Mesh generation with a signed distance function

By

Jaemin Shin

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

Master of Science

in

Mathematics

in the

OFFICE OF GRADUATE STUDIES

of the

KOREA UNIVERSITY

Approved:

Committee Member 1

Committee Member 2

Committee Member 3

Committee in Charge

2011

Contents

Abstract

The primary purpose of this thesis is to have the beginner encourage to understand for elementary step and easy access of the mesh generation, in particular, "Distmesh" algorithm [26]. Forward the explanation of the algorithm, we should construct the distance function, because the "Distmesh" algorithm is based on using the distance from the interface. So we present the methods for making the distance function.

The reinitialization equation make the some initial data for the signed distance function. Note that the initial data and evolved data of reinitialization equation have an identical interface. Although the "Distmesh" algorithm need not the distance function when the implicit interface is sufficiently smooth, we recommend that use the signed distance function. We show a simplified version of the reinitialization method in MATLAB code.

We present techniques for generation of unstructured meshes for geometries specified by a singed distance function. An initial mesh is iteratively improved by solving for a force equilibrium in the element edges, and the boundary nodes are projected to the given interface. The algorithm generalized 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.

Key work : Mesh generation, Distmesh algorithm, Reinitialization.

Acknowledgments

I would like to begin by expressing my thanks to my advisor Junseok Kim. His encouragement and unfailing support have been most valuable to me during these years, and I feel deeply privileged for the opportunity to work in the research group. I also gratefully acknowledge Prof. Woongae Hwang and Prof. Hongjoong Kim of my thesis committee. And I also appreciate all the suggestions and comments from colleagues and friends. Finally, my deepest gratitude goes to my family for supporting and for encouraging and believing in me.

Chapter 1

Introduction

In the numerical simulation, in particular, finite element method (FEM) or finite volume method (FVM), mesh generation is fundamental requirement. These methods have a merit that is natural on the curved shaped domain and on an arbitrary domain. However, it take lots of time to study mesh generation. Mesh generator tend to be complex codes that are nearly inaccessible. They are often just used as "Black boxes". Combining the meshing software and other codes is a tough work for beginner in the mesh generation, so many people suffer hardship and some gives up to progress. To many people, the mesh generation may not be the first priority. We believe that the ability to understand and adapt a mesh generation code is too valuable and potion to lose.

An essential decision is how to represent the geometry about the shape of the region. A signed distance function $d(x, y)$ used the basement of the algorithm for generating the mesh. A signed distance function d is used to represent the shape of the domain. In Chapter 4 we express the reinitialization equation for constructing the signed distance function d. This equation originally developed by the motion in the normal direction. So we first represent the evolution by the motion in an externally generated velocity field in Chapter 3. And we present the implement for constructing the signed distance function.

Chapter 2

Implicit Function and Signed Distance Function

In this chapter we introduce the notion of an implicit function and a signed distance function with a Euclidean metric. Actually the signed distance function is also a sort of the implicit function including the information about the interface position. In these functions, "+" and "−" signs are used to indicate the outside and inside of the surface, respectively. The most important advantage of the signed distance function is that the values at any point mean the smallest distance to the interface. So we explain the fundamental things of the implicit function, then study the signed distance function. Finally we illustrate a number of useful properties, focusing on those that will be of use to us later.

2.1. Implicit Function

First of all, we introduce the implicity surfaces. A good general review can be found in [1]. To help clarify definitions, we discuss for each spatial dimensional. In general, for $\vec{x} \in R^n$, we define the interface as $\partial\Omega = \{\vec{x} \mid \phi(\vec{x}) = 0\}$. And we refer to $\Omega^- = \{\vec{x} \mid \phi(\vec{x}) < 0\}$ as the inside region of the domain and $\Omega^+ = \{\vec{x} \mid \phi(\vec{x}) > 0\}$ as the outside region of the domain.

In general, in R^n, sub-domains are n-dimensional, while the interface has dimension $n-1$. We say that the interface has codimension one. And the implicit function $\phi(\vec{x})$ is defined on all $\vec{x} \in R^n$, and its iso-contour has dimension $n-1$.

For any function $\hat{\phi}(\vec{x})$ and an arbitrary iso-contour $\hat{\phi}(\vec{x}) = a$ for some scalar $a \in R$, we can define $\phi(\vec{x}) = \hat{\phi}(\vec{x}) - a$, so that the $\phi(\vec{x}) = 0$ iso-contour of ϕ is identical to the $\hat{\phi}(\vec{x}) = a$ iso-contour of $\hat{\phi}$. In addition, the functions ϕ and $\hat{\phi}$ have

identical properties up to a scalar translation a. Moveover, the partial derivatives of ϕ are the same as the partial derivatives of $\hat{\phi}$, since the scalar vanishes upon differentiation. Thus, throughout the text all of our implicit functions $\phi(\vec{x})$ will be defined so that the $\phi(\vec{x}) = 0$ iso-contour represents the interface.

2.1.1. Points. In one spatial dimension, suppose we divide the real line into three distinct pieces using the points $x = -1$ and $x = 1$. That is, we define $(-\infty, -1)$, $(-1, 1)$, and $(1, \infty)$ as three separate sub-domains of interest. We refer to $\Omega^- = (-1, 1)$ as the inside part of the domain and $\Omega^+ = (-\infty, -1) \bigcup (1, \infty)$ as the outside part of the domain. The border between the inside and the outside consists of the two points $\partial\Omega = \{-1, 1\}$ and it is called the interface. In this case, the inside and outside regions are one-dimensional space, while the interface is zero-dimensional.

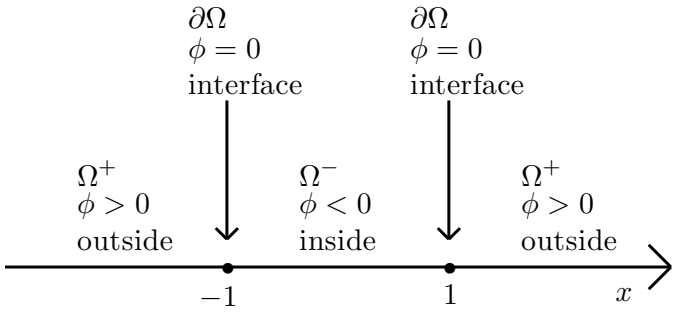


FIGURE 2.1. Implicit representation of the points -1 and 1.

In an explicit interface representation, one explicitly writes down the points that belong to the interface as we did above when defining $\partial\Omega = \{-1, 1\}$. Alternatively, an implicit interface representation defines the interface as the iso-contour of some function. For example, the zero iso-contour of $\phi(x) = x^2 - 1$ is the set of all points where $\phi(x) = 0$; i.e., it is exactly $\partial\Omega = \{-1, 1\}$. This is shown in Figure 2.2.

Consider the functions $\phi(x) = |x| - 1$ and $\phi(x) = x^4 - 1$. These are also implicit functions which have the same interface $\partial\Omega = \{-1, 1\}$, see Figure 2.3. So, there are a lot of implicit function which has identical interface.

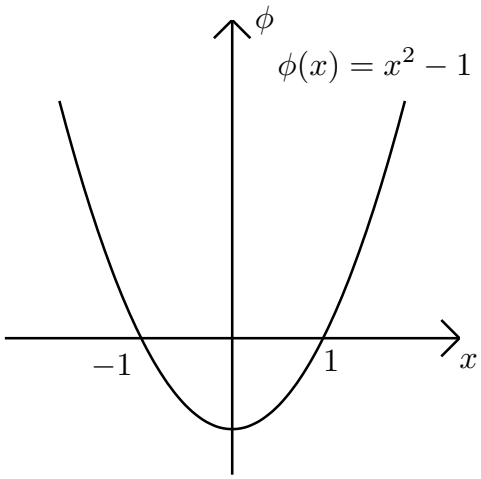


FIGURE 2.2. Implicit function $\phi(x) = x^2 - 1$.

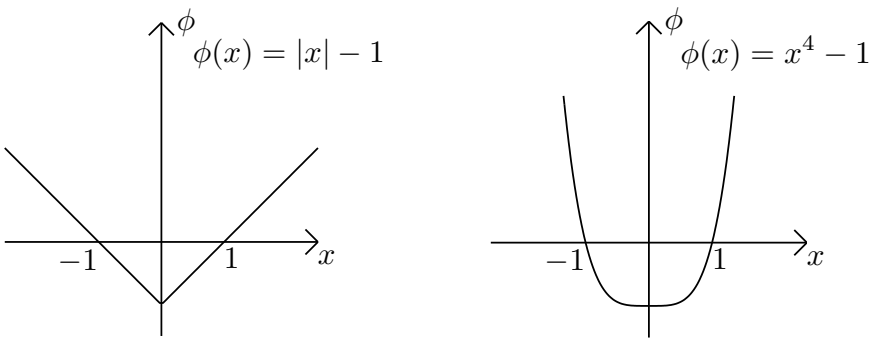


FIGURE 2.3. Two different implicit function having the interface $\partial\Omega = \{-1, 1\}$.

2.1.2. Curves. In two spatial dimensions, the lower dimensional interface is a curve that separates R^2 into sub-domains with nonzero areas. As an example, consider the spacial curve $x^2 + y^2 = 1$ on plane R^2. One explicitly write down the points the interface as $\partial\Omega = \{(x, y)|\ x^2 + y^2 = 1\}$. An implicit interface representation defines the interface as the iso-contour of some function. For example, the zero iso-contour of a paraboloid $\phi(x, y) = x^2 + y^2 - 1$ is the set of all points where $\phi(x, y) = 0$, which is exactly identical, i.e., $\partial\Omega = \{(x, y)|\ \phi(x, y) = 0\}$. In this case,

the interior region is the unit open disk $\Omega^- = \{(x, y)|\ x^2 + y^2 < 1\}$, and the exterior region is $\Omega^+ = \{(x, y)|\ x^2 + y^2 > 1\}$. These regions are depicted in Figure 2.4.

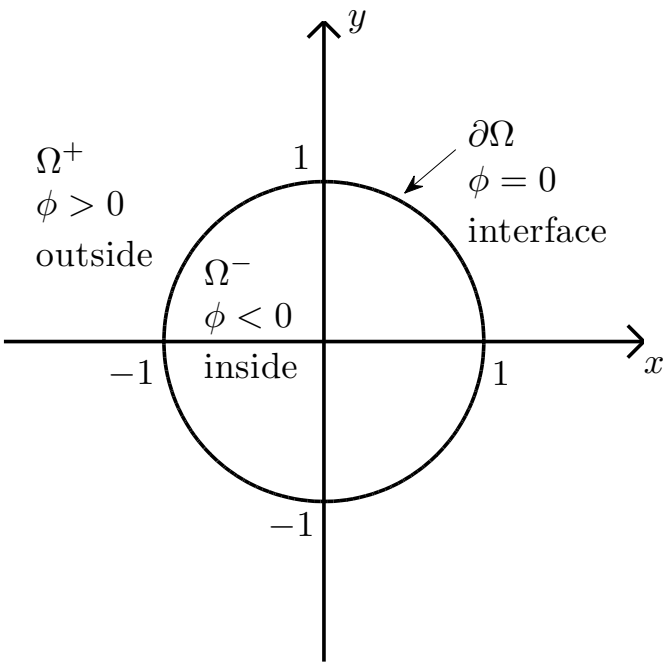


FIGURE 2.4. Implicit representation of the curve $x^2 + y^2 = 1$.

2.1.3. Surfaces. In three spatial dimensions the lower-dimensional interface is a surface that separates R^3 into separate sub-domains with nonzero volumes. As an example, consider $\phi(\vec{x}) = x^2 + y^2 + z^2 - 1$, where the interface is defined by the $\phi(x, y, z) = 0$ iso-contour, which is the boundary of the unit sphere defined as $\partial\Omega = \{(x, y, z) \mid x^2 + y^2 + z^2 = 1\}$. The interior region is the open unit sphere $\Omega^- = \{(x, y, z) \mid x^2 + y^2 + z^2 < 1\}$, and the exterior region is $\Omega^+ = \{(x, y, z) \mid x^2 + y^2 + z^2 > 1\}$. The explicit representation of the interface is $\partial\Omega = \{(x, y, z) \mid x^2 + y^2 + z^2 = 1\}$.

2.2. Signed Distance Function

Until now, we defined implicit functions with $\phi(\vec{x}) < 0$ in the interior region Ω^-, $\phi(\vec{x}) > 0$ in the exterior region Ω^+, and $\phi(\vec{x}) = 0$ on the boundary $\partial\Omega$. From

now on, we discuss signed distance functions. The signed distance functions are also defined to be positive on the exterior region, negative on the interior region, and zero on the boundary.

2.2.1. Distance Function. A distance function $d(\vec{x})$ is defined as

$$d(\vec{x}) = \min_{\vec{x}_I \in \partial\Omega} (|\vec{x} - \vec{x}_I|).$$

Geometrically, d may be constructed as follows. If $\vec{x} \in \partial\Omega$, then $d(\vec{x}) = 0$. Otherwise, for a given point $\vec{x}$, find the point on the boundary set $\partial\Omega$ closest to $\vec{x}$, and label this point $\vec{x}_C$. Then $d(\vec{x}) = |\vec{x} - \vec{x}_C|$.

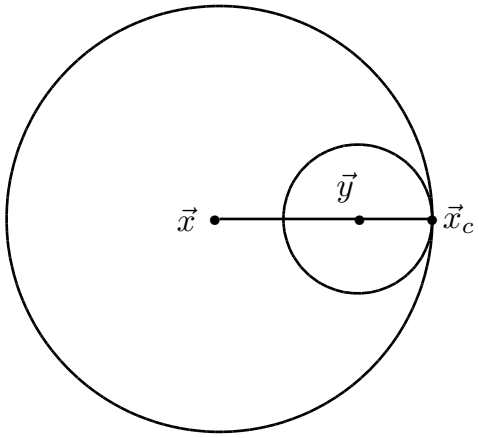


FIGURE 2.5

For a given point $\vec{x}$, supposed that $\vec{x}_C$ is the point on the interface closest to $\vec{x}$. Then for every point $\vec{y}$ on the line segment from $\vec{x}$ to $\vec{x}_C$, $d(\vec{y}) = |\vec{y} - \vec{x}_C|$. To see this, consider Figure 2.5, where $\vec{x}$, $\vec{x}_C$, and an example of a $\vec{y}$ are shown. Since $\vec{x}_C$ is the closest interface point to $\vec{x}$, no other interface points can be inside the large circle drawn about $\vec{x}$ passing through $\vec{x}_C$. Points closer to $\vec{y}$ than $\vec{x}_C$ must reside inside the small circle drawn about $\vec{y}$ passing through $\vec{x}_C$. Since the small circle lies inside the larger circle, no interface points can be inside the smaller circle, and thus $\vec{x}_C$ is the interface point closest to $\vec{y}$.

The line segment from $\vec{x}$ to $\vec{x}_C$ is the shortest path from $\vec{x}$ to the interface. In other words, the path from $\vec{x}$ to $\vec{x}_C$ is the path of steepest descent for the function d. Evaluating $-\nabla d$ at any point on the line segment from $\vec{x}$ to $\vec{x}_C$ gives a vector that points from $\vec{x}$ to $\vec{x}_C$. Furthermore, since d is Euclidean distance,

$$|\nabla d| = 1 \tag{2.1}$$

which is intuitive in the sense that moving twice as close to the interface gives a value of d that is half as big. The above argument leading to equation 2.1 is true for any $\vec{x}$ as long as there is a unique closest point $\vec{x}_C$.

2.2.2. Signed Distance Functions. A signed distance function is an implicit function ϕ with $|\phi(\vec{x})| = d(\vec{x})$ for all $\vec{x}$. Thus, $\phi(\vec{x}) = d(\vec{x}) = 0$ for all $\vec{x} \in \partial\Omega$, $\phi(\vec{x}) = -d(\vec{x})$ for all $\vec{x} \in \Omega^-$ and $\phi(\vec{x}) = d(\vec{x})$ for all $\vec{x} \in \Omega^+$. Signed distance functions share all the properties of implicit functions discussed in the last chapter. In addition, there is new property

$$|\nabla \phi| = 1 \tag{2.2}$$

from equation 2.1. Once again, equation 2.2 is true only in a general sense. It is not true for points that are equidistant from at least two points on the interface. Distance functions have a kink at the interface where $d = 0$ is a local minimum, causing problems in approximating derivatives are monotonic across the interface and can be differentiated there with significantly higher confidence.

Given a point $\vec{x}$, and using the fact that $\phi(\vec{x})$ is the signed distance to the closest point on the interface, we can write

$$\vec{x}_C = \vec{x} - \phi(\vec{x})\overrightarrow{N} \tag{2.3}$$

to calculate the closet point on the interface, where $\overrightarrow{N}$ is the local unit normal at $\vec{x}$. Again, this is true only in a general sense, since equidistant points $\vec{x}$ have more than one closest point $\vec{x}_C$.

