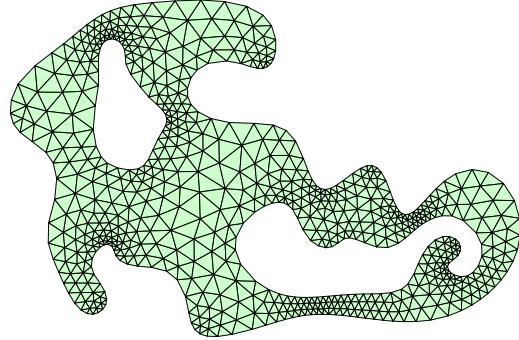


Figure 8: Generation of a mesh size function for a geometry with smooth boundaries.

Here, the feature size is computed explicitly from the geometry description, the curvature adaptation is applied on the boundary nodes, and the size function is computed by gradient limiting with $g = 0.2$. This results in a well-shaped tetrahedral mesh, in the bottom plot.

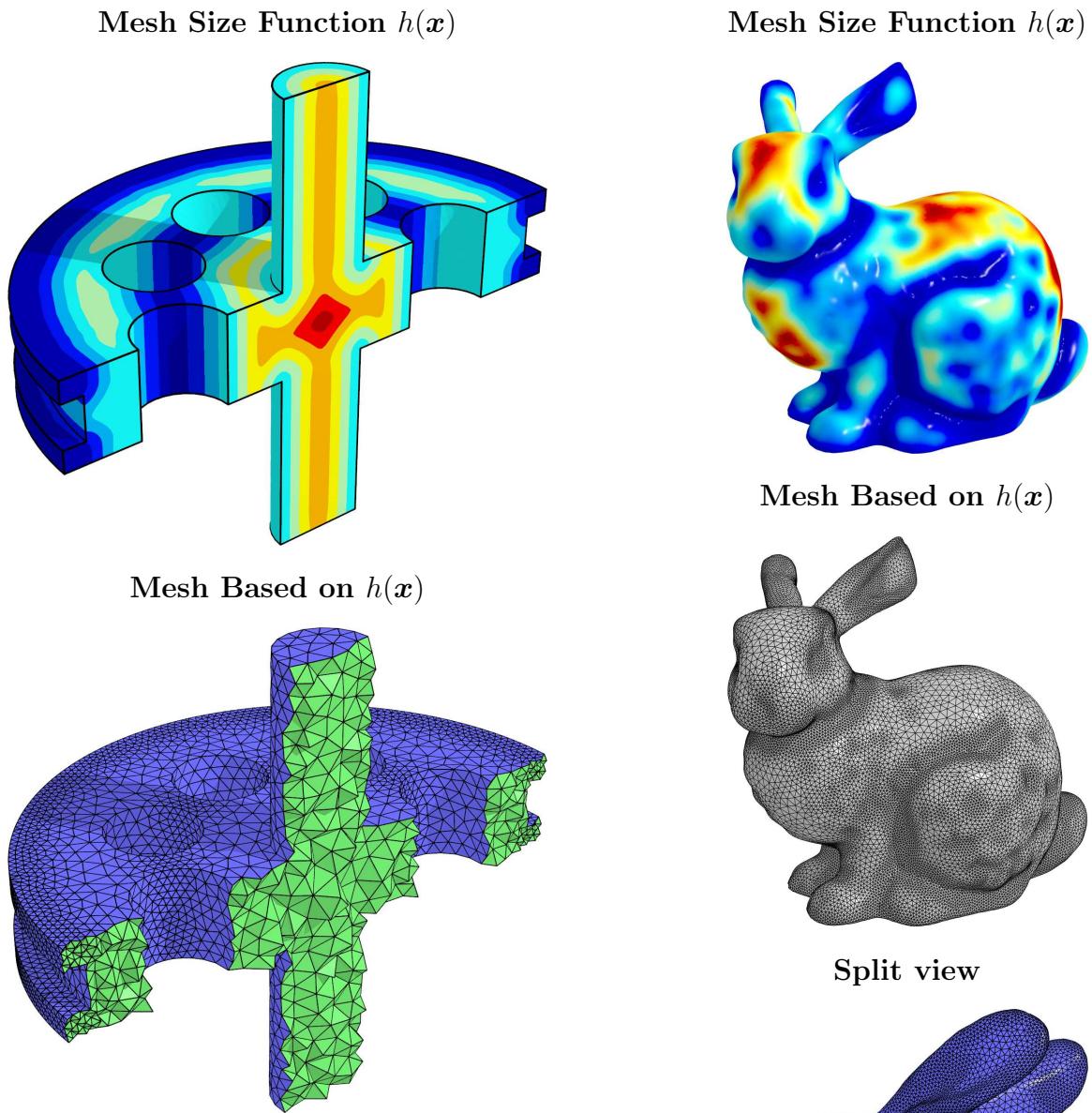


Figure 9: Cross-sections of a 3-D mesh size function and a sample tetrahedral mesh.

A more complex model is shown in Figure 10.¹ We apply gradient limiting with $g = 0.3$ on a size function which is computed automatically, taking into account curvature adaptation and feature size adaptation (from the medial axis, as described before). The plots show the final mesh size function and an example mesh.

Figure 10: A 3-D mesh size function and a sample tetrahedral mesh. Note the small elements in the narrow regions, given by the local feature size, and the smooth increase in element sizes.

¹This model was obtained from the The Stanford 3D Scanning Repository.

# Nodes	Edge Iter.	H-J Iter.	H-J Fast
10,000	0.009s	0.060s	0.006s
40,000	0.068s	0.470s	0.030s
160,000	0.844s	3.625s	0.181s
640,000	6.609s	28.422s	1.453s

Table 3: Performance of the edge-based iterative solver, the Hamilton-Jacobi iterative solver, and the Hamilton-Jacobi fast gradient limiting solver.

6.2 Performance and Accuracy

To study the performance and the accuracy of our algorithms, we consider a simple model problem in $\Omega = (-50, 50) \times (-50, 50)$ with two point-sources, $h(-10, 0) = 1$ and $h(10, 0) = 5$, and $g = 0.3$. The true solution is given by (11), and we solve the problem on a Cartesian grid of varying resolution.

