Mesh Generation for Implicit Geometries

by

Per-Olof Persson

M.S. Engineering Physics, Lund Institute of Technology, 1997

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2005

Author .....
Department of Mathematics
December 8, 2004

Certified by .....
Alan Edelman
Professor of Applied Mathematics
Thesis Supervisor

Certified by .....
Gilbert Strang
Professor of Mathematics
Thesis Supervisor

Accepted by .....
Rodolfo Ruben Rosales
Chairman, Committee on Applied Mathematics

Accepted by .....
Pavel Etingof
Chairman, Department Committee on Graduate Students

Mesh Generation for Implicit Geometries

by

Per-Olof Persson

Submitted to the Department of Mathematics
on December 8, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

We present new techniques for generation of unstructured meshes for geometries specified by implicit functions. An initial mesh is iteratively improved by solving for a force equilibrium in the element edges, and the boundary nodes are projected using the implicit geometry definition. Our algorithm generalizes to any dimension and it typically produces meshes of very high quality. We show a simplified version of the method in just one page of MATLAB code, and we describe how to improve and extend our implementation.

Prior to generating the mesh we compute a mesh size function to specify the desired size of the elements. We have developed algorithms for automatic generation of size functions, adapted to the curvature and the feature size of the geometry. We propose a new method for limiting the gradients in the size function by solving a non-linear partial differential equation. We show that the solution to our gradient limiting equation is optimal for convex geometries, and we discuss efficient methods to solve it numerically.

The iterative nature of the algorithm makes it particularly useful for moving meshes, and we show how to combine it with the level set method for applications in fluid dynamics, shape optimization, and structural deformations. It is also appropriate for numerical adaptation, where the previous mesh is used to represent the size function and as the initial mesh for the refinements. Finally, we show how to generate meshes for regions in images by using implicit representations.

Thesis Supervisor: Alan Edelman
Title: Professor of Applied Mathematics

Thesis Supervisor: Gilbert Strang
Title: Professor of Mathematics

Acknowledgments

I would like to begin by expressing my thanks to my advisors Alan Edelman and Gilbert Strang. Their encouragement and unfailing support have been most valuable to me during these years, and I feel deeply privileged for the opportunity to work with them. I also gratefully acknowledge the other members of my thesis committee, Ruben Rosales and Dan Spielman.

My research has been highly stimulated by discussions with John Gilbert and Jaime Peraire, and I am very grateful for their valuable advise. My good friends and colleagues Pavel Grinfeld and Björn Sjödin always show interest in my work and our discussions have given my many new ideas. I also appreciate all the suggestions and comments from students and researchers around the world who are using my mesh generation software.

My deepest gratitude goes to my wife Kristin, for doing this journey with me and for always encouraging and believing in me. My life would not be complete without our two lovely daughters Ellen and Sara. I am also thankful to our families and friends in Sweden for all their support.

This research was supported in part by Ernold Lundströms stiftelse, Sixten Gemzéus stiftelse, and by an appointment to the Student Research Participation Program at the USARL administered by ORISE.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Creating a mesh is the first step in a wide range of applications, including scientific computing and computer graphics. An unstructured simplex mesh requires a choice of meshpoints (vertex nodes) and a triangulation. The mesh should have small elements to resolve the fine details of the geometry, but larger sizes where possible to reduce the total number of nodes. Furthermore, in numerical applications we need elements of high quality (for example triangles that are almost equilateral) to get accurate discretizations and well conditioned linear systems.

One of the most popular meshing algorithms is the Delaunay refinement method [53], [59], which starts from an initial triangulation and refines until the element qualities are sufficiently high. It works in a bottom-up fashion, by meshing points, curves, surfaces, and finally the volume. The advancing front method [48] also starts from the boundaries, and creates mesh elements along a front moving into the domain. Both these algorithms require geometry boundaries that are specified explicitly, by for example a polygon in two dimensions, a triangulated surface mesh in three dimensions, or more general parameterized boundaries such as rational Bézier curves and surfaces.

For geometries with boundaries described in an implicit form $\phi(\boldsymbol{x}) = 0$, we need other techniques for creating well-shaped meshes. We do not have a direct representation of the boundary, and the bottom-up approaches of Delaunay refinement and advancing front are harder to implement. A more natural way to mesh implicit geometries is to let the mesh generator work directly with the function $\phi(\boldsymbol{x})$. We can

determine if a point $\boldsymbol{x}$ is inside or outside our domain from the sign of $\phi(\boldsymbol{x})$, and we can project points to the closest boundary by using the gradient $\nabla\phi(\boldsymbol{x})$.

These ideas form the basis of our mesh generator. We improve an initial mesh by solving for equilibrium in a truss structure using piecewise linear force-displacement relations. During the node movements, we change the mesh connectivity to improve the element qualities, either by recomputing the Delaunay triangulation or by local updates. At equilibrium, the mesh elements tend to be of very high quality, and the method generalizes to higher dimensions. Our algorithm is easy to understand and to implement, and we describe how to write a complete MATLAB version in only a few dozen lines of code.

The desired sizes of the mesh elements are described by a mesh size function $h(\boldsymbol{x})$, which we use to determine the equilibrium lengths of the edges. We show how to create size functions automatically from discretized implicit geometries. The feature size is computed from the medial axis transform, which we detect as shocks in the distance function. To avoid large variations in element sizes, we limit the gradients in $h(\boldsymbol{x})$ by numerically solving a non-linear partial differential equation, the *gradient limiting equation*. This Hamilton-Jacobi equation simplifies the generation of mesh size functions significantly, since it decouples size constraints at specific locations from the mesh grading requirements.

We have identified many application areas for our mesh generator, based on the iterative formulation and the implicit geometries. We can for example combine it with the level set method to solve problems with evolving interfaces. These moving meshes are particularly well suited for our iterative mesh generation, since at each time step we only have to do a few additional iterations to improve the element qualities. We use this to solve shape optimization problems and to simulate physical phenomena with moving boundaries. A related application is mesh refinement for numerical adaptation, where again we have good initial meshes at each adaptation step.

Finally, we show how to mesh domains that are given in the form of an image. This could be any image such as a photograph, a satellite image, or a three dimensional

image from a computed tomography (CT) scan. Since the image is already an implicit description of the domain, we work directly with this form rather than extracting the boundaries. We use image manipulation techniques to segment and smooth the image, and we compute the distance function using approximate projections and the fast marching method. From this we can create mesh size functions adapted to the geometry, and generate high-quality meshes in both two and three dimensions.

1.1 Prior Work

Many different mesh generation algorithms have been developed, and some of the most popular ones are described in the surveys by Bern and Plassmann [4] and Owen [46]. These methods usually work with explicit geometry representations, although many techniques have been proposed for triangulation of implicit surfaces [7], [19], [20], [65], [69]. Two more recent publications on mesh generation for level sets are Gloth and Vilsmeier [27] and Molino et al [41].

Our iterative mesh generator is closely related to mesh smoothing, for example Laplacian smoothing [23], Winslow smoothing [68], and quality optimization techniques like those in the Mesquite toolkit [12]. These move the nodes to improve the mesh, and possibly also update the mesh connectivities [25]. As far as we know, all these techniques operate on curves, surfaces, and volumes individually, while our method moves all the nodes simultaneously and use the implicit geometry to project the boundary nodes. The bubble mesh technique by Shimada and Gossard [61] also uses the idea of forces between nodes to find good node locations, which are then triangulated. Our improvement technique is most similar to Laplace-Delaunay smoothing [23] and the pliant method by Heckbert and Bossen [11], which also move nodes and update element connectivities.

The techniques we describe for discretization and initialization of distance functions are well known in the level set community, see for example the books by Sethian [56] and Osher and Fedkiw [43]. Local mesh manipulations are used by most mesh generators, including Delaunay refinement. The method of subdividing a regular el-

ement to create the initial mesh is similar to the initial stage of the quadtree and the octree mesh generation algorithms [71]. We are not aware of any other methods to align elements with implicit internal boundaries. Heckbert and Bossen [11] and Borouchaki et al [9], [10] studied anisotropic meshes, and generation of surface meshes is a well studied area as mentioned above.

Prior work on mesh size functions include the work by Peraire et al on the advancing front method and background grids [48], as well as several other later contributions [47], [73], [72], [50]. None of these methods generate mesh size functions for implicit geometries. Skeletonization with distance functions was studied in [36], [62], [52], and some of these claim to achieve subgrid accuracy. We are not aware of any other work related to continuous gradient limiting. Bourochaki et al [10] used simple discrete edge iterations to approximate the true continuous problem.

Shape optimization has been studied by Murat and Simon [42] and Pironneau [49], among others. The idea of using the level set method for the interface motion was used by Sethian and Wiegmann [57], Osher and Santosa, [44], Allaire et al [1], and Wang et al [67]. Our approach of combining the level set method with finite elements on unstructured meshes appears to be new.

There appears to be a growing interest in mesh generation for images, in particular for medical imaging, and a few references are [13], [32], [5]. The thesis ends with a look ahead at applications to medical images.

Chapter 2

A Simple Mesh Generator in MATLAB

2.1 Introduction

Mesh generators tend to be complex codes that are nearly inaccessible. They are often just used as "black boxes". The meshing software is difficult to integrate with other codes – so the user gives up control. We believe that the ability to understand and adapt a mesh generation code (as one might do with visualization, or a finite element or finite volume code, or geometry modeling in computer graphics) is too valuable an option to lose.

Our goal is to develop a mesh generator that can be described in a few dozen lines of MATLAB. We could offer faster implementations, and refinements of the algorithm, but our chief hope is that users will take this code as a starting point for their own work. It is understood that the software cannot be fully state-of-the-art, but it can be simple and effective and public.

An essential decision is how to represent the geometry (the shape of the region). Our code uses a *signed distance function* $d(x, y)$, negative inside the region. We show in detail how to write the distance to the boundary for simple shapes, and how to combine those functions for more complex objects. We also show how to compute the distance to boundaries that are given implicitly by equations $f(x, y) = 0$, or by

values of $d(x, y)$ at a discrete set of meshpoints.

For the actual mesh generation, our iterative technique is based on the physical analogy between a simplex mesh and a truss structure. Meshpoints are nodes of the truss. Assuming an appropriate force-displacement function for the bars in the truss at each iteration, we solve for equilibrium. The forces move the nodes, and (iteratively) the Delaunay triangulation algorithm adjusts the topology (it decides the edges). Those are the two essential steps. The resulting mesh is surprisingly well-shaped, and Figure 2-4 shows examples. Other codes use *Laplacian smoothing* [23] for mesh enhancements, usually without retriangulations. This could be regarded as a force-based method, and related mesh generators were investigated by Bossen and Heckbert [11]. We mention Triangle [58] as a robust and freely available Delaunay refinement code.

The combination of distance function representation and node movements from forces turns out to be good. The distance function quickly determines if a node is inside or outside the region (and if it has moved outside, it is easy to determine the closest boundary point). Thus $d(x, y)$ is used extensively in our implementation, to find the distance to that closest point.

Apart from being simple, it turns out that our algorithm generates meshes of high quality. The edge lengths should be close to the relative size $h(\boldsymbol{x})$ specified by the user (the lengths are nearly equal when the user chooses $h(\boldsymbol{x}) = 1$). Compared to typical Delaunay refinement algorithms, our force equilibrium tends to give much higher values of the mesh quality q, at least for the cases we have studied.

We begin by describing the algorithm and the equilibrium equations for the truss. Next, we present the complete MATLAB code for the two-dimensional case, and describe every line in detail. In Section 2.5, we create meshes for increasingly complex geometries. Finally, we describe the n-dimensional generalization and show examples of 3-D and 4-D meshes.

2.2 The Algorithm

In the plane, our mesh generation algorithm is based on a simple mechanical analogy between a triangular mesh and a 2-D truss structure, or equivalently a structure of springs. Any set of points in the x, y-plane can be triangulated by the Delaunay algorithm [21]. In the physical model, the edges of the triangles (the connections between pairs of points) correspond to bars, and the points correspond to joints of the truss. Each bar has a force-displacement relationship $f(\ell, \ell_0)$ depending on its current length ℓ and its unextended length ℓ_0.

The external forces on the structure come at the boundaries. At every boundary node, there is a reaction force acting normal to the boundary. The magnitude of this force is just large enough to keep the node from moving outside. The positions of the joints (these positions are our principal unknowns) are found by solving for a static force equilibrium in the structure. The hope is that (when $h(x, y) = 1$) the lengths of all the bars at equilibrium will be nearly equal, giving a well-shaped triangular mesh.

To solve for the force equilibrium, collect the x- and y-coordinates of all N meshpoints into an N-by-2 array $\boldsymbol{p}$:

$$\boldsymbol{p} = \begin{bmatrix} \boldsymbol{x} & \boldsymbol{y} \end{bmatrix} \tag{2.1}$$

The force vector $\boldsymbol{F}(\boldsymbol{p})$ has horizontal and vertical components at each meshpoint:

$$\boldsymbol{F}(\boldsymbol{p}) = \begin{bmatrix} \boldsymbol{F}_{\text{int},x}(\boldsymbol{p}) & \boldsymbol{F}_{\text{int},y}(\boldsymbol{p}) \end{bmatrix} + \begin{bmatrix} \boldsymbol{F}_{\text{ext},x}(\boldsymbol{p}) & \boldsymbol{F}_{\text{ext},y}(\boldsymbol{p}) \end{bmatrix} \tag{2.2}$$

where $\boldsymbol{F}_{\text{int}}$ contains the internal forces from the bars, and $\boldsymbol{F}_{\text{ext}}$ are the external forces (reactions from the boundaries). The first column of $\boldsymbol{F}$ contains the x-components of the forces, and the second column contains the y-components.

Note that $\boldsymbol{F}(\boldsymbol{p})$ depends on the topology of the bars connecting the joints. In the algorithm, this structure is given by the Delaunay triangulation of the meshpoints. The Delaunay algorithm determines non-overlapping triangles that fill the convex hull of the input points, such that every edge is shared by at most two triangles, and

the circumcircle of every triangle contains no other input points. In the plane, this triangulation is known to maximize the minimum angle of all the triangles. The force vector $\boldsymbol{F}(\boldsymbol{p})$ is not a continuous function of $\boldsymbol{p}$, since the topology (the presence or absence of connecting bars) is changed by Delaunay as the points move.

The system $\boldsymbol{F}(\boldsymbol{p}) = \boldsymbol{0}$ has to be solved for a set of equilibrium positions $\boldsymbol{p}$. This is a relatively hard problem, partly because of the discontinuity in the force function (change of topology), and partly because of the external reaction forces at the boundaries.

A simple approach to solve $\boldsymbol{F}(\boldsymbol{p}) = \boldsymbol{0}$ is to introduce an artificial time-dependence. For some $\boldsymbol{p}(0) = \boldsymbol{p}_0$, we consider the system of ODEs (in non-physical units!)

$$\frac{d\boldsymbol{p}}{dt} = \boldsymbol{F}(\boldsymbol{p}), \qquad t \geq 0. \tag{2.3}$$

If a stationary solution is found, it satisfies our system $\boldsymbol{F}(\boldsymbol{p}) = \boldsymbol{0}$. The system (2.3) is approximated using the forward Euler method. At the discretized (artificial) time $t_n = n\Delta t$, the approximate solution $\boldsymbol{p}_n \approx \boldsymbol{p}(t_n)$ is updated by

$$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n + \Delta t \boldsymbol{F}(\boldsymbol{p}_n). \tag{2.4}$$

When evaluating the force function, the positions $\boldsymbol{p}_n$ are known and therefore also the truss topology (triangulation of the current point-set). The external reaction forces enter in the following way: All points that go outside the region during the update from $\boldsymbol{p}_n$ to $\boldsymbol{p}_{n+1}$ are moved back to the closest boundary point. This conforms to the requirement that forces act normal to the boundary. The points can move along the boundary, but not go outside.

There are many alternatives for the force function $f(\ell, \ell_0)$ in each bar, and several choices have been investigated [11], [61]. The function $k(\ell_0 - \ell)$ models ordinary linear springs. Our implementation uses this linear response for the *repulsive* forces but it

allows no attractive forces:

$$f(\ell, \ell_0) = \begin{cases} k(\ell_0 - \ell) & \text{if } \ell < \ell_0, \\ 0 & \text{if } \ell \geq \ell_0. \end{cases} \tag{2.5}$$

Slightly nonlinear force-functions might generate better meshes (for example with $k = (\ell + \ell_0)/2\ell_0$), but the piecewise linear force turns out to give good results (k is included to give correct units; we set $k = 1$). It is reasonable to require $f = 0$ for $\ell = \ell_0$. The proposed treatment of the boundaries means that no points are forced to stay at the boundary, they are just prevented from crossing it. It is therefore important that *most of the bars give repulsive forces* $f > 0$, to help the points spread out across the whole geometry. This means that $f(\ell, \ell_0)$ should be positive when ℓ is near the desired length, which can be achieved by choosing ℓ_0 slightly larger than the length we actually desire (a good default in 2-D is 20%, which yields `Fscale`=1.2).

For uniform meshes ℓ_0 is constant. But there are many cases when it is advantageous to have different sizes in different regions. Where the geometry is more complex, it needs to be resolved by small elements (*geometrical adaptivity*). The solution method may require small elements close to a singularity to give good global accuracy (*adaptive solver*). A uniform mesh with these small elements would require too many nodes.

In our implementation, the desired edge length distribution is provided by the user as an *element size function* $h(x, y)$. Note that $h(x, y)$ does not have to equal the actual size; it gives the *relative* distribution over the domain. This avoids an implicit connection with the number of nodes, which the user is not asked to specify. For example, if $h(x, y) = 1 + x$ in the unit square, the edge lengths close to the left boundary ($x = 0$) will be about half the edge lengths close to the right boundary ($x = 1$). This is true regardless of the number of points and the actual element sizes. To find the scaling, we compute the ratio between the mesh area from the actual edge

lengths ℓ_i and the "desired size" (from $h(x,y)$ at the midpoints (x_i, y_i) of the bars):

$$\text{Scaling factor} = \left(\frac{\sum \ell_i^2}{\sum h(x_i, y_i)^2} \right)^{1/2}. \tag{2.6}$$

We will assume here that $h(x,y)$ is specified by the user. It could also be created using adaptive logic to implement the *local feature size*, which is roughly the distance between the boundaries of the region (see example **5** below). For highly curved boundaries, $h(x,y)$ could be expressed in terms of the curvature computed from $d(x,y)$. An adaptive solver that estimates the error in each triangle can choose $h(x,y)$ to refine the mesh for good solutions.

The initial node positions $\boldsymbol{p}_0$ can be chosen in many ways. A random distribution of the points usually works well. For meshes intended to have uniform element sizes (and for simple geometries), good results are achieved by starting from equally spaced points. When a non-uniform size distribution $h(x,y)$ is desired, the convergence is faster if the initial distribution is weighted by probabilities proportional to $1/h(x,y)^2$ (which is the density). Our *rejection method* starts with a uniform initial mesh inside the domain, and discards points using this probability.

2.3 Implementation

The complete source code for the two-dimensional mesh generator is in Figure 2-1. Each line is explained in detail below.

The first line specifies the calling syntax for the function `distmesh2d`:

```
function [p,t]=distmesh2d(fd,fh,h0,bbox,pfix,varargin)
```

This meshing function produces the following outputs:

- The node positions `p`. This N-by-2 array contains the x, y coordinates for each of the N nodes.

- The triangle indices `t`. The row associated with each triangle has 3 integer entries to specify node numbers in that triangle.

```
function [p,t]=distmesh2d(fd,fh,h0,bbox,pfix,varargin)

dptol=.001; ttol=.1; Fscale=1.2; deltat=.2; geps=.001*h0; deps=sqrt(eps)*h0;

% 1. Create initial distribution in bounding box (equilateral triangles)
[x,y]=meshgrid(bbox(1,1):h0:bbox(2,1),bbox(1,2):h0*sqrt(3)/2:bbox(2,2));
x(2:2:end,:)=x(2:2:end,:)+h0/2;                    % Shift even rows
p=[x(:),y(:)];                                     % List of node coordinates

% 2. Remove points outside the region, apply the rejection method
p=p(feval(fd,p,varargin{:})<geps,:);               % Keep only d<0 points
r0=1./feval(fh,p,varargin{:}).^2;                  % Probability to keep point
p=[pfix; p(rand(size(p,1),1)<r0./max(r0),:)];      % Rejection method
N=size(p,1);                                       % Number of points N

pold=inf;                                          % For first iteration
while 1
  % 3. Retriangulation by the Delaunay algorithm
  if max(sqrt(sum((p-pold).^2,2))/h0)>ttol         % Any large movement?
    pold=p;                                        % Save current positions
    t=delaunayn(p);                                % List of triangles
    pmid=(p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:))/3;  % Compute centroids
    t=t(feval(fd,pmid,varargin{:})<-geps,:);       % Keep interior triangles
    % 4. Describe each bar by a unique pair of nodes
    bars=[t(:,[1,2]);t(:,[1,3]);t(:,[2,3])];       % Interior bars duplicated
    bars=unique(sort(bars,2),'rows');              % Bars as node pairs
    % 5. Graphical output of the current mesh
    trimesh(t,p(:,1),p(:,2),zeros(N,1))
    view(2),axis equal,axis off,drawnow
  end

  % 6. Move mesh points based on bar lengths L and forces F
  barvec=p(bars(:,1),:)-p(bars(:,2),:);            % List of bar vectors
  L=sqrt(sum(barvec.^2,2));                        % L = Bar lengths
  hbars=feval(fh,(p(bars(:,1),:)+p(bars(:,2),:))/2,varargin{:});
  L0=hbars*Fscale*sqrt(sum(L.^2)/sum(hbars.^2));   % L0 = Desired lengths
  F=max(L0-L,0);                                   % Bar forces (scalars)
  Fvec=F./L*[1,1].*barvec;                         % Bar forces (x,y components)
  Ftot=full(sparse(bars(:,[1,1,2,2]),ones(size(F))*[1,2,1,2],[Fvec,-Fvec],N,2));
  Ftot(1:size(pfix,1),:)=0;                        % Force = 0 at fixed points
  p=p+deltat*Ftot;                                 % Update node positions

  % 7. Bring outside points back to the boundary
  d=feval(fd,p,varargin{:}); ix=d>0;               % Find points outside (d>0)
  dgradx=(feval(fd,[p(ix,1)+deps,p(ix,2)],varargin{:})-d(ix))/deps; % Numerical
  dgrady=(feval(fd,[p(ix,1),p(ix,2)+deps],varargin{:})-d(ix))/deps; % gradient
  p(ix,:)=p(ix,:)-[d(ix).*dgradx,d(ix).*dgrady];   % Project back to boundary

  % 8. Termination criterion: All interior nodes move less than dptol (scaled)
  if max(sqrt(sum(deltat*Ftot(d<-geps,:).^2,2))/h0)<dptol, break; end
end
```

Figure 2-1: The complete source code for the 2-D mesh generator `distmesh2d.m`.

The input arguments are as follows:

- The geometry is given as a distance function `fd`. This function returns the signed distance from each node location `p` to the closest boundary.
- The (relative) desired edge length function $h(x, y)$ is given as a function `fh`, which returns h for all input points.
- The parameter `h0` is the distance between points in the initial distribution $\boldsymbol{p}_0$. For uniform meshes ($h(x, y) = \text{constant}$), the element size in the final mesh will usually be a little larger than this input.
- The bounding box for the region is an array `bbox`=$[x_{\min}, y_{\min}; x_{\max}, y_{\max}]$.
- The fixed node positions are given as an array `pfix` with two columns.
- Additional parameters to the functions `fd` and `fh` can be given in the last arguments `varargin` (type `help varargin` in MATLAB for more information).

In the beginning of the code, six parameters are set. The default values seem to work very generally, and they can for most purposes be left unmodified. The algorithm will stop when all movements in an iteration (relative to the average bar length) are smaller than `dptol`. Similarly, `ttol` controls how far the points can move (relatively) before a retriangulation by Delaunay.

The "internal pressure" is controlled by `Fscale`. The time step in Euler's method (2.4) is `deltat`, and `geps` is the tolerance in the geometry evaluations. The square root `deps` of the machine tolerance is the Δx in the numerical differentiation of the distance function. This is optimal for one-sided first-differences. These numbers `geps` and `deps` are scaled with the element size, in case someone were to mesh an atom or a galaxy in meter units.

Now we describe steps 1 to 8 in the `distmesh2d` algorithm, as illustrated in Figure 2-2.

1. The first step creates a uniform distribution of nodes within the bounding box of the geometry, corresponding to equilateral triangles:

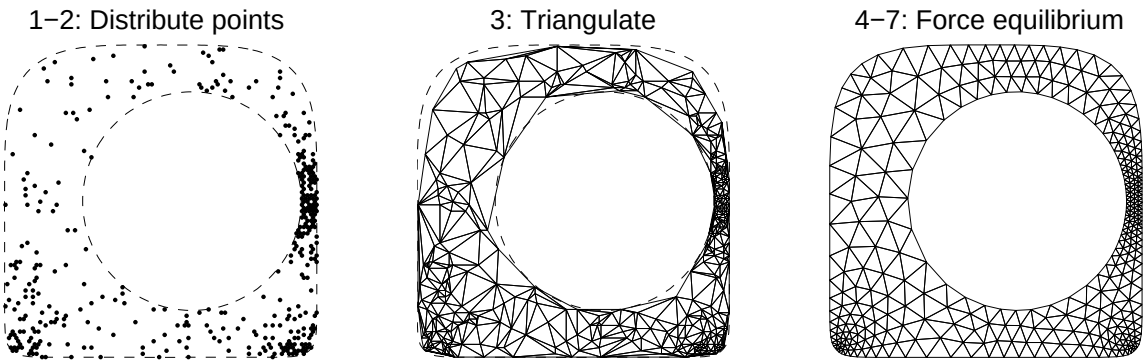

Figure 2-2: The generation of a non-uniform triangular mesh.

```
[x,y]=meshgrid(bbox(1,1):h0:bbox(2,1),bbox(1,2):h0*sqrt(3)/2:bbox(2,2));
x(2:2:end,:)=x(2:2:end,:)+h0/2;                    % Shift even rows
p=[x(:),y(:)];                                     % List of node coordinates
```

The `meshgrid` function generates a rectangular grid, given as two vectors `x` and `y` of node coordinates. Initially the distances are $\sqrt{3}h_0/2$ in the y-direction. By shifting every second row $h_0/2$ to the right, all points will be a distance h_0 from their closest neighbors. The coordinates are stored in the N-by-2 array `p`.

2. The next step removes all nodes outside the desired geometry:

```
p=p(feval(fd,p,varargin{:})<geps,:);               % Keep only d<0 points
```

`feval` calls the distance function `fd`, with the node positions `p` and the additional arguments `varargin` as inputs. The result is a column vector of distances from the nodes to the geometry boundary. Only the interior points with negative distances (allowing a tolerance `geps`) are kept. Then we evaluate $h(x, y)$ at each node and reject points with a probability proportional to $1/h(x, y)^2$:

```
r0=1./feval(fh,p,varargin{:}).^2;                  % Probability to keep point
p=[pfix; p(rand(size(p,1),1)<r0./max(r0),:)];      % Rejection method
N=size(p,1);                                       % Number of points N
```

The user's array of fixed nodes is placed in the first rows of `p`.

