SIAM REVIEW
Vol. 46, No. 2, pp. 329–345

A Simple Mesh Generator in MATLAB*

Per-Olof Persson†
Gilbert Strang†

Abstract. Creating a mesh is the first step in a wide range of applications, including scientific computing and computer graphics. An unstructured simplex mesh requires a choice of meshpoints (vertex nodes) and a triangulation. We want to offer a short and simple MATLAB code, described in more detail than usual, so the reader can experiment (and add to the code) knowing the underlying principles. We find the node locations by solving for equilibrium in a truss structure (using piecewise linear force-displacement relations) and reset the topology by the Delaunay algorithm.

The geometry is described implicitly by its distance function. In addition to being much shorter and simpler than other meshing techniques, our algorithm typically produces meshes of very high quality. We discuss ways to improve the robustness and the performance, but our aim here is simplicity. Readers can download (and edit) the codes from http://math.mit.edu/~persson/mesh.

Key words. mesh generation, distance functions, Delaunay triangulation

AMS subject classifications. 65M50, 65N50

DOI. 10.1137/S0036144503429121

1. Introduction.

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

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

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

*Received by the editors June 3, 2003; accepted for publication (in revised form) December 1, 2003; published electronically May 3, 2004.
http://www.siam.org/journals/sirev/46-2/42912.html

†Department of Mathematics, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139 (persson@math.mit.edu, gs@math.mit.edu).

For the actual mesh generation, our iterative technique is based on the physical analogy between a simplex mesh and a truss structure. Meshpoints are nodes of the truss. Assuming an appropriate force-displacement function for the bars in the truss at each iteration, we solve for equilibrium. The forces move the nodes, and (iteratively) the Delaunay triangulation algorithm adjusts the topology (it decides the edges). Those are the two essential steps. The resulting mesh is surprisingly well shaped, and Figure 5.1 shows examples. Other codes use *Laplacian smoothing* [4] 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 [1]. We mention Triangle [9] as a robust and freely available Delaunay refinement code.

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

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

We begin by describing the algorithm and the equilibrium equations for the truss. Next, we present the complete MATLAB code for the two-dimensional case, and describe every line in detail. In section 5, we create meshes for increasingly complex geometries. Finally, we describe the n-dimensional generalization and show examples of three-dimensional (3-D) and four-dimensional (4-D) meshes.

2. The Algorithm. In the plane, our mesh generation algorithm is based on a simple mechanical analogy between a triangular mesh and a two-dimensional (2-D) truss structure, or equivalently a structure of springs. Any set of points in the x, y-plane can be triangulated by the Delaunay algorithm [3]. In the physical model, the edges of the triangles (the connections between pairs of points) correspond to bars, and the points correspond to joints of the truss. Each bar has a force-displacement relationship $f(\ell, \ell_0)$ depending on its current length ℓ and its unextended length ℓ_0.

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

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

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

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

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

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

Note that $\boldsymbol{F}(\boldsymbol{p})$ depends on the topology of the bars connecting the joints. In the algorithm, this structure is given by the Delaunay triangulation of the meshpoints. The Delaunay algorithm determines nonoverlapping triangles that fill the convex hull of the input points, such that every edge is shared by at most two triangles, and the circumcircle of every triangle contains no other input points. In the plane, this triangulation is known to maximize the minimum angle of all the triangles. The force vector $\boldsymbol{F}(\boldsymbol{p})$ is not a continuous function of $\boldsymbol{p}$, since the topology (the presence or absence of connecting bars) is changed by Delaunay as the points move.

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

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

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

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

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

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

There are many alternatives for the force function $f(\ell, \ell_0)$ in each bar, and several choices have been investigated [1], [11]. The function $k(\ell_0 - \ell)$ models ordinary linear springs. Our implementation uses this linear response for the *repulsive* forces but it allows no attractive forces:

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

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

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

accuracy (*adaptive solver*). A uniform mesh with these small elements would require too many nodes.

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

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

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

The initial node positions $\boldsymbol{p}_0$ can be chosen in many ways. A random distribution of the points usually works well. For meshes intended to have uniform element sizes (and for simple geometries), good results are achieved by starting from equally spaced points. When a nonuniform 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.

3. Implementation. The complete source code for the 2-D mesh generator is in Figure 3.1. Each line is explained in detail below.

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

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

This meshing function produces the following outputs:

- The node positions `p`. This N-by-2 array contains the x, y-coordinates for each of the N nodes.
- The triangle indices `t`. The row associated with each triangle has 3 integer entries to specify node numbers in that triangle.