2.2.3. Examples. In the last chapter we used $\phi(x) = x^2 - 1$ as an implicit representation of an explicit boundary $\partial\Omega = \{-1, 1\}$. A signed distance function representation of these same boundary points is $\phi(x) = |x| - 1$, as shown in Figure 2.6. The signed distance function $\phi(x) = |x| - 1$, gives the same boundary $\partial\Omega = \{-1, 1\}$, interior region Ω^-, and exterior region Ω^+, that the implicit function $\phi(x) = x^2 - 1$ did. However, the signed distance function $\phi(x) = |x| - 1$ has $|\nabla\phi| = 1$ for all $x \in R$ without $x = 0$.

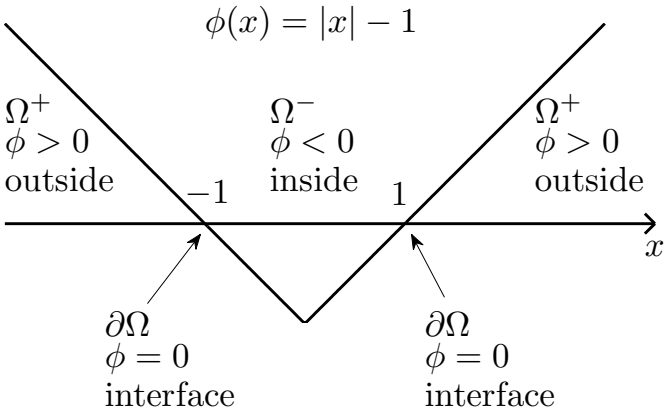


FIGURE 2.6. Implicit representation of a signed distance function

In the spatial dimension we replace the implicit function $\phi(x, y) = x^2 + y^2 - 1$ with the signed distance function $\phi(x, y) = \sqrt{x^2 + y^2} - 1$ in order to implicitly represent the unit circle $\partial\Omega = \{(x, y) \mid x^2 + y^2 = 1\}$. Here $|\nabla\phi(x, y)| = 1$ for all $(x, y) \in R^2$, except $(x, y) = 0$.

In the same manner, we replace the implicit function $\phi(x, y, z) = x^2 + y^2 + z^2 - 1$ with the signed distance function $\phi(x, y, z) = \sqrt{x^2 + y^2 + z^2} - 1$ in order to represent the surface of the unit sphere $\partial\Omega = \{(x, y, z) \mid x^2 + y^2 + z^2 = 1\}$ implicitly.

2.3. Discretization

The implicit representation can be stored with a discretization as well. We discretize a bounded sub-domain $D \subset R^2$ and $\partial\Omega \subset D$. Within this domain, we choose a finite set of points (x_i, y_j) for $i = 1, \cdots, N,\ j = 1, \cdots, M$ to discretely represent the implicit function ϕ. This illustrates a drawback of the implicit surface representation. Although the curve is a one dimension, we need to a two dimensional region D to define the implicit function. Once we have chosen the set of points that make up our discretization, we store the values of the implicity function $\phi(\vec{x})$ at each of these points.

But the implicit discretization does not tells us where the interface is located. Instead, they both give information at sample locations. In the implicit representation we know the values of the implicity function ϕ at only a finite number of points and need to use interpolation to find the values of ϕ else where.

We shall distretize in one dimensional space. Let M is a positive integer, $\Delta x = (b-a)/(M-1)$ be the uniform mesh size. And we define the Cartesian grids as $\{x_i \mid x_i = a + (i-1)\Delta x,\ 1 \le i \le M\}$, see Figure 2.7. The set of data points where the implicit function ϕ is defined is called a mesh grid.

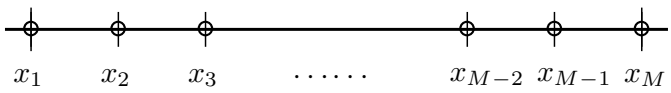


FIGURE 2.7. One-dimensional cell corner mesh grid.

In two dimensional space, let N and M be positive integers, $\Delta x = (b-a)/(M-1)$ and $\Delta y = (d-c)/(N-1)$ be mesh size for each direction. And we define the Cartesian grids as $\{(x_i, y_j) \mid x_i = a + (i-1)\Delta x,\ y_j = c + (j-1)\Delta y,\ 1 \le i \le M,\ 1 \le j \le N\}$. In a uniform Cartesian grid, all the subintervals $[x_i, x_{i+1}]$ are equal in size, and we set $\Delta x = x_{i+1} - x_i$. Likewise, all the subintervals $[y_j, y_{j+1}]$ are equal in size, and we set $\Delta y = y_{j+1} - y_j$. Furthermore, it is usually convenient to choose $\Delta x = \Delta y$ so that the approximation errors are the same in the x-direction as they are in the

y-direction. By definition, Cartesian grids imply the use of a rectangular domain $D = [a, b] \times [c, d]$.

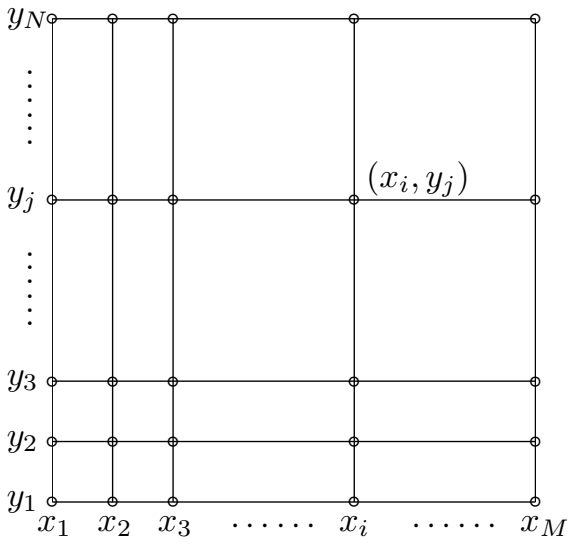


FIGURE 2.8. Two-dimensional cell corner mesh grid.

In three dimensional case, for the computational domain $D = [a, b] \times [c, d] \times [e, f]$, let M, N, and P be positive integers and define $\Delta x = (b - a)/(M - 1)$, $\Delta y = (d - c)/(N - 1)$, and $\Delta z = (e - f)/(P - 1)$. And we can write a uniform Cartesian grid with $\Delta x = \Delta y = \Delta z$ as following set: $\{(x_i, y_j, z_k) \mid x_i = a + (i - 1)\Delta x,\ y_j = c + (j - 1)\Delta y,\ z_k = e + (k - 1)\Delta z,\ 1 \leq i \leq M,\ 1 \leq j \leq N,\ 1 \leq k \leq P\}$. This extension is straightforward generalization from two spatial dimensions. It is the powerful aspect of implicit surfaces that it is straightforward to go from two spatial dimensions to three spatial dimensions.

2.4. Geometrical properties

Initially, the implicit representation might seem wasteful, since the implicit function $\phi(\vec{x})$ is defined on all of R^n, while the interface has only dimension $n-1$. However, we will see that a number of very powerful tools are readily available when we use this representation.

2.4.1. Normal Vector. The gradient of the implicit function is defined as

$$\nabla\phi = \left(\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}\right). \tag{2.4}$$

The gradient $\nabla\phi$ is perpendicular to the iso-contours of ϕ and vector in the direction of increasing ϕ. Therefore, if $\vec{x}_C$ is a point on the zero iso-contour of ϕ, then $\nabla\phi$ evaluated at $\vec{x}_C$ is a vector that points in the same direction as the local unit outward normal vector $\overrightarrow{N}$ to the interface. Thus, the unit outward normal is

$$\overrightarrow{N} = \frac{\nabla\phi}{|\nabla\phi|} \tag{2.5}$$

for points on the interface. When the implicit function is a singed distance function, above equation 2.5 can be simplified to

$$\overrightarrow{N} = \nabla\phi \tag{2.6}$$

for the local unit normal, since $|\nabla\phi| = 1$.

2.4.2. Curvature. The mean curvature of the interface is defined as the divergence of the normal $\overrightarrow{N} = (n_1, n_2, n_3)$,

$$\kappa = \nabla \cdot \overrightarrow{N} = \frac{\partial n_1}{\partial x} + \frac{\partial n_2}{\partial y} + \frac{\partial n_3}{\partial z}, \tag{2.7}$$

so that $\kappa > 0$ for convex regions, $\kappa < 0$ for concave regions, and $\kappa = 0$ for a plane. Substituting equation 2.5 into equation 2.7 gives

$$\kappa = \nabla \cdot \overrightarrow{N} = \nabla \cdot \left(\frac{\nabla\phi}{|\nabla\phi|}\right), \tag{2.8}$$

so that we can writhe the curvature as

$$\kappa = \frac{\phi_{xx}\phi_y^2 - 2\phi_x\phi_y\phi_{xy} + \phi_{yy}\phi_x^2}{(\phi_x^2 + \phi_y^2)^{3/2}}, \tag{2.9}$$

in two dimension, and

$$\begin{aligned}\kappa &= \frac{\phi_x^2\phi_{yy} - 2\phi_x\phi_y\phi_{xy} + \phi_y^2\phi_x x}{(\phi_x^2 + \phi_y^2)^{3/2}} \\ &+ \frac{\phi_x^2\phi_z z - 2\phi_x\phi_z\phi_{xz} + \phi_z^2\phi_{xx}}{(\phi_x^2 + \phi_y^2)^{3/2}} + \frac{\phi_y^2\phi_{zz} - 2\phi_y\phi_z\phi_{yz} + \phi_z^2\phi_{yy}}{(\phi_x^2 + \phi_y^2)^{3/2}},\end{aligned} \tag{2.10}$$

in three dimension. The curvature equation is quite complicate. However, in the signed distance case, it is more simple. If ϕ is signed distance function, the equation 2.8 simplifies to

$$\kappa = \Delta\phi \tag{2.11}$$

for the curvature, where $\Delta\phi$ is the Laplacian of ϕ defined as $\Delta\phi = \phi_{xx} + \phi_{yy}$, in two spatial dimensional, and $\Delta\phi = \phi_{xx} + \phi_{yy} + \phi_{zz}$, in three spatial dimensional.

2.4.3. Blending Surface. Boolean operations for signed distance functions are similar to those for general implicit functions. If ϕ_A and ϕ_B are two different signed distance functions, then $\phi(\vec{x}) = \min(\phi_A(\vec{x}), \phi_B(\vec{x}))$ is the signed distance function representing the union of the interior regions. The function $\phi(\vec{x}) = \max(\phi_A(\vec{x}), \phi_B(\vec{x}))$ is the signed distance function, representing the intersection of the interior regions. The complement of the set defined by $\phi_A(\vec{x})$ has signed distance function $\phi(\vec{x}) = -\phi_A(\vec{x})$. Also, $\phi(\vec{x}) = \max(\phi_A(\vec{x}), -\phi_B(\vec{x}))$ is the signed distance function for the region defined by subtracting the interior of ϕ_B from the interior of ϕ_A.

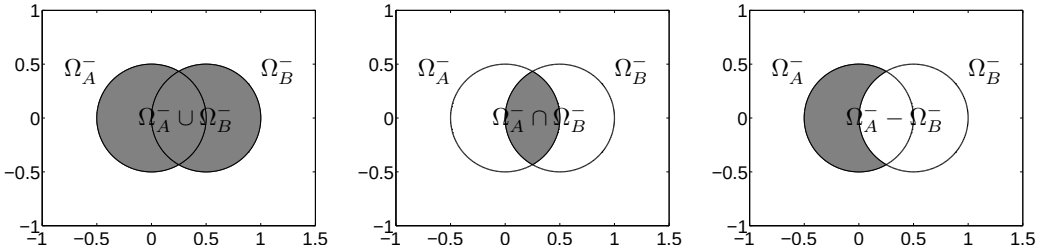
The functions union, difference, and intersection combine two geometries. They use the simplification just mentioned for rectangles, a max or min that ignores "closest corners". We use separate projections to the regions A and B, at distances $\phi_A(x, y)$ and $\phi_B(x, y)$:

$$\text{Union} \quad : \quad \phi_{A\cup B}(x,y) = \min(\phi_A(x,y), \phi_B(x,y)) \tag{2.12}$$

$$\text{Difference} \quad : \quad \phi_{A\setminus B}(x,y) = \max(\phi_A(x,y), -\phi_B(x,y)) \tag{2.13}$$

$$\text{Intersection} \quad : \quad \phi_{A\cap B}(x,y) = \max(\phi_A(x,y), \phi_B(x,y)) \tag{2.14}$$

Variants of these can be used to generate blending surfaces for smooth intersections between two surfaces [33]. For example, consider two implicit functions $f(x,y) = x^2 + y^2 - 0.25$ and $g(x,y) = (x-0.5)^2 + y^2 - 0.25$. Figure 2.9 shows the properties which are union, intersection, and substraction.



(a) Union (b) Intersection (c) Substraction

FIGURE 2.9. Blending surfaces of two circle.

Chapter 3

Motion in an Externally Generated Velocity Field

3.1. Level Set Method

Level set methods add dynamics to implicit surfaces. The key idea is the Hamilton-Jacobi approach to numerical solutions of a time-dependent equation for a moving implicit surface. This was first done in the seminal work of Osher and Sethian [15]. We will discuss this work along the basic convection equation known as the "level set equation". This moves an implicit surface in an externally generated velocity field.

3.2. Convection

Suppose that the velocity of each point on the implicit surface is given as $\overrightarrow{V}(x, y, z)$; i.e., assume that $\overrightarrow{V}(x, y, z)$ is known for every point $(x, y, z) \in D \subset R^3$. Given this velocity field $\overrightarrow{V} = (u, v, w)$, we wish to move all the points on the surface with this velocity.

We use an implicit function ϕ both to represent the interface and to evolve the interface. In order to define the evolution of an implicit function ϕ we use the simple convection or advection equation

$$\frac{\partial \phi}{\partial t} + \overrightarrow{V} \cdot \nabla \phi = 0, \tag{3.1}$$

where t is the time variable and ∇ is the gradient operator defined as

$$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right)$$

in the three dimensional, so that

$$\overrightarrow{V} \cdot \nabla\phi = u\frac{\partial\phi}{\partial x} + v\frac{\partial\phi}{\partial y} + w\frac{\partial\phi}{\partial z}.$$

One and three-dimensional equation is analogously defined. This partial differential equation (PDE) defines the motion of the interface where $\phi(x, y, z) = 0$. It is an Eulerian formulation of the interface evolution, since the interface is captured by the implicit function ϕ. Equation 3.1 is sometimes referred to as the level set equation; it was introduced for numerical interface evolution by Osher and Sethian [15].

3.3. Numerical discretization

Once ϕ and $\overrightarrow{V}$ are defined at every grid point on the Cartesian grid, we can apply numerical methods to evolve ϕ forward in time moving the interface across the grid. At some point in time, say time t^n, let $\phi^n = \phi(t^n)$ represent the current approximation values of ϕ. Updating ϕ in time consists of finding new approximation values of ϕ at every grid point after some time increment Δt. We denote these new values of ϕ by $\phi^{n+1} = \phi(t^{n+1})$, where $t^{n+1} = t^n + \Delta t$.

A rather simple first-order accurate method for the time discretization of equation 3.1 is the forward Euler method given by

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + \overrightarrow{V}^n \cdot \nabla\phi^n = 0, \tag{3.2}$$

where $\overrightarrow{V}^n$ is the given external velocity field at time t^n, and $\nabla\phi^n$ evaluates the gradient operator using the values of ϕ at time t^n. One generally needs to exercise great care when numerically discretizing partial differential equations. We can write the equation 3.2 in expanded form as

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + u^n\phi_x^n + v^n\phi_y^n + w^n\phi_z^n = 0 \tag{3.3}$$

in three dimensional case.

3.4. Upwind Difference

For simplicity, consider the one-dimensional version of equation 3.3,

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} + u^n \phi_x^n = 0, \tag{3.4}$$

where $(\phi_x)_i$ denotes the spatial derivative of ϕ at the point x_i. If $u_i > 0$, the values of ϕ are moving from left to right, and the method of characteristics tells us to look to the left of x_i to determine what value of ϕ will land on the point x_i at the end of a time step. Similarly, if $u_i < 0$, the values of ϕ are moving from right to left, and the method of characteristics implies that we should look to the right to determine an appropriate value of ϕ_i at time t^{n+1}.