3. Now the code enters the main loop, where the location of the N points is iteratively improved. Initialize the variable `pold` for the first iteration, and start the loop (the termination criterion comes later):

```
pold=inf;                                        % For first iteration
while 1
  ...
end
```

Before evaluating the force function, a Delaunay triangulation determines the topology of the truss. Normally this is done for $\boldsymbol{p}_0$, and also every time the points move, in order to maintain a correct topology. To save computing time, an approximate heuristic calls for a retriangulation when the maximum displacement since the last triangulation is larger than `ttol` (relative to the approximate element size ℓ_0):

```
if max(sqrt(sum((p-pold).^2,2))/h0)>ttol         % Any large movement?
  pold=p;                                        % Save current positions
  t=delaunayn(p);                                % List of triangles
  pmid=(p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:))/3;  % Compute centroids
  t=t(feval(fd,pmid,varargin{:})<-geps,:);       % Keep interior triangles
   ...
 end
```

The node locations after retriangulation are stored in `pold`, and every iteration compares the current locations `p` with `pold`. The MATLAB `delaunayn` function generates a triangulation `t` of the convex hull of the point set, and triangles outside the geometry have to be removed. We use a simple solution here – if the centroid of a triangle has $d > 0$, that triangle is removed. This technique is not entirely robust, but it works fine in many cases, and it is very simple to implement.

4. The list of triangles `t` is an array with 3 columns. Each row represents a triangle by three integer indices (in no particular order). In creating a list of edges, each triangle contributes three node pairs. Since most pairs will appear twice (the edges are in two triangles), duplicates have to be removed:

```
bars=[t(:,[1,2]);t(:,[1,3]);t(:,[2,3])];         % Interior bars duplicated
bars=unique(sort(bars,2),'rows');                % Bars as node pairs
```

5. The next two lines give graphical output after each retriangulation. (They can be moved out of the `if`-statement to get more frequent output.) See the MATLAB help texts for details about these functions:

```
trimesh(t,p(:,1),p(:,2),zeros(N,1))
view(2),axis equal,axis off,drawnow
```

6. Each bar is a two-component vector in `barvec`; its length is in `L`.

```
barvec=p(bars(:,1),:)-p(bars(:,2),:);          % List of bar vectors
L=sqrt(sum(barvec.^2,2));                      % L = Bar lengths
```

The desired lengths `L0` come from evaluating $h(x, y)$ at the midpoint of each bar. We multiply by the scaling factor in (2.6) and the fixed factor `Fscale`, to ensure that most bars give repulsive forces $f > 0$ in `F`.

```
hbars=feval(fh,(p(bars(:,1),:)+p(bars(:,2),:))/2,varargin{:});
L0=hbars*Fscale*sqrt(sum(L.^2)/sum(hbars.^2));      % L0 = Desired lengths
F=max(L0-L,0);                                      % Bar forces (scalars)
```

The actual update of the node positions `p` is in the next block of code. The force resultant `Ftot` is the sum of force vectors in `Fvec`, from all bars meeting at a node. A stretching force has positive sign, and its direction is given by the two-component vector in `bars`. The `sparse` command is used (even though `Ftot` is immediately converted to a dense array!), because of the nice summation property for duplicated indices.

```
Fvec=F./L*[1,1].*barvec;                       % Bar forces (x,y comp.)
Ftot=full(sparse(bars(:,[1,1,2,2]),ones(size(F))*[1,2,1,2],[Fvec,-Fvec],N,2));
Ftot(1:size(pfix,1),:)=0;                      % Force = 0 at fixed points
p=p+deltat*Ftot;                               % Update node positions
```

Note that `Ftot` for the fixed nodes is set to zero. Their coordinates are unchanged in `p`.

7. If a point ends up outside the geometry after the update of `p`, it is moved back to the closest point on the boundary (using the distance function). This corresponds to a reaction force normal to the boundary. Points are allowed to move tangentially along the boundary. The gradient of $d(x, y)$ gives the (negative) direction to the closest boundary point, and it comes from numerical differentiation:

```
d=feval(fd,p,varargin{:}); ix=d>0;                    % Find points outside (d>0)
dgradx=(feval(fd,[p(ix,1)+deps,p(ix,2)],varargin{:})-d(ix))/deps; % Num.
dgrady=(feval(fd,[p(ix,1),p(ix,2)+deps],varargin{:})-d(ix))/deps; % gradient
p(ix,:)=p(ix,:)-[d(ix).*dgradx,d(ix).*dgrady];        % Project back to boundary
```

8. Finally, the termination criterion is based on the maximum node movement in the current iteration (excluding the boundary points):

```
if max(sqrt(sum(deltat*Ftot(d<-geps,:).^2,2))/h0)<dptol, break; end
```

This criterion is sometimes too tight, and a high-quality mesh is often achieved long before termination. In these cases, the program can be interrupted manually, or other tests can be used. One simple but efficient test is to compute all the element qualities (see below), and terminate if the smallest quality is large enough.

2.4 Special Distance Functions

The function `distmesh2d` is everything that is needed to mesh a region specified by the distance $d(x, y)$ to the boundary. While it is easy to create distance functions for some simple geometries, it is convenient to define some short help functions (Figure 2-3) for more complex geometries.

The output from `dcircle` is the (signed) distance from `p` to the circle with center `xc`,`yc` and radius `r`. For the rectangle, we take `drectangle` as the minimum distance to the four boundary lines (each extended to infinity, and with the desired negative sign inside the rectangle). This is *not* the correct distance to the four external regions whose nearest points are corners of the rectangle. Our function avoids square roots from distances to corner points, and no meshpoints end up in these four regions when the corner points are fixed (by `pfix`).

The functions `dunion`, `ddiff`, and `dintersect` combine two geometries. They use the simplification just mentioned for rectangles, a max or min that ignores "closest corners". We use separate projections to the regions A and B, at distances $d_A(x, y)$

and $d_B(x, y)$:

$$\textbf{Union}: \qquad d_{A\cup B}(x,y) = \min(d_A(x,y), d_B(x,y)) \tag{2.7}$$

$$\textbf{Difference}: \qquad d_{A\setminus B}(x,y) = \max(d_A(x,y), -d_B(x,y)) \tag{2.8}$$

$$\textbf{Intersection}: \qquad d_{A\cap B}(x,y) = \max(d_A(x,y), d_B(x,y)) \tag{2.9}$$

Variants of these can be used to generate *blending surfaces* for smooth intersections between two surfaces [70]. Finally, `pshift` and `protate` operate on the node array `p`, to translate or rotate the coordinates.

The distance function may also be provided in a discretized form, for example by values on a Cartesian grid. This is common in *level set applications* [45], where partial differential equations efficiently model geometries with moving boundaries. Signed distance functions are created from arbitrary implicit functions using the *reinitialization method* [64]. We can easily mesh these discretized domains by creating $d(x, y)$ from interpolation. The functions `dmatrix` and `hmatrix` in Figure 2-3 use `interp2` to create $d(x, y)$ and $h(x, y)$, and `huniform` quickly implements the choice $h(x, y) = 1$.

Finally, we describe how to generate a distance function (with |gradient| = 1) when the boundary is the zero level set of a given $f(x, y)$. The results are easily generalized to any dimension. For each node $\boldsymbol{p}_0 = (x_0, y_0)$, we need the closest point $\boldsymbol{P}$ on that zero level set – which means that $f(\boldsymbol{P}) = 0$ and $\boldsymbol{P} - \boldsymbol{p}_0$ is parallel to the gradient (f_x, f_y) at $\boldsymbol{P}$:

$$\boldsymbol{L}(\boldsymbol{P}) = \begin{bmatrix} f(x,y) \\ (x-x_0)f_y - (y-y_0)f_x \end{bmatrix} = \boldsymbol{0}. \tag{2.10}$$

We solve (2.10) for the column vector $\boldsymbol{P} = (x, y)$ using the damped Newton's method with $\boldsymbol{p}_0 = (x_0, y_0)$ as initial guess. The Jacobian of $\boldsymbol{L}$ is

$$\boldsymbol{J}(\boldsymbol{P}) = \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{P}} = \begin{bmatrix} f_x & f_y + (x-x_0)f_{xy} - (y-y_0)f_{xx} \\ f_y & -f_x - (y-y_0)f_{xy} + (x-x_0)f_{yy} \end{bmatrix}^{\mathrm{T}} \tag{2.11}$$

(displayed as a transpose for typographical reasons), and we iterate

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k - \alpha \boldsymbol{J}^{-1}(\boldsymbol{p}_k)\boldsymbol{L}(\boldsymbol{p}_k) \tag{2.12}$$

until the residual $\boldsymbol{L}(\boldsymbol{p}_k)$ is small. Then $\boldsymbol{p}_k$ is taken as $\boldsymbol{P}$. The signed distance from (x_0, y_0) to $\boldsymbol{P} = (x, y)$ on the zero level set of $f(x, y)$ is

$$d(\boldsymbol{p}_0) = \text{sign}\left(f(x_0, y_0)\right)\sqrt{(x - x_0)^2 + (y - y_0)^2}. \tag{2.13}$$

The damping parameter α can be set to 1.0 as default, but might have to be reduced adaptively for convergence.

2.5 Examples

Figure 2-4 shows a number of examples, starting from a circle and extending to relatively complicated meshes.

(1) Unit Circle. We will work directly with $d = \sqrt{x^2 + y^2} - 1$, which can be specified as an inline function. For a uniform mesh, $h(x, y)$ returns a vector of 1's. The circle has bounding box $-1 \le x \le 1$, $-1 \le y \le 1$, with no fixed points. A mesh with element size approximately $h_0 = 0.2$ is generated with two lines of code:

```
>> fd=inline('sqrt(sum(p.^2,2))-1','p');
>> [p,t]=distmesh2d(fd,@huniform,0.2,[-1,-1;1,1],[]);
```

The plots ***(1a)***, ***(1b)***, and ***(1c)*** show the resulting meshes for $h_0 = 0.4$, $h_0 = 0.2$, and $h_0 = 0.1$. Inline functions are defined without creating a separate file. The first argument is the function itself, and the remaining arguments name the parameters to the function (`help inline` brings more information). Please note the comment near the end of the paper about the relatively slow performance of inline functions.

Another possibility is to discretize $d(x, y)$ on a Cartesian grid, and interpolate at other points using the `dmatrix` function:

```
>> [xx,yy]=meshgrid(-1.1:0.1:1.1,-1.1:0.1:1.1);      % Generate grid
>> dd=sqrt(xx.^2+yy.^2)-1;                           % d(x,y) at grid points
>> [p,t]=distmesh2d(@dmatrix,@huniform,0.2,[-1,-1;1,1],[],xx,yy,dd);
```

```
function d=dcircle(p,xc,yc,r)                    % Circle
d=sqrt((p(:,1)-xc).^2+(p(:,2)-yc).^2)-r;

function d=drectangle(p,x1,x2,y1,y2)             % Rectangle
d=-min(min(min(-y1+p(:,2),y2-p(:,2)), ...
           -x1+p(:,1)),x2-p(:,1));

function d=dunion(d1,d2)                         % Union
d=min(d1,d2);

function d=ddiff(d1,d2)                          % Difference
d=max(d1,-d2);

function d=dintersect(d1,d2)                     % Intersection
d=max(d1,d2);

function p=pshift(p,x0,y0)                       % Shift points
p(:,1)=p(:,1)-x0;
p(:,2)=p(:,2)-y0;

function p=protate(p,phi)                        % Rotate points around origin
A=[cos(phi),-sin(phi);sin(phi),cos(phi)];
p=p*A;

function d=dmatrix(p,xx,yy,dd,varargin)          % Interpolate d(x,y) in meshgrid matrix
d=interp2(xx,yy,dd,p(:,1),p(:,2),'*linear');

function h=hmatrix(p,xx,yy,dd,hh,varargin)       % Interpolate h(x,y) in meshgrid matrix
h=interp2(xx,yy,hh,p(:,1),p(:,2),'*linear');

function h=huniform(p,varargin)                  % Uniform h(x,y) distribution
h=ones(size(p,1),1);
```

Figure 2-3: Short help functions for generation of distance functions and size functions.

(2) Unit Circle with Hole. Removing a circle of radius 0.4 from the unit circle gives the distance function $d(x,y) = |0.7 - \sqrt{x^2+y^2}| - 0.3$:

```
>> fd=inline('-0.3+abs(0.7-sqrt(sum(p.^2,2)))');
>> [p,t]=distmesh2d(fd,@huniform,0.1,[-1,-1;1,1],[]);
```

Equivalently, $d(x,y)$ is the distance to the difference of two circles:

```
>> fd=inline('ddiff(dcircle(p,0,0,1),dcircle(p,0,0,0.4))','p');
```

(3) Square with Hole. We can replace the outer circle with a square, keeping the circular hole. Since our distance function `drectangle` is incorrect at the corners, we fix those four nodes (or write a distance function involving square roots):

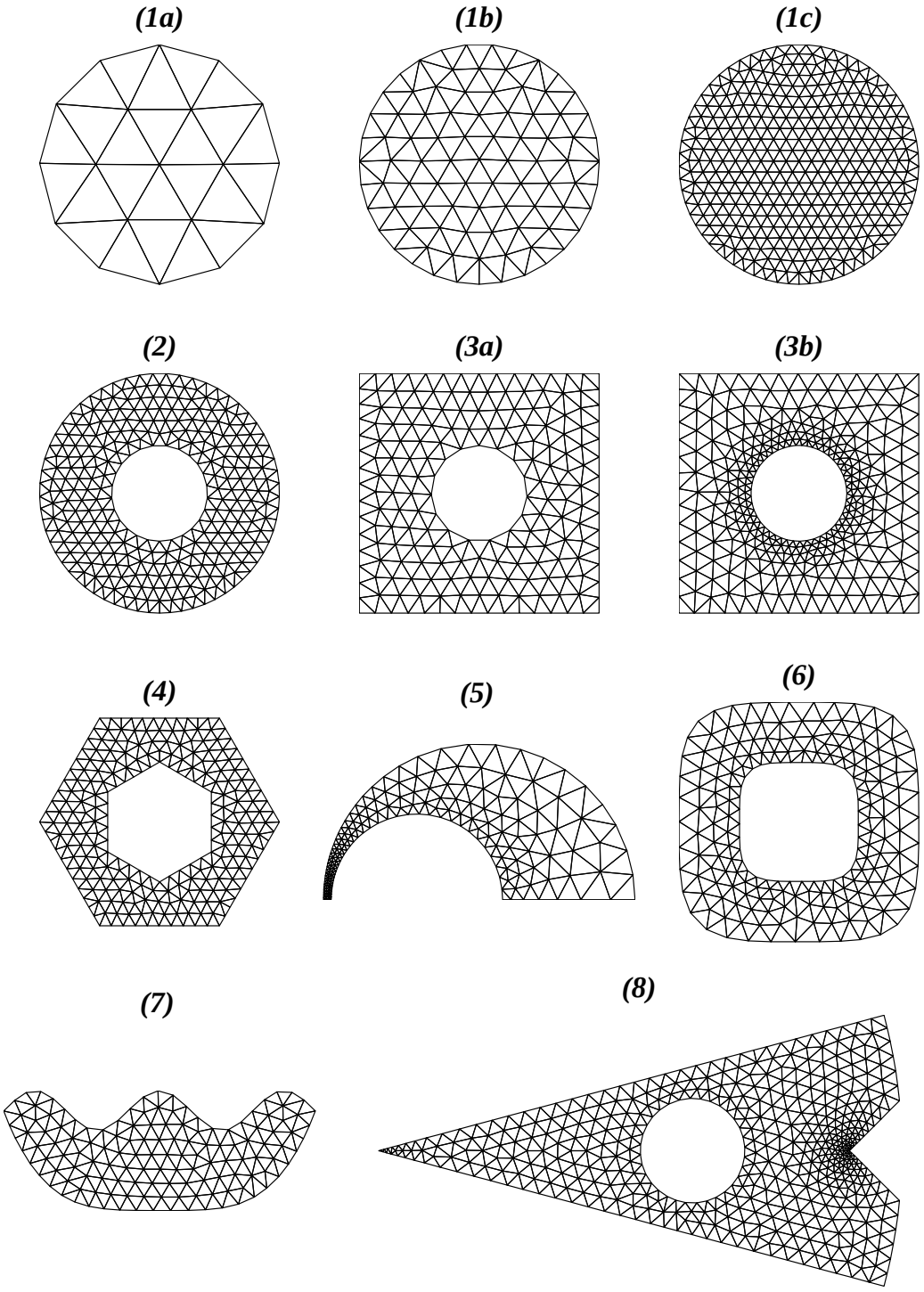

Figure 2-4: Example meshes, numbered as in the text. Examples (3b), (5), (6), and (8) have varying size functions $h(x, y)$. Examples (6) and (7) use Newton's method (2.12) to construct the distance function.

```
>> fd=inline('ddiff(drectangle(p,-1,1,-1,1),dcircle(p,0,0,0.4))','p');
>> pfix=[-1,-1;-1,1;1,-1;1,1];
>> [p,t]=distmesh2d(fd,@huniform,0.15,[-1,-1;1,1],pfix);
```

A non-uniform $h(x, y)$ gives a finer resolution close to the circle (mesh ***(3b)***):

```
>> fh=inline('min(4*sqrt(sum(p.^2,2))-1,2)','p');
>> [p,t]=distmesh2d(fd,fh,0.05,[-1,-1;1,1],pfix);
```

(4) Polygons. It is easy to create `dpoly` (not shown here) for the distance to a given polygon, using MATLAB's `inpolygon` to determine the sign. We mesh a regular hexagon and fix its six corners:

```
>> phi=(0:6)'/6*2*pi;
>> pfix=[cos(phi),sin(phi)];
>> [p,t]=distmesh2d(@dpoly,@huniform,0.1,[-1,-1;1,1],pfix,pfix);
```

Note that `pfix` is passed twice, first to specify the fixed points, and next as a parameter to `dpoly` to specify the polygon. In plot ***(4)***, we also removed a smaller rotated hexagon by using `ddiff`.

(5) Geometric Adaptivity. Here we show how the distance function can be used in the definition of $h(x, y)$, to use the local feature size for geometric adaptivity. The half-plane $y > 0$ has $d(x, y) = -y$, and our $d(x, y)$ is created by an intersection and a difference:

$$d_1 = \sqrt{x^2 + y^2} - 1 \tag{2.14}$$

$$d_2 = \sqrt{(x + 0.4)^2 + y^2} - 0.55 \tag{2.15}$$

$$d = \max(d_1, -d_2, -y). \tag{2.16}$$

Next, we create two element size functions to represent the finer resolutions near the circles. The element sizes h_1 and h_2 increase with the distances from the boundaries (the factor 0.2 gives a ratio 1.2 between neighboring elements):

$$h_1(x, y) = 0.15 - 0.2 \cdot d_1(x, y), \tag{2.17}$$

$$h_2(x, y) = 0.06 + 0.2 \cdot d_2(x, y). \tag{2.18}$$

These are made proportional to the two radii to get equal angular resolutions. Note the minus sign for d_1 since it is negative inside the region. The local feature size is the distance between boundaries, and we resolve this with at least three elements:

$$h_3(x, y) = (d_2(x, y) - d_1(x, y))/3. \tag{2.19}$$

Finally, the three size functions are combined to yield the mesh in plot ***(5)***:

$$h = \min(h_1, h_2, h_3). \tag{2.20}$$

The initial distribution had size $h_0 = 0.05/3$ and four fixed corner points.

(6), (7) Implicit Expressions. We now show how distance to level sets can be used to mesh non-standard geometries. In ***(6)***, we mesh the region between the level sets 0.5 and 1.0 of the superellipse $f(x, y) = (x^4 + y^4)^{\frac{1}{4}}$. The example in ***(7)*** is the intersection of the following two regions:

$$y \leq \cos(x) \quad \text{and} \quad y \geq 5\left(\frac{2x}{5\pi}\right)^4 - 5, \tag{2.21}$$

with $-5\pi/2 \leq x \leq 5\pi/2$ and $-5 \leq y \leq 1$. The boundaries of these geometries are not approximated by simpler curves, they are represented exactly by the given expressions. As the element size h_0 gets smaller, the mesh automatically fits to the exact boundary, without any need to refine the representation.

(8) More complex geometry. This example shows a somewhat more complicated construction, involving set operations on circles and rectangles, and element sizes increasing away from two vertices and the circular hole.

2.6 Mesh Generation in Higher Dimensions

Many scientific and engineering simulations require 3-D modeling. The boundaries become surfaces (possibly curved), and the interior becomes a volume instead of an area. A simplex mesh uses tetrahedra.

Our mesh generator extends to any dimension n. The code `distmeshnd.m` is given in www-math.mit.edu/∼persson/mesh. The truss lies in the higher-dimensional space, and each simplex has $\binom{n+1}{2}$ edges (compared to three for triangles). The initial distribution uses a regular grid. The input $\boldsymbol{p}$ to Delaunay is N-by-n. The ratio `Fscale` between the unstretched and the average actual bar lengths is an important parameter, and we employ an empirical dependence on n. The post-processing of a tetrahedral mesh is somewhat different, but the MATLAB visualization routines make this relatively easy as well. For more than three dimensions, the visualization is not used at all.

In 2-D we usually fix all the corner points, when the distance functions are not accurate close to corners. In 3-D, we would have to fix points along intersections of surfaces. A choice of edge length along those curves might be difficult for non-uniform meshes. An alternative is to generate "correct" distance functions, without the simplified assumptions in `drectangle`, `dunion`, `ddiff`, and `dintersect`. This handles all convex intersections, and the technique is used in the cylinder example below.

The extended code gives 3-D meshes with very satisfactory edge lengths. There is, however, a new problem in 3-D. The Delaunay algorithm generates *slivers*, which are tetrahedra with reasonable edge lengths but almost zero volume. These slivers could cause trouble in finite element computations, since interpolation of the derivatives becomes inaccurate when the Jacobian is close to singular.

All Delaunay mesh generators suffer from this problem in 3-D. The good news is that techniques have been developed to remove the bad elements, for example face swapping, edge flipping, and Laplacian smoothing [25]. A promising method for sliver removal is presented in [15]. Recent results [39] show that slivers are not a big problem in the Finite Volume Method, which uses the dual mesh (the Voronoi graph). It is not clear how much damage comes from isolated bad elements in finite element computations [60]. The slivery meshes shown here give nearly the same accuracy for the Poisson equation as meshes with higher minimum quality.

Allowing slivers, we generate the tetrahedral meshes in Figure 2-5.

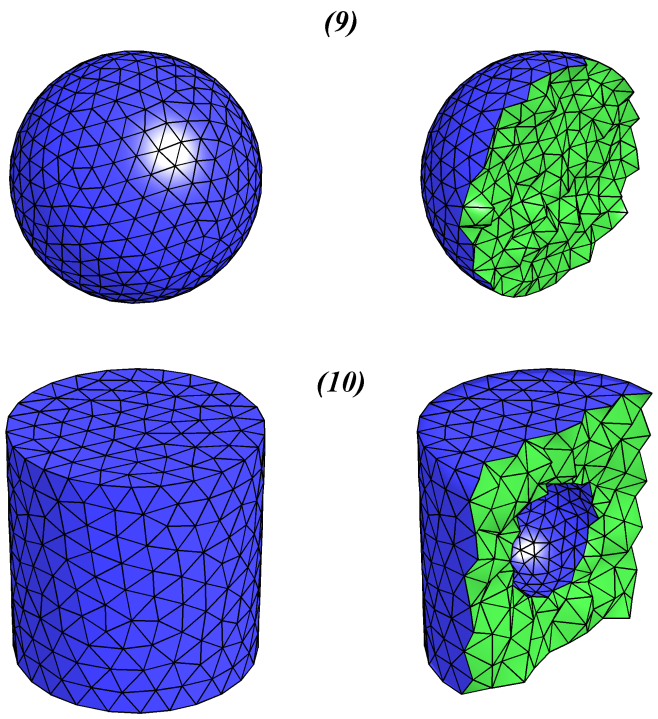

Figure 2-5: Tetrahedral meshes of a ball and a cylinder with a spherical hole. The left plots show the surface meshes, and the right plots show cross-sections.

(9) Unit Ball. The ball in 3-D uses nearly the same code as the circle:

```
>> fd=inline('sqrt(sum(p.^2,2))-1','p');
>> [p,t]=distmeshnd(fd,@huniform,0.15,[-1,-1,-1;1,1,1],[]);
```

This distance function `fd` automatically sums over three dimensions, and the bounding box has two more components. The resulting mesh has 1,295 nodes and 6,349 tetrahedra.

(10) Cylinder with Spherical Hole. For a cylinder with radius 1 and height 2, we create d_1, d_2, d_3 for the curved surface and the top and bottom:

$$d_1(x,y,z) = \sqrt{x^2+y^2} - 1 \tag{2.22}$$

$$d_2(x,y,z) = z - 1 \tag{2.23}$$

$$d_3(x,y,z) = -z - 1. \tag{2.24}$$

An approximate distance function is then formed by intersection:

$$d_{\approx} = \max(d_1, d_2, d_3). \tag{2.25}$$

This would be sufficient if the "corner points" along the curves $x^2 + y^2 = 1, z = \pm 1$ were fixed by an initial node placement. Better results can be achieved by correcting our distance function using distances to the two curves:

$$d_4(x, y, z) = \sqrt{d_1(x, y, z)^2 + d_2(x, y, z)^2} \tag{2.26}$$

$$d_5(x, y, z) = \sqrt{d_1(x, y, z)^2 + d_3(x, y, z)^2}. \tag{2.27}$$

These functions should be used where the intersections of d_1, d_2 and d_1, d_3 overlap, that is, when they both are positive:

$$d = \begin{cases} d_4, & \text{if } d_1 > 0 \text{ and } d_2 > 0 \\ d_5, & \text{if } d_1 > 0 \text{ and } d_3 > 0 \\ d_{\approx}, & \text{otherwise.} \end{cases} \tag{2.28}$$

Figure 2-5 shows a mesh for the difference between this cylinder and a ball of radius 0.5. We use a finer resolution close to this ball, $h(x, y, z) = \min(4\sqrt{x^2 + y^2 + z^2} - 1, 2)$, and $h_0 = 0.1$. The resulting mesh has 1,057 nodes and 4,539 tetrahedra.

(11) 4-D Hypersphere. To illustrate higher dimensional mesh generation, we create a simplex mesh of the unit ball in 4-D. The nodes now have four coordinates and each simplex element has five nodes. We also fix the center point $\boldsymbol{p} = (0, 0, 0, 0)$.

```
>> fd=inline('sqrt(sum(p.^2,2))-1','p');
>> [p,t]=distmeshnd(fd,@huniform,0.2,[-ones(1,4);ones(1,4)],zeros(1,4));
```

With $h_0 = 0.2$ we obtain a mesh with $3,458$ nodes and $60,107$ elements.

It is hard to visualize a mesh in four dimensions! We can compute the total mesh volume $V_4 = 4.74$, which is close to the expected value of $\pi^2/2 \approx 4.93$. By extracting all tetrahedra on the surface, we can compare the hyper-surface area $S_4 = 19.2$ to the

surface area $2\pi^2 \approx 19.7$ of a 4-D ball. The deviations are because of the simplicial approximation of the curved surface.