In Table 3 we compare the execution times for three different solvers – edge-based iterations, Hamilton-Jacobi iterations, and the Hamilton-Jacobi fast gradient limiting solver. The edge-based iterative solver loops until convergence over all neighboring nodes i, j and updates the size function locally by $h_j \leftarrow \min(h_j, h_i + g|\mathbf{x}_j - \mathbf{x}_i|)$ (assuming $h_j > h_i$). The iterative Hamilton-Jacobi solver is based on the iteration (16) with a tolerance of about two digits. All algorithms are implemented in C++ using the same optimizations, and the tests were done on a PC with an Athlon XP 2800+ processor.

The table shows that the iterative Hamilton-Jacobi solver is about five times slower than the simple edge-based iterations. This is because the update formula for the edge-based iterations is simpler (all edge lengths are the same) and since the Hamilton-Jacobi solver requires more iterations for high accuracy (although their asymptotic behavior should be the same). The fast solver is better than the iterative solvers, and the difference gets bigger with increasing problem size (since it is asymptotically faster). Note that these background meshes are relatively large and that all solvers probably are sufficiently fast in many practical situations.

We also mention that simple algorithms based on the explicit expression (11) for convex domains or geometric searches for non-convex domains might be faster for a small number of point-sources. However, these methods are not practical for larger problems because of the $\mathcal{O}(n^2)$ complexity.