Here, we define the backward and forward difference as follows:

$$\left(D^- \phi\right)_i = \frac{\phi_i - \phi_{i-1}}{\Delta x}, \tag{3.5}$$

$$\left(D^+ \phi\right)_i = \frac{\phi_{i+1} - \phi_i}{\Delta x}. \tag{3.6}$$

And clearly $D^- \phi_i$ should be used to approximate $(\phi_x)_i$ when $u_i > 0$. In contrast, $D^+ \phi_i$ cannot possibly give a good approximation, since it fails to contain the information to the left of x_i that dictates the new value of ϕ_i. Similar reasoning indicates that $D^+ \phi$ should be used to approximate ϕ_x when $u_i < 0$. This method of choosing an approximation to the spatial derivatives based on the sign of u is known as upwind difference or upwind method. When $u_i = 0$, the $u_i(\phi_x)_i$ term vanishes, and ϕ_x does not need to be approximated. This is a first-order accurate discretization of the spatial, since $D^- \phi_i$ and $D^+ \phi_i$ are first-order accurate approximations of the derivative, it is called that the errors are $O(\Delta x)$.

The combination of the forward Euler time discretization with the upwind difference scheme is a consistent finite difference approximation to the partial differential equation 3.1, since the approximation error converges to zero as $\Delta t \to 0$ and $\Delta x \to 0$. According to the Lax-Richtmyer equivalence theorem a finite difference approximation to a linear partial differential equation is convergent, i.e., the correct solution is obtained as $\Delta t \to 0$ and $\Delta x \to 0$, if and only if it is both consistent and stable.

Stability guarantees that small errors in the approximation are not amplified as the solution is marched forward in time.

3.5. Implementation of the upwind method

For fast understanding the upwind method, we present short MATLAB code in below and the result of code is depicted in Figure 3.3 for each iteration. In this example, the parameters is as follows: the initial data $\phi_0 = e^{-10(x+1)^2}$, the spatial size $\Delta x = 0.05$, the temporal size $\Delta t = 0.05$, and the domain $D = [-2, 2]$.

```
% 1. Assignment
dx = 0.05; x = -2:dx:2; phi = exp(-10*(x+1).^2);
Vn = zeros(size(phi)+[0 2]);
data = zeros(size(phi)+[0,2]);
phi_x_minus = zeros(size(phi)+[0,2]);
phi_x_plus = zeros(size(phi)+[0,2]);
phi_x = zeros(size(phi)+[0,2]);

it=0; iterations = 10; t=0; dt = 0.05;
while it < iterations
    % 2. External Velocity Field and Initial Setting for calculation
    % The boundary condition - Neumann Condition
    Vn(2:end-1) = 1.0;
    data(2:end-1) = phi;
    data(1) = data(2); data(end) = data(end-1);

    % 3. Calculate the the gradient
    phi_x_minus(2:end-1) = (data(2:end-1)-data(1:end-2))/dx;
    phi_x_plus(2:end-1) = (data(3:end)-data(2:end-1))/dx;
    phi_x = phi_x_minus.*(Vn > 0) + phi_x_minus.*(Vn < 0)+0*(Vn == 0);

    % 4. New approximation
    delta = Vn.*phi_x;
    phi=phi-dt*delta(2:end-1);
    it=it+1; t=t+dt;
end
```

FIGURE 3.1. MATLAB code for the upwind method

Now we describe steps 1 to 4 in the upwind algorithm code.

1. Initial setting the parameters and the assignment the using the vector. In the assignment step, the vector is expended as 2 size because of the boundary setting.

```
dx = 0.05; x = -2:dx:2; phi = exp(-10*(x+1).^2);
Vn = zeros(size(phi)+[0 2]);
data = zeros(size(phi)+[0,2]);
phi_x_minus = zeros(size(phi)+[0,2]);
phi_x_plus = zeros(size(phi)+[0,2]);
phi_x = zeros(size(phi)+[0,2]);
```

L. Now the code enters the main loop, where the ϕ^n is iteratively evolved. Initialize the variable **it** for the first iteration, and the termination criterion is given by **iteration**. At the last of the loop, the variable **it** and **t** is updated.

```
it=0; iterations = 10; t=0; dt = 0.05;
while it < iterations
    ...
    it=it+1; t=t+dt;
end
```

2. The externally generated velocity is setting in here, $\overrightarrow{V} = (u) = 1$. And *data* is store the current ϕ^n at $2 : \text{end} - 1$, the 1 and end components are used the boundary points. In this case, we present the Neumann boundary condition for reasonable results.

```
Vn(2:end-1) = 1.0;
data(2:end-1) = phi;
data(1) = data(2); data(end) = data(end-1);
```

3. The variables phi_x_minus and phi_x_plus store the candidates $D^-\phi_x$ and $D^+\phi_x$. In this example, we use the backward and forward difference equation, respectively. And we use the upwind scheme for choosing ϕ_x.

```
phi_x_minus(2:end-1) = (data(2:end-1)-data(1:end-2))/dx;
phi_x_plus(2:end-1) = (data(3:end)-data(2:end-1))/dx;
phi_x = phi_x_minus.*(Vn > 0) + phi_x_minus.*(Vn < 0)+0*(Vn == 0);
```

4. Finally, we calculate the new approximation ϕ^{n+1}.

```
delta = Vn.*phi_x;
phi=phi-dt*delta(2:end-1);
```

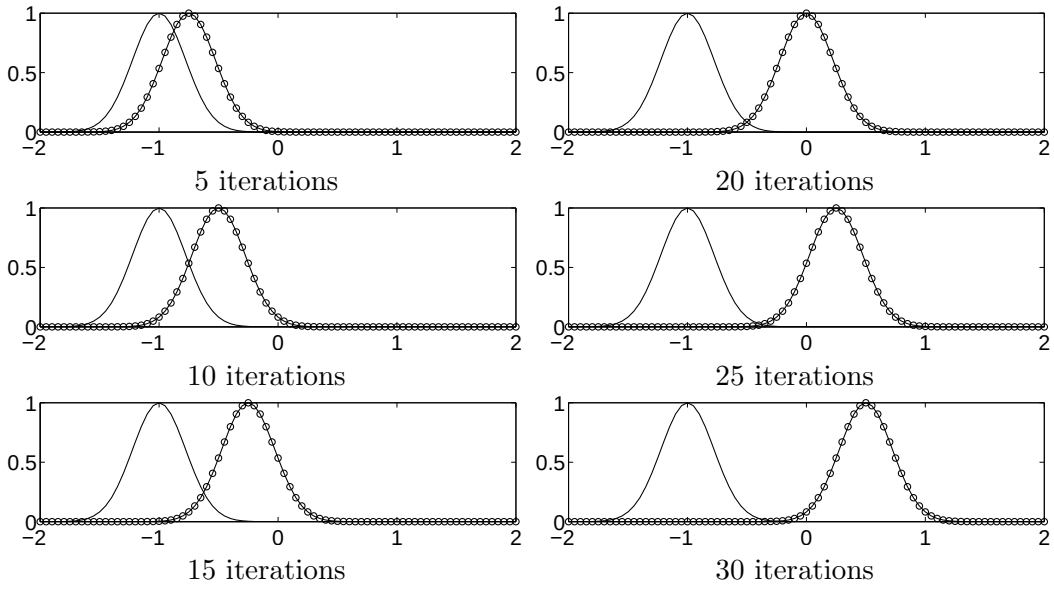


FIGURE 3.2. Example of the upwind simulation $u = 1$.

3.6. CFL condition

Stability can be enforced using the Courant-Friedreichs-Lewy (CFL) condition. This condition means that the numerical wave speed of $\Delta x/\Delta t$ must be at least as fast as the physical wave speed $|u|$, i.e., $\Delta x/\Delta t > |u|$. This leads us to the CFL time step restriction of

$$\Delta t \leq \frac{\Delta x}{\max\{|u|\}}, \tag{3.7}$$

where $\max\{|u|\}$ is chosen to be the largest value of $|u|$ over the entire Cartesian grid. Equation 3.7 is usually enforced by choosing a CFL number α with

$$\Delta t \left(\frac{\max\{|u|\}}{\Delta x} \right) = \alpha \tag{3.8}$$

and $0 < \alpha \leq 1$. In the upwind example code, figure 3.1, the CFL condition number α is 1. A common near-optimal choice is $\alpha = 0.9$, and a common conservative choice is $\alpha = 0.5$. A multidimensional CFL condition can be written as

$$\Delta t \max \left\{ \frac{|u|}{\Delta x} + \frac{|v|}{\Delta y} + \frac{|w|}{\Delta z} \right\} = \alpha, \tag{3.9}$$

and

$$\Delta t \left(\frac{\max\{|\overrightarrow{V}|\}}{\min\{\Delta x, \Delta y, \Delta z\}} \right) = \alpha \tag{3.10}$$

is also quite popular. More details on consistency, stability, and convergence can be found in basic textbooks on the numerical solution of partial differential equations, see ref. [16].

3.7. Hamilton-Jacobi ENO

The first-order accurate upwind scheme described in the last section can be improved upon by using a more accurate approximation for ϕ_x^- and ϕ_x^+. The velocity u is sill used to decide whether ϕ_x^- or ϕ_x^+ is used, but the approximations for ϕ_x^- or ϕ_x^+ can be improved significantly.

(ENO) polynomial interpolation of data for the numerical solution of conservation laws. Their basic idea was to compute numerical flux functions using the smoothest possible polynomial interpolants. The actual numerical implementation of this idea was improved considerably by Shu and Osher in [17] and [18], where the numerical flux functions were constructed directly from a divided difference table of the pointwise data. In [15], Osher and Sethian realized that Hamilton-Jacobi equations in one spatial dimension are integrals of conservation laws. They used this fact to extend the ENO method for the numerical discretization of conservation laws to Hamilton-Jacobi equations such as equation 3.1. This Hamilton-Jacobi ENO (HJ ENO) method allows one to extend first-order accurate upwind differencing to higher-order spatial accuracy by providing better numerical approximations to ϕ_x^- or ϕ_x^+

Proceeding along the lines of [17] and [18], we use the smoothest possible polynomial interpolation to find ϕ and then differentiate to get ϕ_x. As is standard with Newton polynomial interpolation (see any undergraduate numerical analysis text,

e.g., [19]). The first divided differences of ϕ are defined midway between grid nodes as

$$D^1_{i+\frac{1}{2}}\phi = \frac{\phi_{i+1} - \phi_i}{\Delta x}, \tag{3.11}$$

where we are assuming that the mesh spacing is uniformly Δx. Note that $D^1_{i-\frac{1}{2}}\phi = (D^-\phi)_i$ and $D^1_{i+\frac{1}{2}}\phi = (D^+\phi)_i$, i.e., the first divided differences, are the backward and forward difference approximations to the derivatives. The second divided differences are defined at the grid nodes as

$$D^2_i\phi = \frac{D^1_{i+\frac{1}{2}}\phi - D^1_{i-\frac{1}{2}}\phi}{\Delta x}, \tag{3.12}$$

while the third divided differences

$$D^3_{i+\frac{1}{2}}\phi = \frac{D^2_{i+1}\phi - D^2_i\phi}{\Delta x} \tag{3.13}$$

are defined midway between the grid nodes.

The divided differences are used to reconstruct a polynomial of the form

$$\phi(x) = Q_0(x) + Q_1(x) + Q_2(x) + Q_3(x) \tag{3.14}$$

that can be differentiated and evaluated at x_i to find $(\phi^-_x)_i$ and $(\phi^+_x)_i$. That is, we use

$$\phi_x(x_i) = Q'_1(x_i) + Q'_2(x_i) + Q'_3(x_i) \tag{3.15}$$

to define $(\phi^-_x)_i$ and $(\phi^+_x)_i$. Note that the constant $Q_0(x)$ term vanishes upon differentiation.

To find ϕ^-_x we start with $k = i - 1$, and to find ϕ^+_x we start with $k = i$. Then we define

$$Q_1(x) = (D^1_{k+\frac{1}{2}})(x - x_i), \tag{3.16}$$

so that

$$Q_1'(x_i) = D^1_{k+\frac{1}{2}}\phi, \tag{3.17}$$

implying that the contribution from $Q_1'(x_i)$ in equation 3.15 is the backward difference in the case of ϕ_x^- in equation 3.15 is the backward difference in the case of ϕ_x^+. In other words, first-order accurate polynomial interpolation is exactly first-order upwinding. Improvements are obtained by including the $Q_2'(x_i)$ and $Q_3'(x_i)$ terms in equation 3.15, leading to second- and third-order accuracy, respectively.

Looking at the divided difference table and noting that $D^1_{k+\frac{1}{2}}\phi$ was chosen for first-order accuracy, we have two choices for the second-order accurate correction. We could include the next point to the left and use $D^2_k\phi$, or we could include the next point to the right and use $D^2_{k+1}\phi$. The key observation is that smooth slowly varying data tend to produce small numbers in divided difference tables, while discontinuous or quickly varying data tend to produce large numbers in divided difference tables. This is obvious in the sense that the differences measure variation in the data. Comparing $|D^2_k\phi|$ to $|D^2_{k+1}\phi|$ indicates which of the polynomial interpolants has more variation. We would like to avoid interpolating near large variations such as discontinuities or steep gradients, since they cause overshoots in the interpolating function, leading to numerical errors in the approximation of the derivative. Thus, if $|D^2_k\phi| \leq |D^2_{k+1}\phi|$, we set $c = D^2_k\phi/2$ and $k^\star = k - 1$; otherwise, we set $c = D^2_{k+1}\phi/2$ and $k^\star = k$. Then we define

$$Q_2(x) = c(x - x_k)(x - x_{k+1}), \tag{3.18}$$

so that

$$Q_2'(x_i) = c(2(i-k)-1)\Delta x \tag{3.19}$$

is the second-order accurate correction to the approximation of ϕ_x in equation (3.18). If we stop here, i.e., omitting the Q_3 term, we have a second-order accurate method for approximating ϕ_x^- and ϕ_x^+. Note that $k^\star$ has not yet been used. It is defined below for use in calculating the third-order accurate correction.

Similar to the second-order accurate correction, the third-order accurate correction is obtained by comparing $|D^3_{k^\star+\frac{1}{2}}\phi|$ and $|D^3_{k^\star+\frac{3}{2}}\phi|$. If $|D^3_{k^\star+\frac{1}{2}}\phi| \le |D^3_{k^\star+\frac{3}{2}}\phi|$, we set $c^\star = D^3_{k^\star+\frac{1}{2}}\phi/3$; otherwise, we set $c^\star = D^3_{k^\star+\frac{3}{2}}\phi/3$. Then we define

$$Q_3(x) = c^\star(x - x_{k^\star})(x - x_{k^\star+1})(x - x_{k^\star+2}), \tag{3.20}$$

so that

$$Q_3'(x_i) = c^\star(3(i-k^\star)^2 - 6(i-k^\star) + 2)(\Delta x)^2 \tag{3.21}$$

is the third-order accurate correction to the approximation of ϕ_x in equation (3.18).

3.8. Implementation

In this section, we present the detail explanation for the second order ENO scheme. The third order ENO scheme is in the next Chapter with the reinitialization.

```
% 1. Assignment
dx = 0.05; x = -2:dx:2; phi(size(x))=0;
Vn = zeros(size(phi)+[0 4]);
delta = zeros(size(phi)+[0 4]);
data = zeros(size(phi)+[0 4]);
phi_x_minus = zeros(size(phi)+[0 4]);
phi_x_plus = zeros(size(phi)+[0 4]);
phi_x = zeros(size(phi)+[0 4]);

% 2. Initialization
ix=x>-1.5 & x<-0.5; phi(ix)=0.5-abs(x(ix)+1);
```

```
phi0 = phi;

it=0; iterations = 100; t=0; dt = 0.025;
while it < iterations
    % 3. External Velocity Field and Initial Setting for iteration
    % The boundary condition - Periodic  condition
    Vn(3:end-2) = 1; data(3:end-2) = phi;
    data(2) = data(end-2); data(1) = data(end-3);
    data(end-1) = data(1); data(end) = data(2);

    % 4. Calculate the gradient
    phi_x_minus= der_ENO2_minus(data, dx);
    phi_x_plus = der_ENO2_plus(data, dx);
    phi_x = phi_x_minus.*(Vn > 0) + phi_x_minus.*(Vn < 0)+0*(Vn == 0);

    % 5. New approximation
    delta = Vn.*phi_x;
    phi = phi - dt * delta(3:end-2);
    it = it+1; t = t+dt;
end
```

Now we describe steps 1 to 5 in the upwind algorithm code.

1. Initial setting the parameters and the assignment the using the vector. In the assignment step, the vector is expended as 4 size because of the boundary setting.