The input arguments are as follows:

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

```
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 meshpoints 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
```

Fig. 3.1 *The complete source code for the 2-D mesh generator* `distmesh2d.m`. *This code can be downloaded from http://math.mit.edu/~persson/mesh.*

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.

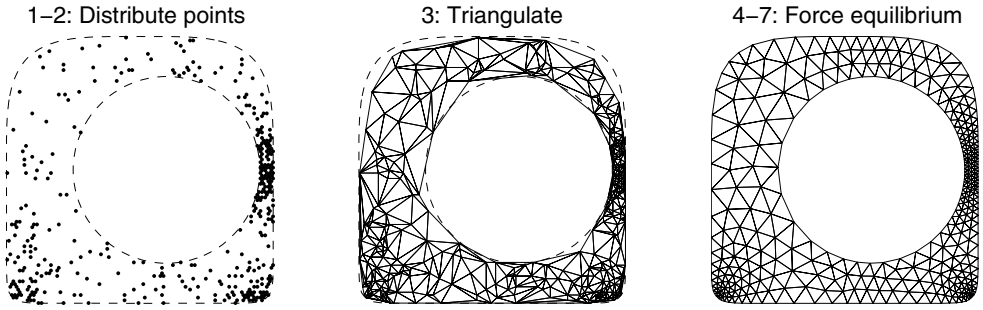

Fig. 3.2 *The generation of a nonuniform triangular mesh.*

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

Now we describe steps 1 to 8 in the `distmesh2d` algorithm, as illustrated in Figure 3.2.

Step 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-by-2 array `p`.

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

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

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

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

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

Step 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.

Step 4. The list of triangles `t` is an array with three 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
```

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

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

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

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

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

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

The actual update of the node positions `p` is in the next block of code. The force resultant `Ftot` is the sum of force vectors in `Fvec`, from all bars meeting at a node. A stretching force has positive sign, and its direction is given by the two-component vector in `bars`. The `sparse` command is used (even though `Ftot` is immediately

converted to a dense array!), because of the nice summation property for duplicated indices.

```
Fvec=F./L*[1,1].*barvec;                                % Bar forces (x,y-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`.