Next we compare the accuracy of the edge-based solver and Hamilton-Jacobi discretizations of first and second order accuracy. The true solution is given by (11), and an algorithm based on this expression would of course be exact to full precision. Figure 11 shows solutions

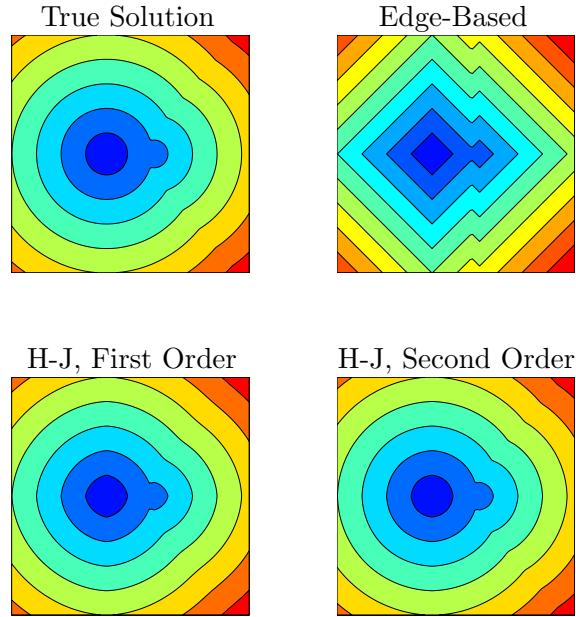


Figure 11: Comparison of the accuracy of the discrete edge-based solver and the continuous Hamilton-Jacobi solver on a Cartesian background mesh. The edge-based solver does not capture the continuous nature of the propagating fronts.

for a 100×100 grid, and it is clear that the edge-based solver is highly inaccurate since it does not take into account the continuous nature of the problem. It has a maximum error of 7.79, compared to 0.38 and 0.10 for the Hamilton-Jacobi solvers. This is similar to the error in solving the Eikonal equation using Dijkstra's shortest path algorithm instead of the continuous fast marching method [11]. The error with the edge-based solver might be even larger for unstructured background meshes which often have low element qualities.

7. OTHER APPLICATIONS

In this section we show some special applications of mesh size functions and the gradient limiting equation – numerical adaptation, mesh generation for images, and non-constant g values.

7.1 Numerical Adaptation

Numerical adaptation is a technique for solving PDEs using mesh size functions that are automatically generated to reduce the discretization error. From an error estimator in each element, a new mesh size function is computed. The mesh can then be updated, either by local refinements or remeshing. The procedure is repeated until the desired accuracy is achieved.

One problem when regenerating the mesh is that the size function $h(\mathbf{x})$ from the adaptive solver might be highly irregular. The error estimation often varies between neighboring elements, giving high gradients also in the size function. A simple solution is to smooth the size function, e.g. using Laplacian smoothing. However, this introduces large deviations from the original size function, even where the gradient is small. A better method is to use gradient limiting and solve (8) on the same unstructured mesh that the size function is defined on, see [28] for further details.

Figure 12 shows an example of adaptive meshing for a compressible flow simulation over a bump at Mach 0.95. We solve the Euler equations with a finite volume solver, and use a simple adaptive scheme based on second-derivatives of the density to determine new size functions [1]. These resolve the shock accurately but the sizes increase sharply away from the shock, giving low-quality triangles (top figure). After gradient limiting the mesh size function is well-behaved and a high-quality mesh can be generated (bottom figure). We have also generated meshes for this problem using the advancing front method. With the original size function we were unable to create a mesh because of the large gradients, but after gradient limiting we obtained a well-shaped mesh.

7.2 Meshing Images

Images are special cases of implicit geometry definitions, since the boundaries of objects in the image are not available in an explicit form. These object boundaries can be detected by edge detection methods [29], but these typically work on a pixel level and do not produce smooth boundaries. A more natural approach is to keep the image-based representation, and form an implicit function with a level set representing the boundary.

Before doing this, we have to identify the objects that should be part of the domain, in other words to *segment* the image. Many methods have been developed for this, and we use the standard tools available in image manipulation programs. This will result in a new, binary image, which represents our domain. We also mention that image segmentation based on the level set method, for example Chan and Vese's active contours without edges [30], might be a good alternative, since they produce distance functions directly from the segmentation.