   ```
   dx = 0.05; x = -2:dx:2; phi(size(x))=0;
   Vn = zeros(size(phi)+[0 4]);
   delta = zeros(size(phi)+[0 4]);
   data = zeros(size(phi)+[0 4]);
   phi_x_minus = zeros(size(phi)+[0 4]);
   phi_x_plus = zeros(size(phi)+[0 4]);
   phi_x = zeros(size(phi)+[0 4]);
   ```

2. Initialization for ϕ_0 which is triangle shape.

   ```
   ix=x>-1.5 & x<-0.5;
   phi(ix)=0.5-abs(x(ix)+1);
   ```

L. Now the code enters the main loop, where the ϕ^n is iteratively evolved. Initialize the variable **it** for the first iteration, and the termination criterion is given by **iteration**. At the last of the loop, the variable **it** and **t** is updated.

   ```
   it=0; iterations = 100; t=0; dt = 0.025;
   while it < iterations
   ```

```
    ...
    it = it+1; t = t+dt;
end
```

3. The externally generated velocity is setting in here, $\overrightarrow{V} = (u) = 1$. And *data* is store the current ϕ^n at $\mathbf{3 : end - 1}$, the **1, 2** and $\mathbf{end - 1, end}$ components are used the boundary points. In this case, we present the Periodic boundary condition for reasonable results.

```
Vn(3:end-2) = 1; data(3:end-2) = phi;
data(2) = data(end-2); data(1) = data(end-3);
data(end-1) = data(1); data(end) = data(2);
```

4. The variables phi_x_minus and phi_x_plus store the candidates $D^-\phi_x$ and $D^+\phi_x$. In this example, we use the ENO difference scheme. And we use the upwind scheme for choosing ϕ_x.

```
phi_x_minus= der_ENO2_minus(data, dx);
phi_x_plus = der_ENO2_plus(data, dx);
phi_x = phi_x_minus.*(Vn > 0) + phi_x_minus.*(Vn < 0)+0*(Vn == 0);
```

5. Finally, we calculate the new approximation ϕ^{n+1}.

```
delta = Vn.*phi_x;
phi = phi - dt * delta(3:end-2);
```

The function codes **der_ENO2_minus** and **der_ENO2_plus** are accompanying.

```
function  data_x = der_ENO2_minus(data, dx)
data_x = zeros(size(data));

D1 = (data(2:end)-data(1:end-1))/dx;
D2 = (D1(2:end)-D1(1:end-1))/dx;
absD2 = abs(D2);

for i=1:(length(data)-4)
    k = i-1;
    Q1p = D1(k+2);
    if absD2(k+1) <= absD2(k+2)
        c = D2(k+1)/2;
    else
        c = D2(k+2)/2;
    end
    Q2p = c*(2*(i-k)-1)*dx;
    data_x(i+2) = Q1p+Q2p;
```

```
end
```

```
function [data_x] = der_ENO2_plus(data, dx)
data_x = zeros(size(data));

D1 = (data(2:end)-data(1:end-1))/dx;
D2 = (D1(2:end)-D1(1:end-1))/dx;
absD2 = abs(D2);

for i=1:(length(data)-4)
    k = i;
    Q1p = D1(k+2);
    if absD2(k+1) <= absD2(k+2)
        c = D2(k+1)/2;
    else
        c = D2(k+2)/2;
    end
    Q2p = c*(2*(i-k)-1)*dx;
    data_x(i+2) = Q1p+Q2p;
end
```

Figure 3.3 is the results when we carry out the code in the MATLAB.

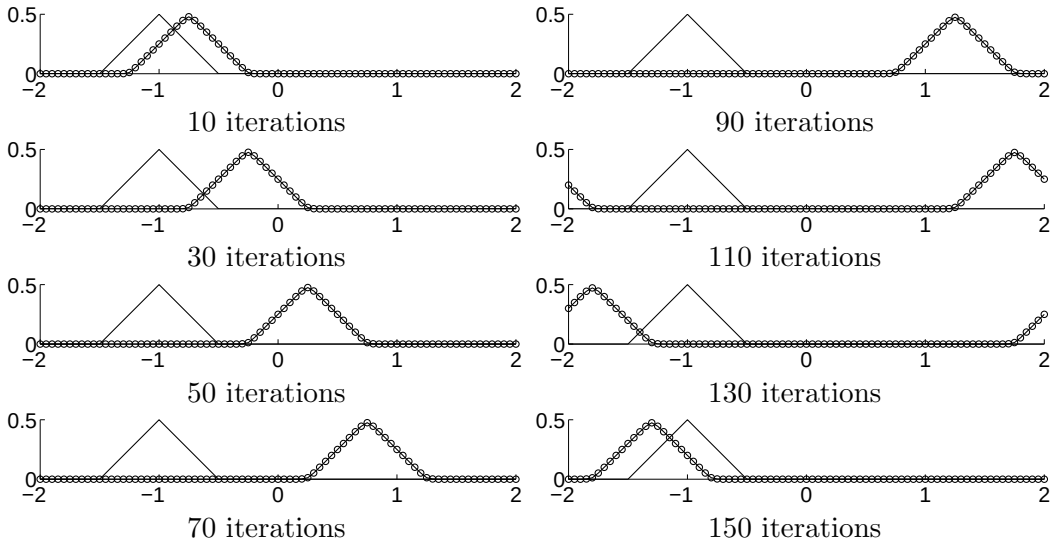


FIGURE 3.3. Example of the upwind simulation $u = 1$.

Chapter 4

Constructing Signed Distance Functions

4.1. Introduction

In the mesh generation in Chapter 5, we need an implicit surface function ϕ, and it can be made simple when ϕ is a signed distance function. for this reason, we dedicate this chapter to numerical techniques for constructing approximate signed distance functions. These techniques can be applied to the initial data in order to initialize ϕ to a signed distance function.

Even if a particular numerical approach doesn't seem to depend on how accurately ϕ approximates a signed distance function, one needs to remember that ϕ can develop noisy features and steep gradients that are not amenable to finite difference approximations. For this reason, it is always advisable to reinitialize occasionally so that ϕ stays smooth enough to approximate its spatial derivatives with some degree of accuracy.

4.2. Motion in the Normal Direction

We discuss the motion of an interface under an internally generated velocity field for constant motion in the normal direction. This velocity field is defined by $\overrightarrow{V} = a\overrightarrow{N}$ or $V_n = a$, where a is a constant. The corresponding level set equation (i.e., equation (4.4)) is

$$\phi_t + a|\nabla\phi| = 0, \tag{4.1}$$

where a can be of either sign. When $a > 0$ the interface moves in the normal direction, and when $a < 0$ the interface moves opposite the normal direction. When $a = 0$ this equation reduces to the trivial $\phi_t = 0$, where ϕ is constant for all time.

4.2.1. Numerical Discretization. First of all, we consider

$$|\nabla\phi| = \frac{|\nabla\phi|^2}{|\nabla\phi|} = \frac{\nabla\phi}{|\nabla\phi|} \cdot \nabla\phi = \overrightarrow{N} \cdot \nabla\phi. \tag{4.2}$$

For instructive purposes, suppose we plug $\overrightarrow{V} = a\overrightarrow{N}$ into equation 3.1 and try a simple upwind differencing approach. That is, we can write the equation 4.1 as

$$\frac{\partial\phi}{\partial t} + \left(\frac{a\phi_x}{|\nabla\phi|}, \frac{a\phi_y}{|\nabla\phi|}, \frac{a\phi_z}{|\nabla\phi|}\right) \cdot \nabla\phi = 0, \tag{4.3}$$

and we will discretize as follows:

$$\frac{\phi^{n+1} - \phi^n}{\partial t} + \left(\frac{a\phi_x^n}{|\nabla\phi^n|}, \frac{a\phi_y^n}{|\nabla\phi^n|}, \frac{a\phi_z^n}{|\nabla\phi^n|}\right) \cdot \nabla\phi^n = 0 \tag{4.4}$$

with forward difference. The CFL condition for equation 4.4 is

$$\frac{\Delta t}{|\nabla\phi^n|}\left(\frac{|a\phi_x^n|}{\Delta x} + \frac{|a\phi_y^n|}{\Delta y} + \frac{|a\phi_z^n|}{\Delta z}\right) = \alpha \tag{4.5}$$

with $0 < \alpha \leq 1$ corresponding to equation 3.9.

4.3. Godunov's Scheme

In [25], Godunov proposed a numerical method that gives the exact solution to the Riemann problem for one-dimensional conservation laws with piecewise constant initial data. Let us examine the Godunov method in detail. First, assume $a > 0$. If ϕ_x^- and ϕ_x^+ are both positive, then use $\phi_x = \phi_x^-$. If ϕ_x^- and ϕ_x^+ are both negative, use $\phi_x = \phi_x^+$. If $a\phi_x^+ \leq 0$ and $a\phi_x^- \geq 0$, treat the expansion by setting $\phi_x = 0$. If $a\phi_x^+ \geq 0$ and $a\phi_x^- \leq 0$, treat the shock by setting ϕ_x to either ϕ_x^- or ϕ_x^+,, depending on which gives the largest magnitude for $a\phi_x$. Note that when $\phi_x^- = \phi_x^+ = 0$ both of the last two cases are activated, and both consistently give $\phi_x = 0$. We also have the following elegant formula due to Rouy and Tourin [11]: $\phi_x^2 \approx \max\left(\max(\phi_x^-, 0)^2, \min(\phi_x^+, 0)^2\right)$ when $a > 0$, and $\phi_x^2 \approx \max\left(\min(\phi_x^-, 0)^2, \max(\phi_x^+, 0)^2\right)$ when $a < 0$.

4.4. Implement

Next, we describe the MATLAB code for motion in normal direction with the Hamilton-Jacobi ENO scheme in two spacial dimension. The code exactly generate result, figure 4.2. The extension to the three dimension remain for readers.

```
dx = 0.05; dy = 0.05; alpha = 0.5;
[xx,yy]=meshgrid(-2:dx:2,-2:dy:2);
phi  = (xx.^2).^(1/3) + (yy.^2).^(1/3) -1;
delta = zeros(size(phi)+4);
Vn = zeros(size(S_phi)+4);

S_phi = phi./sqrt(phi.^2 + dx.^2);
Vn(3:end-2,3:end-2) = -1;

it=0; t=0; iterations = 10;
while (it < iterations)
    [delta, Vx, Vy] = evolve_normal_ENO2(phi, dx, dy, Vn);
    maxs = max(max(Vx/dx+Vy/dy));
    dt = alpha/(maxs+(maxs==0));
    phi = phi - dt*delta;
    it = it+1; t = t+dt;
end
```

```
function [delta, Vx, Vy] = evolve_normal_ENO2(phi, dx, dy, Vn)
data = zeros(size(phi)+4);
phi_x_minus = zeros(size(phi)+4);
phi_x_plus = zeros(size(phi)+4);
phi_y_minus = zeros(size(phi)+4);
phi_y_plus = zeros(size(phi)+4);
phi_x = zeros(size(phi)+4);
phi_y = zeros(size(phi)+4);

data(3:end-2,3:end-2) = phi;

for i=1:size(phi,1)
phi_x_minus(i+2,:) = der_ENO2_minus(data(i+2,:), dx);
phi_x_plus(i+2,:) = der_ENO2_plus(data(i+2,:), dx);
phi_x(i+2,:) = select_der_normal(Vn(i+2,:), phi_x_minus(i+2,:), phi_x_plus(i+2,:));
end

for j=1:size(phi,2)
phi_y_minus(:,j+2) = der_ENO2_minus(data(:,j+2), dy);
phi_y_plus(:,j+2) = der_ENO2_plus(data(:,j+2), dy);
phi_y(:,j+2) = select_der_normal(Vn(:,j+2), phi_y_minus(:,j+2), phi_y_plus(:,j+2));
end

abs_grad_phi = sqrt(phi_x.^2 + phi_y.^2);
```

```
Vx = abs(Vn.*phi_x ./ (abs_grad_phi+dx*dx*(abs_grad_phi == 0)));
Vy = abs(Vn.*phi_y ./ (abs_grad_phi+dx*dx*(abs_grad_phi == 0)));
Vx = Vx(3:end-2,3:end-2);
Vy = Vy(3:end-2,3:end-2);

delta = Vn.*abs_grad_phi;
delta = delta(3:end-2,3:end-2);
```

```
function [data_x] = der_ENO2_minus(data, dx)
data_x = zeros(size(data));

data(2) = 2*data(3)-data(4);
data(1) = 2*data(2)-data(3);
data(end-1) = 2*data(end-2)-data(end-3);
data(end) = 2*data(end-1)-data(end-2);

D1 = (data(2:end)-data(1:end-1))/dx;
D2 = (D1(2:end)-D1(1:end-1))/dx;
absD2 = abs(D2);

for i=1:(length(data)-4)
    k = i-1;
    Q1p = D1(k+2);
    if absD2(k+1) <= absD2(k+2)
        c = D2(k+1)/2;
    else
        c = D2(k+2)/2;
    end
    Q2p = c*(2*(i-k)-1)*dx;
    data_x(i+2) = Q1p+Q2p;
end
```

```
function [data_x] = der_ENO2_plus(data, dx)
data_x = zeros(size(data));

data(2) = 2*data(3)-data(4);
data(1) = 2*data(2)-data(3);
data(end-1) = 2*data(end-2)-data(end-3);
data(end) = 2*data(end-1)-data(end-2);

D1 = (data(2:end)-data(1:end-1))/dx;
D2 = (D1(2:end)-D1(1:end-1))/dx;
absD2 = abs(D2);

for i=1:(length(data)-4)
    k = i;
    Q1p = D1(k+2);
    if absD2(k+1) <= absD2(k+2)
        c = D2(k+1)/2;
```

```
    else
        c = D2(k+2)/2;
    end

    Q2p = c*(2*(i-k)-1)*dx;
    data_x(i+2) = Q1p+Q2p;
end
```

```
function [der] = select_der_normal(Vn, der_minus, der_plus)
der = zeros(size(der_plus));

for i=1:numel(Vn)
    Vn_der_m = Vn(i)*der_minus(i);
    Vn_der_p = Vn(i)*der_plus(i);

    if Vn_der_m <= 0 & Vn_der_p <= 0
        der(i) = der_plus(i);
    elseif Vn_der_m >= 0 & Vn_der_p >= 0
        der(i) = der_minus(i);
    elseif Vn_der_m <= 0 & Vn_der_p >= 0
        der(i) = 0;
    elseif Vn_der_m >= 0 & Vn_der_p <= 0
        if abs(Vn_der_p) >= abs(Vn_der_m)
            der(i) = der_plus(i);
        else
            der(i) = der_minus(i);
        end
    end
end
```

Figure 4.1 shows the evolution of a rose-shaped interface $r = \sin 9\theta$ as it moves normal to itself in the outward direction.

Figure 4.2 shows the evolution of a star-shaped interface $\phi(x, y) = |x|^{1/3} + |y|^{1/3} - 1$ as it moves normal to itself in the inward direction.

4.5. Crossing Times

One of the difficulties associated with the direct computation of signed distance functions is locating and discretizing the interface. This can be avoided in the following fashion. Consider a point $\vec{x} \in \Omega^+$. If $\vec{x}$ does not lie on the interface, we wish to know how far from the interface it is so that we can set $\phi(\vec{x}) = +d$. If we move the interface in the normal direction using equation (4.1) with $a = 1$, the

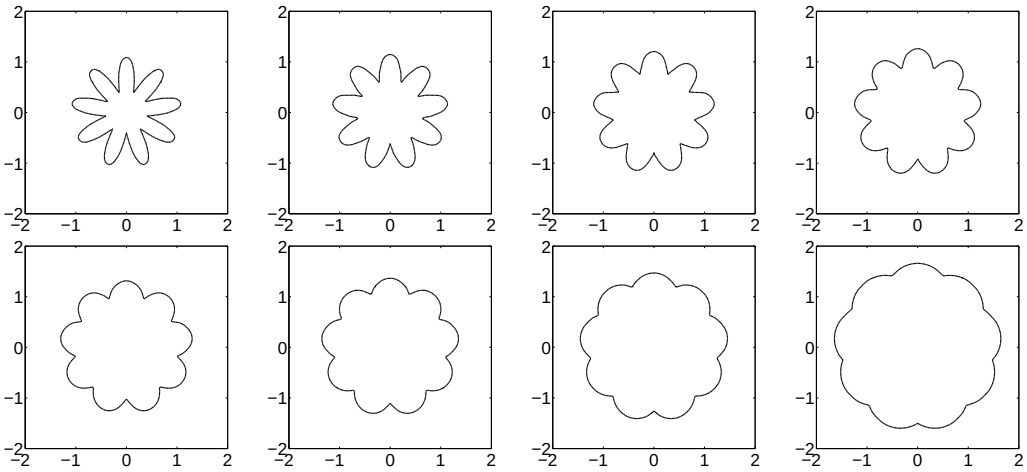


FIGURE 4.1. Evolution of a nine leaved rose-shaped interface as it moves normal to itself in the outward direction. The iteration number is written at each figures.