The correctness of the mesh can also be tested by solving Poisson's equation $-\nabla^2 u = 1$ in the four-dimensional domain. With $u = 0$ on the boundary, the solution is $u = (1 - r^2)/8$, and the largest error with linear finite elements is $\|e\|_\infty = 5.8 \cdot 10^{-4}$. This result is remarkably good, considering that many of the elements probably have very low quality (some elements were bad in 3-D before postprocessing, and the situation is likely to be much worse in 4-D).

2.7 Mesh Quality

The plots of our 2-D meshes show that the algorithm produces triangles that are almost equilateral. This is a desirable property when solving PDEs with the finite element method. Upper bounds on the errors depend only on the smallest angle in the mesh, and if all angles are close to 60°, good numerical results are achieved. The survey paper [24] discusses many measures of the "element quality". One commonly used quality measure is the ratio between the radius of the largest inscribed circle (times two) and the smallest circumscribed circle:

$$q = 2\frac{r_{\text{in}}}{r_{\text{out}}} = \frac{(b+c-a)(c+a-b)(a+b-c)}{abc} \tag{2.29}$$

where a, b, c are the side lengths. An equilateral triangle has $q = 1$, and a degenerate triangle (zero area) has $q = 0$. As a rule of thumb, if all triangles have $q > 0.5$ the results are good.

For a single measure of uniformity, we use the standard deviation of the ratio of actual sizes (circumradii of triangles) to desired sizes given by $h(x, y)$. That number is normalized by the mean value of the ratio since h only gives relative sizes.

The meshes produced by our algorithm tend to have exceptionally good element quality and uniformity. All 2-D examples except ***(8)*** with a sharp corner have every $q > 0.7$, and average quality greater than 0.96. This is significantly better than

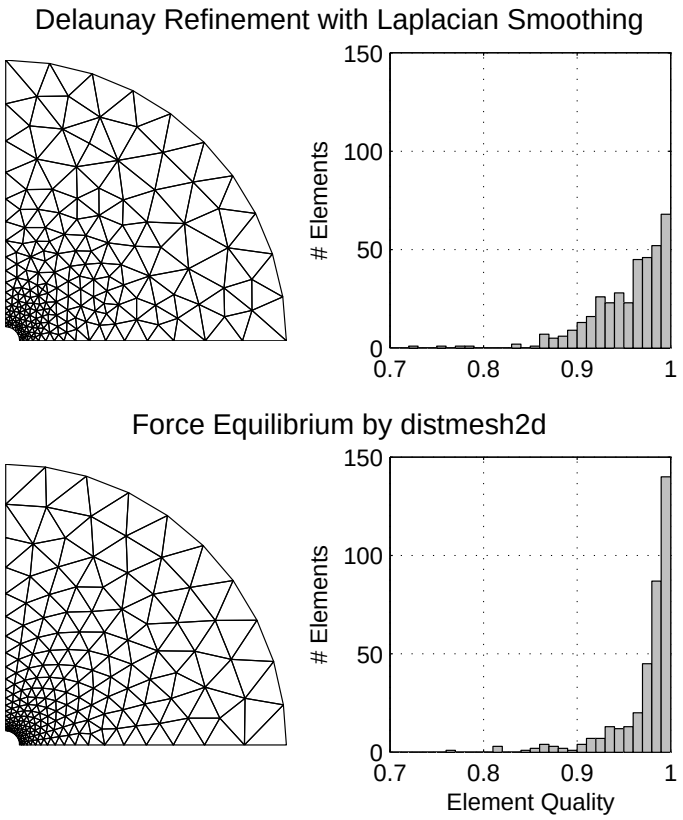

Figure 2-6: Histogram comparison with the Delaunay refinement algorithm. The element qualities are higher with our force equilibrium, and the element sizes are more uniform.

a typical Delaunay refinement mesh with Laplacian smoothing. The average size deviations are less than 4%, compared to $10 - 20\%$ for Delaunay refinement.

A comparison with the Delaunay refinement algorithm is shown in Figure 2-6. The top mesh is generated with the mesh generator in the PDE Toolbox, and the bottom with our generator. Our force equilibrium improves both the quality and the uniformity. This remains true in 3-D, where quality improvement methods such as those in [25] must be applied to both mesh generators.

Chapter 3

An Advanced Mesh Generator

The mesh generator described in the previous chapter is remarkably short and simple, and although we had to make a few compromises to achieve this the code still handles a large class of meshing applications. In this chapter we show how to improve our mesh generator, by increasing the performance and the robustness of the algorithm, and generalizing it for generation of other types of meshes. But the main underlying ideas are the same – force equilibrium in a truss structure and boundary projections using implicit geometries. In our implementation we have used the C++ programming language for many of these improvements, since the operations are hard to vectorize and a for-loop based MATLAB code would be too slow.

3.1 Discretized Geometry Representations

Our mesh generator needs two functions to mesh a domain, the signed distance function $d(\boldsymbol{x})$ and the mesh size function $h(\boldsymbol{x})$. In Chapter 2, we used closed-form expressions for these functions and showed how to write relatively complex geometries as combinations of simple functions. But as the complexity of the model grows, this representation becomes inefficient and a discretized form of $d(\boldsymbol{x})$ and $h(\boldsymbol{x})$ is preferable.

The idea behind the discretization is simple. We store the function values at a finite set of points $\boldsymbol{x}_i$ (node points) and use interpolation to approximate the function

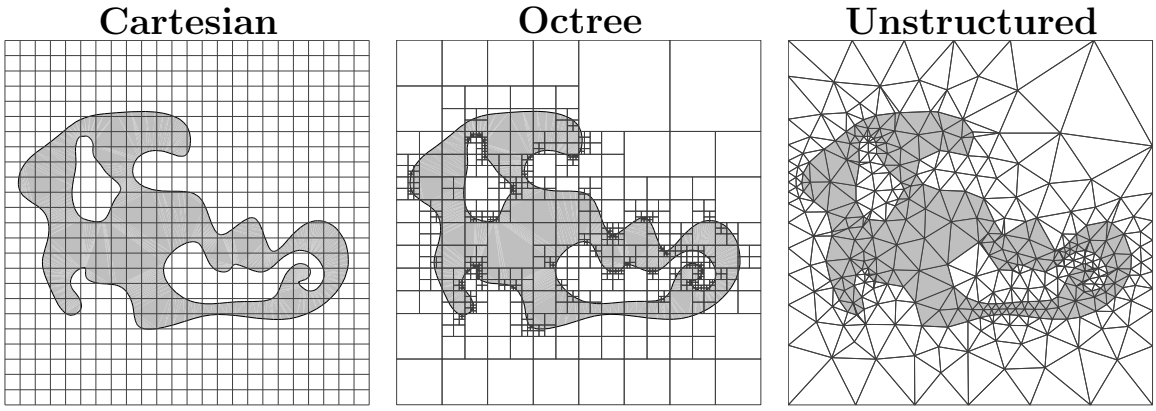

Figure 3-1: Background grids for discretization of the distance function and the mesh size function.

for arbitrary $\boldsymbol{x}$. These node points and their connectivities are part of the *background mesh* and below we discuss different options.

3.1.1 Background Meshes

The most simple background mesh is a Cartesian grid (Figure 3-1, left). 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 $\boldsymbol{x}$ we find the enclosing rectangle and the local coordinates by a few scalar operations, and use bilinear interpolation within the rectangle. We also find $\nabla u(\boldsymbol{x})$ in a similar way, and avoid the numerical differentiation we used in our MATLAB code. Higher order schemes can be used for increased accuracy.

This scheme is very simple to implement (we mentioned the help functions `dmatrix` and `hmatrix` in Chapter 2), and the Cartesian grid is particularly good for implementing level set schemes and fast marching methods (see below and Chapter 5). 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 3-1, center). The cells are still squares 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 $h(\boldsymbol{x})$ has large variations.

The adapted grid is conveniently stored in an octree data structure, and the cell enclosing an arbitrary point $\boldsymbol{x}$ is found in a time proportional to the logarithm of the number of cells. For the projections in our mesh generator, most nodes remain in the same cell as in the previous iterations, and the time to find the cell can be reduced by taking advantage of this. Within the cell we again use bilinear or higher order interpolation.

A third possibility is to discretize using an arbitrary unstructured mesh (Figure 3-1, right). This provides the freedom of using varying resolution over the domain, and the asymptotic storage requirements are similar to the octree grid. An additional advantage with unstructured meshes is that it can be aligned with the domain boundaries, making the projections highly accurate. This 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 previous time step (or iteration) is used.

Finding the triangle (or tetrahedron) enclosing an arbitrary point $\boldsymbol{x}$ can still be done in logarithmic time, but the algorithm is slower and more complicated. We can again take advantage of the fact that the nodes move slowly and are likely to remain in the same cell as in the previous iteration. The interpolation can be piecewise linear or higher order within each element.

If we assume that all boundary nodes that are projected are located within a small distance of the actual boundary, we do not need a mesh of the entire domain but only a narrow band of elements around the boundary. In our MATLAB code we also had to determine the sign of $d(\boldsymbol{x})$ in the entire domain, but this step can be avoided since the boundary points can be found from the connectivities. However,

the total memory requirement is still proportional to the length/area of the boundary curve/surface, as for the full octree or unstructured grid.

3.1.2 Initialization of the Distance Function

Before interpolating the distance function and the mesh size function on the background mesh, their values at the nodes of this grid must be calculated. For closed-form expressions this is easy, and since we only evaluate them once for each node point, the performance is good even for complex expressions.

For higher efficiency we can compute $d(\boldsymbol{x})$ for the nodes in a narrow band around the domain boundary, and use the *Fast Marching Method* (Sethian [55], see also Tsitsiklis [66]) to calculate the distances at all the remaining node points. The computed values are considered "known values", and the nodes neighboring these 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 $n \log n$ 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 the narrow band of background grid nodes around the triangle 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 [38], 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 [64] 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 update the nodes close the boundary, and use the fast marching method for the rest of the domain. If the implicit function is sufficiently smooth, we can also use the approximate projections described below to avoid reinitialization.

3.2 Approximate Projections

For a signed distance function d, the projection $\boldsymbol{x} \leftarrow \boldsymbol{x} - d(\boldsymbol{x})\nabla d(\boldsymbol{x})$ is exact. For more general implicit functions $\phi(\boldsymbol{x})$ this is no longer true, and we have discussed various ways to modify ϕ into a distance function. The most general approach of Section 2.4 computes distances to an arbitrary implicit boundary by solving a system of nonlinear equations. In the previous section we mentioned the reinitialization equation and the fast marching method, which can be used for discretized distance functions. Here, we show how to modify the projections in the mesh generator to handle general implicit geometry descriptions.

3.2.1 Problem Statement

The (exact) projection can be defined in the following way: Given a point $\boldsymbol{p}$ and a function $\phi(\boldsymbol{p})$, we want to find a correction $\Delta\boldsymbol{p}$ such that

$$\phi(\boldsymbol{p} + \Delta\boldsymbol{p}) = 0, \tag{3.1}$$

$$\Delta\boldsymbol{p} \parallel \nabla\phi(\boldsymbol{p} + \Delta\boldsymbol{p}). \tag{3.2}$$

Note that the correction should be parallel to the gradient at the boundary point $\boldsymbol{p} + \Delta\boldsymbol{p}$, not at the initial point location $\boldsymbol{p}$. We can write (3.2) in terms of an additional parameter t, to obtain the system

$$\phi(\boldsymbol{p} + \Delta\boldsymbol{p}) = 0 \tag{3.3}$$

$$\Delta\boldsymbol{p} + t\nabla\phi(\boldsymbol{p} + \Delta\boldsymbol{p}) = 0. \tag{3.4}$$

These equations can also be derived by considering the constrained optimization problem

$$\begin{cases} \min_{\Delta\boldsymbol{p}} \ |\Delta\boldsymbol{p}|^2 \\ \phi(\boldsymbol{p} + \Delta\boldsymbol{p}) = 0 \end{cases} \tag{3.5}$$

and rewrite it using the Lagrange multiplier t. We will use this viewpoint when computing distances to Bézier surfaces in Section 3.6.2.

For a general exact projection, we solve (3.3) using Newton iterations. This approach was described in Section 2.4, where eliminating t gives a system in $\Delta \boldsymbol{p}$ only. These iterations might be expensive, and we now discuss how to approximate the projections by assuming a smooth implicit function ϕ.

3.2.2 First Order Approximation

A first order approximation can be derived by replacing ϕ with its truncated Taylor expansion at $\boldsymbol{p}$:

$$\phi(\boldsymbol{p} + \Delta \boldsymbol{p}) \approx \phi + \nabla \phi \cdot \Delta \boldsymbol{p} \tag{3.6}$$

(ϕ and $\nabla \phi$ in the right hand side are implicitly assumed to be evaluated at $\boldsymbol{p}$). (3.4) then becomes

$$\Delta \boldsymbol{p} + t \nabla \phi = 0. \tag{3.7}$$

Insert into (3.6) and set to zero:

$$\phi - t \nabla \phi \cdot \nabla \phi = 0 \Rightarrow t = \frac{\phi}{|\nabla \phi|^2}, \tag{3.8}$$

and

$$\Delta \boldsymbol{p} = \frac{\phi}{|\nabla \phi|^2} \nabla \phi. \tag{3.9}$$

This is a very simple modification to get first order accuracy. Compared to the true distance function we simply divide by the squared length of the gradient. Below we show how to incorporate this into the MATLAB code of the previous chapter by only one additional line of code.

3.2.3 Second Order Approximation

We can derive a higher order approximate projection by including more terms in the truncated Taylor expansion of ϕ. For simplicity we show this derivation in two dimensions. For a point (x, y) and a small displacement $(\Delta x, \Delta y)$ we set

$$\phi(x + \Delta x, y + \Delta y) \approx \phi + \Delta x\phi_x + \Delta y\phi_y + \frac{\Delta x^2}{2}\phi_{xx} + \Delta x\Delta y\phi_{xy} + \frac{\Delta y^2}{2}\phi_{yy} \quad (3.10)$$

As before, ϕ and its derivatives are evaluated at the original point (x, y). (3.4) becomes

$$\Delta x + t(\phi_x + \Delta x\phi_{xx} + \Delta y\phi_{xy}) = 0 \quad (3.11)$$

$$\Delta y + t(\phi_y + \Delta x\phi_{xy} + \Delta y\phi_{yy}) = 0. \quad (3.12)$$

Solve for $\Delta x, \Delta y$:

$$\Delta x = \frac{(\phi_y\phi_{xy} - \phi_x\phi_{yy})t^2 - \phi_x t}{(\phi_{xx}\phi_{yy} - \phi_{xy}^2)t^2 + (\phi_{xx} + \phi_{yy})t + 1} \quad (3.13)$$

$$\Delta y = \frac{(\phi_x\phi_{xy} - \phi_y\phi_{xx})t^2 - \phi_y t}{(\phi_{xx}\phi_{yy} - \phi_{xy}^2)t^2 + (\phi_{xx} + \phi_{yy})t + 1}. \quad (3.14)$$

Insert into (3.10), set to zero, multiply by denominator, and simplify to obtain a fourth degree polynomial in t:

$$p_4t^4 + p_3t^3 + p_2t^2 + p_1t + p_0 = 0 \quad (3.15)$$

with

$$\begin{aligned}
p_0 &= \phi \\
p_1 &= 2\phi(\phi_{xx} + \phi_{yy}) - \phi_x^2 - \phi_y^2 \\
p_2 &= \phi\phi_{xx}^2 + \phi\phi_{yy}^2 + 4\phi\phi_{yy}\phi_{xx} - 2\phi_y^2\phi_{xx} - 2\phi_x^2\phi_{yy} - 2\phi\phi_{xy}^2 - \\
&\quad \frac{1}{2}\phi_y^2\phi_{yy} - \frac{1}{2}\phi_x^2\phi_{xx} + 3\phi_x\phi_{xy}\phi_y \\
p_3 &= -\phi_x^2\phi_{yy}^2 - 2\phi\phi_{xy}^2\phi_{xx} - \phi_y^2\phi_{yy}\phi_{xx} + 2\phi_x\phi_{xx}\phi_{xy}\phi_y - 2\phi\phi_{xy}^2\phi_{yy} - \phi_y^2\phi_{xx}^2 +
\end{aligned}$$

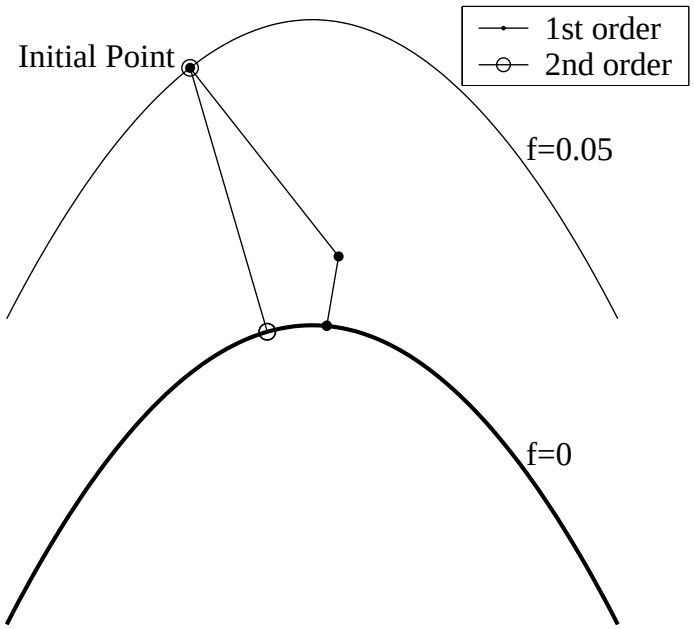

Figure 3-2: Comparison of first and second order projection schemes.

$$
\begin{aligned}
& 2\phi\phi_{yy}^2\phi_{xx} - \phi_x^2\phi_{xx}\phi_{yy} + 2\phi_x\phi_{xy}\phi_y\phi_{yy} + 2\phi\phi_{xx}^2\phi_{yy} \\
p_4 = & -\phi_x\phi_{xy}^3\phi_y + \phi\phi_{yy}^2\phi_{xx}^2 + \phi\phi_{xy}^4 - 2\phi\phi_{xy}^2\phi_{yy}\phi_{xx} + \frac{1}{2}\phi_y^2\phi_{xx}\phi_{xy}^2 - \frac{1}{2}\phi_y^2\phi_{xx}^2\phi_{yy} - \\
& \frac{1}{2}\phi_x^2\phi_{yy}^2\phi_{xx} + \frac{1}{2}\phi_x^2\phi_{yy}\phi_{xy}^2 + \phi_x\phi_{xy}\phi_y\phi_{yy}\phi_{xx}
\end{aligned} \tag{3.16}
$$

Solve (3.16) for the real root t with smallest magnitude and insert in (3.13),(3.14) to obtain $\Delta x, \Delta y$.

3.2.4 Examples

A comparison of the first and the second order projections is shown in Figure 3-2. The point $(x, y) = (0.23, \sin(2\pi \cdot 0.23) + 0.05)$ is projected onto the zero level set of $\phi(x, y) = y - \sin(2\pi x)$. The projections are repeated until the projected points are close to $\phi = 0$.

We can see how the first order method initially moves in the gradient direction at x, y instead of at the boundary, and ends up far away from the closest boundary point. The second order method moves in a direction very close to the exact one. However, our experience is that the first order projections are sufficiently accurate

for our mesh generator, especially when highly curved boundaries are well resolved and ϕ is relatively smooth. Note that we do not really require the projections to be exact, we simply want to move the point to any nearby boundary point.

The first order method is trivial to incorporate into our MATLAB code. The projection using the distance function:

```
p(ix,:)=p(ix,:)-[d(ix).*dgradx,d(ix).*dgrady];
```

is replaced by (3.9):

```
dgrad2=dgradx.^2+dgrady.^2;
p(ix,:)=p(ix,:)-[d(ix).*dgradx./dgrad2,d(ix).*dgrady./dgrad2];
```

This is a significant improvement of the code, since we can "cheat" when we generate the distance function. For example, an ellipse with semimajor/semiminor axes a, b is given implicitly by:

$$\phi(x, y) = \frac{x^2}{a^2} + \frac{y^2}{b^2} - 1 = 0. \tag{3.17}$$

But computing the real distances $d(x, y)$ to the boundary $\phi(x, y) = 0$ is rather hard. With our first order modification we can use ϕ to mesh the domain (with $a = 2$ and $b = 1$):

```
fd=inline('p.^2*[a;b].^(-2)-1','p','a','b');
[p,t]=distmesh2d(fd,@huniform,0.2,[-2,-1;2,1],[],2,1);
```

The result is a high quality mesh of the ellipse (Figure 3-3), where the largest deviation from the true boundary is only $1.8 \cdot 10^{-4}$. If higher accuracy is desired, the code can easily be modified to apply the projections several times.

3.3 Mesh Manipulation

In our MATLAB code the connectivities of the mesh are always computed using the Delaunay triangulation. Every time we update the connectivities a complete triangulation is computed, even if only a few edges were modified. Furthermore, we do not have any control over the generated elements, if we for example want to force edges to be aligned with the boundaries (the constrained Delaunay triangulation [16]).

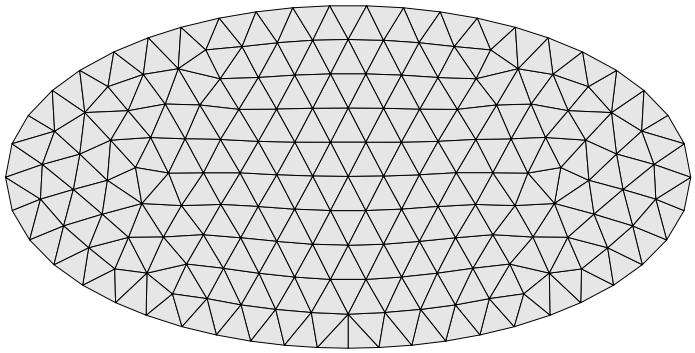

Figure 3-3: An ellipse represented implicitly and meshed using first order approximate projections.

To achieve higher performance and robustness, the triangulation can be controlled explicitly by local manipulation of the connectivities. In this section we describe how this improves the element updates during the iterations, and how we can control the node density and the generation of the initial mesh. In the next section we use these results to create other types of meshes, including anisotropic meshes and surface meshes.

3.3.1 Local Connectivity Updates

During the iterations, we update the connectivities to maintain a good triangulation of the nodes. But most of the triangles are usually of high quality, and the retriangulations then modify only a few of the mesh elements (in particular if a good initial mesh is used, or when the algorithm is close to convergence). We can save computations by starting from the previous connectivities and only update the bad triangles, and one way to do this is by local connectivity updates.

In the flipping algorithm for computing the Delaunay triangulation [4] we loop over all the edges of the mesh and consider "flipping" the edge between neighboring triangles (Figure 3-4, top). This decision can be made based on the standard Delaunay in-circle condition, or some other quality norm. These iterations will terminate and they produce the Delaunay triangulation of the nodes [4]. To obtain high performance, we need a data structure that provides pointers to the neighbors of each triangle, and

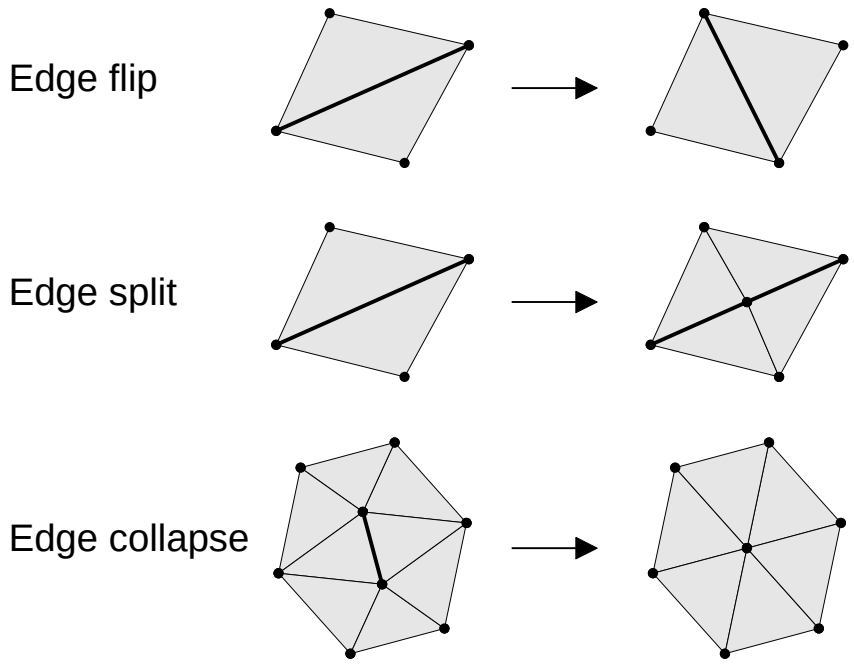

Figure 3-4: Local mesh operations. The edge flip alters the connectivity but not the nodes, the split inserts one new node, and the collapse removes one node.

these have to be modified whenever we change the mesh.

The local updates improve the performance significantly, since we do not have to recompute the entire Delaunay triangulation. We find the triangles that can be improved by an edge flip, and leave the rest of the mesh intact. In our current implementation we search through all the elements, but the element qualities could be stored in a priority queue giving very fast access to the bad elements.

Another advantage with the local updates is that we can keep the topology of the initial mesh. For example, by excluding edges along the boundary from the edge flips, we will never produce elements that cross the boundaries (unless they do so in the initial mesh). The projections are also much faster, since we always know which nodes are part of the boundary, and we do not have to compute the distance function at the interior nodes.

In three dimensions, the connectivities can be improved by similar local updates. In [25], so called "face swapping" and "edge flipping" was introduced, although it is not known if these always produce a good mesh. We have not yet implemented this,

we use the full Delaunay triangulation for all our tetrahedral meshes. However, we believe that the local updates in three dimensions have a good potential for generating high-quality meshes, even avoiding the bad sliver elements. In Chapter 2, we post-processed our tetrahedral meshes with these local updates and Laplacian smoothing to remove most of the slivers, but integrating the updates with our mesh generator might directly produce high-quality meshes.

3.3.2 Density Control

In our original algorithm, we relied on the initial mesh to give a good density of nodes according to the size function $h(\boldsymbol{x})$, and during the iterations we never added or removed any nodes. However, in some cases it is desirable to control the node density. During the generation of the initial mesh we can for example start with a trivial uniform mesh given by a Cartesian background grid. Another case is when we apply the mesh generator to moving meshes and adaptive solvers in Chapter 5, where we want to keep a previous mesh as initial mesh, but the geometry $d(\boldsymbol{x})$ and the size function $h(\boldsymbol{x})$ have changed.

When we retriangulate the nodes using Delaunay (as in our MATLAB code), density control simply means that we add new nodes and delete existing nodes as we wish. The desired size function $h(\boldsymbol{x})$ at the edges is compared with the actual edge lengths, and if they differ more than a tolerance, nodes can be inserted or removed. The retriangulation will use the new set of nodes and the mesh generator will rearrange them to improve the qualities.

When the mesh is manipulated locally, we have to be a little more careful when inserting and removing nodes. The mesh has to remain a valid representation of the domain after the modifications, and the data structures representing the element neighbors need to be updated. One simple way to do the node insertion is to split an edge (Figure 3-4, center). This divides the edge in half and connects the new node to the two opposite corners. The resulting four triangles are likely to have lower quality, but the mesh generator will improve the node locations and modify the connectivity.