Given a binary image A with values 0 for pixels outside the domain and 1 for those inside, we smooth the image with a low-pass filter and create the signed distance function for the domain using approximate projections and the fast marching method (see [4] for more details).

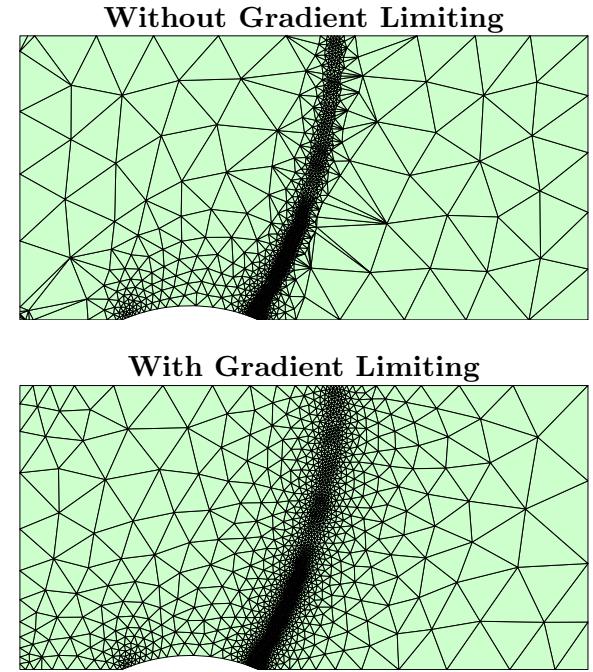


Figure 12: Numerical adaptation for compressible flow over a bump at Mach 0.95. The second-derivative based error estimator resolves the shock accurately, but gradient limiting is required to generate a new mesh of high quality.

Figure 13 shows an example with a picture of a few objects taken with a standard digital camera. We isolate the objects using the segmentation feature of an image manipulation program, and create a binary mask. Next we create the distance function as described above, and a good mesh size function based on curvature, feature size from the medial axis (shown in Figure 3), and gradient limiting. For the skeletonization we increase κ_{tol} to compensate for the slightly noisy distance function close to the boundary.

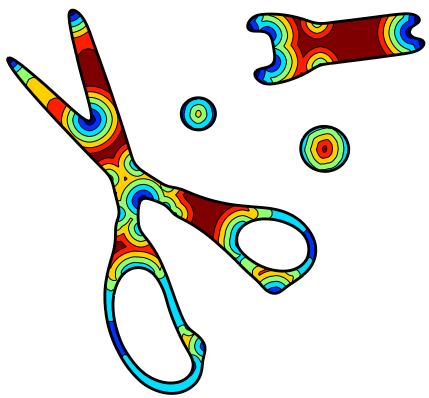
All techniques used for meshing the two dimensional images extend directly to higher dimensions. The image is then a three-dimensional array of pixels, and the binary mask selects a subvolume. Examples of this are the sampled density values produced by computed tomography (CT) scans in medical imaging, which we created mesh size functions and tetrahedral meshes for in [4].

7.3 Space and Solution Dependent g

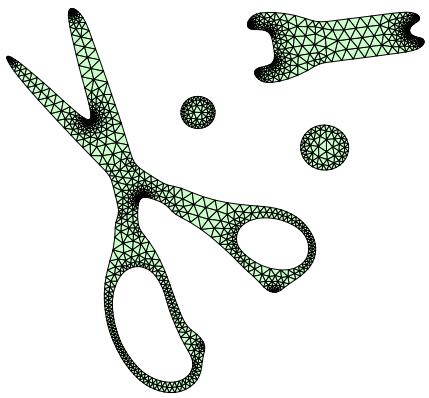
The solution of the gradient limiting equation remains well-defined if we make $g(\mathbf{x})$ a function of space. The numerical schemes in Section 5.2 are still valid, and we replace for example g in (16) with g_{ijk} . An application