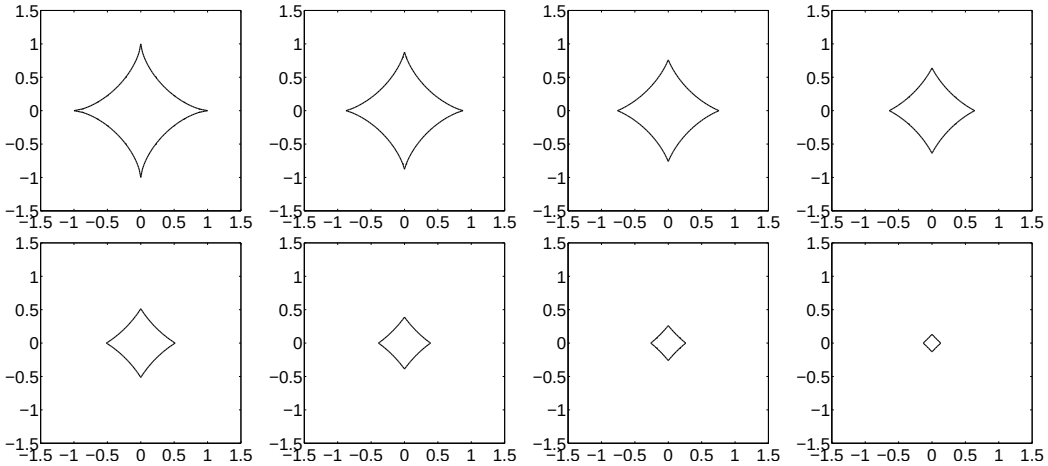


FIGURE 4.2. Evolution of a star-shaped interface as it moves normal to itself in the inward direction. The iteration number is written at each figures.

interface eventually crosses over $\vec{x}$, changing the local value of ϕ from positive to negative. If we keep a time history of the local values of ϕ at $\vec{x}$, we can find the exact time when ϕ was equal to zero using interpolation in time. This is the time

it takes the zero level set to reach the point $\vec{x}$, and we call that time t_0 the crossing time. Since equation (4.1) moves the level set normal to itself with speed $a = 1$, the time it takes for the zero level set to reach a point $\vec{x}$ is equal to the distance the interface is from $\vec{x}$. That is, the crossing time t_0 is equal to the distance d. For points $\vec{x} \in \Omega^-$, the crossing time is similarly determined using $a = -1$ in equation (4.1).

In a series of papers, [8] and [9], Kimmel and Bruckstein introduced the notion of using crossing times in image-processing applications. For example, [9] used equation (4.1) with $a = 1$ to create shape offsets, which are distance functions with distance measured from the boundary of an image. The idea of using crossing times to solve some general Hamilton- Jacobi equations with Dirichlet boundary conditions was later generalized and rigorized by Osher [10].

4.6. Reinitialization Equation

In [11], Rouy and Tourin proposed a numerical method for solving $|\nabla\phi| = f(\vec{x})$ for a spatially varying function f derived from the intensity of an image. In the trivial case of $f(\vec{x}) = 1$, the solution ϕ is a signed distance function. They added $f(\vec{x})$ to the right-hand side of equation (4.1) as a source term to obtain

$$\phi_t + |\nabla\phi| = f(\vec{x}) \tag{4.6}$$

which is evolved in time until a steady state is reached. At steady state, the values of ϕ cease to change, implying that $\phi_t = 0$. Then equation (4.6) reduces to $|\nabla\phi| = f(\vec{x})$, as desired. Since only the steady-state solution is desired, [11] used an accelerated iteration method instead of directly evolving equation (4.6) forward in time.

Equation (4.6) propagates information in the normal direction, so information flows from smaller values of ϕ to larger values of ϕ. This equation is of little use in reinitializing the level set function, since the interface location will be influenced by the negative values of ϕ. That is, the $\phi = 0$ isocontour is not guaranteed to stay

fixed, but will instead move around as it is influenced by the information flowing in from the negative values of ϕ. One way to avoid this is to compute the signed distance function for all the grid points adjacent to the interface. Then

$$\phi_t + |\nabla\phi| = 1 \tag{4.7}$$

can be solved in Ω^+ to update ϕ based on those grid points adjacent to the interface. And we can update the values of ϕ in Ω^- by solving

$$\phi_t - |\nabla\phi| = -1 \tag{4.8}$$

to steady state. Equations (4.7) and (4.8) reach steady state rather quickly, since they propagate information at speed 1 in the direction normal to the interface. For example, if $\Delta t = 0.5\Delta x$, it takes only about 10 time steps to move information from the interface to 5 grid cells away from the interface.

In [12], Sussman, Smereka, and Osher put all this together into a reinitialization equation

$$\phi_t + S(\phi_0)(|\nabla\phi| - 1) = 0, \tag{4.9}$$

where $S(\phi_0)$ is a sign function defined as follows:

$$S(\phi_0) = \begin{cases} 1, & \text{if } \vec{x} \in \Omega^+, \\ 0, & \text{if } \vec{x} \in \partial\Omega, \\ -1, & \text{if } \vec{x} \in \Omega^-. \end{cases} \tag{4.10}$$

We want ϕ to stay identically equal to zero. Using this equation, there is no need to initialize any points near the interface for use as boundary conditions. The points near the interface in Ω^+ use the points in Ω^- as boundary conditions, while the points in Ω^- conversely look at those in Ω^+. This circular loop of dependencies eventually balances out, and a steady-state signed distance function is obtained. As long as ϕ is relatively smooth and the initial data are somewhat balanced across the interface, this method works rather well.

4.7. Numerical Discretization of Reinitialization Equation

A nice feature of using this procedure to reinitialize is that the level set function is reinitialized near the front first. To see this we rewrite equation 4.9 as

$$\frac{\partial \phi}{\partial t} + \overrightarrow{V} \cdot \nabla \phi = S(\phi), \tag{4.11}$$

where

$$\overrightarrow{V} = S(\phi)\frac{\nabla \phi}{|\nabla \phi|}.$$

It is evident that equation 4.11 is a nonlinear hyperbolic equation with the characteristic velocities pointing outwards from the interface in the direction of the normal. This means that ϕ will be reinitialized to $|\nabla \phi| = 1$ near the interface first.

In discretizing equation 4.9, the $S(\phi_0)|\nabla \phi|$ term is treated as motion in the normal direction as described in the first part of this Chapter as below.

$$S(\phi_0)|\nabla \phi| = S(\phi_0)\frac{\nabla \phi}{|\nabla \phi|} \cdot \nabla \phi, \tag{4.12}$$

so that equation 4.11 is corresponding to

$$\frac{\partial \phi}{\partial t} + S(\phi_0)\frac{\nabla \phi}{|\nabla \phi|} \cdot \nabla \phi = S(\phi_0). \tag{4.13}$$

Here $S(\phi_0)$ is constant for all time and can be thought of as a spatially varying "a" term. Numerical tests indicate that better results are obtained when $S(\phi_0)$ is numerically smeared out, so [12] used

$$S(\phi_0) = \frac{\phi_0}{\sqrt{\phi_0^2 + (\Delta x)^2}} \tag{4.14}$$

as a numerical approximation. Numerical smearing of the sign function decreases its magnitude, slowing the propagation speed of information near the interface. This probably aids the balancing out of the circular dependence on the initial data as well, since it produces characteristics that do not look as far across the interface for their information. We recommend using Godunov's method for discretizing the

hyperbolic $S(\phi_0)|\nabla\phi|$ term. After finding a numerical approximation to $S(\phi_0)|\nabla\phi|$, we combine it with the remaining $S(\phi_0)$ source term at each grid point.

4.8. Implement

```
dx = 0.01; dy = 0.01; iterations = 10; alpha = 0.5;

[xx,yy]=meshgrid(-2:dx:2,-2:dy:2);
phi  = xx.^2 + yy.^2 - 1;

S_phi = phi./sqrt(phi.^2 + dx.^2);
Vn_ext = zeros(size(S_phi)+6);
Vn_ext(4:end-3,4:end-3) = S_phi;

it=0; t=0;
while (it < iterations)
    [delta, H1_abs, H2_abs] = evolve_normal_ENO3(phi, dx, dy, Vn_ext);
    maxs = max(max(H1_abs/dx + H2_abs/dy));
    dt = alpha/(maxs+(maxs==0));
    phi = phi + dt*(S_phi - delta);
    it = it+1; t = t+dt;
end
```

```
function [delta, H1_abs, H2_abs] = evolve_normal_ENO3(phi, dx, dy, Vn)
delta = zeros(size(phi)+6);
data_ext = zeros(size(phi)+6);
data_ext(4:end-3,4:end-3) = phi;

phi_x_minus = zeros(size(phi)+6);
phi_x_plus = zeros(size(phi)+6);
phi_y_minus = zeros(size(phi)+6);
phi_y_plus = zeros(size(phi)+6);
phi_x = zeros(size(phi)+6);
phi_y = zeros(size(phi)+6);
for i=1:size(phi,1)
phi_x_minus(i+3,:) = der_ENO3_minus(data_ext(i+3,:), dx);
phi_x_plus(i+3,:) = der_ENO3_plus(data_ext(i+3,:), dx);
phi_x(i+3,:) = select_der_normal(Vn(i+3,:), phi_x_minus(i+3,:), phi_x_plus(i+3,:));
end

% then scan the columns
for j=1:size(phi,2)
phi_y_minus(:,j+3) = der_ENO3_minus(data_ext(:,j+3), dy);
phi_y_plus(:,j+3) = der_ENO3_plus(data_ext(:,j+3), dy);
phi_y(:,j+3) = select_der_normal(Vn(:,j+3), phi_y_minus(:,j+3), phi_y_plus(:,j+3));
end

abs_grad_phi = sqrt(phi_x.^2 + phi_y.^2);
```

```
H1_abs = abs(Vn.*phi_x ./ (abs_grad_phi+dx*dx*(abs_grad_phi == 0)));
H2_abs = abs(Vn.*phi_y ./ (abs_grad_phi+dx*dx*(abs_grad_phi == 0)));
H1_abs = H1_abs(4:end-3,4:end-3);
H2_abs = H2_abs(4:end-3,4:end-3);

delta = Vn.*abs_grad_phi;
delta = delta(4:end-3,4:end-3);
```

```
function [data_x] = der_ENO3_minus(data, dx)
data_x = zeros(size(data));

data(3) = 2*data(4)-data(5);
data(2) = 2*data(3)-data(4);
data(1) = 2*data(2)-data(3);
data(end-2) = 2*data(end-3)-data(end-4);
data(end-1) = 2*data(end-2)-data(end-3);
data(end) = 2*data(end-1)-data(end-2);

D1 = (data(2:end)-data(1:end-1))/dx;
D2 = (D1(2:end)-D1(1:end-1))/dx;
absD2 = abs(D2);
D3 = (D2(2:end)-D2(1:end-1))/dx;
absD3 = abs(D3);

for i=1:(length(data)-6)
    k = i-1;
    Q1p = D1(k+3);
    if absD2(k+2) <= absD2(k+3)
        kstar = k-1;
        c = D2(k+2)/2;
    else
        kstar = k;
        c = D2(k+3)/2;
    end
    Q2p = c*(2*(i-k)-1)*dx;
    if absD3(kstar+2) <= absD3(kstar+3)
        cstar = D3(kstar+2)/3;
    else
        cstar = D3(kstar+3)/3;
    end
    Q3p = cstar*( 3*(i-kstar)*(i-kstar) - 6*(i-kstar) + 2 )*dx*dx;
    data_x(i+3) = Q1p+Q2p+Q3p;
end
```

```
function [data_x] = der_ENO3_plus(data, dx)
data_x = zeros(size(data));

data(3) = 2*data(4)-data(5);
data(2) = 2*data(3)-data(4);
```

```
data(1) = 2*data(2)-data(3);
data(end-2) = 2*data(end-3)-data(end-4);
data(end-1) = 2*data(end-2)-data(end-3);
data(end) = 2*data(end-1)-data(end-2);

D1 = (data(2:end)-data(1:end-1))/dx;
D2 = (D1(2:end)-D1(1:end-1))/dx;
absD2 = abs(D2);
D3 = (D2(2:end)-D2(1:end-1))/dx;
absD3 = abs(D3);

for i=1:(length(data)-6)
    k = i;
    Q1p = D1(k+3);
    if absD2(k+2) <= absD2(k+3)
        kstar = k-1;
        c = D2(k+2)/2;
    else
        kstar = k;
        c = D2(k+3)/2;
    end
    Q2p = c*(2*(i-k)-1)*dx;
    if absD3(kstar+2) <= absD3(kstar+3)
        cstar = D3(kstar+2)/3;
    else
        cstar = D3(kstar+3)/3;
    end
    Q3p = cstar*( 3*(i-kstar)*(i-kstar) - 6*(i-kstar) + 2 )*dx*dx;
    data_x(i+3) = Q1p+Q2p+Q3p;
end
```

```
function [der] = select_der_normal(Vn, der_minus, der_plus)

der = zeros(size(der_plus));

for i=1:numel(Vn)
    Vn_der_m = Vn(i)*der_minus(i);
    Vn_der_p = Vn(i)*der_plus(i);

    if Vn_der_m <= 0 & Vn_der_p <= 0
        der(i) = der_plus(i);
    elseif Vn_der_m >= 0 & Vn_der_p >= 0
        der(i) = der_minus(i);
    elseif Vn_der_m <= 0 & Vn_der_p >= 0
        der(i) = 0;
    elseif Vn_der_m >= 0 & Vn_der_p <= 0
        if abs(Vn_der_p) >= abs(Vn_der_m)
            der(i) = der_plus(i);
        else
            der(i) = der_minus(i);
        end
```

```
    end
end
```

Chapter 5

Mesh generator in MATLAB

In this Chapter, we present the a simple mesh generator which is described in a few dozen lines of MATLAB [26]. We could offer detail explanation for each implementations, but our chief hope is that users will take this code as a starting point for their own work. So when you sufficiently understand the below contents, you should study more advanced method.

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

5.1. Force Equilibrium

For the actual mesh generation, our iterative technique is based on the physical analogy between a mesh and a truss structure. Mesh points are nodes of the truss. Assuming an appropriate force-displacement function for the bars in the truss at each iteration, we solve for equilibrium.

For example, the below truss structure in figure 5.2 consider the 6 mesh points which are setting by 5 point fixed in equidistance and 1 point is laid. Let's thick a kind of spring force f_k for each bar. We wish to change the only 1 point to be the below summation is zero.

$$f = \sum_{k=1}^{5} f_k \tag{5.1}$$

Figure 5.2 shows the process to find the position of one point to be the force equilibrium state.

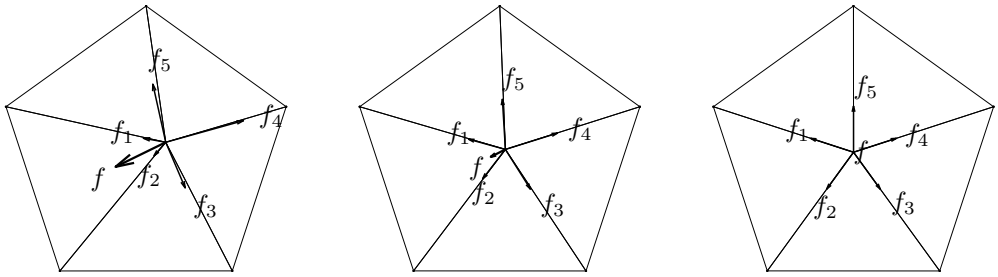


FIGURE 5.1. Example of the process to the force equilibrium.

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. In the physical model, the edges of the triangles correspond to bars, and the points correspond to joints of the truss. Each bar has a force-displacement relationship $f(l, l_0)$ depending on its current length l and its expected 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 are found by solving for a static force equilibrium in the structure. To solve for the force equilibrium, collect the x- and y-coordinates of all N mesh points into an $N \times 2$ array $\boldsymbol{p}$:

$$\boldsymbol{p} = \Big[\ x,\ y\ \Big]. \tag{5.2}$$

And we can write the force vector $\boldsymbol{F}(\boldsymbol{p})$:

$$\boldsymbol{F}(\boldsymbol{p}) = \Big[\boldsymbol{F}_{int,x}(\boldsymbol{p}), \boldsymbol{F}_{int,y}(\boldsymbol{p})\Big] + \Big[\boldsymbol{F}_{ext,x}(\boldsymbol{p}), \boldsymbol{F}_{ext,y}(\boldsymbol{p})\Big] \tag{5.3}$$

where $\boldsymbol{F}_{int}$ contains the internal forces from the bars, and $\boldsymbol{F}_{ext}$ are the external forces generated by 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.