Deleting a node can be done in a similar way, for example by merging two neigh-

boring nodes into one (Figure 3-4, bottom). This operation is more complicated than the edge split, since all elements referring to the deleted node have to be changed. However, the pointers to neighboring elements are sufficient for doing this in a limited number of operations, independent of the total mesh size. Another issue with node deletion is that we have to make sure the mesh is still valid. For example, we can not merge two boundary nodes connected by an internal edge.

3.3.3 The Initial Mesh

The first step of our iterative mesh generator is to generate the initial locations of the node points and their connectivities. In the MATLAB code this is done by first creating a regular grid with the given spacing h_0. Points outside the domain are removed, and for non-uniform size functions points are kept with a probability proportional to the desired density. The new connectivities are computed in the first iteration by a Delaunay triangulation. This technique is easy to implement and usually generates good results, but it can be improved in several ways.

One drawback with this approach is that a large number of points might be discarded, either because they are outside the domain or because the size function is highly non-uniform. For example, if the desired sizes differ by several orders of magnitude, the initial uniform mesh might not even fit in the memory. This is also true for geometries that fill a small portion of their bounding box. One way to solve the problem is to subdivide the region into smaller boxes (for example using an octree data structure) and apply the technique individually in each box.

The second issue is the connectivity computation by the Delaunay triangulation. This step is expensive and might not generate conforming elements (edges can cross the boundary). In two dimensions we can use a constrained Delaunay triangulation [16], but in higher dimensions such a triangulation might not exist.

In our C++ code we generate the initial mesh by a new technique, which is more robust and efficient. We begin by enclosing the entire geometry with one large element (a regular triangle). This mesh is then repeatedly refined until the edges are smaller than the sizes given by $h(\boldsymbol{x})$. These refinements can be made using local

refinement techniques [51], but we use a simpler method where we simply split an edge by dividing two neighboring triangles and flip edges to improve the quality.

During the refinements, elements that are completely outside the domain can be removed since they will never be part of the final mesh. We can detect this using the distance function, a sufficient condition is $d > \ell_{\max}$, where $\ell_{\max}$ is the longest edge of the element. After the refinements, we remove outside elements if $d > 0$ at the element centroid (as in the MATLAB code).

With a good data structure and routines that operate locally, this procedure requires a time proportional to the number of nodes, and it returns a complete mesh including both the nodes and their connectivity.

We also mention that for a discretized implicit geometry definition, the initial mesh can be generated directly from the discretization. For example, a 2-D Cartesian background grid can easily be split into triangles. At the boundaries, we can generate elements of poor quality that fit to the boundary (by splitting the boundary cells), or we can let the mesh generator move the nodes to the boundaries as before. In either case, the quality of the mesh is not important since it will be improved by the iterations, and the node density can be controlled by the density control described above.

3.4 Internal Boundaries

In finite element calculations it is often desirable to have elements that are aligned with given internal boundaries. This makes it possible to, for example, solve partial differential equations with discontinuities in the material coefficients. The internal boundaries divide the geometry into several subdomains, which are connected only through the common node points on the boundaries.

One way to obtain elements that are aligned with internal boundaries is to mesh the boundaries separately before meshing the entire domain, and fix the location of these generated node points and boundary elements. This is essentially the bottom-up approach used by other mesh generators such as Delaunay refinement, and it relies

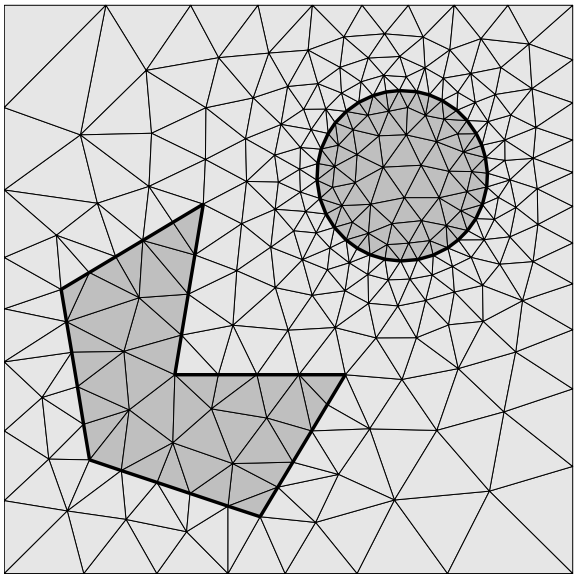

Figure 3-5: An example of meshing with internal boundaries.

on an explicit representation of the internal boundaries.

A solution more in the spirit of our mesh generator is to represent the internal boundaries implicitly, by another distance function $d_I(\boldsymbol{x})$ (as before, with approximate projections this could be any smooth implicit function $\phi_I(\boldsymbol{x})$). We then project internal boundary points using this function in the same way as before. The difficulty is to determine which points to project, since we now have points on both sides of the boundary. A simple solution for our MATLAB code is to find edges that cross the internal boundary and project the closer endpoint of the edge. This is not entirely robust though, and a better solution is to start with an initial mesh that aligns with the boundaries, and keep track of these nodes during the iterations. If a new node tries to cross the boundary it is added to the list of boundary nodes and is projected. Density control as described above might be required, in particular to remove boundary nodes.

An example is shown in Figure 3-5. The square geometry has two internal boundaries, consisting of a circle and a polygon. Note that the same $d_I(\boldsymbol{x})$ can represent all internal boundaries, as long as they do not cross.

3.5 Anisotropic Meshes

Up to now we have considered the generation of isotropic meshes, where the lengths of all the edges in an element are approximately equal. Sometimes it is desirable to use anisotropic elements, where the edge length depends on the orientation of the edge. One example is in computational fluid dynamics, where solution fields with boundary layers or shocks have large variations in one direction but not in the other. By using anisotropic elements we can resolve the solution accurately with few elements. Another application of anisotropic meshes is when the mesh is transformed from a parameter space to real space, for example with parameterized surfaces (see Section 3.6.2). The parameterization might distort the elements but this can be compensated for by generating an appropriate anisotropic mesh in the parameter space.

We can extend our mesh generator to generate anisotropic meshes by introducing a local metric tensor $\mathcal{M}$ instead of the scalar mesh size function h. In this metric, all desired edge lengths are one, and assuming that $\mathcal{M}$ is constant over an edge $\boldsymbol{u}$, we can compute the actual edge length from

$$\ell(\boldsymbol{u}) = \sqrt{\boldsymbol{u}^T \mathcal{M} \boldsymbol{u}}. \tag{3.18}$$

In two dimensions, $\mathcal{M}$ has the form

$$\mathcal{M}(x, y) = \begin{pmatrix} a(x, y) & b(x, y) \\ b(x, y) & c(x, y) \end{pmatrix}. \tag{3.19}$$

For the special case of an isotropic size function $h(x, y)$, the corresponding metric is $\mathcal{M} = I/h^2$. An anisotropic metric with sizes h_x, h_y in the x, y-directions is represented by $\mathcal{M} = \text{diag}(1/h_x^2, 1/h_y^2)$. In the general case, $\mathcal{M}$ can be written in terms of its

eigendecomposition as

$$\mathcal{M} = \mathcal{R} \begin{pmatrix} 1/h_1^2 & 0 \\ 0 & 1/h_2^2 \end{pmatrix} \mathcal{R}^{-1} \tag{3.20}$$

where $\mathcal{R}$ is a rotation matrix and h_1, h_2 are the desired sizes along the directions of the column vectors of $\mathcal{R}$.

To incorporate anisotropy into our mesh generator, we replace the size function h by $\mathcal{M}$, for example by the three functions $a(x, y), b(x, y), c(x, y)$. In the calculation of the edge lengths in the force function, we set all desired sizes to one and compute actual lengths by (3.18) with $\mathcal{M}$ averaged at the two end points. The metric also changes the connectivity updates, where we can use a modified form of the in-circle test, or a length based quality test with $\mathcal{M}$ averaged over the element nodes. We also modify the edge lengths in the density control, if applied.

An example is shown in Figure 3-6. We generate a mesh of the unit circle, where the mesh is very fine in the radial direction near the boundary. This can be expressed using (3.20) with the sizes

$$h_1 = \min(h_{\min} + g(1 - \sqrt{x^2 + y^2}), h_{\max}) \tag{3.21}$$

$$h_2 = h_{\max}, \tag{3.22}$$

where $h_{\min} = 0.01$, $h_{\max} = 0.2$, and $g = 0.3$, and the rotation picks out the normal and tangential directions:

$$\mathcal{R} = \frac{1}{\sqrt{x^2 + y^2}} \begin{pmatrix} x & -y \\ y & x \end{pmatrix}. \tag{3.23}$$

The generated mesh can accurately represent a boundary layer of thickness ~ 0.01, but with a total of only 500 node points.

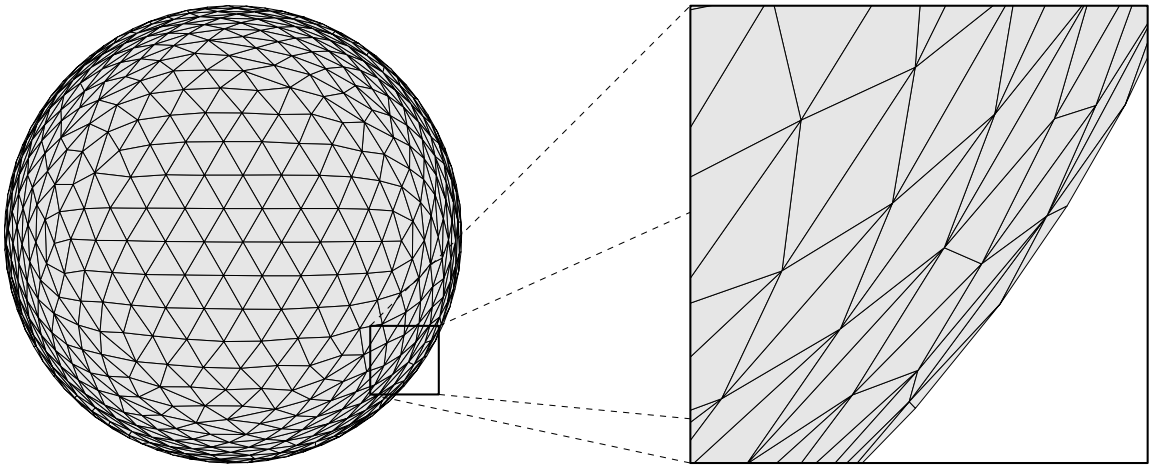

Figure 3-6: An anisotropic mesh of the unit circle, with small edges in the radial direction close to the boundary.

3.6 Surface Meshes

Generating a surface mesh means we are only interested in the boundary surface of a domain in three dimensions, or the discretization of the boundary curves in two dimensions. This is of interest when only the surface mesh is required, for example with boundary element methods and in computer graphics applications, but also as a preprocessing step for generating a volume mesh (tetrahedral) with for example the Delaunay refinement method.

3.6.1 Implicit Surfaces

For surfaces given in an explicit parameterized form, we can create a triangular mesh in the parameter plane and map the nodes to the surface. This is a common representation of CAD geometries, and we discuss it further in the next section. Here, we consider the implicit specification of the boundaries as the zero level set of a function $\phi(\boldsymbol{x})$ in $\mathbb{R}^3$, just like before. This eliminates the need to form the parameterization and to divide the domain into patches, by working directly with the implicit description.

We propose the following simple modification to generate surface meshes: Project all the nodes after every iteration. This corresponds to assigning reaction forces normal to the boundary of exactly the right magnitude to keep the points at the

boundary. The actual mesh generation then proceeds almost as in the two dimensional case, but with the nodes moving in three dimensions.

One problem with this approach to surface meshing is the generation of the connectivities. The Delaunay triangulator would generate tetrahedra in the entire convex hull. Even if we discarded all elements except for the boundary triangulation, it would give poor element qualities. Instead, we use our explicit mesh modifications from Section 3.3. We flip edges to improve triangle qualities, where the qualities are defined for triangles embedded in $\mathbb{R}^3$. Some care has to be taken to avoid inverting the elements.

The initial mesh can be created with the same techniques as in Section 3.3.3, but keeping only the triangles on the surface. For a discretized geometry, such as a Cartesian or Octree description, it is easy to triangulate the surface in each cell of the background grid, and let the density control coarsen and/or refine the mesh. In computer graphics a popular algorithm for this is the marching cube method [37].

An example mesh is shown in Figure 3-7. The difference between a sphere and a cylinder is formed by smoothed set operations [70]:

$$\phi(\boldsymbol{x}) = g(-d_1, R) + g(d_2, R) - 1 \tag{3.24}$$

$$d_1 = \sqrt{x^2 + y^2 + z^2} - R_1 \tag{3.25}$$

$$d_2 = \sqrt{x^2 + z^2} - R_2 \tag{3.26}$$

$$g(s, R) = \begin{cases} 4(s - R)^2(9R/4 - s)/(9R^3) & \text{if } s \leq R, \\ 0 & \text{otherwise,} \end{cases} \tag{3.27}$$

with $R = 0.5$, $R_1 = 1.2$, and $R_2 = 0.5$. The size function is based on curvature and gradient limiting (see Chapter 4), and the mesh is generated by the method described above.

Note that this implicit representation can only be used for closed surfaces. It might be possible to mesh an open subset of the surface using a second implicit function that defines the new boundaries (similar to the handling of the internal boundaries described before). We have not worked out the details on how to do these projections.

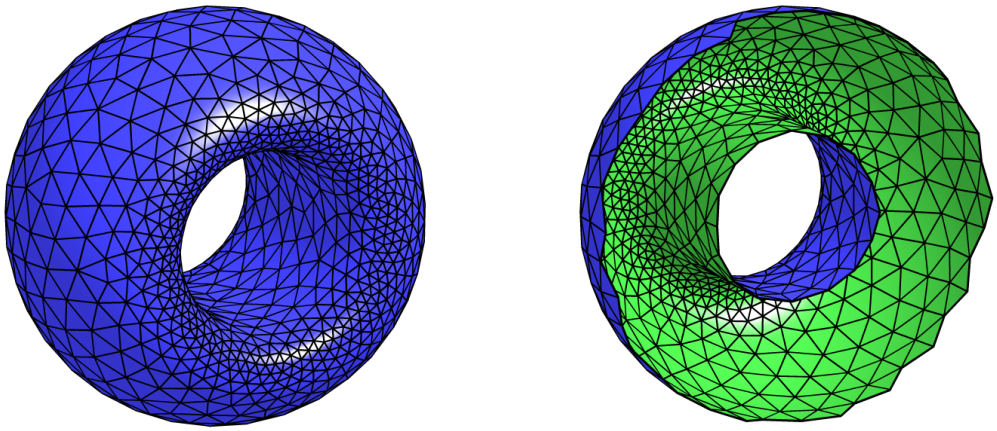

Figure 3-7: A triangular surface mesh, full mesh (left) and split view (right).

3.6.2 Explicit Surfaces

We now show how to mesh a parameterized surface patch, for example a rational Bézier surface. We are given an explicit mapping $\boldsymbol{r}(u, v)$ from the parameter space (u, v) to $\mathbb{R}^3$:

$$\boldsymbol{r}(u, v) = (X(u, v), Y(u, v), Z(u, v)). \tag{3.28}$$

To mesh this parameterized surface means we want to triangulate a region Ω in the (u, v) space such that when the nodes $\boldsymbol{p}_i = (u_i, v_i)$ are mapped to $\boldsymbol{r}(\boldsymbol{p}_i)$, the corresponding triangles are of high quality, according to some quality norm for triangles in $\mathbb{R}^3$. In general, Ω is a subset of the definition space $(u, v) \in [0, 1] \times [0, 1]$ (a "trimmed surface") . In our setting, it is natural to describe Ω implicitly by $\phi(u, v) \leq 0$.

We could in principle find an implicit function representing the surface $\boldsymbol{r}(u, v)$, for example by implicitization [54], and use the same projection techniques as we described in the previous section. In general this results in high-degree polynomials, and it is not clear how to handle the boundaries $\phi(u, v) = 0$. Instead, we keep the explicit formulation and do the projections by solving for the smallest distance from a point to the surface.

A straight forward method is to mesh the domain Ω in parameter space without considering the mapping. The resulting mesh in real space is typically of low quality, since the mapping deforms the elements. We could compensate for this by generating an appropriate anisotropic mesh in the parameter space, using the techniques described in Section 3.5. However, we have found it easier to take advantage of our force- and projection-based mesh generation, and create a high-quality mesh directly in $\mathbb{R}^3$. This approach is particulatly advantageous for highly distorted mappings or degenerate surface patches, which are common in practical CAD applications.

The force calculations and the connectivity updates are done exactly as for the implicit surface meshes, as described in Section 3.6.1. After the update, each node $\boldsymbol{p}_i = (x_i, y_i, z_i)$ is projected back to the surface by solving for (u_i, v_i) that minimizes

$$\min_{u_i, v_i} \ |\boldsymbol{r}(u_i, v_i) - \boldsymbol{p}_i|^2 \tag{3.29}$$

and setting $\boldsymbol{p}_i \leftarrow \boldsymbol{r}(u_i, v_i)$. We solve (3.29) with a damped Newton's method, which converges very fast because of the good initial condition from the previous node locations. The first and second derivatives of $\boldsymbol{r}(u, v)$ can be computed by explicit differentiation or numerically.

For the boundary nodes, which we detect from their connectivity or by $\phi(u_i, v_i) > 0$, we could project using our usual projections in the (u, v)-plane. However, for non-orthogonal mappings this gives highly inaccurate results, since in $\mathbb{R}^3$ the points are moved tangentially to the boundary. Instead we project using a constrained optimization:

$$\begin{cases} \min_{u_i, v_i} \ |\boldsymbol{r}(u_i, v_i) - \boldsymbol{p}_i|^2 \\ \phi(u, v) = 0 \end{cases} \tag{3.30}$$

We rewrite (3.30) as a system of non-linear equations in terms of a Lagrange multiplier t, and solve using Newton's method as before. We can simplify this by a first order approximation $\phi(u_i, v_i) = \phi_0 + \phi_x(u_i - u_i^0) + \phi_y(v_i - v_i^0)$, where (u_i^0, v_i^0) are the initial parameter values for node i, and ϕ_x, ϕ_y are the components of the gradient evaluated

at (u_i^0, v_i^0). The constraint $\phi(u_i, v_i) = 0$ then gives us a relation between u_i and v_i, $\phi_x(u_i - u_i^0) + \phi_y(v_i - v_i^0) = 0$, and we can apply the projection in two separate steps. First we project (u_i, v_i) back orthogonally to the boundary in the (u, v) space using the usual first order approximation:

$$(\tilde{u}_i, \tilde{v}_i) \leftarrow (u_i, v_i) - \frac{\phi(u_i, v_i)}{|\nabla\phi(u_i, v_i)|^2}\nabla\phi(u_i, v_i). \tag{3.31}$$

Next we solve a scalar non-linear optimization problem where we search only in the direction orthogonal to $\nabla\phi = (\phi_x, \phi_y)$:

$$\min_t \ |\boldsymbol{r}(\tilde{u}_i - t\phi_y, \tilde{v}_i + t\phi_x) - \boldsymbol{p}_i|^2, \tag{3.32}$$

and finally we set $\boldsymbol{p}_i \leftarrow \boldsymbol{r}(\tilde{u}_i - t\phi_y, \tilde{v}_i + \phi_x t)$. If this approximation is inaccurate (that is, $|\phi(\tilde{u}_i - t\phi_y, \tilde{v}_u + \phi_x t)|$ is too large), we can repeat the projections, use a second-order approximation of $\phi(u_i, v_i)$, or solve the full non-linear system of equations (3.30) with repeated evaluations of $\phi(u_i, v_i)$.

As an example, we generate a mesh for a bi-quadratic rational Bézier surface. These have the form

$$\boldsymbol{r}(u, v) = \frac{\sum_{i=0}^{2}\sum_{j=0}^{2} w_{ij}\boldsymbol{b}_{ij}B_{i,2}(u)B_{j,2}(v)}{\sum_{i=0}^{2}\sum_{j=0}^{2} w_{ij}B_{i,2}(u)B_{j,2}(v)}, \tag{3.33}$$

where the basis functions $B_{i,n}$ are Bernstein polynomials

$$B_{i,n}(t) = \frac{n!}{i!(n-i)!}(1-t)^{n-i}t^i, \qquad i = 0, \ldots, n, \tag{3.34}$$

$\boldsymbol{b}_{ij}$ are control points, and w_{ij} are weights. Figure 3-8 (top) shows the mapping of $(u, v) \in [0, 1] \times [0, 1]$ to a surface together with its control points $\boldsymbol{b}_{ij}$. In the middle plot, we show how a high-quality mesh in the parameter space gets deformed by the mapping. Finally, in the bottom plot we show the result after force equilibrium in $\mathbb{R}^3$ and first-order projections as described above. This mesh is of course distorted in the parameter space, but we did not have to explicitly work with this anisotropy.

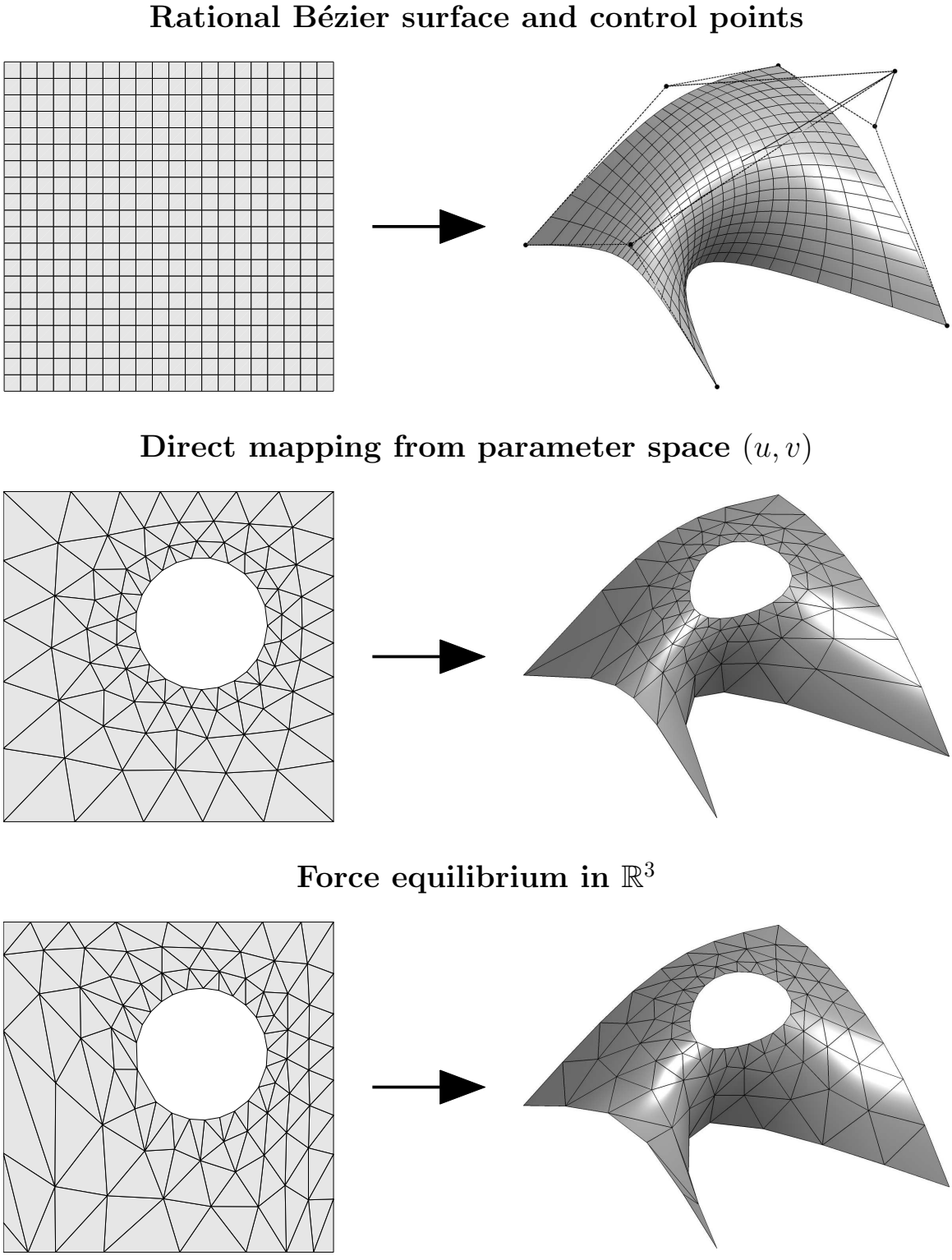

Figure 3-8: Mesh generation for a Bézier surface patch. Finding a force equilibrium in $\mathbb{R}^3$ gives a high-quality mesh directly.

Chapter 4

Mesh Size Functions

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* $h(\boldsymbol{x})$ which is determined by many factors. At curved boundaries, $h(\boldsymbol{x})$ should be small to resolve the curvature. In region 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, $h(\boldsymbol{x})$ 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 $\nabla h(\boldsymbol{x})$.

In many mesh generation algorithms it is advantageous if an appropriate mesh size function $h(\boldsymbol{x})$ is known prior to computing the mesh. This includes our mesh generator that we developed in Chapter 2 and 3, but also the advancing front method [48] and the paving method for quadrilateral meshes [6]. The popular Delaunay refinement algorithm [53], [59] 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 [47], [73], [72]. A common solution is to represent the size function in

a discretized form on a background grid and obtain the actual values of $h(\boldsymbol{x})$ by interpolation, as described in Section 3.1.1.

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.

4.1 Problem Statement

We define our mesh size function $h(\boldsymbol{x})$ for a given geometry by the following five properties:

1. **Curvature Adaptation** On the boundaries, we require $h(\boldsymbol{x}) \leq 1/K|\kappa(\boldsymbol{x})|$, 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, $h(\boldsymbol{x}) \leq \mathrm{lfs}(\boldsymbol{x})/R$. The local feature size $\mathrm{lfs}(\boldsymbol{x})$ is, loosely speaking, half the width of the geometry at $\boldsymbol{x}$,. The parameter R gives half the number of elements across narrow regions of the geometry.

3. **Non-geometric Adaptation** An additional external spacing function $h_{\mathrm{ext}}(\boldsymbol{x})$ might be given by an adaptive numerical solver or as a user-specified function. We then require that $h(\boldsymbol{x}) \leq h_{\mathrm{ext}}(\boldsymbol{x})$.

4. **Grading Limiting** The grading requirement means that the size of two neighboring elements in a mesh should not differ more than a factor G, or $h_i \leq G h_j$ 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 $|\nabla h(\boldsymbol{x})| \leq G - 1 \equiv g$

(an alternative definition is $g \equiv \log G$, depending on the interpretation of the element sizes).

5. **Optimality** In addition to the above requirements (which are all upper bounds), we require that $h(\boldsymbol{x})$ is as large as possible at all points.