Step 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
```

Step 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.

4. Special Distance Functions. The function `distmesh2d` is everything that is needed to mesh a region specified by the distance $d(x, y)$ to the boundary. While it is easy to create distance functions for some simple geometries, it is convenient to define some short help functions (Figure 4.1) for more complex geometries.

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

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

$$\text{(4.1)} \qquad \textbf{Union}: \qquad d_{A\cup B}(x,y) = \min(d_A(x,y), d_B(x,y)).$$

$$\text{(4.2)} \qquad \textbf{Difference}: \qquad d_{A\setminus B}(x,y) = \max(d_A(x,y), -d_B(x,y)).$$

$$\text{(4.3)} \qquad \textbf{Intersection}: \qquad d_{A\cap B}(x,y) = \max(d_A(x,y), d_B(x,y)).$$

Variants of these can be used to generate *blending surfaces* for smooth intersections between two surfaces [13]. Finally, `pshift` and `protate` operate on the node array `p`, to translate or rotate the coordinates.

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

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

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

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

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

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

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

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

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

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

Fig. 4.1 *Short help functions for generation of distance functions and size functions.*

The distance function may also be provided in a discretized form, for example, by values on a Cartesian grid. This is common in *level set applications* [8], where PDEs efficiently model geometries with moving boundaries. Signed distance functions are created from arbitrary implicit functions using the *reinitialization method* [12]. We can easily mesh these discretized domains by creating $d(x, y)$ from interpolation. The functions `dmatrix` and `hmatrix` in Figure 4.1 use `interp2` to create $d(x, y)$ and $h(x, y)$, and `huniform` quickly implements the choice $h(x, y) = 1$.

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

$$\boldsymbol{L}(\boldsymbol{P}) = \begin{bmatrix} f(x,y) \\ (x-x_0)f_y - (y-y_0)f_x \end{bmatrix} = \boldsymbol{0}. \tag{4.4}$$

We solve (4.4) for the column vector $\boldsymbol{P} = (x, y)$ using the damped Newton's method with $\boldsymbol{p}_0 = (x_0, y_0)$ as initial guess. The Jacobian of $\boldsymbol{L}$ is

$$\boldsymbol{J}(\boldsymbol{P}) = \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{P}} = \begin{bmatrix} f_x & f_y + (x-x_0)f_{xy} - (y-y_0)f_{xx} \\ f_y & -f_x - (y-y_0)f_{xy} + (x-x_0)f_{yy} \end{bmatrix}^{\mathrm{T}} \tag{4.5}$$

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

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k - \alpha \boldsymbol{J}^{-1}(\boldsymbol{p}_k)\boldsymbol{L}(\boldsymbol{p}_k) \tag{4.6}$$

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

$$d(\boldsymbol{p}_0) = \text{sign}\,(f(x_0, y_0))\,\sqrt{(x - x_0)^2 + (y - y_0)^2}. \tag{4.7}$$

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

5. Examples. Figure 5.1 shows a number of examples, starting from a circle and extending to relatively complicated meshes.

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