5.2. Delaunay Triangulation

Any set of points in the xy-plane can be triangulated by the Delaunay algorithm [32]. 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.

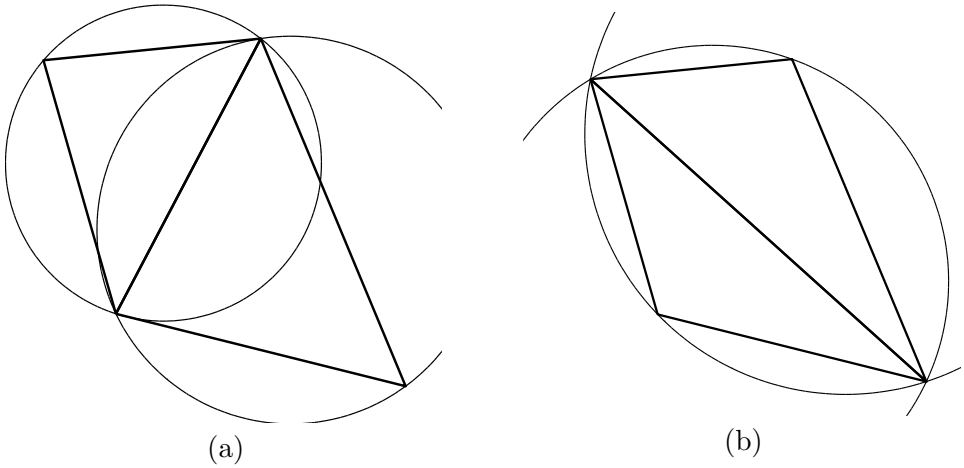


FIGURE 5.2. Delaunay Triangulation: (a) Correct and (b) Wrong

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 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 forces move the nodes, and (iteratively) the Delaunay triangulation algorithm adjusts the topology (it decides the edges). Those are the two essential steps. Other codes use Laplacian smoothing [28] 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 [29]. We mention Triangle [30] as a robust and freely available Delaunay refinement code.

5.3. The Algorithm

The system $\boldsymbol{F}(\boldsymbol{p}) = 0$ has to be solved for a set of equilibrium positions $\boldsymbol{p}$. A simple approach to solve $\boldsymbol{F}(\boldsymbol{p}) = 0$ is to introduce an artificial time-dependence. For some $\boldsymbol{p}(0) = \boldsymbol{p}_0$, we consider the system of ODEs

$$\frac{d\boldsymbol{p}}{dt} = \boldsymbol{F}(\boldsymbol{p}(t)), \qquad t \geq 0. \tag{5.4}$$

If a steady stable solution is found, it satisfies our system $\boldsymbol{F}(\boldsymbol{p}) = 0$. The system equation 5.4 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

$$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n + \Delta t \boldsymbol{F}(\boldsymbol{p}_n). \tag{5.5}$$

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.

The implementation uses this linear response for the repulsive forces but it allows no attractive forces:

$$f(l, l_0) = \begin{cases} k(l_0 - l) & \text{if } l \leq l_0, \\ 0 & \text{if } l > l_0. \end{cases} \tag{5.6}$$

The function $k(l_0 - l)$ models ordinary linear springs. There are many alternatives for the force function $f(l, l_0)$ in each bar, and several choices have been investigated [29], [31]. It is reasonable to require $f = 0$ for $l = l_0$. 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 $f(l, l_0)$ 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

5.4. Mesh Size Function

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. The edge lengths should be close to the relative size $h(x)$ specified by the user (the lengths are nearly equal when the user chooses $h(x) = 1$). 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. The solution method may require small elements close to a singularity to give good global accuracy. A uniform mesh with these small elements would require too many nodes.

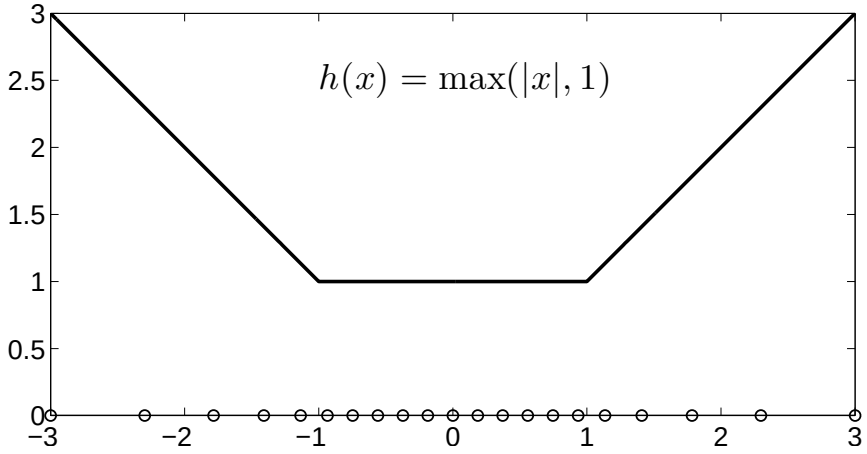


FIGURE 5.3. The relation between the mesh size function $h(x)$ and generating mesh point length

In the 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 ($x = 1$). Figure 5.3 shows the relationship the mesh size function h and the actual mesh size. 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):

$$\text{Scaling factor} = \left(\frac{\sum l_i^2}{\sum h(x_i, y_i)^2} \right)^{1/2}. \tag{5.7}$$

We 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 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.

5.5. Mesh Generation in Two Dimension

The complete source code for the two-dimensional mesh generator is in below. 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 $\boldsymbol{p}$.
- 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
```

The input arguments are as follows:

- The geometry is given as a signed distance function **fd**.
- The mesh size function $h(x, y)$ is given as a function **fh**.
- The parameter **h0** is the distance between points in the initial distribution $\boldsymbol{p}_0$.
- The bounding box for the region is an array $\mathbf{bbox} = [\mathbf{x_{min}}, \mathbf{y_{min}}; \mathbf{x_{max}}, \mathbf{y_{max}}]$.
- The fixed node positions are given as an array **pfix**
- Additional parameters to the functions **fd** and **fh** can be given in the last arguments **varargin**.

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** which is k value in equation 5.6. The time step in Eulers method 5.5 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.

Now we describe steps 1 to 8 in the **distmesh2d** algorithm.

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

   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 $N \times 2$ array $\boldsymbol{p}$.

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 $\boldsymbol{p}$ 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 users array of fixed nodes is placed in the first rows of $\boldsymbol{p}$.

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. 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 equation 5.7 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 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
```

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; % 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. 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
```

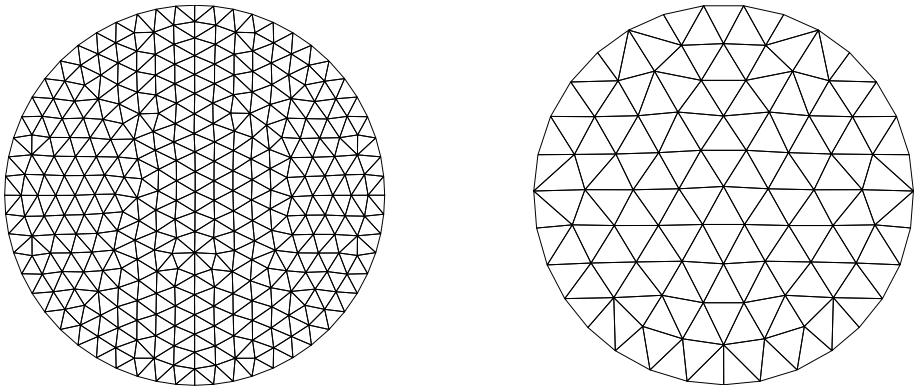
This criterion is sometimes too tight, and a high-quality mesh is often achieved long before termination. In these cases, the program can be interrupted manually, or other tests can be used. One simple but efficient test is to compute all the element qualities (see below), and terminate if the smallest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

5.6. Examples in two dimension

5.6.1. Unit Circle. We work directly with $d = \sqrt{x^2+y^2}-1$, which is a signed distance function of the unit circle $x^2 + y^2 = 1$. For a uniform mesh, $h(x, y) = 1$ is used for mesh generation. The two function is stored in the Cartesian mesh grid with spatial size $\Delta x = \Delta y = 0.1$. The circle has bounding box $-1 \le x \le 1$, $-1 \le y \le 1$, with no fixed points. Actually, you can set the bounding box more large or small. A mesh with element size approximately $h_0 = 0.1$ is generated with below code.

```
[xx yy] = meshgrid(-2:0.1:2, -2:0.1:2);
dd = sqrt(xx.^2 + yy.^2) - 1;
hh = ones(size(dd));
[p,t]=distmesh2d(@dmatrix,@hmatrix,0.1,[-1,-1;1,1],[], xx, yy, dd, hh);
```

5.6.2. Unit Circle with Hole. Removing a circle of radius 0.5 from the unit circle gives the distance function $d(x, y) = \max(\sqrt{x^2+y^2} - 1, 0.5 - \sqrt{x^2+y^2})$. And this distance function is equivalent as $d(x, y) = |0.75 - \sqrt{x^2+y^2}| - 0.25$. And another example has a distance function $d(x, y) = \max(\sqrt{x^2+y^2} - 1, 0.5 -$



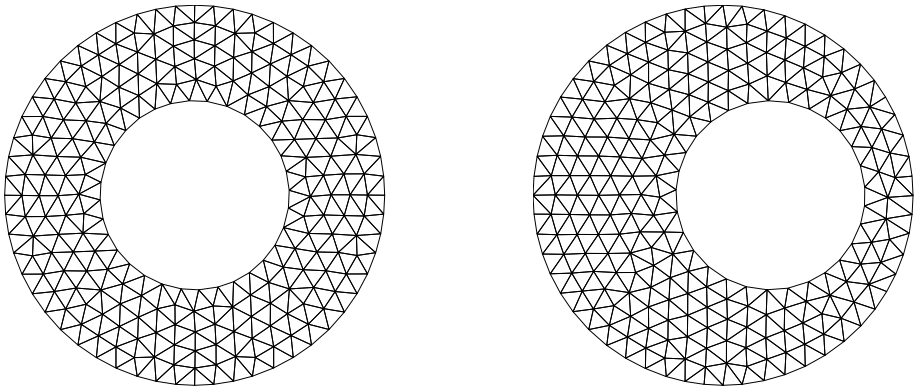
(a) $h_0 = 0.1$ (b)$h_0 = 0.2$

FIGURE 5.4. Unit circle mesh

$\sqrt{(x-0.25)^2+y^2}$). For a uniform mesh, $h(x,y) = 1$ is used for mesh generation. The two function is stored in the Cartesian mesh grid with spatial size $\Delta x = \Delta y = 0.1$. The circle has bounding box $-1 \le x \le 1$, $-1 \le y \le 1$, with no fixed points. Actually, you can set the bounding box more large or small. A mesh with element size approximately $h_0 = 0.1$ is generated with below code.

```
[xx yy] = meshgrid(-2:0.1:2,-2:0.1:2);
da=sqrt(xx.^2 + yy.^2)-1;
db=0.5-sqrt(xx.^2+yy.^2);
dd=max(da,db);
hh=ones(size(dd));
[p,t]=distmesh2d(@dmatrix,@hmatrix,0.1,[-1,-1;1,1],[],xx,yy,dd,hh);
```

5.6.3. Square with Hole. We can replace the outer circle with a square, keeping the circular hole. For make the rectangular cone, consider the distance function $d_A(x,y) = \max(|x|-1, |y|-1)$ which has the inside region $-1 \le x \le 1$ and $-1 \le y \le 1$. And the distance function of outer circle is $d_B(x,y) = 0.5 - \sqrt{x^2+y^2}$. And the intersection of $d_A(x,y)$ and $d_B(x,y)$ give the geometry of the square with hole. For a uniform mesh, $h(x,y) = 1$ is used for mesh generation. The two function is stored in the Cartesian mesh grid with spatial size $\Delta x = \Delta y = 0.1$. The circle has bounding box $-1 \le x \le 1$, $-1 \le y \le 1$, with 4 corner fixed points. Actually, you can

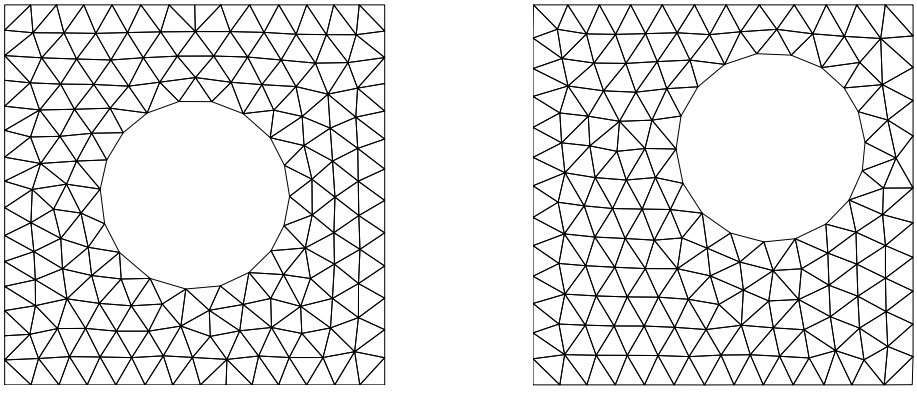


(a) (b)

FIGURE 5.5. Unit Circle with Hole

set the bounding box more large or small. A mesh with element size approximately $h_0 = 0.1$ is generated with below code.

```
[xx yy] = meshgrid(-2:0.1:2,-2:0.1:2);
da=max(abs(xx)-1,abs(yy)-1);
db=0.5-sqrt(xx.^2+yy.^2);
dd=max(da,db);
hh=ones(size(dd));
[p,t]=distmesh2d(@dmatrix,@hmatrix,0.15,[-1,-1;1,1],[-1,-1;-1,1;1,1;1,-1],xx,yy,dd,hh);
```



(a) (b)

FIGURE 5.6. Square with Hole

5.7. Mesh Generation in Three Dimension

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

The two dimensional mesh generator extends to any dimension by Per-Olof Persson. And the code **distmeshnd.m** is given in **www-math.mit.edu/∼ persson/mesh**. We just modify the "distmesh2d" code for easy understanding of mesh generation in three spatial dimensional.

```
function [p,t]=distmesh3d(fdist,fh,h0,box,fix,varargin)
ptol=.001; ttol=.1; Fscale=1.1; deltat=.1; geps=1e-1*h0; deps=sqrt(eps)*h0;

% 1. Create initial distribution in bounding box
[x,y,z]=ndgrid(box(1,1):h0:box(2,1),box(1,2):h0:box(2,2),box(1,3):h0:box(2,3));
p=[x(:),y(:),z(:)];

% 2. Remove points outside the region, apply the rejection method
p=p(feval(fdist,p,varargin{:})<geps,:);
r0=feval(fh,p,varargin{:});
p=[fix; p(rand(size(p,1),1)<min(r0)^3./r0.^3,:)];
N=size(p,1);

count=0; pold=inf;
while 1
    % 3. Retriangulation by Delaunay
    if max(sqrt(sum((p-pold).^2,2)))>ttol*h0
        pold=p;
        t=delaunayn(p);
        pmid=(p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:)+p(t(:,4),:))/4;
        t=t(feval(fdist,pmid,varargin{:})<-geps,:);
        % 4. Describe each edge by a unique pair of nodes
        pair=[t(:,[1,2]);t(:,[1,3]);t(:,[1,4]);t(:,[2,3]);t(:,[2,4]);t(:,[3,4])];
        pair=unique(sort(pair,2),'rows');
        % 5. Graphical output of the current mesh
        if mod(count,5)==0
            simpplot(p,t,'p(:,2)>0');
            view(-45,15); drawnow
        end
    else
        count=count+1;
    end

    % 6. Move mesh points based on edge lengths L and forces F
    bars=p(pair(:,1),:)-p(pair(:,2),:);
    L=sqrt(sum(bars.^2,2));
```

```
    L0=feval(fh,(p(pair(:,1),:)+p(pair(:,2),:))/2,varargin{:});
    L0=L0*Fscale*(sum(L.^3)/sum(L0.^3))^(1/3);
    F=max(L0-L,0);
    Fbar=[bars,-bars].*repmat(F./L,1,2*3);
    dp=full(sparse(pair(:,[ones(1,3),2*ones(1,3)]),ones(size(pair,1),1)*[1:3,1:3]...
        ,Fbar,N,3));
    dp(1:size(fix,1),:)=0;
    p=p+deltat*dp;

    % 7. Bring outside points back to the boundary
    d=feval(fdist,p,varargin{:}); ix=d>0;
    gradd=zeros(sum(ix),3);
    dgrad2 = zeros(sum(ix),1);
    for ii=1:3
        a=zeros(1,3);
        a(ii)=deps;
        d1x=feval(fdist,p(ix,:)+ones(sum(ix),1)*a,varargin{:});
        gradd(:,ii)=(d1x-d(ix))/deps;
    end
    p(ix,:)=p(ix,:)-d(ix)*ones(1,3).*gradd;

    % 8. Termination criterion
    if max(deltat*sqrt(sum(dp(d<-geps,:).^2,2)))<ptol*h0, break; end
end
```

Now we describe steps 1 to 8 in the **distmesh3d** algorithm.

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

   ```
   [x,y,z]=ndgrid(box(1,1):h0:box(2,1),box(1,2):h0:box(2,2),box(1,3):h0:box(2,3));
   p=[x(:),y(:),z(:)];
   ```

   The mesh grid function generates a rectangular grid, given as two vectors x and y of node coordinates. The coordinates are stored in the $N \times 3$ array $\boldsymbol{p}$.

2. The next step removes all nodes outside the desired geometry. Only the interior points with negative distances (allowing a tolerance **geps**) are kept.

   ```
   p=p(feval(fdist,p,varargin{:})<geps,:);
   ```

   Then we evaluate $h(x, y)$ at each node and reject points with a probability proportional to $1/h(x, y)^2$. The array of fixed nodes is placed in the first rows of $\boldsymbol{p}$.

   ```
   r0=feval(fh,p,varargin{:});
   ```

```
p=[fix; p(rand(size(p,1),1)<min(r0)^3./r0.^3,:)];
N=size(p,1);
```

L. 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.