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

4.2 Curvature Adaptation

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

$$\begin{cases} h_{\text{curv}}(\boldsymbol{x}) = 1/K|\kappa(\boldsymbol{x})|, & \text{if } \phi(\boldsymbol{x}) = 0, \\ \infty, & \text{if } \phi(\boldsymbol{x}) \neq 0, \end{cases} \tag{4.1}$$

where $\kappa(\boldsymbol{x})$ is the curvature at $\boldsymbol{x}$. 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 $h(\boldsymbol{x})$ 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, 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:

$$\kappa = \nabla \cdot \frac{\nabla\phi}{|\nabla\phi|} = \frac{\phi_{xx}\phi_y^2 - 2\phi_y\phi_x\phi_{xy} + \phi_{yy}\phi_x^2}{(\phi_x^2 + \phi_y^2)^{3/2}}. \tag{4.2}$$

In 3-D similar expressions give the mean curvature H and the Gaussian curvature K, from which the principal curvatures are obtained as $\kappa = H \pm \sqrt{H^2 - K}$. 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 (4.1) 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 (4.1). 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 $\boldsymbol{x}_{ij}$. This point is generally not located on the boundary, but a distance $|\phi_{ij}|$ away. If we set $h_{\text{curv}}(\boldsymbol{x}_{ij}) = 1/(K|\kappa_{ij}|)$ we get two sources of errors:

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

$$\kappa_{\text{bound}} = \frac{1}{\frac{1}{\kappa_{ij}} + \phi_{ij}} = \frac{\kappa_{ij}}{1 + \kappa_{ij}\phi_{ij}} \tag{4.3}$$

  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 $\boldsymbol{x}_{ij}$ 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 this expression involving an 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:

$$h_{\text{curv}}(\boldsymbol{x}_{ij}) = \begin{cases} \left| \frac{1+\kappa_{ij}\phi_{ij}}{K\kappa_{ij}} \right| - g\phi_{ij}, & |\phi_{ij}| \leq 2\Delta x, \\ \infty, & |\phi_{ij}| > 2\Delta x. \end{cases} \tag{4.4}$$

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.

4.3 Feature Size Adaptation

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

$$\begin{cases} h_{\text{lfs}}(\boldsymbol{x}) = \text{lfs}(\boldsymbol{x})/R, & \text{if } \phi(\boldsymbol{x}) \leq 0, \\ \infty, & \text{if } \phi(\boldsymbol{x}) > 0. \end{cases} \tag{4.5}$$

The *local feature size* $\text{lfs}(\boldsymbol{x})$ is a measure of the distance between nearby boundaries. It is defined by Ruppert [53] as "the larger distance from $\boldsymbol{x}$ 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 [2]) that the local feature size at a boundary point $\boldsymbol{x}$ is equal to the smallest distance between $\boldsymbol{x}$ 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.

This 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:

$$\text{lfs}_{\text{MA}}(\boldsymbol{x}) = |\phi(\boldsymbol{x})| + |\phi_{\text{MA}}(\boldsymbol{x})|, \tag{4.6}$$

where $\phi(\boldsymbol{x})$ is the distance function for the domain and $\phi_{\text{MA}}(\boldsymbol{x})$ is the distance to its medial axis (MA). The distances $\phi_{\text{MA}}(\boldsymbol{x})$ are always positive, but we take its absolute value to emphasize that we always add positive distances.

The expression (4.6) obviously reduces to the definition in [2] at boundary points $\boldsymbol{x}$, since then $\phi(\boldsymbol{x}) = 0$. For a narrow region with parallel boundaries, $\text{lfs}(\boldsymbol{x})$ 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 (4.6), we have to compute the *medial axis transform* $\phi_{\text{MA}}(\boldsymbol{x})$ in addition to the given distance function $\phi(\boldsymbol{x})$. If we know the location of the medial axis we can use the techniques described in Section 3.1.2, for example explicit 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 [8], are based on explicit representations of the geometry. Kimmel et al [36] 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 [62] used a divergence based formulation combined with a *thinning* process to guarantee a correct topology. Telea and Wijk [52] 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 slightly 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 [62]). 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 (4.6) 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 minimize the number of user parameters, since the algorithm must 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 a parameter giving the smallest resolved curvature.

Our method is based on a simple idea: 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 4-1). 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 4.1. 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)
- 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 circle of radius $1/|\kappa|$, where κ is the largest curvature at the two points. With one unit separation

Algorithm 4-1: Skeletonization using Distance Function

Description: Compute the crossing between grid edges and the medial axis.
Input: Grid $\boldsymbol{x}_{ijk}$, discretized distance function ϕ_{ijk}, parameters γ and κ_{tol}
Output: Medial axis crossings $\boldsymbol{p}_i$ and distances to neighboring nodes $\phi_{\text{MA}}(\boldsymbol{x}_{ijk})$

Normalize grid points $\boldsymbol{x}_{ijk}$ and ϕ_{ijk} to have unit grid spacing
Compute $\nabla\phi_{ijk}$ with one-sided difference approximations
Compute maximal principal curvature κ_{ijk} from ϕ_{ijk} with difference approximations
for all consecutive six nodes $\boldsymbol{x}_{i-2:i+3,j,k}$
Define $\phi_1, \ldots, \phi_6 = \phi_{i-2,j,k}, \ldots, \phi_{i+3,j,k}$
Fit parabola $p_1(s)$ to the data points $(s, \phi) = (-2, \phi_1), (-1, \phi_2), (0, \phi_3)$
Fit parabola $p_2(s)$ to the data points $(s, \phi) = (1, \phi_4), (2, \phi_5), (3, \phi_6)$
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 at $\boldsymbol{x}_{i,j,k}$ and $\boldsymbol{x}_{i+1,j,k}$

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

if $\alpha < 1 - \gamma^2 \max(\kappa_1^2, \kappa_2^2, \kappa_{\text{tol}}^2)/2$:
Accept $\boldsymbol{p} = \boldsymbol{x}_{ijk} + \boldsymbol{e}_1 h s_0$ as a medial axis point
Compute medial axis normal

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

Compute the distance to the two neighboring points

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

end if
end if
end for
Within each interval $\boldsymbol{x}_{i-2:i+3,j,k}$ keep p_i with largest $d\Delta p/ds(s_0)$
Repeat for consecutive nodes in y- and z-direction

Table 4.1: The algorithm for detecting the medial axis in a discretized distance function and computing the distances to neighboring nodes.

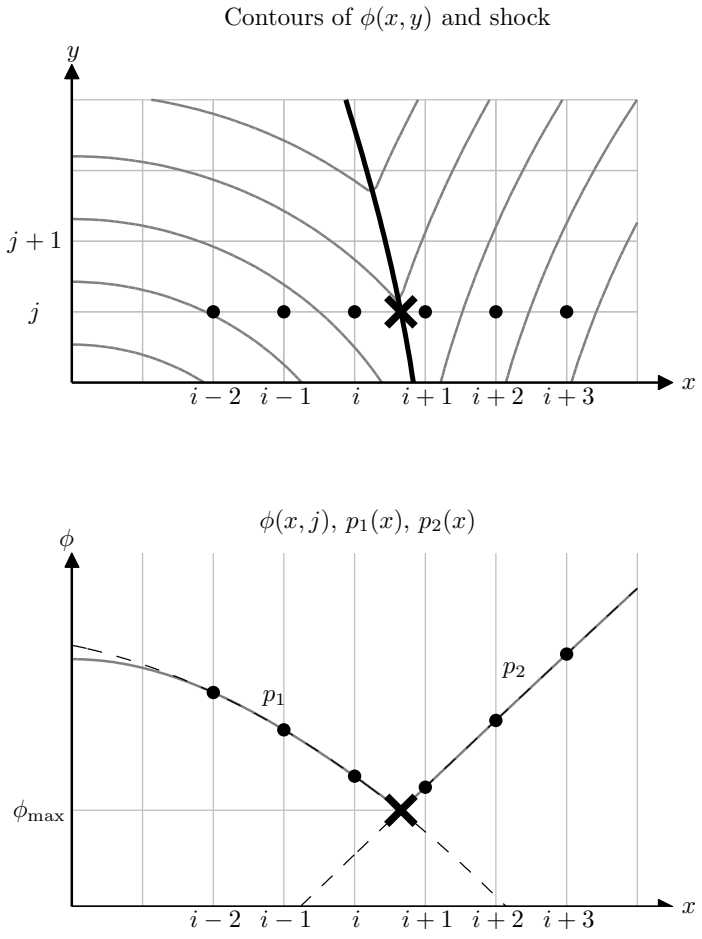

Figure 4-1: Detection of shock in the distance function $\phi(x, y)$ along the edge $(i, j), (i+1, j)$. The location of the shock is given by the crossing of the two parabolas $p_1(x)$ and $p_2(x)$.

between the points and an angle θ between the fronts, this dot product is

$$\cos\theta = 1 - 2\sin^2(\theta/2) = 1 - 2(|\kappa|/2)^2 = 1 - \kappa^2/2 \tag{4.7}$$

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

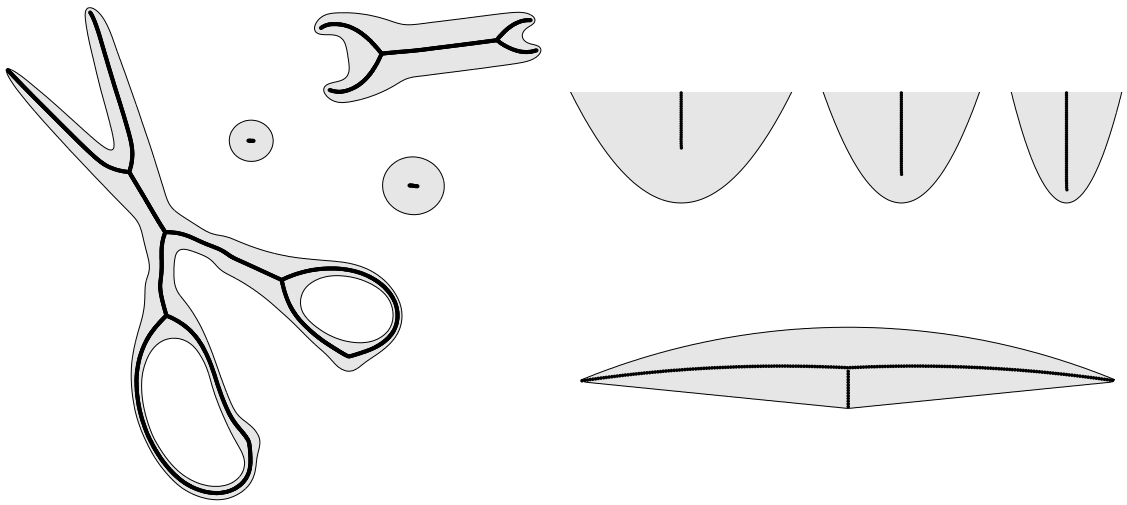

Figure 4-2: Examples of medial axis calculations for some planar geometries.

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 $\phi_{\mathrm{MA}}(\boldsymbol{x})$ in the entire domain using the fast marching method.

Some examples of medial axis detections are shown in Figure 4-2. Note how the three parabolas (top right) are handled correctly with the curvature dependent tolerances.

4.4 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 $|\nabla h(\boldsymbol{x})| \leq g$ with $g \equiv G - 1$. In some simple cases, this can be built into the size function explicitly. For example, a "point-source" size constraint $h(\boldsymbol{y}) = h_0$ in a convex domain can be extended as $h(\boldsymbol{x}) = h_0 + g|\boldsymbol{x} - \boldsymbol{y}|$, 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 4-6), or to non-constant g (Figures 4-10 and 4-11).

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 [10]. When the iterations converge, the solution satisfies $|\nabla h(\boldsymbol{x})| \leq g$ 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 [26]. This data structure is used in the quadtree meshing algorithm [71], 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 $\boldsymbol{x}$, it is natural to formulate the gradient limiting as a PDE with solution $h(\boldsymbol{x})$ independently of the actual background mesh. We can see many benefits in doing this:

- The analytical solution is exactly the optimal gradient limited size function $h(\boldsymbol{x})$ that we want, as shown by Theorem 4.4.1. 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.

4.4.1 The Gradient Limiting Equation

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

$$\frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, g), \tag{4.8}$$

with initial condition

$$h(t = 0, \boldsymbol{x}) = h_0(\boldsymbol{x}). \tag{4.9}$$

When $|\nabla h| \leq g$, (4.8) 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 (4.8), showing that it is indeed the optimal solution:

Theorem 4.4.1. *Let $\Omega \subset \mathbb{R}^n$ be a bounded convex domain, and $I = (0, T)$ a given time interval. The steady-state solution $h(x) = \lim_{T\to\infty} h(x, T)$ to*

$$\begin{cases} \frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, g) & (x, t) \in \Omega \times I \\ h(x, t)|_{t=0} = h_0(x) & x \in \Omega \end{cases} \tag{4.10}$$

is

$$h(x) = \min_y (h_0(y) + g|x - y|). \tag{4.11}$$

Proof. The Hopf-Lax theorem [34] 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

$$u(x,t) = \min_y \left[u_0(y) + tF^*\left((x-y)/t\right)\right], \tag{4.12}$$

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

For our equation (4.10), rewrite as $\frac{\partial h}{\partial t} + F(\nabla h) = 0$, with $F(w) = |w| - \min(|w|, g)$. The conjugate function is

$$\begin{aligned} F^*(u) &= \max_w(wu - F(w)) \\ &= \max_w(wu - |w| + \min(|w|, g)) \\ &= \begin{cases} g|u|, & \text{if } |u| < 1, \\ +\infty & \text{if } |u| \geq 1. \end{cases} \end{aligned} \tag{4.13}$$

Using (4.12), we get

$$\begin{aligned} h(x,t) &= \min_y \left[h_0(y) + tF^*\left((x-y)/t\right)\right] \\ &= \min_{\substack{y \\ |x-y| \leq t}} (h_0(y) + g|x-y|). \end{aligned} \tag{4.14}$$

Let $t \to \infty$ to get the steady-state solution to (4.10):

$$h(x) = \min_y(h_0(y) + g|x-y|). \tag{4.15}$$

□

Note that the solution (4.11) is composed of infinitely many point-source solutions as described before. We could in principle define an algorithm based on (4.11) 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 (4.10) using efficient Hamilton-Jacobi

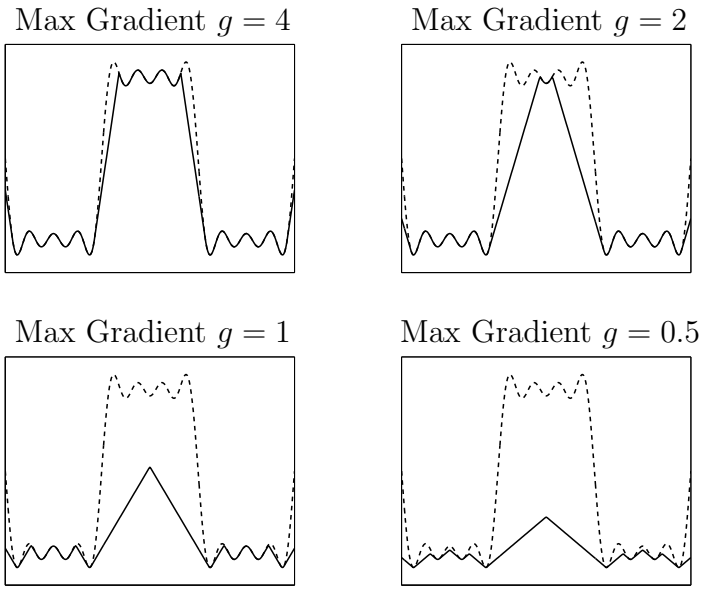

Figure 4-3: Illustration of gradient limiting by $\partial h/\partial t + |\nabla h| = \min(|\nabla h|, g)$. The dashed lines are the initial conditions h_0 and the solid lines are the gradient limited steady-state solutions h for different parameter values g.

solvers.

The gradient limiting is illustrated by a one dimensional example in Figure 4-3, where (4.10) 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 $h_0(\boldsymbol{x})$. Our solution is not necessarily smooth, but it is continuous and $|\nabla h| \leq g$ everywhere.

4.4.2 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.

On a Cartesian background grid, the equation (4.8) can be solved with just a few

lines of code using the following iteration:

$$h_{ijk}^{n+1} = h_{ijk}^{n} + \Delta t \left(\min(\nabla_{ijk}^{+}, g) - \nabla_{ijk}^{+} \right) \tag{4.16}$$

where

$$\begin{aligned} \nabla_{ijk}^{+} = \big[& \max(D^{-x} h_{ijk}^{n}, 0)^2 + \min(D^{+x} h_{ijk}^{n}, 0)^2 + \\ & \max(D^{-y} h_{ijk}^{n}, 0)^2 + \min(D^{+y} h_{ijk}^{n}, 0)^2 + \\ & \max(D^{-z} h_{ijk}^{n}, 0)^2 + \min(D^{+z} h_{ijk}^{n}, 0)^2 \big]^{1/2} \end{aligned} \tag{4.17}$$

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 $\Delta h(\boldsymbol{x})$ 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 (4.16) 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 (4.16) is first-order accurate in space, and higher accuracy can be achieved by using a second-order solver. See [45] and [33] for details.

For faster solution of (4.8) we use a modified version of the fast marching method (see Section 3.1.2). The main idea for solving our PDE (4.8) 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

Algorithm 4-2: Fast Gradient Limiting

Description: Solve (4.8) on a Cartesian grid
Input: Initial discretized h_0, grid spacing Δx
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}^{new} in $\nabla_{ijk}^{+} = g$ from (4.17)
      Set $h_{ijk} \leftarrow \min(h_{ijk}, h_{ijk}^{\text{new}})$
    end if
  end for
end while

Table 4.2: 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.

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 (4.17). 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 4.2. 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 [55] 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 [3]. 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 [35], and in [18] 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 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.

4.4.3 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 (4.11), and we solve the problem on a Cartesian grid of varying resolution.

In Table 4.3 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 $h_j \leftarrow \min(h_j, h_i + g|\boldsymbol{x}_j - \boldsymbol{x}_i|)$ (assuming $h_j > h_i$). The iterative Hamilton-Jacobi solver is based on the iteration (4.16) with a tolerance of about two digits. All algorithms are implemented in C++

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

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 (4.11) 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 (4.11), and an algorithm based on this expression would of course be exact to full precision. Figure 4-4 shows solutions 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 [55]. The error with the edge-based solver might be even larger for unstruc-

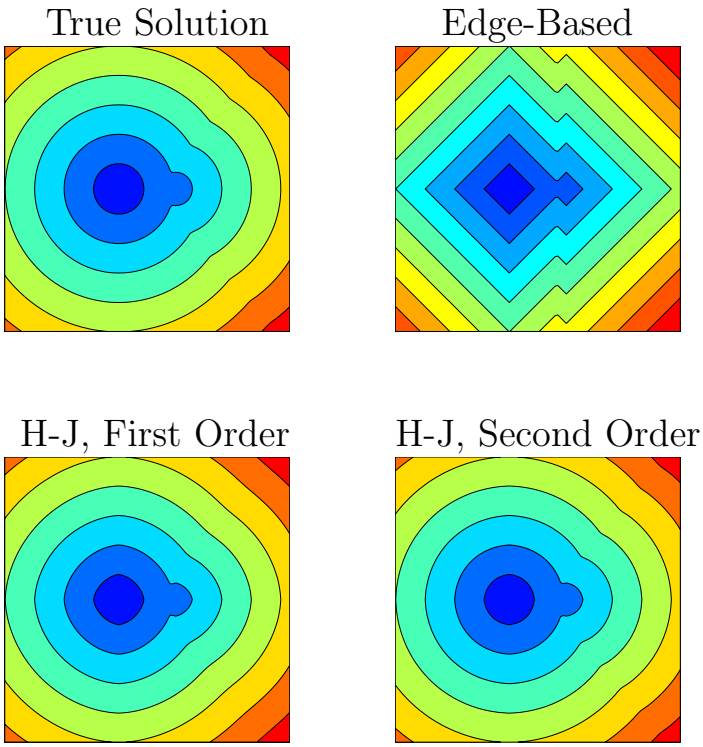

Figure 4-4: Comparison of the accuracy of the discrete edge-based solver and the continuous Hamilton-Jacobi solver on a Cartesian background mesh. The edge-based solver does not capture the continuous nature of the propagating fronts.

tured background meshes which often have low element qualities.

4.5 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 $h_{\text{ext}}(\boldsymbol{x})$ is provided as input. Our final size function must be smaller than these at each point in space:

$$h_0(\boldsymbol{x}) = \min(h_{\text{curv}}(\boldsymbol{x}), h_{\text{lfs}}(\boldsymbol{x}), h_{\text{ext}}(\boldsymbol{x})) \tag{4.18}$$

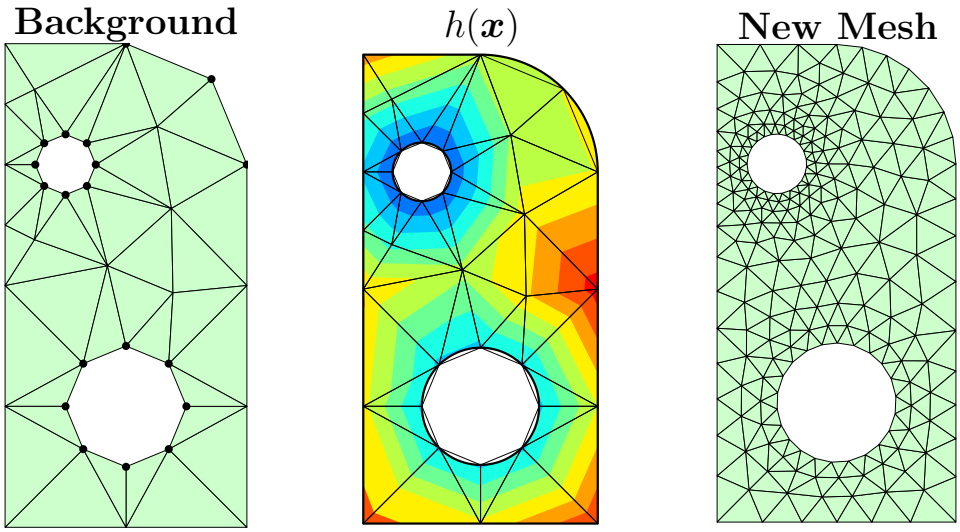

Figure 4-5: Example of gradient limiting with an unstructured background grid. The size function is given at the curved boundaries and computed by (4.8) at the remaining nodes.

Finally, we apply the gradient limiting algorithm from Section 4.4 on h_0 to get the mesh size function h, by solving:

$$\frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, G_c) \tag{4.19}$$

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

We now show a number of examples, with different geometries, background grids, and feature size definitions.

4.5.1 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 4-5, we set $h_0(\boldsymbol{x})$ 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

$$h(x) = \min_i (h_i + g\phi_i(\boldsymbol{x})). \tag{4.20}$$

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 (4.20) 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.

In Figure 4-6 we show a size function for a geometry with a narrow slit, again generated using the unstructured gradient limiting solver. The initial size function $h_0(\boldsymbol{x})$ 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 (4.20), where more expensive geometric search routines would have to be used.

A more complicated example is shown in Figure 4-7 (left plots). 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 4.3. 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 4-2 to get a well-behaved size function. We also use curvature adaptation as before. Note that this mesh size function would be very expensive 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 bound-

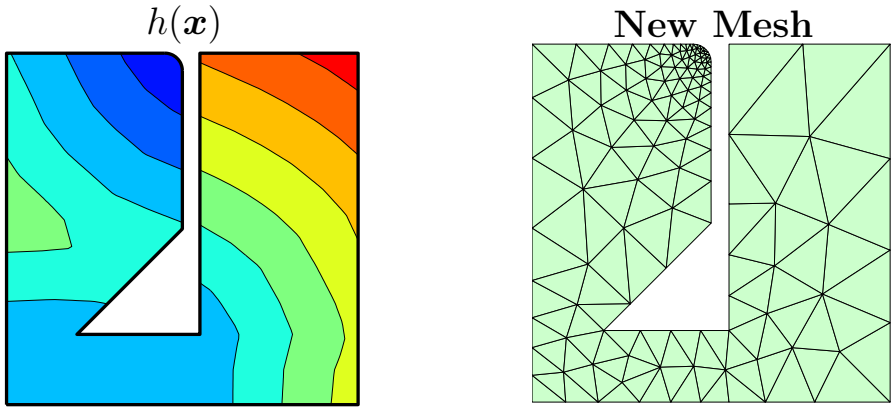

Figure 4-6: Another example of gradient limiting, showing that non-convex regions are handled correctly. The small sizes at the curved boundary do not affect the region at the right, since there are no connections across the narrow slit.

aries in Figure 4-7 (right plots). 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 (4.4) since the grid is not aligned with the boundaries.

The PDE-based formulation generalizes to arbitrary dimensions, and in Figure 4-8 we show a 3-D example. 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.

A more complex model is shown in Figure 4-9.[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.

4.5.2 Space and Solution Dependent g

The solution of the gradient limiting equation remains well-defined if we make $g(\boldsymbol{x})$ a function of space. The numerical schemes in Section 4.4.2 are still valid, and we

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

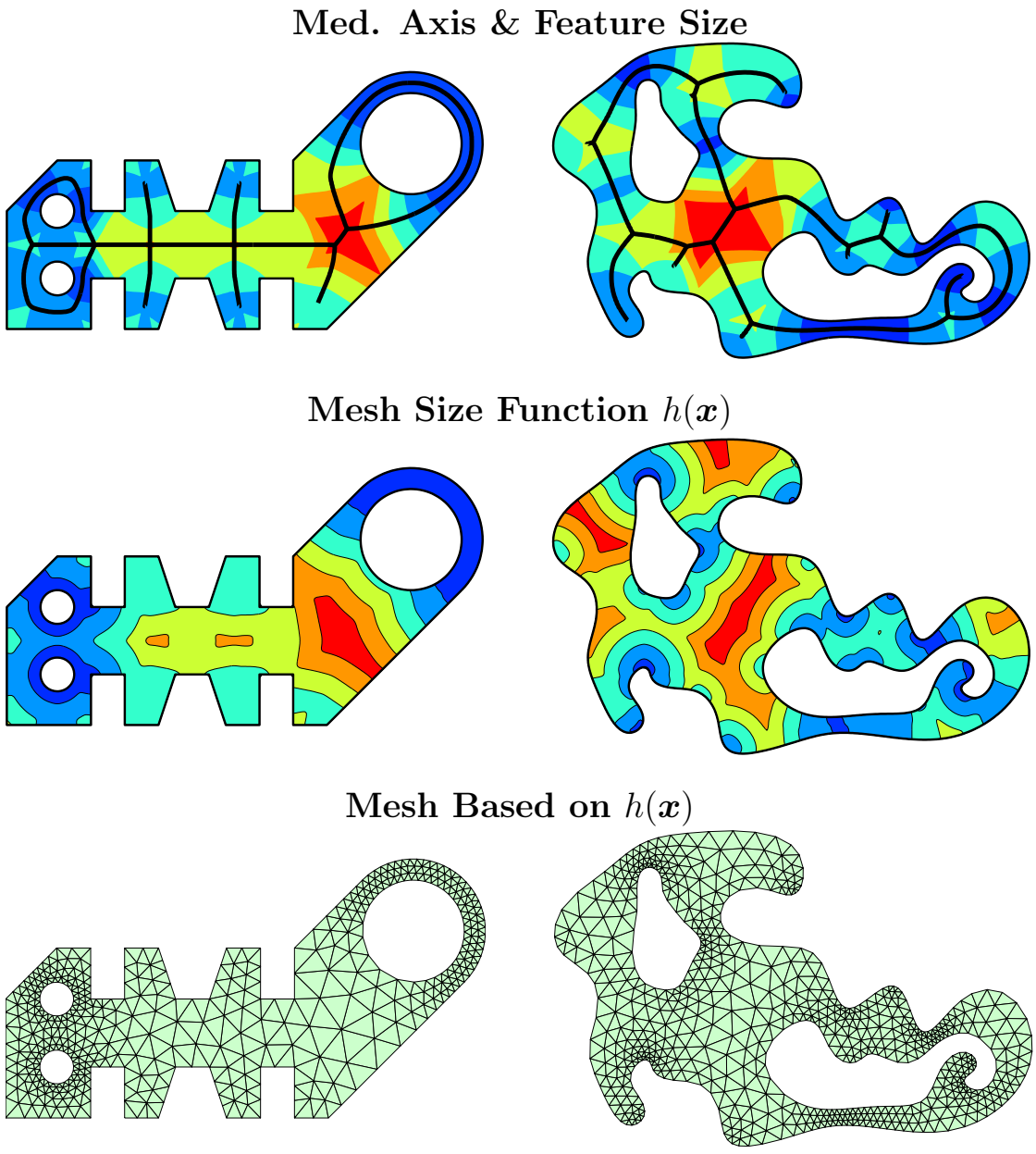

Figure 4-7: Mesh size functions taking into account both feature size, curvature, and gradient limiting. The feature size is computed as the sum of the distance function and the distance to the medial axis.