Original Image

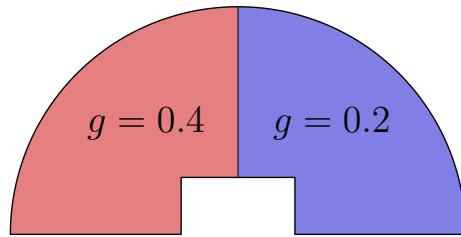


Size Function

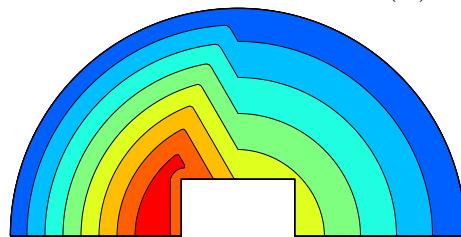


Triangular Mesh

Maximum Gradient $g(\mathbf{x})$



Mesh Size Function $h(\mathbf{x})$



Mesh Based on $h(\mathbf{x})$

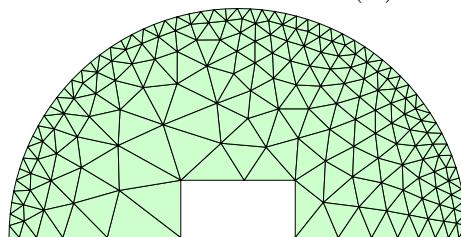


Figure 14: Gradient limiting with space-dependent $g(\mathbf{x})$.

of this is when some regions of the geometry require higher element qualities, and therefore also a smaller maximum gradient in the size function.

Figure 14 shows a simple example. The initial mesh size h_0 is based on curvatures and feature sizes. The left and the right parts of the region have different values of g , and the gradient limiting generates a new size function h satisfying $|\nabla h| \leq g(\mathbf{x})$ everywhere.

Another possible extension is to let g be a function of the solution $h(\mathbf{x})$ (although it is then not clear if the gradient limiting equation has one unique solution). This can be used, for example, to get a fast increase for small element sizes but smaller variations for large elements. In a numerical solver this might be compensated by the smaller truncation error for the small elements. A simple example is shown in Figure 15, where $g(h)$ varies smoothly between 0.6 (for small elements) and 0.2 (for large elements).

Figure 13: Meshing objects in an image. The segmentation is done with an image manipulation program, the distance function is computed by smoothing and approximate projections, and the size function uses the curvature, the feature size, and gradient limiting.

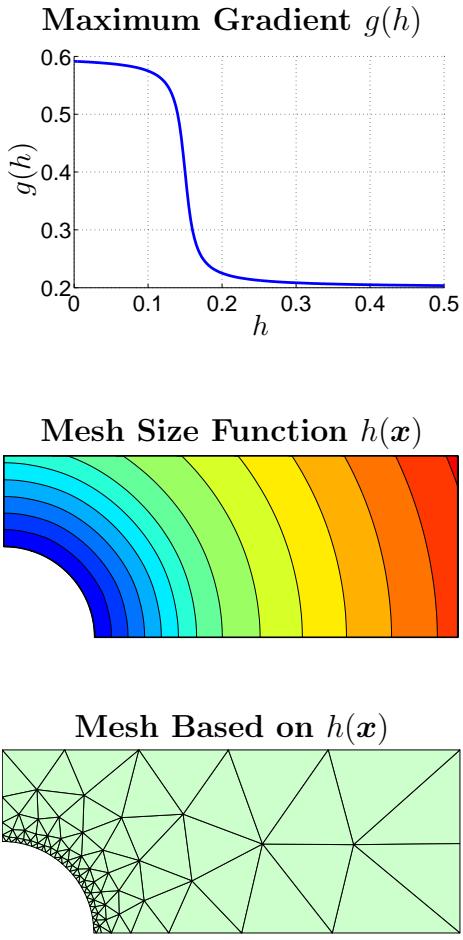


Figure 15: Gradient limiting with solution-dependent $g(h)$. The distances between the level sets of $h(\mathbf{x})$ are smaller for small h , giving a faster increase in mesh size.