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

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

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

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

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

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

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

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

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

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

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

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

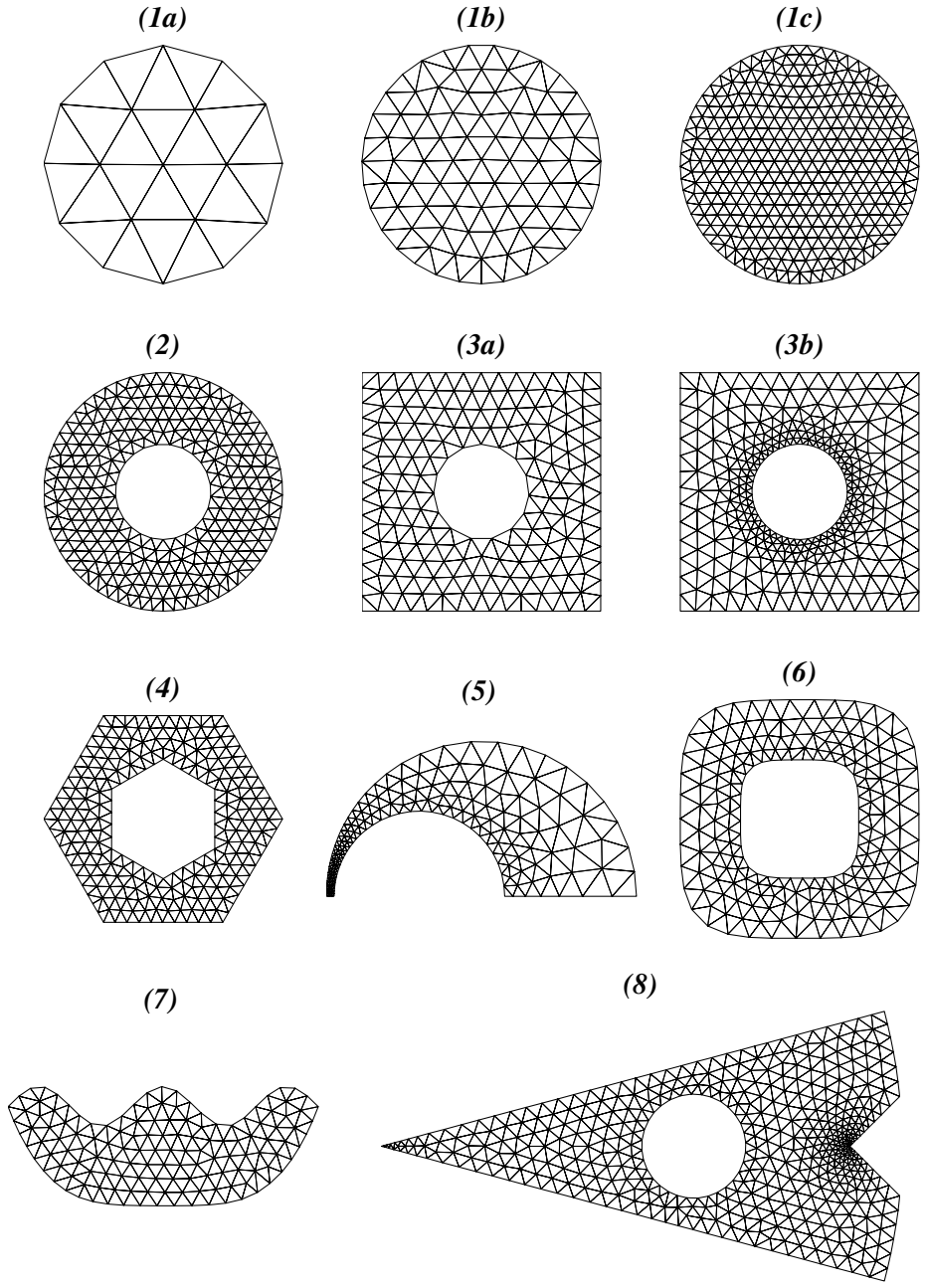

Fig. 5.1 *Example meshes, numbered as in the text. Examples* (3b), (5), (6), *and* (8) *have varying size functions* $h(x, y)$. *Examples* (6) *and* (7) *use Newton's method* (4.6) *to construct the distance function.*

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

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

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

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

$$d_1 = \sqrt{x^2 + y^2} - 1, \tag{5.1}$$

$$d_2 = \sqrt{(x + 0.4)^2 + y^2} - 0.55, \tag{5.2}$$

$$d = \max(d_1, -d_2, -y). \tag{5.3}$$

Next, we create two element size functions to represent the finer resolutions near the circles. The element sizes h_1 and h_2 increase with the distances from the boundaries (the factor 0.3 gives a ratio 1.3 between neighboring elements):

$$h_1(x, y) = 0.15 - 0.2 \cdot d_1(x, y), \tag{5.4}$$

$$h_2(x, y) = 0.06 + 0.2 \cdot d_2(x, y). \tag{5.5}$$

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

$$h_3(x, y) = (d_2(x, y) - d_1(x, y))/3. \tag{5.6}$$

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

$$h = \min(h_1, h_2, h_3). \tag{5.7}$$

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

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

$$y \le \cos(x) \quad \text{and} \quad y \ge 5\left(\frac{2x}{5\pi}\right)^4 - 5, \tag{5.8}$$

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

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

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

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

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

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

All Delaunay mesh generators suffer from this problem in three dimensions. The good news is that techniques have been developed to remove the bad elements, for example, face swapping, edge flipping, and Laplacian smoothing [6]. A promising method for sliver removal is presented in [2]. Recent results [7] show that slivers are not a big problem in the finite volume method, which uses the dual mesh (the Voronoi graph). It is not clear how much damage comes from isolated bad elements in finite element computations [10]. The slivery meshes shown here give nearly the same accuracy for the Poisson equation as meshes with higher minimum quality.

Allowing slivers, we generate the tetrahedral meshes in Figure 6.1.

(9) Unit Ball. The ball in three dimensions uses nearly the same code as the circle:

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

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

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

$$d_1(x,y,z) = \sqrt{x^2+y^2} - 1, \tag{6.1}$$

$$d_2(x,y,z) = z - 1, \tag{6.2}$$

$$d_3(x,y,z) = -z - 1. \tag{6.3}$$

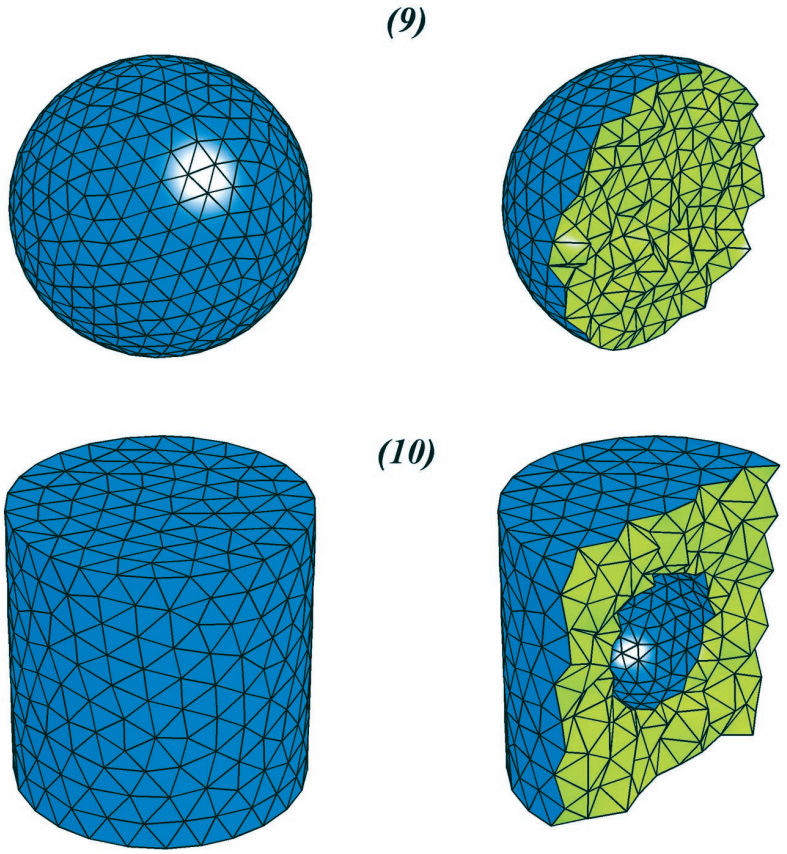