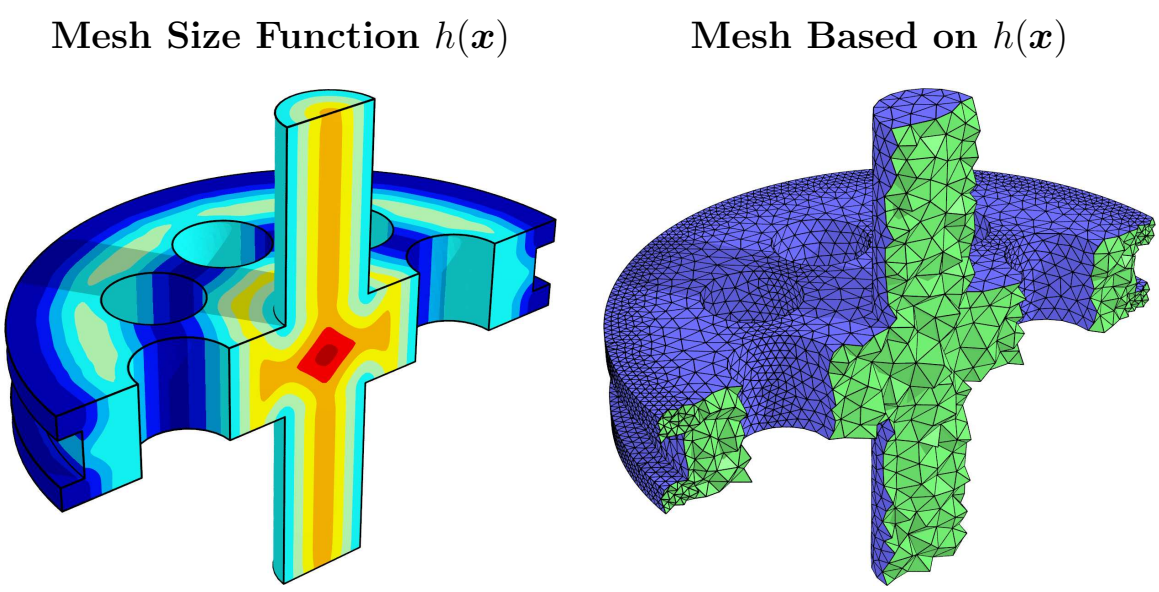

Figure 4-8: Cross-sections of a 3-D mesh size function and a sample tetrahedral mesh.

replace for example g in (4.16) with g_{ijk}. An application 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 4-10 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 $|\nabla h| \leq g(\boldsymbol{x})$ everywhere.

Another possible extension is to let g be a function of the solution $h(\boldsymbol{x})$ (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 4-11, where $g(h)$ varies smoothly between 0.6 (for small elements) and 0.2 (for large elements).

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.

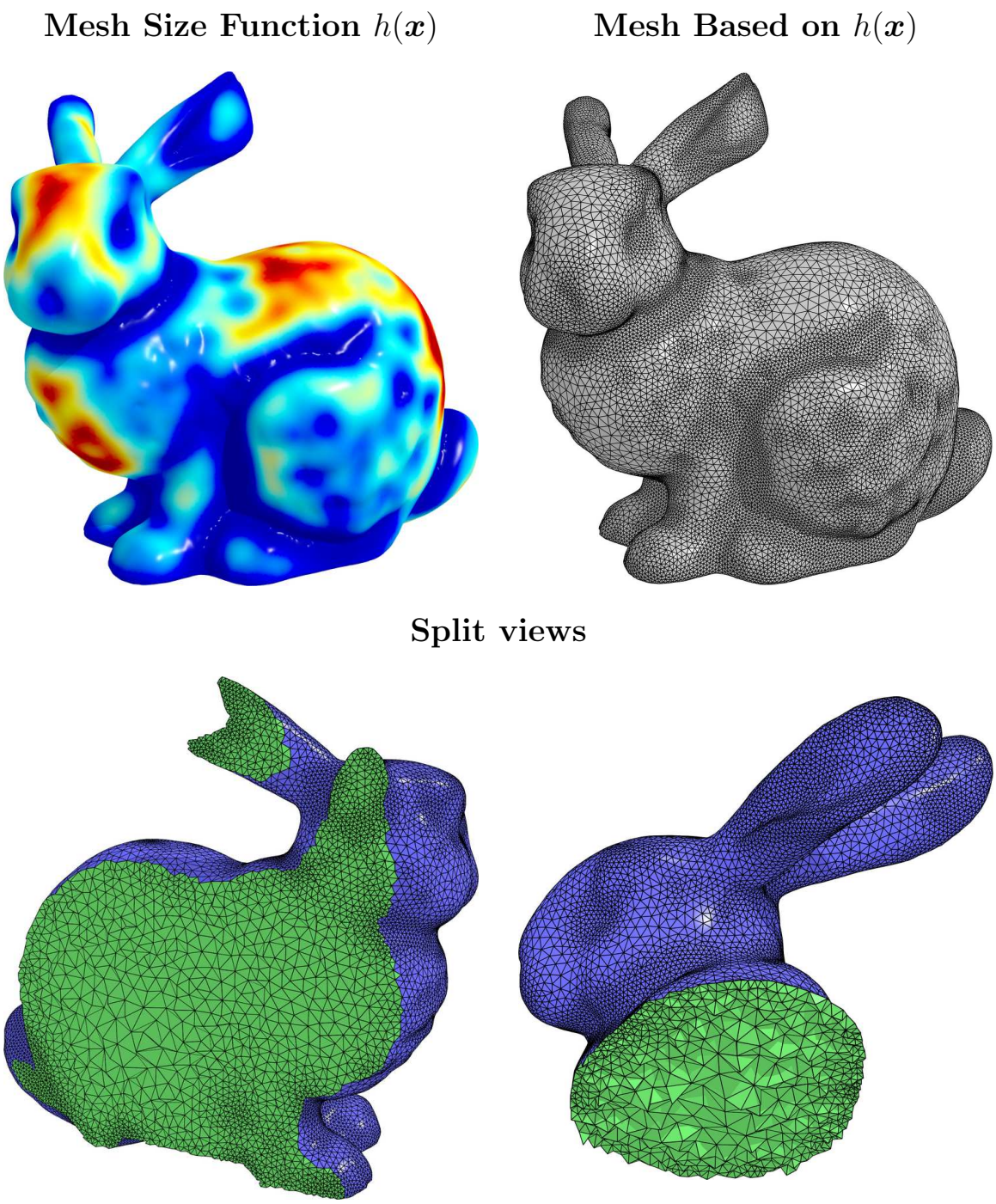

Figure 4-9: A 3-D mesh size function and a sample tetrahedral mesh. Note the small elements in the narrow regions, given by the local feature size, and the smooth increase in element sizes.

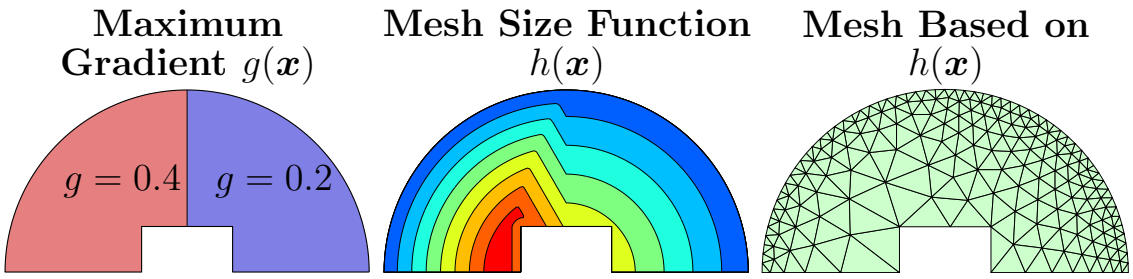

Figure 4-10: Gradient limiting with space-dependent $g(\boldsymbol{x})$.

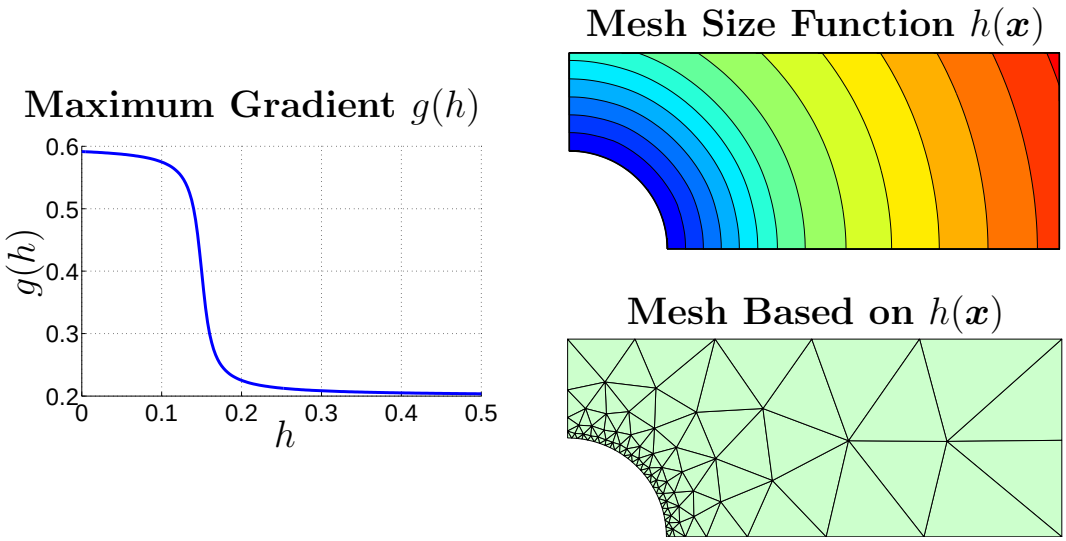

Figure 4-11: Gradient limiting with solution-dependent $g(h)$. The distances between the level sets of $h(\boldsymbol{x})$ are smaller for small h, giving a faster increase in mesh size.

Chapter 5

Applications

In this chapter we show several applications that are particularly well suited for our mesh generator. The iterative node movement is appropriate for meshing geometries that change with time ("moving meshes"), and we show how to combine the level set method with the finite element method for applications in shape optimization, stress-induced instabilities, and fluid dynamics. We can also improve remeshing for numerical adaptation, where our gradient limiting equation is solved on the unstructured background mesh from the previous iteration. Finally, we show how implicit geometries can be used for meshing objects in images, in two and three dimensions.

The figures in this chapter bring out the key points of each application, more powerfully than the words. The central problem is to remesh adaptively and quickly, maintaining high quality.

5.1 Moving Meshes

When our mesh generator creates a sequence of meshes for a moving geometry, we always have a good initial configuration by the mesh from the previous time step. Typically we only need a few additional iterations to obtain a new high quality mesh. At each step, we need an initial guess for the location of the mesh points. For the first mesh we use one of the methods in Section 3.3.3 or the simple rejection technique in our MATLAB code. For the subsequent meshes, a fast start is obtained by displacing

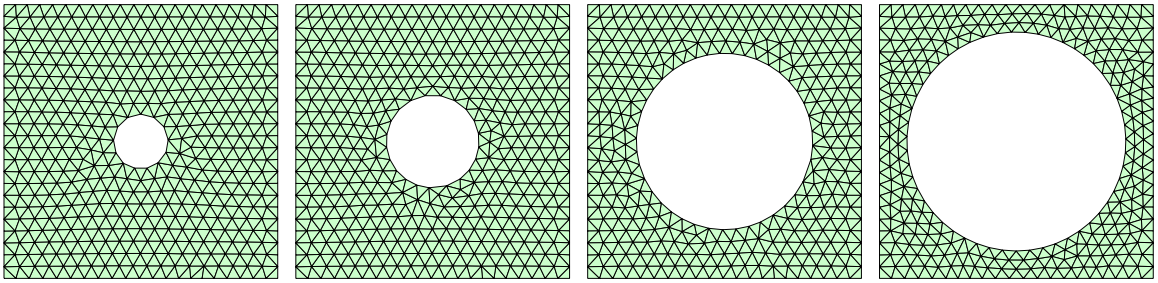

Figure 5-1: Example of moving mesh, shown at four different times. The point density control is used to keep the mesh size uniform when the area changes.

the mesh points a distance $\boldsymbol{v}(\boldsymbol{p})\Delta t$ for each mesh point $\boldsymbol{p}$, where $\boldsymbol{v}(\boldsymbol{p})$ is the velocity of the node point and Δt is the time interval between the two geometries.

Moving meshes are best visualized as animations. Please visit www-math.mit.edu/~persson/mesh to view our movies. For the illustrations here, we show meshes at a few different times. As a simple example of a moving mesh, we show a geometry consisting of a square having a circular hole with a radius that changes with time, see Figure 5-1. Note how the density control ensures that the element sizes are approximately equal even though the geometry area changes drastically.

One benefit of the algorithm is that mesh elements far away from the moving interface are left essentially unmodified. This allows easier and more accurate solution transfer between the meshes and better opportunities for mesh compression. We also take advantage of this fact to improve the performance of our algorithm. We assign a stiffness to each mesh edge (the constant k in (2.5)) that increases with the distance from the moving interface. A few mesh elements away we set k to infinity, which means these nodes do not move at all. We can then ignore them when solving for force equilibrium, which gives a significant performance improvement. The technique is illustrated by the example in Figure 5-2, where we mesh a circular hole moving through a rectangle. Only elements in a thin layer close to the circle are allowed to move at each step, but the element qualities remain very high.

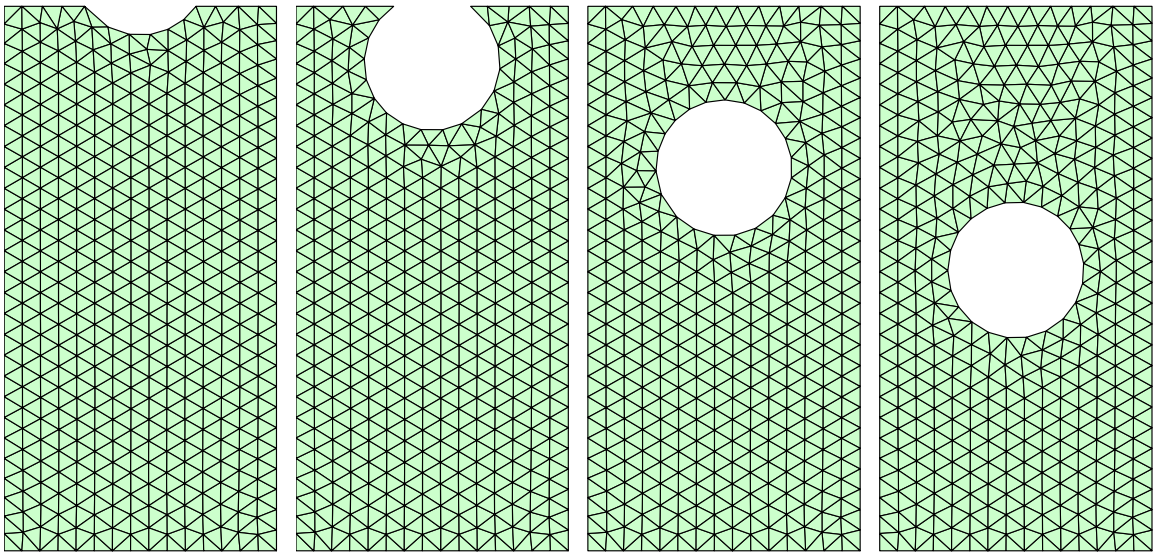

Figure 5-2: Only mesh points close to the moving interface are allowed to move. This improves the performance dramatically and provides easier solution transfer between the old and new grids.

5.1.1 Level Sets and Finite Elements

Since our mesh generator is based on distance functions we can use the level set method [45] to propagate the geometry boundary according to a given velocity field. In this way we combine the benefits of the level set method (robust interface propagation, entropy solutions, topology changes, easy extension to higher dimensions) with the flexibility of general purpose finite element calculations on unstructured meshes.

Our moving algorithm starts with an initial geometry, represented by a discretized implicit function $\phi(\boldsymbol{x})$ as before. The geometry boundary is then evolved in time according to a velocity field $\boldsymbol{v}(\boldsymbol{x})$ or a normal velocity field $F(\boldsymbol{x})$. These fields are typically dependent on a solution of a physical problem, which in turn depends on the current geometry. With our unstructured meshes we can use the finite element method to solve these physical problems.

The actual propagation of the boundary is done using the level set method, which solves hyperbolic PDEs on the Cartesian grid using entropy satisfying numerical

schemes. For a velocity field $\boldsymbol{v}$ it solves the convection equation

$$\phi_t + \nabla\phi \cdot \boldsymbol{v} = 0 \tag{5.1}$$

and for a normal field F it solves the *level set equation*

$$\phi_t + F|\nabla\phi| = 0 \tag{5.2}$$

(both $\boldsymbol{v}$ and T may depend on space and time). These equations are solved using numerical discretizations, [56], [43]. We use different schemes for motion due to curvature and for general, curvature independent motion. For the general motion, we use a first order finite difference approximation on the Cartesian grid:

$$\phi_{ijk}^{n+1} = \phi_{ijk}^{n} + \Delta t_1 \left(\max(F,0)\nabla_{ijk}^{+} + \min(F,0)\nabla_{ijk}^{-}\right), \tag{5.3}$$

where

$$\begin{aligned}\nabla_{ijk}^{+} = \big[&\max(D^{-x}\phi_{ijk}^{n},0)^2 + \min(D^{+x}\phi_{ijk}^{n},0)^2 + \\ &\max(D^{-y}\phi_{ijk}^{n},0)^2 + \min(D^{+y}\phi_{ijk}^{n},0)^2 + \\ &\max(D^{-z}\phi_{ijk}^{n},0)^2 + \min(D^{+z}\phi_{ijk}^{n},0)^2\big]^{1/2},\end{aligned} \tag{5.4}$$

$$\begin{aligned}\nabla_{ijk}^{-} = \big[&\min(D^{-x}\phi_{ijk}^{n},0)^2 + \max(D^{+x}\phi_{ijk}^{n},0)^2 + \\ &\min(D^{-y}\phi_{ijk}^{n},0)^2 + \max(D^{+y}\phi_{ijk}^{n},0)^2 + \\ &\min(D^{-z}\phi_{ijk}^{n},0)^2 + \max(D^{+z}\phi_{ijk}^{n},0)^2\big]^{1/2}.\end{aligned} \tag{5.5}$$

Here, D^{-x} is the backward difference operator in the x-direction, D^{+x} the forward difference operator, etc. For the curvature dependent part of F, we use central difference approximations when computing the curvature. For further details and higher order schemes, see [56]. After evolving ϕ, it generally does not remain a signed distance function. We reinitialize by the techniques described in Section 3.1.2, for example by

explicit updates of the nodes close to the boundary $\phi(\boldsymbol{x}) = 0$ and the fast marching method for the remaining nodes.

We now show applications using moving meshes and implicit geometries. There are many application areas and here we will focus on two shape optimization problems, stress-induced instabilities, and fluid dynamics with moving boundaries.

5.1.2 Shape Optimization

Our first example comes from structural vibration control, and it was solved by Osher and Santosa using level set techniques on Cartesian grids [44]. We consider the eigenvalue problem in a region Ω with two materials:

$$-\Delta u = \lambda \rho(\boldsymbol{x}) u, \qquad x \in \Omega \tag{5.6}$$

$$u = 0, \qquad x \in \partial\Omega \tag{5.7}$$

The density is constant in the two subregions S and $\Omega \setminus S$:

$$\rho(\boldsymbol{x}) = \begin{cases} \rho_1 & \text{for } x \notin S \\ \rho_2 & \text{for } x \in S, \end{cases} \tag{5.8}$$

and we minimize λ_1 or λ_2 subject to the area constraint $\|S\| = K$.

We represent the boundary of S by a signed distance function on a Cartesian grid. To find the optimal distribution $\rho(\boldsymbol{x})$, we mesh the inside and the outside of the region using the techniques for internal boundaries in Section 3.4. We then solve the eigenvalue problem (5.6)-(5.7) using linear finite elements on our unstructured mesh. To decrease the ith eigenvalue, we compute a descent direction $\delta\phi = -F(\boldsymbol{x})|\nabla\phi|$, where the normal velocity is calculated using the current solution λ_i, u_i (see [44] for details):

$$F = \frac{\lambda_i(\rho_2 - \rho_1)}{\int_\Omega \rho u_i^2 \, dx} u_i^2. \tag{5.9}$$

Finally, we interpolate this velocity field to the Cartesian mesh, and use the level set method to propagate the interface. This velocity field is generally not mass conserving, and we implement the conservation constraint $\|S\| = K$ by solving for a Lagrange multiplier ν such that the velocity field $F + \nu$ conserves mass. This is a nonlinear problem in the unknown ν, which we solve using Newton's method.

Figure 5-3 shows the minimization of the first and the second eigenvalue on a sample geometry. Note how the dark region is split into two separate regions in minimizing λ_2. This automatic treatment of topology changes is one of the benefits of the level set method. By using unstructured meshes and the finite element method we achieve the following additional benefits:

- The material discontinuity is handled with high accuracy since the mesh is aligned with the interface between the two densities.

- We handle arbitrary outer geometries, again with high accuracy. Normally the level set method is used only on rectangular grids.

- The graded mesh sizes give asymptotically more efficient simulations.

Our second example of shape optimization comes from structural design improvement. The geometry in Figure 5-4 is clamped at the left edge, and a vertical force is applied at the midpoint of the right edge. We solve a linear elastostatic problem for the displacements $\boldsymbol{u}$:

$$\begin{cases} -\mathrm{div}(Ae(\boldsymbol{u})) = 0, & \text{in } \Omega \\ \boldsymbol{u} = 0, & \text{on } \Gamma_D \\ (Ae(\boldsymbol{u})) = \boldsymbol{g}, & \text{on } \Gamma_N. \end{cases} \tag{5.10}$$

The linear operator A comes from Hooke's law, and $\boldsymbol{g}$ are the boundary forces applied on a part Γ_N of the boundary. The remaining boundary Γ_D is fixed.

The optimization minimizes the compliance, which is the work done by the exter-

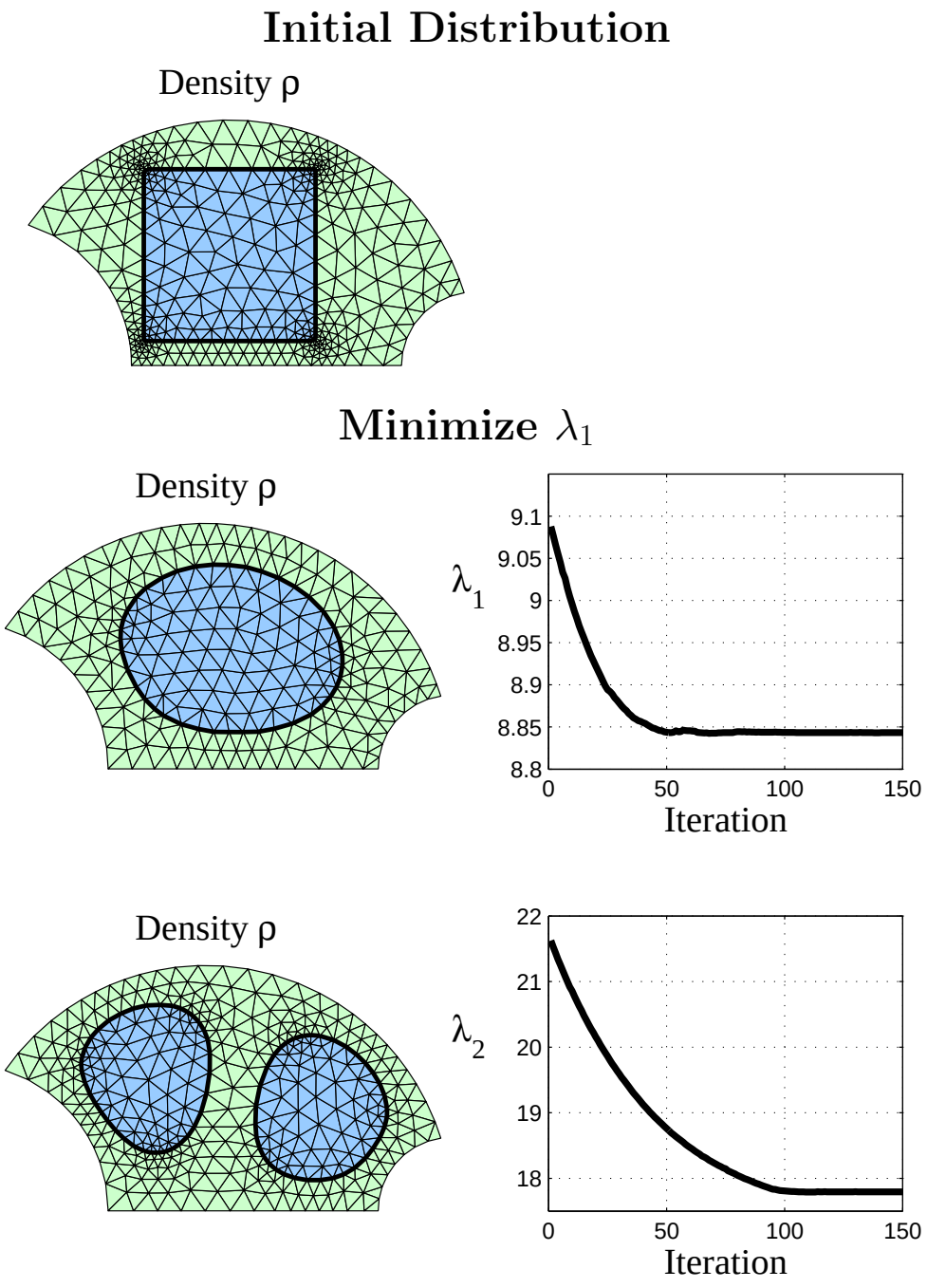

Figure 5-3: Finding an optimal distribution of two different densities (light gray/green indicates low density) to minimize eigenvalues.

nal forces $\boldsymbol{g}$ or the total elastic energy:

$$\text{Minimize} \int_{\Gamma_N} \boldsymbol{g} \cdot \boldsymbol{u}\, ds = \int_{\Omega} Ae(\boldsymbol{u}) \cdot e(\boldsymbol{u})\, dx. \tag{5.11}$$

We also impose the area constraint

$$\|\Omega\| = K, \tag{5.12}$$

where K is significantly smaller than the initial geometry area. The steepest descent direction is given by the velocity in the normal direction

$$F(\boldsymbol{x}) = -Ae(\boldsymbol{u}) \cdot e(\boldsymbol{u}), \tag{5.13}$$

see [1] for a derivation. Sethian and Wiegmann solved this problem using level set techniques together with the immersed interface method [57]. Allaire, Jouve, and Toader used a similar technique but solved the linear elastostatic problem using an Ersatz material approach [1]. Since we have high-quality unstructured meshes at each iteration, we can solve the physical problem using the finite element method. We discretize (5.10) using second-order triangular finite elements, and solve for the displacements with a direct sparse solver. The energy calculation for the steepest descent direction (5.13) is done on the unstructured mesh, and then interpolated to the Cartesian mesh for the interface evolution. The optimal structure is shown in the bottom plots of Figure 5-4.