```
pold=inf;
while 1
    ...
end
```

3. 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)))>ttol*h0
    pold=p;
    t=delaunayn(p);
    pmid=(p(t(:,1),:)+p(t(:,2),:)+p(t(:,3),:)+p(t(:,4),:))/4;
    t=t(feval(fdist,pmid,varargin{:})<-geps,:);
    ...
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 $d < 0$, then 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 6 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.

```
pair=[t(:,[1,2]);t(:,[1,3]);t(:,[1,4]);t(:,[2,3]);t(:,[2,4]);t(:,[3,4])];
pair=unique(sort(pair,2),'rows');
```

5. The next two lines give graphical output after each retriangulation. The code about the **simpplot** function is written in Appendix.

```
simpplot(p,t,'p(:,2)>0');
view(-45,15); drawnow
```

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

```
bars=p(pair(:,1),:)-p(pair(:,2),:);
L=sqrt(sum(bars.^2,2));
```

The desired lengths **L0** come from evaluating $h(x, y)$ at the midpoint of each bar. We multiply by the scaling factor in equation 5.7 and the fixed factor **Fscale**, to ensure that most bars give repulsive forces $f > 0$ in **F**.

```
L0=feval(fh,(p(pair(:,1),:)+p(pair(:,2),:))/2,varargin{:});
L0=L0*Fscale*(sum(L.^3)/sum(L0.^3))^(1/3);
F=max(L0-L,0);
```

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 **Fbar**, 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.

```
Fbar=[bars,-bars].*repmat(F./L,1,2*3);
dp=full(sparse(pair(:,[ones(1,3),2*ones(1,3)]),ones(size(pair,1),1)*[1:3,1:3]...
    ,Fbar,N,3));
dp(1:size(fix,1),:)=0;
p=p+deltat*dp;
```

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 $\mathbf{p}$, 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(fdist,p,varargin{:}); ix=d>0;
gradd=zeros(sum(ix),3);
dgrad2 = zeros(sum(ix),1);
for ii=1:3
    a=zeros(1,3);
    a(ii)=deps;
    d1x=feval(fdist,p(ix,:)+ones(sum(ix),1)*a,varargin{:});
    gradd(:,ii)=(d1x-d(ix))/deps;
end
p(ix,:)=p(ix,:)-d(ix)*ones(1,3).*gradd;
```

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

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

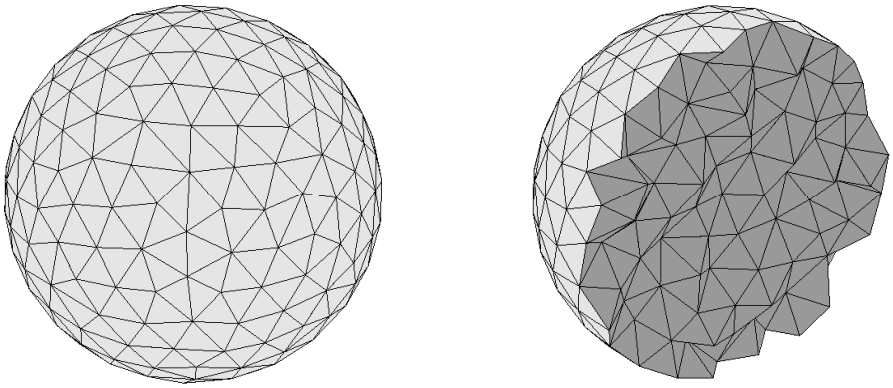
This criterion is sometimes too tight, and a high-quality mesh is often achieved long before termination. In these cases, the program can be interrupted manually, or other tests can be used. One simple but efficient test is to compute all the element qualities (see below), and terminate if the smallest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

5.8. Example in three dimension

5.8.1. Unit Sphere. The sphere uses nearly the same manner as the circle. The corresponding signed distance function $d(x, y, z) = \sqrt{x^2 + y^2 + z^2} - 1$ of the unit sphere is used. The mesh size function $h(x, y, z) = 1$ is used for the simplicity. The circle has bounding box $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, and $-1 \leq z \leq 1$, with no

fixed points. And the three dimensional uniform Cartesian grid with $\Delta x = \Delta y = \Delta z = 0.1$. The expected mesh size is chosen $h_0 = 0.2$.

```
[xx,yy,zz] = ndgrid(-2:0.1:2,-2:0.1:2,-2:0.1:2);
dd = sqrt(xx.^2 + yy.^2 + zz.^2) - 1;
hh = ones(size(dd));
[p,t]=distmesh3d(@dmatrix3d,@hmatrix3d,0.2,[-1,-1,-1;1,1,1],[],xx,yy,zz,dd,hh);
```



(a) (b)

FIGURE 5.7. Unit Sphere mesh

5.9. Caution

There is a constraint between the mesh size and the element size. The Cartesian grid is given, so that the mesh size is fixed. For choosing the element size, we has a constraint as follow: $h_0 > \sqrt{(\Delta x)^2 + (\Delta y)^2}$ in two spacial dimension, and $h_0 > \sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2}$ in three spacial dimension. The typical example is shown below codes. This is same simulation in section 5.6.1 without the spatial size, $\Delta x = \Delta y = 0.2$. Although the result figure is close the solution, it is not converge.

```
[xx yy] = meshgrid(-2:0.2:2, -2:0.2:2);
dd = sqrt(xx.^2 + yy.^2) - 1;
hh = ones(size(dd));
```

```
[p,t]=distmesh2d(@dmatrix,@hmatrix,0.1,[-1,-1;1,1],[], xx, yy, dd, hh);
```

5.10. 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 $|\nabla h(x)| \leq g$ 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 $h(y) = h_0$ in a convex domain can be extended as $h(x) = h_0 + g|x - y|$, and similarly for other shapes such as edges.

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 [35]. When the iterations converge, the solution satisfies $|\nabla h(x)| \leq g$ 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 [37]. This data structure is used in the quadtree meshing algorithm [36], 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.

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

$$\frac{\partial h}{\partial t} + |\nabla h| = \min(|\nabla h|, g), \tag{5.8}$$

with initial condition

$$h(t=0,x) = h_0(x). \tag{5.9}$$

When $|\nabla h| \leq g$, equation (5.8) gives that $h_t = 0$, 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 R^n and constant g, we can derive an analytical expression for the solution to equation (5.8), showing that it is indeed the optimal solution:

THEOREM 5.10.1. *Let $\Omega \subset R^n$ 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*

$$\begin{cases} \frac{\partial h}{\partial t} + |\nabla h| = min(|\nabla h|, g), & (x,t) \in \Omega \times I, \\ h(x,t)|_{t=0} = h_0(x), & x \in \Omega. \end{cases} \tag{5.10}$$

is

$$h(x) = \min_y (h_0(y) + g|x-y|). \tag{5.11}$$

Note that the solution equation (5.11) is composed of infinitely many point-source solutions as described before. We could in principle define an algorithm based on equation (5.11) 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 can solve an equation (5.10) using efficient Hamilton-Jacobi solvers.

5.11. Examples for Gradient Limiting

5.11.1. Unit Circle. A signed distance function $d = \sqrt{x^2+y^2} - 1$ is used, which has the interface, the unit circle $x^2 + y^2 = 1$. To make fine near the interface for generating more close the curve, the mesh size function is chosen $h(x,y) = 1 - 5d(x,y)$. The Cartesian mesh grid with spatial size $\Delta x = \Delta y = 0.05$ is set. The

circle has bounding box $-1 \le x \le 1$, $-1 \le y \le 1$, with no fixed points. Actually, you can set the bounding box more large or small. We wish to get the mesh point with element size approximately $h_0 = 0.05$ in the boundary.

```
[xx yy] = meshgrid(-2:0.05:2, -2:0.05:2);
dd = sqrt(xx.^2 + yy.^2) - 1;
hh = ones(size(dd)) - 5.0*dd;
[p,t]=distmesh2d(@dmatrix,@hmatrix,0.05,[-1,-1;1,1],[], xx, yy, dd, hh);
```

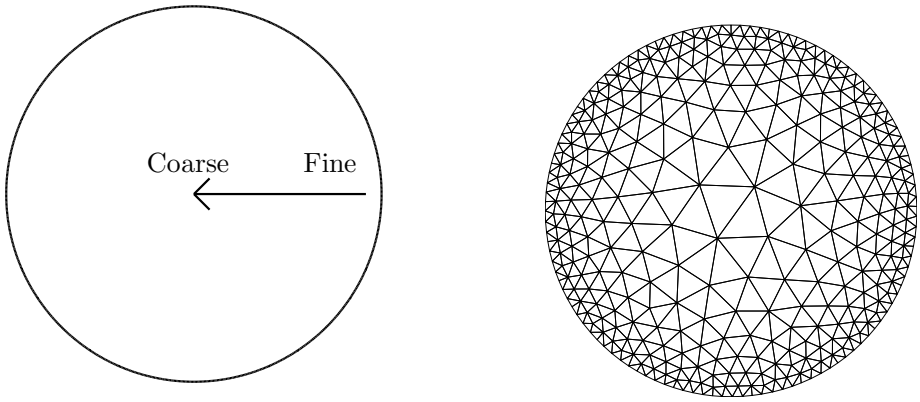


FIGURE 5.8

5.11.2. Unit Circle and Square with Hole. Removing a circle of radius 0.5 from the unit circle gives the distance function $d(x, y) = \max(\sqrt{x^2 + y^2} - 1, 0.5 - \sqrt{x^2 + y^2})$. For a adaptive mesh, $h(x, y) = 4\sqrt{x^2 + y^2}$ is used for mesh generation. The Above mesh size function has 1 value near the inner circle, and 4 value near the outer circle. The two function is stored in the Cartesian mesh grid with spatial size $\Delta x = \Delta y = 0.05$. The circle has bounding box $-1 \le x \le 1$, $-1 \le y \le 1$, with no fixed points. Actually, you can set the bounding box more large or small. In here, an element size $h_0 = 0.05$ means the mesh size near the inner hole.

```
[xx yy] = meshgrid(-2:0.05:2,-2:0.05:2);
da=sqrt(xx.^2 + yy.^2)-1;
db=0.5-sqrt(xx.^2+yy.^2);
dd=max(da,db);
hh=4*sqrt(xx.^2+yy.^2);
[p,t]=distmesh2d(@dmatrix,@hmatrix,0.05,[-1,-1;1,1],[],xx,yy,dd,hh);
```

We can replace the outer circle with a square, keeping the circular hole. For make the rectangular cone, consider the distance function $d_A(x, y) = \max(|x| - 1, |y| - 1)$ which has the inside region $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$. And the distance function of outer circle is $d_B(x, y) = 0.5 - \sqrt{x^2 + y^2}$. And the intersection of $d_A(x, y)$ and $d_B(x, y)$ give the geometry of the square with hole. The curvature near the inner circle is high more in the square, so the mesh size function $h(x, y) = 4\sqrt{x^2 + y^2}$ is considerable. The two function is stored in the Cartesian mesh grid with spatial size $\Delta x = \Delta y = 0.05$. The circle has bounding box $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, with 4 corner fixed points. Actually, you can set the bounding box more large or small. The initially $h_0 = 0.05$ is used, but the rejection method and the mesh size function effect like as figure 5.9.

```
[xx yy] = meshgrid(-2:0.05:2,-2:0.05:2);
da=max(abs(xx)-1,abs(yy)-1);
db=0.5-sqrt(xx.^2+yy.^2);
dd=max(da,db);
hh=4*sqrt(xx.^2+yy.^2);
[p,t]=distmesh2d(@dmatrix,@hmatrix,0.05,[-1,-1;1,1],[-1,-1;-1,1;1,1;1,-1],xx,yy,dd,hh);
```

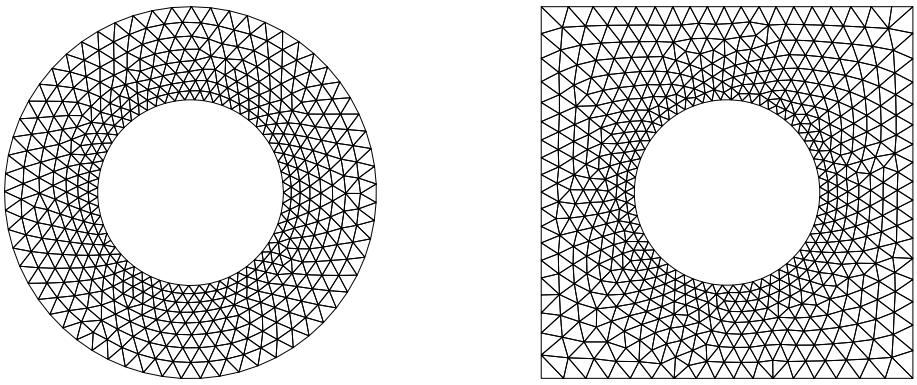


FIGURE 5.9

5.11.3. Cylinder with Spherical Hole. For a cylinder with radius 1 and the height from $z = -1$ to $z = 1$, we write d_1, d_2, and d_3 for the cylinder surface and the top and bottom.

$$d_1(x, y, z) = \sqrt{x^2 + y^2} - 1, \tag{5.12}$$

$$d_2(x, y, z) = z - 1, \tag{5.13}$$

$$d_3(x, y, z) = -z - 1. \tag{5.14}$$

And approximate distance function is then formed by intersection

$$\widetilde{d} = \max(d_1, \max(d_2, d_3)). \tag{5.15}$$

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 distance function using distances to the two curves.

$$d_4(x, y, z) = \sqrt{d_1(x, y, z)^2 + d_2(x, y, z)^2}, \tag{5.16}$$

$$d_5(x, y, z) = \sqrt{d_1(x, y, z)^2 + d_3(x, y, z)^2}. \tag{5.17}$$

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.

$$d = \begin{cases} d_4, & \text{if } d_1 > 0 \text{ and } d_2 > 0, \\ d_5, & \text{if } d_1 > 0 \text{ and } d_3 > 0, \\ \widetilde{d}, & \text{otherwise.} \end{cases} \tag{5.18}$$

Figure 5.10 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 $h_0 = 0.1$. Commend code is:

```
[p,t]=distmesh3d(@fd,@fh,0.1,[-1,-1,-1;1,1,1],[]);
```

In this case, we did not use the Cartesian grid, alternatively use the analytical distance function as below function code **fd** and **fh**.