In the iterative solver, we replace g with $g(h_{ijk})$, and if the iterations converge we have obtained a solution. In the fast solver, we solve a (scalar) non-linear equation $\nabla_{ijk}^+ = g(h_{ijk})$ at every update.

8. CONCLUSIONS

We have presented new techniques for automatic generation of mesh size functions. The distance function is used to compute the medial axis transform, from which the local feature size is derived. We introduced a new, continuous formulation of the gradient limiting procedure, which is an important part in the generation of good mesh size functions. We showed several examples of high-quality meshes generated with our mesh size functions, and we gave an example of gradient limiting for an adaptive finite element solver and mesh generation for regions in images.

We give several suggestions for future development:

- Other feature size algorithms. We have experimented with a PDE-based algorithm that convect the distance function along its characteristics, and a simpler direct algorithm that finds the largest spheres without first extracting the medial axis.
- Other background meshes for the feature size. We have worked exclusively with Cartesian and octree grids for the medial axis based feature size calculations, and a version for unstructured meshes would be useful.
- Implementing a fast marching based solver for triangular/tetrahedral background meshes. The methods described in [26] and [27] should be applicable in a straightforward way.
- Extending the gradient limiting to anisotropic mesh size functions. There might be a PDE similar to the gradient limiting equation (or a system of PDEs) based on general metrics [20].
- Adaptive generation of background meshes. Zhu et al [8] discussed an intuitive, iterative approach for refinement of background meshes. With our PDE-based formulation, we can achieve this in a strict and systematic way by applying error estimators for numerical adaptive solvers [31] on the discretized solution $h(\mathbf{x})$.

References

- [1] Peraire J., Vahdati M., Morgan K., Zienkiewicz O.C. “Adaptive Remeshing for Compressible Flow Computations.” *J. Comput. Phys.*, vol. 72, no. 2, 449–466, 1987
- [2] Blacker T.D., Stephenson M.B. “Paving: A New Approach to Automated Quadrilateral Mesh Generation.” *Internat. J. Numer. Methods Engrg.*, vol. 32, 811–847, 1991
- [3] Persson P.O., Strang G. “A Simple Mesh Generator in MATLAB.” *SIAM Review*, vol. 46, no. 2, 329–345, June 2004
- [4] Persson P.O. *Mesh Generation for Implicit Geometries*. Ph.D. thesis, Massachusetts Institute of Technology, 2005
- [5] Ruppert J. “A Delaunay refinement algorithm for quality 2-dimensional mesh generation.” *J. Algorithms*, vol. 18, no. 3, 548–585, 1995
- [6] Shewchuk J.R. “Delaunay refinement algorithms for triangular mesh generation.” *Comput. Geom.*, vol. 22, no. 1-3, 21–74, 2002