Again we see advantages with our general meshes. The Neumann conditions at most of the boundaries are handled easily and accurately with the finite element method. The graded meshes resolve the fine details while having a minimum of total number of nodes. Finally, in this example we used standard Lagrange elements, but the finite element method is better developed than finite difference methods for advanced elasticity calculations and provides specialized elements.

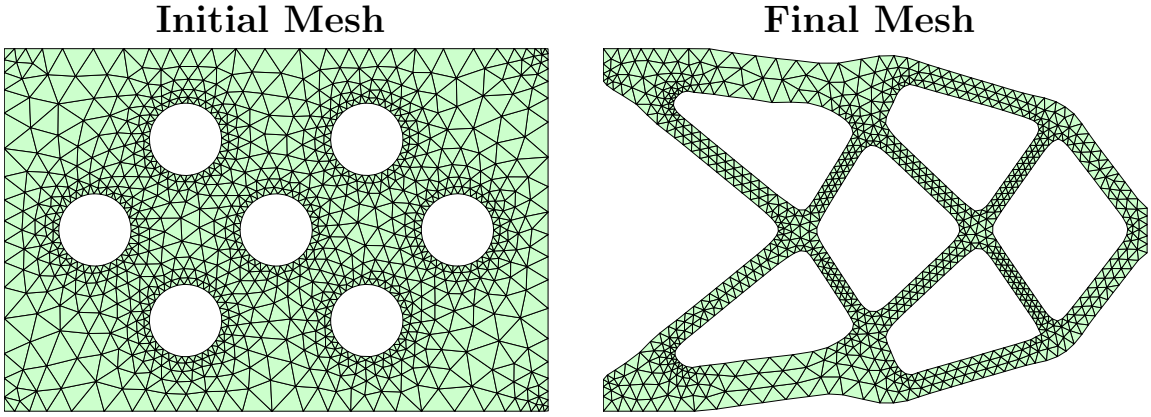

Figure 5-4: Structural optimization for minimum of compliance. The structure is clamped at the left edge and loaded vertically at the right edge midpoint. Note how the initial topology is modified by the algorithm.

5.1.3 Stress-Induced Instabilities

Our next example is a numerical study of Stress Driven Rearrangement Instabilities (SDRI). This phenomenon appears in epitaxial growth of solid nanofilms, for example InAs (Indium Arsenide) grown on a GaAs (Gallium Arsenide) substrate. The stress is induced by the misfit in the two lattice parameters. This results in a morphological instability where so-called quantum dots are formed on the surface, see Figure 5-5 for an experimental result. Our simulations are based on numerical analysis of the mathematical problem formulated and analyzed by M. Grinfeld [30].

We consider a thin sheet of material, with an initially almost flat upper surface. Our quasi-static interface evolution is based on minimization of the total energy, due to the elastic energy density ε and the surface tension σ:

$$E = \int_{\Omega} \varepsilon(\boldsymbol{x})\, dV + \int_{\Gamma_N} \sigma\, dS. \tag{5.14}$$

From this the descent direction can be derived, to give our interface evolution equation

$$F(\boldsymbol{x}) = \varepsilon(\boldsymbol{x}) - \sigma\kappa(\boldsymbol{x}), \tag{5.15}$$

with surface curvature $\kappa(\boldsymbol{x})$. For more details see [30].

We evolve the interface with level set techniques in the same way as before. At each

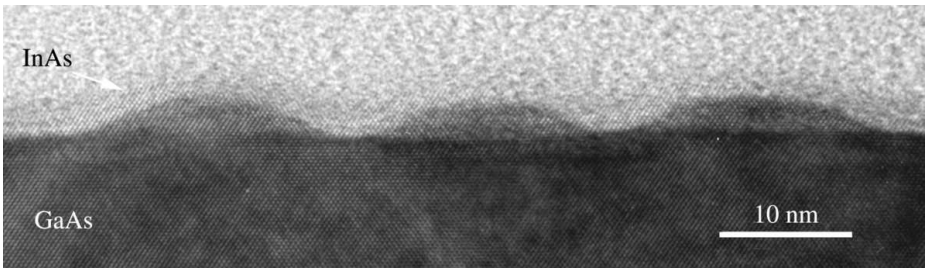

Figure 5-5: Transmission Electron Microscopy (TEM) image of defect-free InAs quantum dots.

step of the quasi-static interface evolution, we generate unstructured meshes for the domain $\phi(\boldsymbol{x}) \leq 0$ using our iterative techniques. We then discretize the elastostatic equations using the finite element method with second-order Lagrange elements. The resulting sparse linear system is solved with a direct sparse solver.

The stress is imposed by applying a prestraining ε_x on the discretized system. We write the total displacement field as a sum of a given stretched field $U_0 = \varepsilon_x X$ and an unknown, periodic perturbation U. We then solve for U in $KU = -KU_0$, where K is the stiffness matrix with boundary conditions incorporated.

Figure 5-6 shows the results of a two dimensional simulation. The distance function is represented on a block of dimensions 4×1, discretized with a grid of size 257×97. Initially, the surface is located a distance 0.66 from the bottom of the domain, and the height is perturbed at each node in the x, y-plane by normal distributed random numbers with standard deviation 0.0025. The material has Young's modulus $E = 1$, Poisson's ratio $\nu = 0.3$, and surface tension $\sigma = 0.20, 0.10, 0.05$ in the three simulations.

The boundary conditions specify the displacement in the z-direction $w = 0$ at the bottom face, and all displacements u, v, w are periodic at the left/right and the front/back faces. We use a timestep $\Delta t_1 = 0.05\sigma$ for the curvature independent part, and $\Delta t_2 = \Delta t_1/10$ for the motion by curvature.

The plots in Figure 5-6 show the meshes for the initial and the final boundary configurations, and the elastic energy density, where the color is based on a logarithmic scale. Animations of the quasi-static time evolution are provided at www-

math.mit.edu/∼persson/qdots, and more details including the results of our three dimensional simulations can be found in [31].

5.1.4 Fluid Dynamics

Our final example of moving meshes comes from computational fluid dynamics. We solve the incompressible Navier-Stokes equations

$$\frac{\partial \boldsymbol{u}}{\partial t} - \nu \nabla^2 \boldsymbol{u} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} + \nabla P = \boldsymbol{f} \tag{5.16}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{5.17}$$

on a domain Ω that changes with time. Here, ν is the dynamic viscosity and P the dynamic pressure, obtained by dividing the viscosity and the pressure by the density. In our two dimensional example, the fluid velocities $\boldsymbol{u} = (u, v)$ are specified on the entire boundary of Ω to be equal to the velocity of the boundary. Other variants are possible, allowing inflows, outflows, and free boundaries. We do not need any boundary conditions for the dynamic pressure P, but to make it unique we set it to zero at a few corner points.

We discretize (5.16-5.17) in space using the finite element method, with P2-P1 elements that satisfy the LBB stability condition. For the time evolution we use a Lagrangian approach, where at each time step we move all the mesh nodes according to the current velocity field $\boldsymbol{u}$. The total derivative $\partial \boldsymbol{u}/\partial t + \boldsymbol{u} \cdot \nabla \boldsymbol{u}$ then reduces to a simple partial derivative, and we do not have to discretize the nonlinear convection term $\boldsymbol{u} \cdot \nabla \boldsymbol{u}$. This formulation is also natural for problems with moving boundaries, since the boundary nodes have the same velocity as the boundary and therefore follow it in its motion.

A serious complication with Lagrangian node movement is that the mesh deforms significantly after a few time steps. We then do a few iterations with our mesh generator to improve the mesh. To compensate for this nonphysical node movement, we use a simple approach of interpolation between the meshes. This introduces additional numerical diffusion into the system, but it gives reasonable velocity fields for

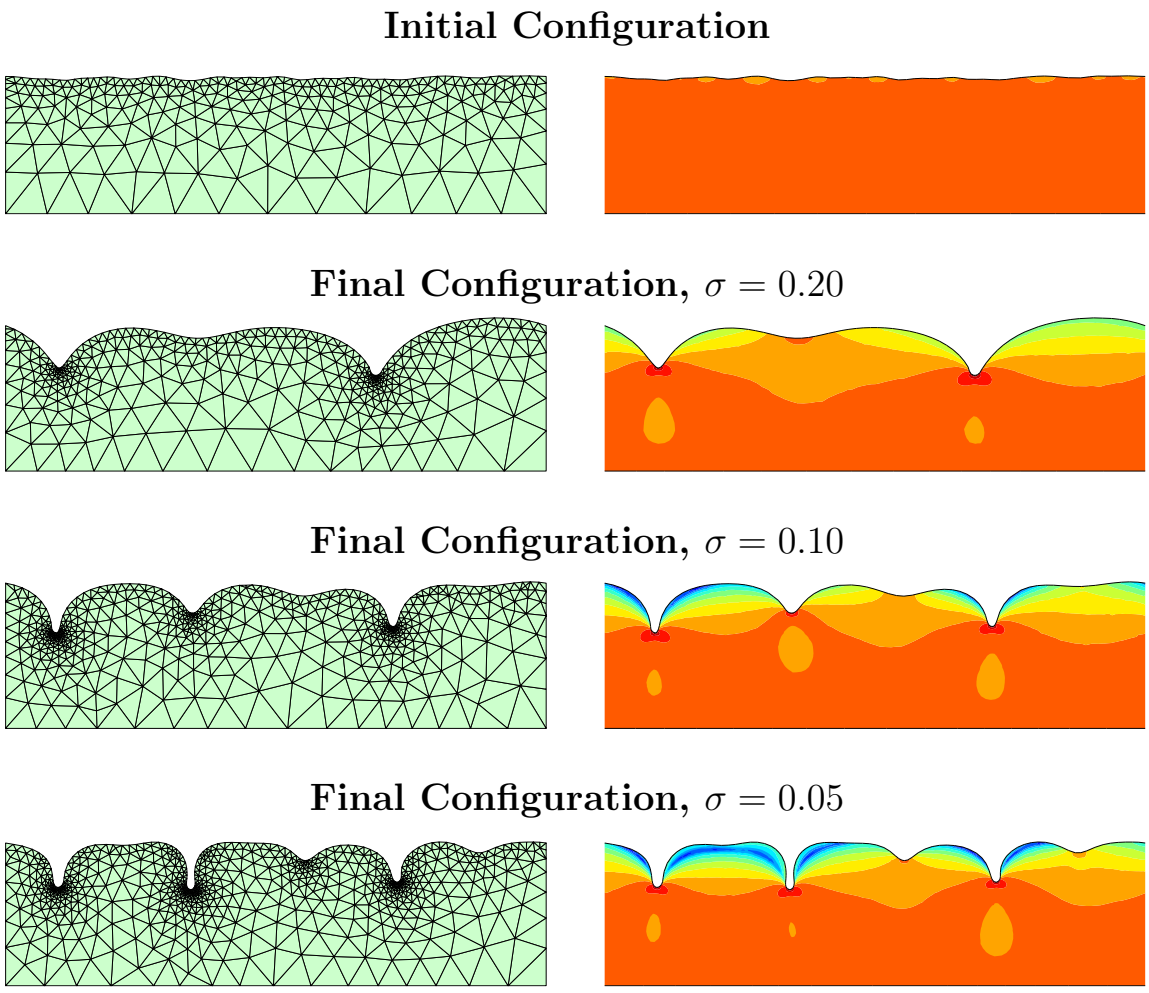

Figure 5-6: Results of the two dimensional simulations of the Stress Driven Rearrangement Instability. The left plots show the meshes and the right plots show the elastic energy densities (logarithmic color scale).

fine meshes and small time steps, and the purpose here is to demonstrate the mesh generation. More sophisticated techniques include L_2-projections or the Arbitrary Lagrangian-Eulerian (ALE) method [63].

The viscous term $-\nu\nabla^2\boldsymbol{u}$ is handled by an implicit time integration using backward Euler. Finally, the incompressibility is enforced using Chorin's projection method [17]. At the end of each time step we project the velocities onto a divergence free space by solving for an "artificial pressure" in the Pressure Poisson Equation (PPE) $\nabla^2 P = -\nabla \cdot (\boldsymbol{u} \cdot \nabla \boldsymbol{u})$, and subtracting its gradient from the velocities. This simple scheme is only first order accurate in space, but higher methods are available. We use a discrete form of the projection method with nearly consistent mass matrix, see [29].

In our example, the domain is a square with a rotating object inside. The rotation angle $\theta(t)$ is a given function of time, and our boundary conditions are that $\boldsymbol{u} = \boldsymbol{0}$ at the outer boundaries, and $\boldsymbol{u} = \boldsymbol{u}_{\text{object}}$ at the moving object boundaries, where

$$\boldsymbol{u}_{\text{object}}(x, y, t) = (-y\theta'(t),\ x\theta'(t)). \tag{5.18}$$

We integrate in time until the smallest element quality is below a threshold, when we improve the mesh and interpolate the velocities. There are better ways to do this, for example by only updating the nodes close to elements of poor quality in order to minimize the numerical diffusion due to interpolation.

The resulting meshes during the first time steps are shown in Figure 5-7, both before and after the mesh improvements. This is an example where it is important to maintain high element qualities, since we then can take more time steps before having to retriangulate.

5.2 Meshing for Numerical Adaptation

An adaptive finite element solver starts from an initial mesh, solves the physical problem, and estimates the error in each mesh element. From this a new mesh size

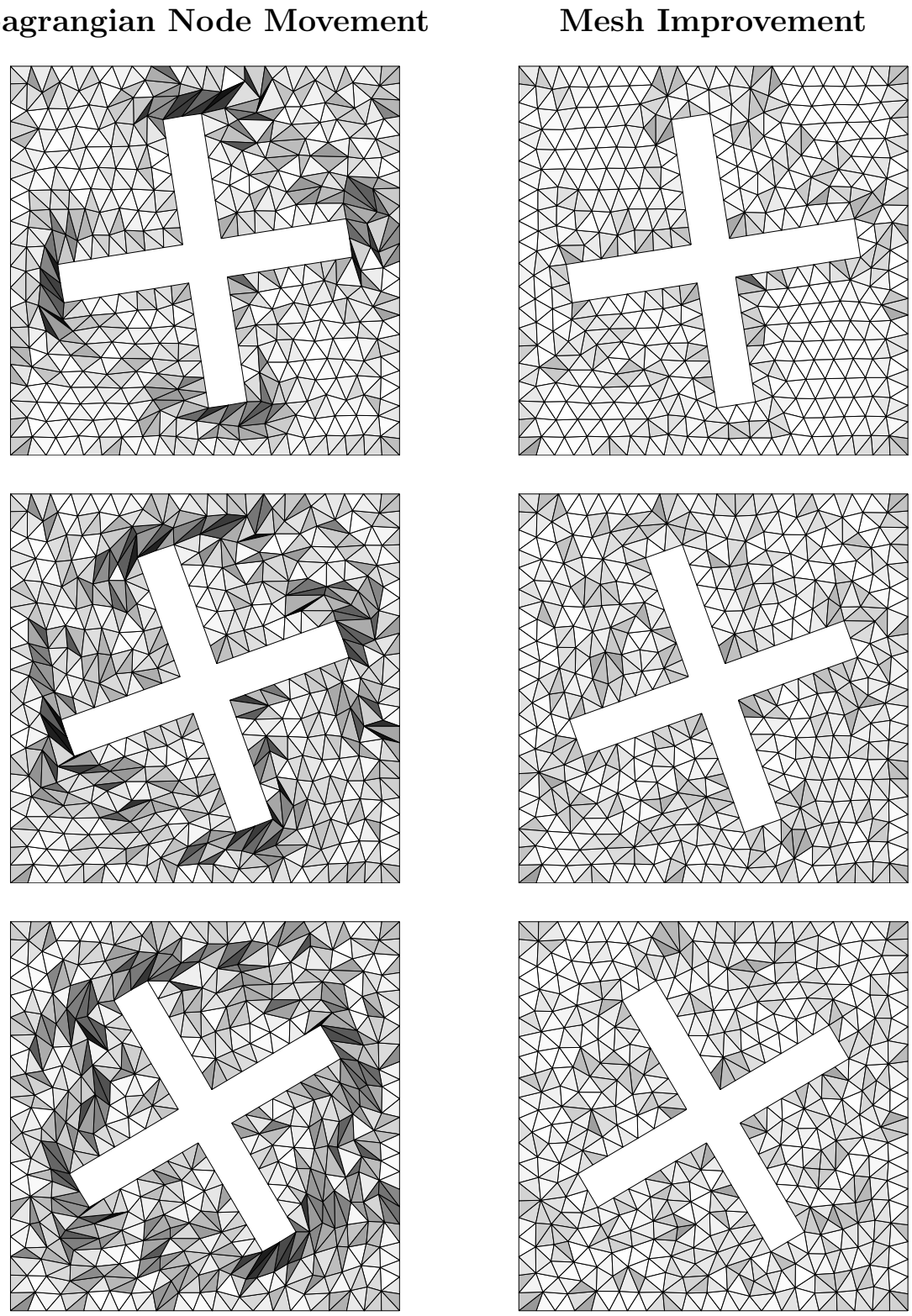

Figure 5-7: Solving the Navier-Stokes equation on a domain with moving boundaries. The nodes are moved with the fluid velocities until the element qualities (darkness of the triangles) are too low, when we retriangulate using our force equilibrium.

function can be derived, for example by equidistributing the error across the domain. The challenge for the mesh generator is to create a new high-quality mesh, conforming to the size function and other geometrical constraints.

One approach is to refine the existing mesh, by splitting elements that are too large, and possibly also coarsening small elements. These local refinement techniques are efficient, robust, and provide simple solution transfer between the meshes. The refinement can be made in a way that completely avoids bad elements, but the average qualities usually drop during the process.

An alternative is to remesh the domain by generating a new mesh from scratch based on the desired size function. This technique has been considered expensive, but it can produce meshes of very high quality if the size function is well-behaved. However, the size functions arising from adaptive solvers may have large gradients and they have to modified before the refinement, at least for mesh generators that rely on good size functions.

We propose to use our new techniques for mesh size functions and mesh generation for the remeshing. The mesh size function from the adaptive solver is gradient limited, by solving (4.8) on the mesh from the previous iteration. We then apply a simple density control scheme, without considering the resulting element quality. Finally, the qualities are improved by iterating toward a force equilibrium.

To illustrate the technique, we solve Poisson's equation with a delta source and estimate the error in the energy norm [22], see Figure 5-8. The initial mesh and the gradient limited size function are shown in left plot. Next, we apply a density control by splitting and collapsing edges (center plot). Note that this mesh does not have to be of high quality, or have good connectivity, so any simple scheme can be used. Finally we solve for force equilibrium (right plot).

We now show two examples of numerical adaptation and remeshing using our methods. Our first example solves a simple convective model problem on a square geometry:

$$\boldsymbol{v} = [1, -2\pi A \cos 2\pi x] \tag{5.19}$$

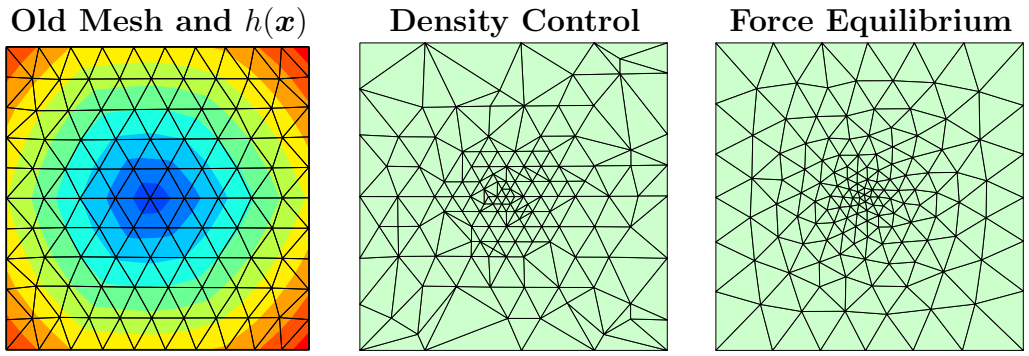

Figure 5-8: The steps of the remeshing algorithm. First, a gradient limited size function $h(\boldsymbol{x})$ is generated by solving (4.10) on the old mesh. Next, the node density is controlled by edge splitting and merging. Finally, we solve for a force equilibrium in the edges using forward Euler iterations.

(with $A = 0.3$) and solve for u in the following advection problem:

$$\boldsymbol{v} \cdot \nabla u(x, y) = 0, \qquad (x, y) \in (-1, 1) \times (-1, 1) \tag{5.20}$$

with boundary conditions

$$u(x, -1) = 0, \tag{5.21}$$

$$u(x, 1) = 1, \tag{5.22}$$

$$u(-1, y) = \begin{cases} 1, & \text{if } y \geq 0 \\ 0, & \text{if } y < 0. \end{cases} \tag{5.23}$$

The exact solution to this problem has a jump:

$$u(x, y) = \begin{cases} 1, & \text{if } y \geq A \sin 2\pi x \\ 0, & \text{if } y < A \sin 2\pi x. \end{cases} \tag{5.24}$$

We discretize (5.20)-(5.23) using piecewise linear finite elements with streamline-diffusion stabilization. To obtain an accurate numerical solution, the discontinuity along $y = A \sin 2\pi x$ has to be resolved. We do this using numerical adaptation in the L_2-norm, see [22]. The size function from the adaptive scheme is highly irregular,

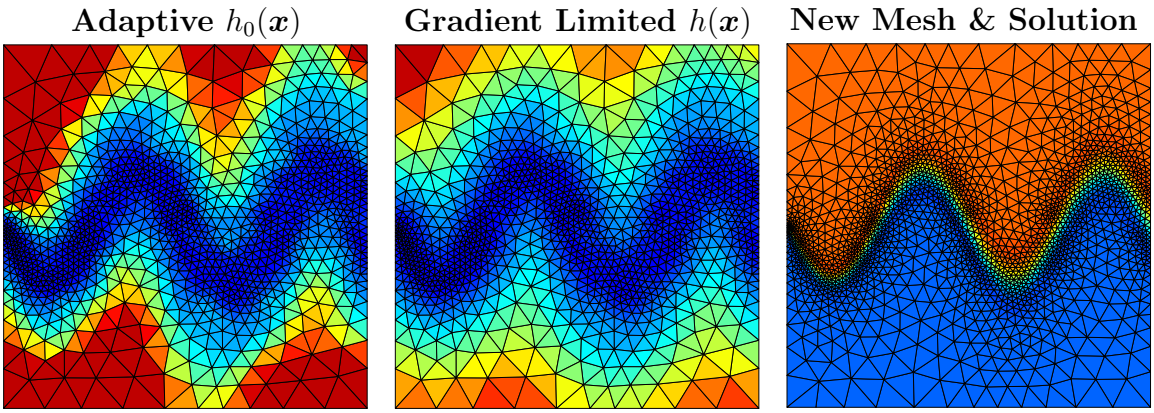

Figure 5-9: An example of numerical adaptation for solution of (5.20)-(5.23).

and in particular it specifies large variations in element sizes which would give low-quality triangles. After gradient limiting the mesh size function is well-behaved and a high-quality mesh can be generated (in the right plot of Figure 5-9).

Our next example is a compressible flow simulation over a bump at Mach 0.95. We solve the Euler equations using the NSC2KE solver [40], and use a simple adaptive scheme based on second-derivatives of the density [48] to determine new size functions. These resolve the shock accurately but the sizes increase sharply away from the shock. With gradient limiting a high quality mesh is generated.

5.3 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 [28], but these typically work on 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

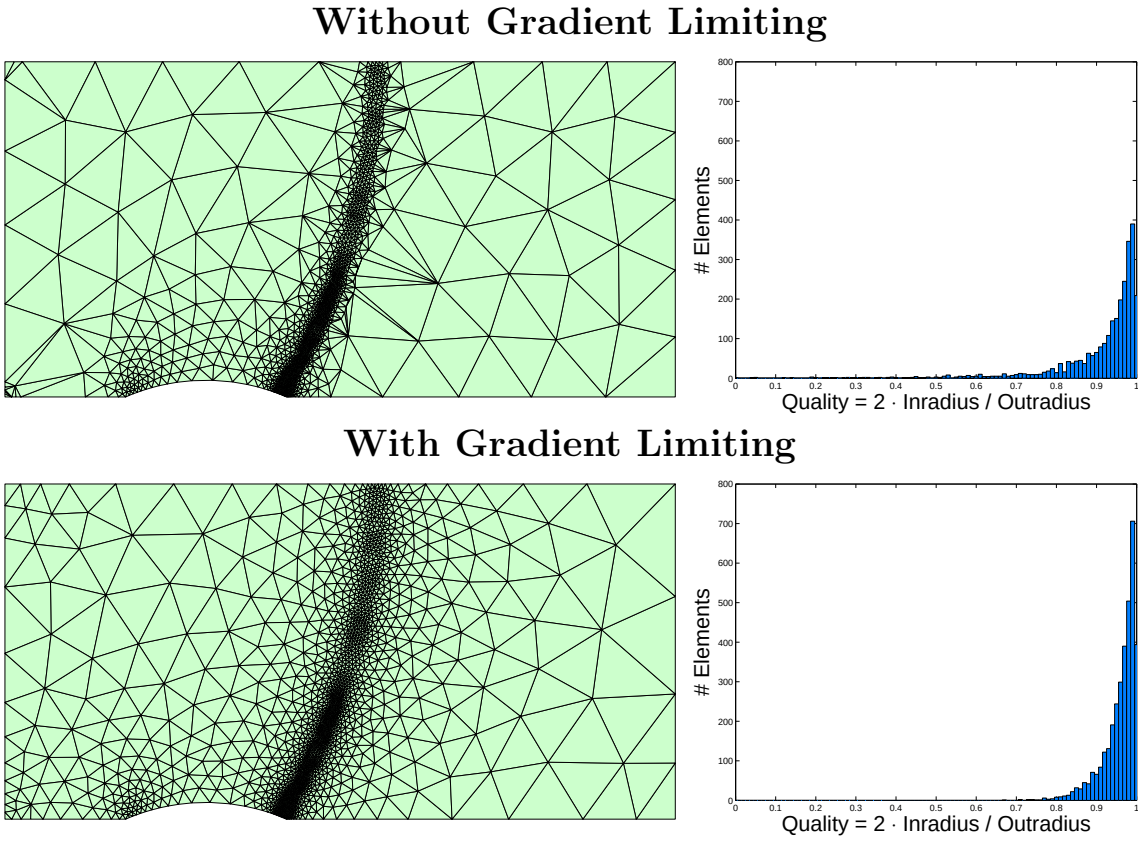

Figure 5-10: Numerical adaptation for compressible flow over a bump at Mach 0.95. The second-derivative based error estimator resolves the shock accurately, but gradient limiting is required to generate a new mesh of high quality.