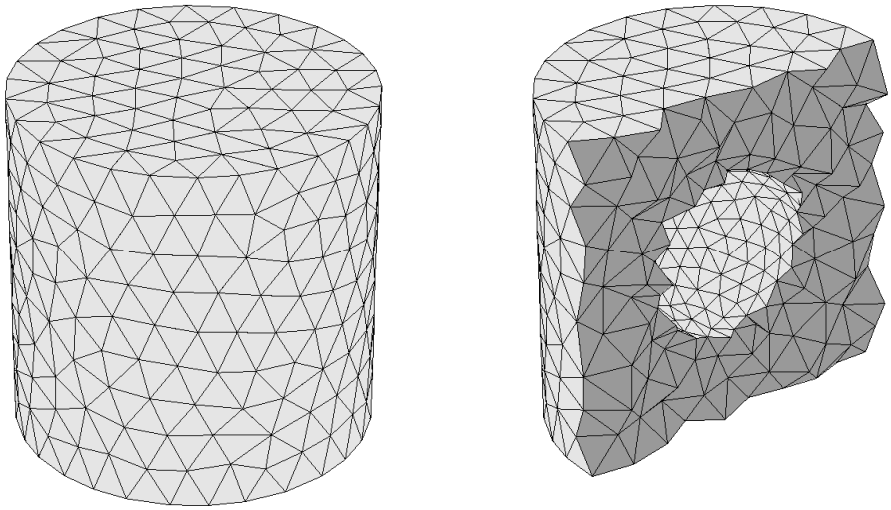
```
function d=fd(p)

r=sqrt(p(:,1).^2+p(:,2).^2);
z=p(:,3);

d1=r-1;
d2=z-1;
d3=-z-1;
d4=sqrt(d1.^2+d2.^2);
d5=sqrt(d1.^2+d3.^2);
d=max(max(d1,d2),d3);
ix=d1>0 & d2>0;
d(ix)=d4(ix);
ix=d1>0 & d3>0;
d(ix)=d5(ix);

dsphere = 0.5-sqrt(p(:,1).^2+p(:,2).^2+p(:,3).^2);
d=max(d,dsphere);
```

```
function h=fh(p)
h1=4*sqrt(sum(p.^2,2))-1;
h=min(h1,2);
```



(a) (b)

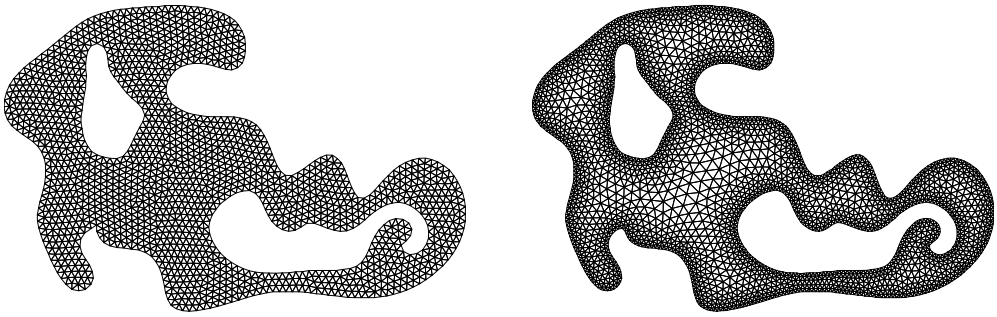
FIGURE 5.10. Cylinder with Spherical Hole

Chapter 6

Results of Mesh Generation

6.1. Lake-Shaped Map

In [27], author show the various methods for making automatical mesh size function, and show the simulation. Figure 6.1 is taken in the reference figure [27] to show the evolution. The image segmentation method in [38] is used to get the figure. Then we evolve the a singed distance function from the phase-field data which is given in the segment method. The 512×512 mesh grid is used in the computational domain $(0,1) \times (0,1)$. Figure 6.1 (a) is uniform mesh and (b) is adaptive mesh for making detail in the curve.



(a) (b)

FIGURE 6.1. Generated mesh

6.2. Red Blood Cell

In [34], Timothy and Philip present two mathematical models that describe human red blood cells (RBCs). In this reference, there are corresponding Cartesian equations of a discocyte and somatocye.

6.2.1. Discocyte. Discocyte shapes are generated by the corresponding Cartesian equation as follows:

$$\phi(x, y, z) = (x^2 + y^2)^2 + P(x^2 + y^2) + Qz^2 + R \tag{6.1}$$

where the coefficients P, Q, and R are given by

$$\begin{aligned} P &= (1 - 2m)d^2/4m, \\ Q &= (1 - m)d^4/16a^2m, \\ R &= (m - 1)d^4/16m). \end{aligned}$$

We use the uniform Cartesian mesh grid with increment $\Delta x = \Delta y = \Delta z = 0.1$ and the computational domain is $[-4.5, 4.5] \times [-4.5, 4.5] \times [-1.5, 1.5]$. And the initial mesh size of the mesh point is $h_0 = 0.3$.

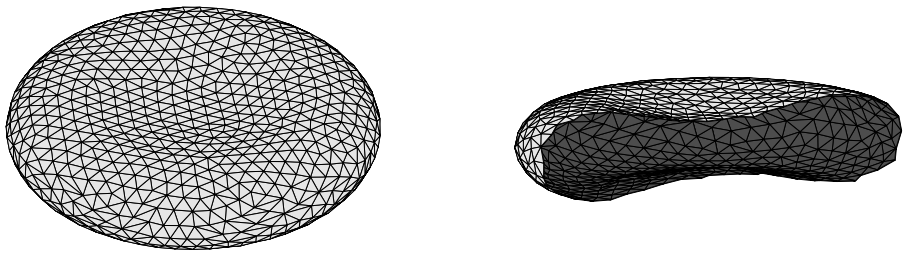


FIGURE 6.2

6.2.2. Stomatocyte. Stomatocytes are modelled in two halves where $p = 3$ and 2 for the top and bottom of the cell, respectively. The Cartesian expression of $p = 3$ is:

$$\phi(x, y, z) = (x^2 + y^2)^4 + A(x^2 + y^2)^3 + B(x^2 + y^2)^2 + C(x^2 + y^2) + Dz^2 + E = 0 \quad (6.2)$$

where A, B, C, D, and E are

$$\begin{aligned} A &= -(4m_1 - 3)d^2/4m_1, \\ B &= 3(1 - 3m_1 + 2m_2)d^4/16m_1^2, \\ C &= -(m_1 - 1)^2(4m_1 - 1)d^6/64m_1^3, \\ D &= (1 - m_1)^3 d^8/256a_1^2 m_1^3, \\ E &= (m_1 - 1)^3 d^8/256m_1^3. \end{aligned} \quad (6.3)$$

Similarly the Cartesian expression when $p = 2$ is

$$\phi(x, y, z) = (x^2 + y^2)^3 + F(x^2 + y^2)^2 + G(x^2 + y^2) + Hz^2 + I = 0 \quad (6.4)$$

where the coefficients F, G, H, and I are

$$\begin{aligned} F &= -(3m_2 - 2)d^2/4m_2, \\ G &= (1 - 4m_2 + 3m_2^2)d^4/16m_2^2, \\ H &= (1 - m_2)^2 d^6/64a_2^2 m_2^2, \\ I &= -(m_2 - 1)^2 d^6/64m_2^2. \end{aligned} \quad (6.5)$$

In equations 6.3 and 6.5, m_1 and m_2 denote the specific values of m that generate each half of the stomatocyte surface. We use the uniform Cartesian mesh grid with increment $\Delta x = \Delta y = \Delta z = 0.1$ and the computational domain is $[-4.5, 4.5] \times [-4.5, 4.5] \times [-1.5, 1.5]$. And the initial mesh size of the mesh point is $h_0 = 0.3$.

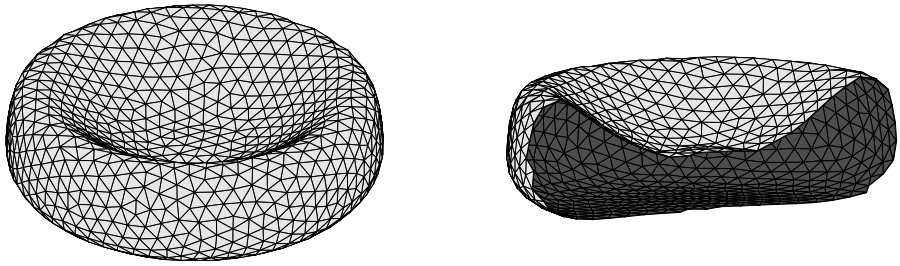


FIGURE 6.3

Chapter 7

Conclusion

We devote the represent the mesh generators and show the intuitive examples to easy access of the generating code and for understanding deeply. And we organize the MATLAB code for making a signed distance function given some initial data. The evolved data is stored in the uniform Cartesian grid points and it can be composed with the Distmesh code. Therefore, we are applied in two cases: the lake-shaped map and the red blood cells. Apart from being simple of the code, it seems that the algorithm generates meshes of high quality.

Appendix A

The used codes

```
function simpplot(p,t,expr,bcol,icol)

if nargin<4, bcol=.9*ones(1,3); end
if nargin<5, icol=.6*ones(1,3); end

dim=size(p,2);
switch dim
    case 2
        trimesh(t,p(:,1),p(:,2),0*p(:,1),'facecolor','none','edgecolor','k');
        view(2)
        axis equal
        axis off
    case 3
        tri1=surftri(p,t);
        if nargin>2 & ~isempty(expr)
            incl=find(eval(expr));
            t=t(any(ismember(t,incl),2),:);
            tri1=tri1(any(ismember(tri1,incl),2),:);
            tri2=surftri(p,t);
            tri2=setdiff(tri2,tri1,'rows');
            h=trimesh(tri2,p(:,1),p(:,2),p(:,3));
            set(h,'facecolor',icol,'edgecolor','k');
            hold on
        end
        h=trimesh(tri1,p(:,1),p(:,2),p(:,3));
        hold off
        set(h,'facecolor',bcol,'edgecolor','k');
        axis equal
        cameramenu
    otherwise
        error('Unimplemented dimension.');
end
```

```
function tri=surftri(p,t)

faces=[t(:,[1,2,3]);
    t(:,[1,2,4]);
    t(:,[1,3,4]);
    t(:,[2,3,4])];
```

```
node4=[t(:,4);t(:,3);t(:,2);t(:,1)];
faces=sort(faces,2);
[foo,ix,jx]=unique(faces,'rows');
vec=histc(jx,1:max(jx));
qx=find(vec==1);
tri=faces(ix(qx),:);
node4=node4(ix(qx));

v1=p(tri(:,2),:)-p(tri(:,1),:);
v2=p(tri(:,3),:)-p(tri(:,1),:);
v3=p(node4,:)-p(tri(:,1),:);
ix=find(dot(cross(v1,v2,2),v3,2)>0);
tri(ix,[2,3])=tri(ix,[3,2]);
```

```
function d=dmatrix(p,xx,yy,dd,varargin)
d=interp2(xx,yy,dd,p(:,1),p(:,2),'*linear');

function h=hmatrix(p,xx,yy,dd,hh,varargin)
h=interp2(xx,yy,hh,p(:,1),p(:,2),'*linear');

function d=dmatrix3d(p,xx,yy,zz,dd,varargin)
d=interpn(xx,yy,zz,dd,p(:,1),p(:,2),p(:,3),'*linear');

function h=hmatrix3d(p,xx,yy,zz,dd,hh,varargin)
h=interpn(xx,yy,zz,hh,p(:,1),p(:,2),p(:,3),'*linear');
```

Bibliography

[1] Bloomenthal, J., Bajaj, C., Blinn, J., Cani-Gascuel, M.-P., Rockwood, A., Wyvill, B., and Wyvill, G., Introduction to Implicit Surfaces, Morgan Kaufmann Publishers Inc., San Francisco (1997).

[2] Osher, S. and Sethian, J., Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton.Jacobi Formulations, J. Comput. Phys. 79, 12-49 (1988).

[3] Chopp, D., Computing Minimal Surfaces via Level Set Curvature Flow, J. Comput. Phys. 106, 77-91 (1993).

[4] Mulder, W., Osher, S., and Sethian, J., Computing Interface Motion in Compressible Gas Dynamics, J. Comput. Phys. 100, 209-228 (1992).

[5] Adalsteinsson, D. and Sethian, J., A Fast Level Set Method for Propagating Interfaces, J. Comput. Phys. 118, 269-277 (1995).

[6] Peng, D., Merriman, B., Osher, S., Zhao, H.-K., and Kang, M., A PDE-Based Fast Local Level Set Method, J. Comput. Phys. 155, 410-438 (1999).

[7] Merriman, B., Bence, J., and Osher, S., Motion of Multiple Junctions: A Level Set Approach, J. Comput. Phys. 112, 334-363 (1994).

[8] Bruckstein, A., On Shape from Shading, Comput. Vision Graphics Image Process. 44, 139-154 (1988).

[9] Kimmel, R. and Bruckstein, A., Shape Offsets via Level Sets, Computer Aided Design 25, 154-162 (1993).

[10] Osher, S., A Level Set Formulation for the Solution of the Dirichlet Problem for Hamilton.Jacobi Equations, SIAM J. Math. Anal. 24, 1145-1152 (1993).

[11] Rouy, E. and Tourin, A., A Viscosity Solutions Approach to Shape-From- Shading, SIAM J. Num. Anal. 29, 867-884 (1992).

[12] Sussman, M., Smereka, P., and Osher, S., A Level Set Approach for Computing Solutions to Incompressible Two-Phase Flow, J. Comput. Phys. 114, 146-159 (1994).

[13] Russo, G. and Smereka, P., A Remark on Computing Distance Functions, J. Comput. Phys. 163, 51-67 (2000).

[14] Fedkiw, R., Aslam, T., Merriman, B., and Osher, S., A Non-Oscillatory Eulerian Approach to Interfaces in Multimaterial Flows (The Ghost Fluid Method), J. Comput. Phys. 152, 457-492 (1999).

[15] Osher, S. and Sethian, J., Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton.Jacobi Formulations, J. Comput. Phys. 79, 12.49 (1988).

[16] Strikwerda, J., Finite Difference Schemes and Partial Differential Equations, Wadsworth & Brooks/Cole Advanced Books and Software, Pacific Grove, California (1989).

[17] Shu, C.W. and Osher, S., Efficient Implementation of Essentially Non- Oscillatory Shock Capturing Schemes, J. Comput. Phys. 77, 439.471 (1988).

[18] Shu, C.W. and Osher, S., Efficient Implementation of Essentially Non- Oscillatory Shock Capturing Schemes II (two), J. Comput. Phys. 83, 32.78 (1989).

[19] Heath, M., Scientific Computing, The McGraw-Hill Companies Inc. (1997).

[20] Osher, S. and Shu, C.-W., High Order Essentially Non-Oscillatory Schemes for Hamilton.Jacobi Equations, SIAM J. Numer. Anal. 28, 902.921 (1991).
[21] Liu, X.-D., Osher, S., and Chan, T., Weighted Essentially Non-Oscillatory Schemes, J. Comput. Phys. 126, 202.212 (1996).
[22] Jiang, G.-S. and Peng, D., Weighted ENO Schemes for Hamilton Jacobi Equations, SIAM J. Sci. Comput. 21, 2126.2143 (2000).
[23] Jiang, G.-S. and Shu, C.-W., Efficient Implementation of Weighted ENO Schemes, J. Comput. Phys. 126, 202.228 (1996).
[24] Fedkiw, R., Merriman, B., and Osher, S., Simplified Upwind Discretization of Systems of Hyperbolic Conservation Laws Containing Advection Equations, J. Comput. Phys. 157, 302.326 (2000).
[25] Godunov, S.K., A Finite Difference Method for the Computation of Discontinuous Solutions of the Equations of Fluid Dynamics, Mat. Sb. 47, 357-393 (1959).
[26] Persson,P.,and Strang, G. A Simple Mesh Generator in Matlab, SIAM Review, 329-346. (2004)
[27] Persson,P., PDE-based gradient limiting for mesh size functions, Proceedings of 13th international meshing roundtable, 2004, ann.jussieu.fr
[28] David A. Field. Laplacian smoothing and delaunay triangulations. Comm. in Applied Numerical Methods, 4, 709-712, (1988).
[29] 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).
[30] J.R. Shewchuk, Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator, Lecture Notes in Computer Science, 1148,in: M.C. Lin, D. Manocha, Editors, Applied Computational Geometry: Towards Geometric Engineering, First ACM Workshop on Applied Computational Geometry, Springer-Verlag, Berlin, pages 203-222. (1996).
[31] 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.
[32] Herbert Edelsbrunner. Geometry and topology for mesh generation, volume 7 of Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, 2001.
[33] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. The Visual Computer, 2(4):227-234, 1986.
[34] Timothy J. Larkin and Philip W. Kuchel. Mathematical Models of Naturally "Morphed" Human Erythrocytes: Stomatocytes and Echinocytes, Bulletin of Mathematical Biology Volume 72, Number 6, 1323-1333.
[35] 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.
[36] 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.
[37] Pascal J. Frey and Loc Marechal. Fast adaptive quadtree mesh generation. In Proceedings of the 7th International Meshing Roundtable, pages 211.224. Sandia Nat. Lab., October 1998.
[38] Yibao Li and Junseok Kim, A fast and accurate numerical method for medical image segmentation, J. KSIAM Vol 14, No. 4, pp. 201-210, 2010.