Fig. 6.1 *Tetrahedral meshes of a ball and a cylinder with a spherical hole. The left plots show the surface meshes, and the right plots show cross sections.*

An approximate distance function is then formed by intersection:

$$d_{\approx} = \max(d_1, d_2, d_3). \tag{6.4}$$

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

$$d_4(x, y, z) = \sqrt{d_1(x, y, z)^2 + d_2(x, y, z)^2}, \tag{6.5}$$

$$d_5(x, y, z) = \sqrt{d_1(x, y, z)^2 + d_3(x, y, z)^2}. \tag{6.6}$$

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

$$d = \begin{cases} d_4 & \text{if } d_1 > 0 \text{ and } d_2 > 0, \\ d_5 & \text{if } d_1 > 0 \text{ and } d_3 > 0, \\ d_{\approx} & \text{otherwise.} \end{cases} \tag{6.7}$$

Figure 6.1 shows a mesh for the difference between this cylinder and a ball of radius 0.5. We use a finer resolution close to this ball, $h(x, y, z) = \min(4\sqrt{x^2 + y^2 + z^2} - 1, 2)$, and $h_0 = 0.1$. The resulting mesh has 1,057 nodes and 4,539 tetrahedra.

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

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

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

It is hard to visualize a mesh in four dimensions! We can compute the total mesh volume $V_4 = 4.74$, which is close to the expected value of $\pi^2/2 \approx 4.93$. By extracting all tetrahedra on the surface, we can compare the hypersurface area $S_4 = 19.2$ to the surface area $2\pi^2 \approx 19.7$ of a 4-D ball. The deviations are due to the simplicial approximation of the curved surface.

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

7. Mesh Quality. The plots of our 2-D meshes show that the algorithm produces triangles that are almost equilateral. This is a desirable property when solving PDEs with the finite element method. Upper bounds on the errors depend only on the smallest angle in the mesh, and if all angles are close to 60°, good numerical results are achieved. The survey paper [5] discusses many measures of the "element quality." One commonly used quality measure is the ratio between the radius of the largest inscribed circle (times two) and the smallest circumscribed circle:

$$q = 2\frac{r_{\text{in}}}{r_{\text{out}}} = \frac{(b + c - a)(c + a - b)(a + b - c)}{abc}, \tag{7.1}$$

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

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

The meshes produced by our algorithm tend to have exceptionally good element quality and uniformity. All 2-D examples except (8) with a sharp corner have every $q > 0.7$ and average quality greater than 0.96. This is significantly better than a typical Delaunay refinement mesh with Laplacian smoothing. The average size deviations are less than 4%, compared to 10–20% for Delaunay refinement.

A comparison with the Delaunay refinement algorithm is shown in Figure 7.1. The top mesh is generated with the mesh generator in the PDE Toolbox, and the bottom with our generator. Our force equilibrium improves both the quality and the uniformity. This remains true in three dimensions, where quality improvement methods such as those in [6] must be applied to both mesh generators.

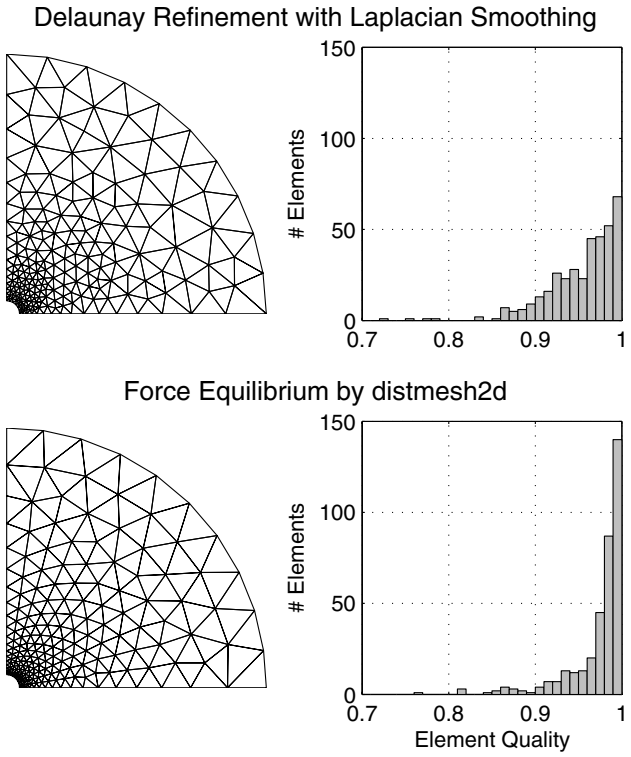

Fig. 7.1 *Histogram comparison with the Delaunay refinement algorithm. The element qualities are higher with our force equilibrium, and the element sizes are more uniform.*