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 [14], 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 create the signed distance function for the domain by the following steps:

1. Smooth the image with a low-pass filter, for example a simple averaging filter. This gives a band of a few pixels where the image is smooth, and in particular our implicit boundary $A(\boldsymbol{x}) = 0.5$ is smooth.

2. For pixels close to the boundary, compute central difference approximations of

the derivatives and find the approximate signed distances using the projections in Section 3.2.

3. Use the fast marching method to obtain the distance function for the entire domain.

Our first example is a picture of a few objects taken with a standard digital camera, see Figure 5-11. We isolate the objects using the segmentation feature of a 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, and gradient limiting. For the skeletonization we increase κ_{tol} to compensate for the slightly noisy distance function close to the boundary.

Next we show how to mesh geographical areas in a satellite image. In Figure 5-12, we use the same techniques as above to generate a mesh of Lake Superior[1].

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.

The meshes in Figure 5-13 are created from a CT scan of the iliac bone.[2] The top mesh is a uniform surface mesh, and the bottom mesh is a full tetrahedral mesh with graded element sizes.

[1]Image courtesy of MODIS Rapid Response Project at NASA/GSFC.

[2]The image datasets used in this experiment were from the Laboratory of Human Anatomy and Embryology, University of Brussels (ULB), Belgium.

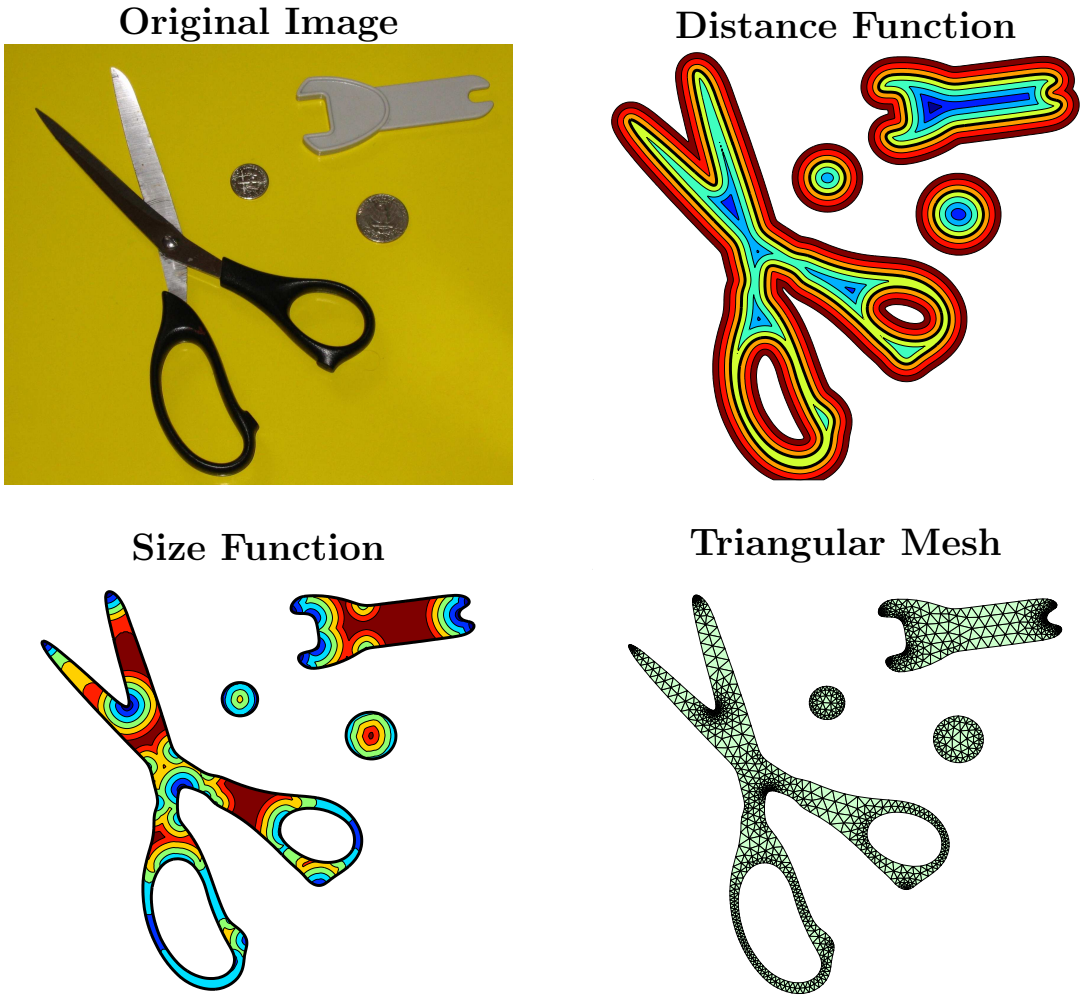

Figure 5-11: Meshing objects in an image. The segmentation is done with an image manipulation program, the distance function is computed by smoothing and approximate projections, and the size function uses the curvature, the feature size, and gradient limiting.

Original Image

Triangular Mesh

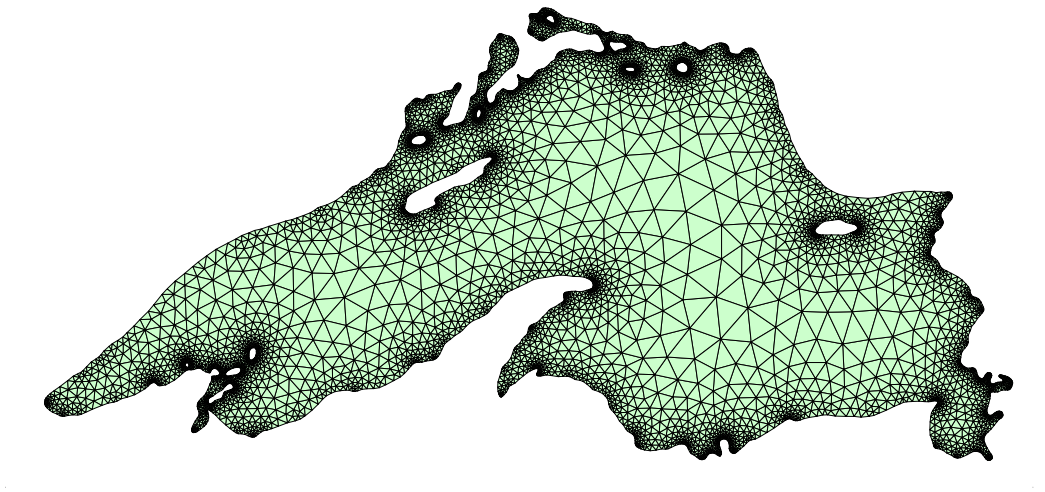

Figure 5-12: Meshing a satellite image of Lake Superior.

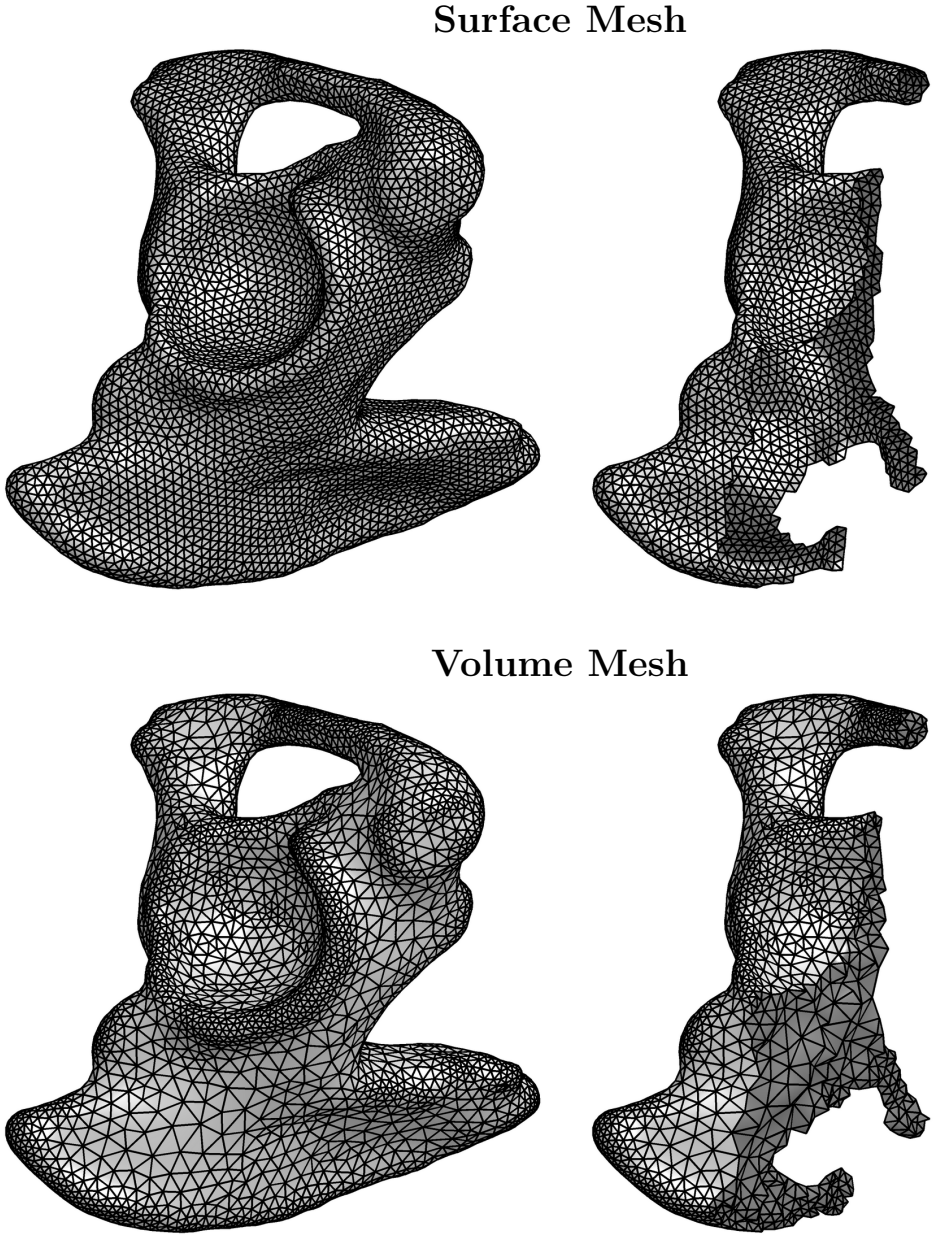

Figure 5-13: Meshing the iliac bone. The top plots show a uniform surface mesh, and the bottom plots show a tetrahedral volume mesh, created with an automatically computed mesh size function.

Chapter 6

Conclusions and Future Work

We have presented techniques for mesh generation using implicit functions. Many extensions are possible, but the main idea of iterative node movements and boundary projections appears to be successful for generating high-quality meshes. The simplicity is an important factor, and we believe that our short MATLAB code will help users understand mesh generation and integrate it with their own codes. The accurate size functions we compute using the medial axis and the gradient limiting are essential for achieving highest possible mesh qualities. They will likely also improve other mesh generators that rely on a-priori size functions.

We have demonstrated how many applications can benefit from our mesh generator. In particular, it is appropriate for problems that require frequent remeshing such as moving boundary problems and numerical adaptation. It appears that implicit geometries are becoming increasingly popular, for example in level set methods, computer graphics, and image processing. We hope that many of these applications will find advantages with our techniques.

During our work we have identified many possibilities for future work, and we list some of these ideas below.

Space-time meshes A space-time mesh is a mesh of the higher-dimensional space consisting of both space and time, for example a tetrahedral mesh for two space dimensions and time. Using these, efficient numerical methods can be created,

in particular for moving boundary problems. It might be possible to create these higher dimensional elements directly during our iterative node movements and connectivity updates.

Sliver removal In all our examples of tetrahedral meshes we have used the Delaunay triangulation, which produces poor elements called slivers. We then try to eliminate these using the standard techniques of face swapping and edge flipping. If we instead use these connectivity updates in the mesh generator, it might produce high quality elements directly.

No update of good elements Usually most of the elements are of high quality already after a few iterations, and a significant speed-up might be possible by excluding these from the updates. Appropriate data structures can be used to find the poor elements and their neighboring nodes, for example a priority queue. Note that this will be particularly useful for moving meshes, since then only the elements close to the moving boundary are deformed. In Section 5.1 we implemented this manually by adding stiffness away from the boundary, but a quality based approach would automatically detect which elements we need to update.

Increased robustness Although our mesh generator usually produces high-quality meshes, there is no guarantee that it will terminate. We have implemented some additional control logic to make it reliable when we create thousands of meshes, such as in our moving mesh applications. We also use other termination criteria based on element qualities. With some additional work in this area the mesh generator might become robust enough to be used as a black-box.

Quadrilateral and hexahedral meshes Perhaps our force equilibrium and connectivity updates can be used to generate high-quality quadrilateral or hexahedral meshes.

More gradient limiting Our gradient limiting equation might have application in other areas, where smoothing is traditionally used but gradient limiting is the

desired operation, for example in signal and image processing. The fast gradient limiting solvers can be extended to unstructured meshes, and the methods described in [35] and [18] should be applicable in a straightforward way. We would also like to extend the gradient limiting equation to anisotropic mesh size functions, and there might be a similar PDE (or a system of PDEs) based on general metrics [10]. Finally, with the PDE based formulation it is possible that error estimators for numerical adaptive solvers can be applied on the discretized solution $h(\boldsymbol{x})$ for adaptive generation of background meshes.

Moving meshes without background grid In our applications we use the level set method on a Cartesian or octree background grid to evolve interfaces. But since we generate an unstructured mesh for the domain $\phi(\boldsymbol{x}) \leq 0$, it might be possible to solve the level set equation on this mesh using unstructured Hamilton-Jacobi solvers [3], and represent the distance function on the previous mesh when generating the new mesh. This would result in a hybrid explicit-implicit approach, since the domain is represented explicitly at each step by the unstructured mesh, but the implicit level set equation gives robust interface propagation, entropy solutions, and automatic handling of topology changes.

No size function For other mesh generators such as the advancing front method it is essential to have a good size function, since the elements are created one at a time starting from the boundaries. But in our iterative approach, we could in principle try to determine good mesh sizes during the iterations, for example based on element qualities. This would eliminate the need for an a-priori mesh size function.

More applications We would like to study other shape optimization problems, for example acoustic, aerodynamic, and photonics applications. In computational fluid dynamics there are several interesting extensions, for example free boundary flows, and using space-time meshes instead of the Lagrangian approach (as described above). Other similar application areas are fluid-structure interaction and contact problems, where the distance function provides a fast way to detect

interfaces in contact. Finally, mesh generation for images is an important topic with many applications, and a complete, user-friendly package for this would be most welcome in the medical imaging community.

Bibliography

[1] Grégoire Allaire, François Jouve, and Anca-Maria Toader. A level-set method for shape optimization. *C. R. Math. Acad. Sci. Paris*, 334(12):1125–1130, 2002.

[2] Nina Amenta, Marshall Bern, and David Eppstein. The crust and the β-skeleton: Combinatorial curve reconstruction. *Graphical Models & Image Processing*, 60/2(2):125–135, March 1998.

[3] Timothy J. Barth and James A. Sethian. Numerical schemes for the Hamilton-Jacobi and level set equations on triangulated domains. *J. Comput. Phys.*, 145(1):1–40, 1998.

[4] Marshall Bern and Paul Plassmann. Mesh generation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science, 1999.

[5] G. Berti. Image-based unstructured 3D mesh generation for medical applications. In *Proceedings of ECCOMAS 2004*, 2004.

[6] Ted D. Blacker and Michael B. Stephenson. Paving: A new approach to automated quadrilateral mesh generation. *Internat. J. Numer. Methods Engrg.*, 32:811–847, 1991.

[7] Jules Bloomenthal. Polygonization of implicit surfaces. *Comput. Aided Geom. Design*, 5(4):341–355, 1988.

[8] H. Blum. Biological shape and visual science (part I). *Journal of Theoretical Biology*, 38:205–287, 1973.

[9] Houman Borouchaki, Paul Louis George, Frédéric Hecht, Patrick Laug, and Eric Saltel. Delaunay mesh generation governed by metric specifications. I. Algorithms. *Finite Elem. Anal. Des.*, 25(1-2):61–83, 1997.

[10] Houman Borouchaki, Frediric Hecht, and Pascal J. Frey. Mesh gradation control. In *Proceedings of the 6th International Meshing Roundtable*, pages 131–141. Sandia Nat. Lab., October 1997.

[11] Frank J. Bossen and Paul S. Heckbert. A pliant method for anisotropic mesh generation. In *Proceedings of the 5th International Meshing Roundtable*, pages 63–76. Sandia Nat. Lab., 1996.

[12] M. Brewer, L. Diachin, P. Knupp, T. Leurent, and D. Melander. The Mesquite mesh quality improvement toolkit. In *Proceedings of the 12th International Meshing Roundtable*, pages 239–259. Sandia Nat. Lab., 2003.

[13] J. R. Cebral and Rainald Lohner. From medical images to CFD meshes. In *Proceedings of the 8th International Meshing Roundtable*, pages 321–331. Sandia Nat. Lab., October 1999.

[14] T. Chan and L. Vese. Active contours without edges. *IEEE Trans. Image Processing*, 10(2):266–277, 2001.

[15] Siu-Wing Cheng, Tamal K. Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. Sliver exudation. *J. ACM*, 47(5):883–904, 2000.

[16] L. Paul Chew. Constrained Delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.

[17] Alexandre Joel Chorin. Numerical solution of the Navier-Stokes equations. *Math. Comp.*, 22:745–762, 1968.

[18] Paul Covello and Garry Rodrigue. A generalized front marching algorithm for the solution of the eikonal equation. *J. Comput. Appl. Math.*, 156(2):371–388, 2003.

[19] Luiz Henrique de Figueiredo, Jonas de Miranda Gomes, Demetri Terzopoulos, and Luiz Velho. Physically-based methods for polygonization of implicit surfaces. In *Proceedings of the conference on Graphics interface '92*, pages 250–257. Morgan Kaufmann Publishers Inc., 1992.

[20] Mathieu Desbrun, Nicolas Tsingos, and Marie-Paule Gascuel. Adaptive sampling of implicit surfaces for interactive modelling and animation. *Computer Graphics Forum*, 15(5):319–325, 1996.

[21] Herbert Edelsbrunner. *Geometry and topology for mesh generation*, volume 7 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 2001.

[22] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational differential equations*. Cambridge University Press, Cambridge, 1996.

[23] David A. Field. Laplacian smoothing and delaunay triangulations. *Comm. in Applied Numerical Methods*, 4:709–712, 1988.

[24] David A. Field. Qualitative measures for initial meshes. *Internat. J. Numer. Methods Engrg.*, 47:887–906, 2000.

[25] Lori A. Freitag and Carl Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *Internat. J. Numer. Methods Engrg.*, 40(21):3979–4002, 1997.

[26] Pascal J. Frey and Loïc Marechal. Fast adaptive quadtree mesh generation. In *Proceedings of the 7th International Meshing Roundtable*, pages 211–224. Sandia Nat. Lab., October 1998.

[27] O. Gloth and R. Vilsmeier. Level Sets as Input for Hybrid Mesh Generation. In *Proceedings of the 9th International Meshing Roundtable*, pages 137–146. Sandia Nat. Lab., October 2000.

[28] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2 edition, 2002.

[29] P. M. Gresho and R. L. Sani. *Incompressible Flow and the Finite Element Method*. John Wiley and Sons, 2000.

[30] M. Grinfeld. 2-dimensional islanding atop stressed solid helium and epitaxial-films. *Physical Review B*, 49(12):8310–8319, 1994.

[31] M. Grinfeld, P. Grinfeld, H. Kojima, J. Little, R. Masutomi, P.-O. Persson, and T. Zheleva. The stress driven rearrangement instabilities in electronic materials and in helium crystals. In *MRS Proceedings*, November 2004.

[32] Dave Hale. Atomic images - a method for meshing digital images. In *Proceedings of the 10th International Meshing Roundtable*, pages 185–196. Sandia Nat. Lab., October 2001.

[33] Ami Harten, Bjorn Engquist, Stanley Osher, and Sukumar R. Chakravarthy. Uniformly high order accurate essentially non-oscillatory schemes. *J. Comput. Phys.*, 71(2):231–303, 1987.

[34] Eberhard Hopf. Generalized solutions of non-linear equations of first order. *J. Math. Mech.*, 14:951–973, 1965.

[35] R. Kimmel and James A. Sethian. Fast marching methods on triangulated domains. In *Proceedings of the National Academy of Sciences*, volume 95, pages 8341–8435, 1998.

[36] Ron Kimmel, Doron Shaked, Nahum Kiryati, and Alfred M. Bruckstein. Skeletonization via distance maps and level sets. *Computer Vision and Image Understanding: CVIU*, 62(3):382–391, 1995.

[37] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th*

annual conference on Computer graphics and interactive techniques, volume 21:4, pages 163–169, 1987.

[38] Sean Patrick Mauch. *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, Caltech, 2003.

[39] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. On the radius-edge condition in the control volume method. *SIAM J. Numer. Anal.*, 36(6):1690–1708, 1999.

[40] B. Mohammadi. Fluid dynamics computation with NSC2KE - a user guide, release 1.0. Technical Report 0164, INRIA, 1994.

[41] N. Molino, R. Bridson, J. Teran, and R. Fedkiw. A Crystalline, Red Green Strategy for Meshing Highly Deformable Objects with Tetrahedra. In *Proceedings of the 12th International Meshing Roundtable*, pages 103–114. Sandia Nat. Lab., October 2003.

[42] F. Murat and S. Simon. Etudes de problèmes d'optimal design. In *Lecture Notes in Computer Science*, volume 41, pages 54–62. Springer Verlag, Berlin, 1976.

[43] Stanley Osher and Ronald Fedkiw. *Level set methods and dynamic implicit surfaces*, volume 153 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2003.

[44] Stanley Osher and Fadil Santosa. Level set methods for optimization problems involving geometry and constraints. I. Frequencies of a two-density inhomogeneous drum. *J. Comput. Phys.*, 171(1):272–288, 2001.

[45] Stanley Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *J. Comput. Phys.*, 79(1):12–49, 1988.

[46] Steven J. Owen. A survey of unstructured mesh generation technology. In *Proceedings of the 7th International Meshing Roundtable*, pages 239–267. Sandia Nat. Lab., 1998.

[47] Steven J. Owen and Sunil Saigal. Surface mesh sizing control. *Internat. J. Numer. Methods Engrg.*, 47(1-3):497–511, 2000.

[48] J. Peraire, M. Vahdati, K. Morgan, and O. C. Zienkiewicz. Adaptive remeshing for compressible flow computations. *J. Comput. Phys.*, 72(2):449–466, 1987.

[49] Olivier Pironneau. *Optimal shape design for elliptic systems*. Springer Series in Computational Physics. Springer-Verlag, New York, 1984.

[50] W. Quadros, S. Owen, M. Brewer, and K. Shimada. Finite element mesh sizing for surfaces using skeleton. In *Proceedings of the 13th International Meshing Roundtable*, pages 389–400. Sandia Nat. Lab., September 2004.

[51] María-Cecilia Rivara. Mesh refinement processes based on the generalized bisection of simplices. *SIAM J. Numer. Anal.*, 21(3):604–613, 1984.

[52] Martin Rumpf and Alexandru Telea. A continuous skeletonization method based on level sets. In *Proceedings of the symposium on Data Visualisation 2002*, pages 151–ff. Eurographics Association, 2002.

[53] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[54] T. W. Sederberg. *Implicit and Parametric Curves and Surfaces for Computer Aided Geometric Design*. PhD thesis, Purdue University, August 1983.

[55] James A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. Nat. Acad. Sci. U.S.A.*, 93(4):1591–1595, 1996.

[56] James A. Sethian. *Level set methods and fast marching methods*, volume 3 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, second edition, 1999.

[57] James A. Sethian and Andreas Wiegmann. Structural boundary design via level set and immersed interface methods. *J. Comput. Phys.*, 163(2):489–528, 2000.

[58] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1996.

[59] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Comput. Geom.*, 22(1-3):21–74, 2002.

[60] Jonathan Richard Shewchuk. What is a good linear element? Interpolation, conditioning, and quality measures. In *Proceedings of the 11th International Meshing Roundtable*, pages 115–126. Sandia Nat. Lab., 2002.

[61] Kenji Shimada and David C. Gossard. Bubble mesh: automated triangular meshing of non-manifold geometry by sphere packing. In *SMA '95: Proceedings of the Third Symposium on Solid Modeling and Applications*, pages 409–419, 1995.

[62] Kaleem Siddiqi, Sylvain Bouix, Allen Tannenbaum, and Staven Zucker. The hamilton-jacobi skeleton. In *International Conference on Computer Vision (ICCV)*, pages 828–834, 1999.

[63] Erwin Stein, René de Borst, and Thomas J.R. Hughes, editors. *Encyclopedia of Computational Mechanics*, volume Volume 1: Fundamentals, chapter 14, Arbitrary Lagrangian-Eulerian Methods. John-Wiley and Sons, 2004.

[64] Mark Sussman, Peter Smereka, and Stanley Osher. A level set approach for computing solutions to incompressible two-phase flow. *J. Comput. Phys.*, 114(1):146–159, 1994.

[65] Richard Szeliski and David Tonnesen. Surface modeling with oriented particle systems. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 185–194. ACM Press, 1992.

[66] John N. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Trans. Automat. Control*, 40(9):1528–1538, 1995.

[67] Michael Yu Wang, Xiaoming Wang, and Dongming Guo. A level set method for structural topology optimization. *Comput. Methods Appl. Mech. Engrg.*, 192(1-2):227–246, 2003.

[68] Alan M. Winslow. Numerical solution of the quasilinear Poisson equation in a nonuniform triangle mesh. *J. Comput. Phys.*, 1:149–172, 1967.

[69] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277. ACM Press, 1994.

[70] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[71] M. A. Yerry and M. S. Shephard. A modified quadtree approach to finite element mesh generation. *IEEE Comp. Graph. Appl.*, 3(1):39–46, 1983.

[72] Jin Zhu. A new type of size function respecting premeshed entities. In *Proceedings of the 11th International Meshing Roundtable*, pages 403–413. Sandia Nat. Lab., September 2003.

[73] Jin Zhu, Ted Blacker, and Rich Smith. Background overlay grid size functions. In *Proceedings of the 11th International Meshing Roundtable*, pages 65–74. Sandia Nat. Lab., September 2002.