- [7] Owen S.J., Saigal S. “Surface mesh sizing control.” *Internat. J. Numer. Methods Engrg.*, vol. 47, no. 1-3, 497–511, 2000
- [8] Zhu J., Blacker T., Smith R. “Background Overlay Grid Size Functions.” *Proceedings of the 11th International Meshing Roundtable*, pp. 65–74. Sandia Nat. Lab., September 2002
- [9] Zhu J. “A New Type of Size Function Respecting Premeshed Entities.” *Proceedings of the 11th International Meshing Roundtable*, pp. 403–413. Sandia Nat. Lab., September 2003
- [10] Osher S., Sethian J.A. “Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations.” *J. Comput. Phys.*, vol. 79, no. 1, 12–49, 1988
- [11] Sethian J.A. “A fast marching level set method for monotonically advancing fronts.” *Proc. Nat. Acad. Sci. U.S.A.*, vol. 93, no. 4, 1591–1595, 1996
- [12] Tsitsiklis J.N. “Efficient algorithms for globally optimal trajectories.” *IEEE Trans. Automat. Control*, vol. 40, no. 9, 1528–1538, 1995
- [13] Mauch S.P. *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations*. Ph.D. thesis, Caltech, 2003
- [14] Sussman M., Smereka P., Osher S. “A level set approach for computing solutions to incompressible two-phase flow.” *J. Comput. Phys.*, vol. 114, no. 1, 146–159, 1994
- [15] Amenta N., Bern M. “Surface reconstruction by Voronoi filtering.” *SCG ’98: Proceedings of the fourteenth annual symposium on Computational geometry*, pp. 39–48. ACM Press, 1998
- [16] Blum H. “Biological Shape and Visual Science (Part I).” *Journal of Theoretical Biology*, vol. 38, 205–287, 1973
- [17] Kimmel R., Shaked D., Kiryati N., Bruckstein A.M. “Skeletonization via Distance Maps and Level Sets.” *Computer Vision and Image Understanding: CVIU*, vol. 62, no. 3, 382–391, 1995
- [18] Siddiqi K., Bouix S., Tannenbaum A., Zucker S. “The Hamilton-Jacobi Skeleton.” *International Conference on Computer Vision (ICCV)*, pp. 828–834. 1999
- [19] Rumpf M., Telea A. “A continuous skeletonization method based on level sets.” *Proceedings of the symposium on Data Visualisation 2002*, pp. 151–ff. Eurographics Association, 2002
- [20] Borouchaki H., Hecht F., Frey P.J. “Mesh Gradiation Control.” *Proceedings of the 6th International Meshing Roundtable*, pp. 131–141. Sandia Nat. Lab., October 1997
- [21] Frey P.J., Marechal L. “Fast Adaptive Quadtree Mesh Generation.” *Proceedings of the 7th International Meshing Roundtable*, pp. 211–224. Sandia Nat. Lab., October 1998
- [22] Yerry M.A., Shephard M.S. “A Modified Quadtree Approach to Finite Element Mesh Generation.” *IEEE Comp. Graph. Appl.*, vol. 3, no. 1, 39–46, 1983
- [23] Hopf E. “Generalized solutions of non-linear equations of first order.” *J. Math. Mech.*, vol. 14, 951–973, 1965
- [24] Harten A., Engquist B., Osher S., Chakravarthy S.R. “Uniformly High Order Accurate Essentially Non-Oscillatory Schemes.” *J. Comput. Phys.*, vol. 71, no. 2, 231–303, 1987
- [25] Barth T.J., Sethian J.A. “Numerical schemes for the Hamilton-Jacobi and level set equations on triangulated domains.” *J. Comput. Phys.*, vol. 145, no. 1, 1–40, 1998
- [26] Kimmel R., Sethian J.A. “Fast Marching Methods on Triangulated Domains.” *Proceedings of the National Academy of Sciences*, vol. 95, pp. 8341–8435. 1998
- [27] Covello P., Rodrigue G. “A generalized front marching algorithm for the solution of the eikonal equation.” *J. Comput. Appl. Math.*, vol. 156, no. 2, 371–388, 2003
- [28] Persson P.O. “Size Functions and Mesh Generation for High-Quality Adaptive Remeshing.” *Proc. of the Third MIT Conference on Computational Fluid and Solid Mechanics*. Cambridge, MA, 2005
- [29] Gonzalez R.C., Woods R.E. *Digital Image Processing*. Prentice Hall, 2 edn., 2002
- [30] Chan T., Vese L. “Active contours without edges.” *IEEE Trans. Image Processing*, vol. 10, no. 2, 266–277, 2001
- [31] Eriksson K., Estep D., Hansbo P., Johnson C. *Computational differential equations*. Cambridge University Press, Cambridge, 1996