8. Future Improvements. Our code is short and simple, and it appears to produce high quality meshes. In this version, its main disadvantages are slow execution and the possibility of nontermination. Our experiments show that it can be made more robust with additional control logic. The termination criterion should include a quality estimate to avoid iterating too often. Our method to remove elements outside the region (evaluation of $d(\boldsymbol{x})$ at the centroid) can be improved upon, and the cases when the mesh does not respect the boundaries should be detected. A better scaling between h and the actual edge lengths could give more stable behavior for highly nonuniform meshes. We decided not to include this extra complexity here.

We have *vectorized* the MATLAB code to avoid for-loops, but a pure C-code is still between one and two magnitudes faster. One reason is the slow execution of the inline functions (a standard file-based MATLAB function is more than twice as fast). An implicit method could solve (2.3), instead of forward Euler.

We think that the algorithm can be useful in other areas than pure mesh generation. The distance function representation is effective for moving boundary problems, where the mesh generator is linked to the numerical solvers. The simplicity of our method should make it an attractive choice.

We hope readers will use this code and adapt it. The second author emphasizes that the key ideas and the advanced MATLAB programming were contributed by the first author. Please tell us about significant improvements.

REFERENCES

[1] F. J. Bossen and P. S. Heckbert, *A pliant method for anisotropic mesh generation*, in Proceedings of the 5th International Meshing Roundtable, Sandia National Laboratories, Albuquerque, NM, 1996, pp. 63–74.

[2] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng, *Sliver exudation*, in Proceedings of the Fifteenth Annual Symposium on Computational Geometry, ACM, New York, 1999, pp. 1–13.

[3] H. Edelsbrunner, *Geometry and Topology for Mesh Generation*, Cambridge University Press, Cambridge, UK, 2001.

[4] D. Field, *Laplacian smoothing and Delaunay triangulations*, Comm. Appl. Numer. Methods, 4 (1988), pp. 709–712.

[5] D. Field, *Qualitative measures for initial meshes*, Internat. J. Numer. Methods Engrg., 47 (2000), pp. 887–906.

[6] L. Freitag and C. Ollivier-Gooch, *Tetrahedral mesh improvement using swapping and smoothing*, Internat. J. Numer. Methods Engrg., 40 (1997), pp. 3979–4002.

[7] G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington, *On the radius-edge condition in the control volume method*, SIAM J. Numer. Anal., 36 (1999), pp. 1690–1708.

[8] S. Osher and J. A. Sethian, *Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations*, J. Comput. Phys., 79 (1988), pp. 12–49.

[9] J. R. Shewchuk, *Triangle: Engineering a* 2D *Quality Mesh Generator and Delaunay Triangulator*, in Applied Computational Geometry: Towards Geometric Engineering, M. C. Lin and D. Manocha, eds., Lecture Notes in Comput. Sci. 1148, Springer-Verlag, New York, 1996, pp. 203–222.

[10] J. R. Shewchuk, *What is a good linear element? Interpolation, conditioning, and quality measures*, in Proceedings of the 11th International Meshing Roundtable, Sandia National Laboratories, Albuquerque, NM, 2002, pp. 115–126.

[11] K. Shimada and D. 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, 1995, pp. 409–419.

[12] M. Sussman, P. Smereka, and S. Osher, *A levelset approach for computing solutions to incompressible two-phase flow*, J. Comput. Phys., 114 (1994), pp. 146–159.

[13] B. Wyvill, C. McPheeters, and G. Wyvill, *Data structure for soft objects*, The Visual Computer, 2 (1986), pp. 227–234.