How to make a mesh

John Burkardt
Department of Scientific Computing
Florida State University

....

....

http://people.sc.fsu.edu/∼jburkardt/presentations/. . .
. . . mesh_2014_fsu.pdf

11 April 2014

Meshing analyzes geometry.

SURVEY: A Township Survey

Meshing can organize a territory into patches.
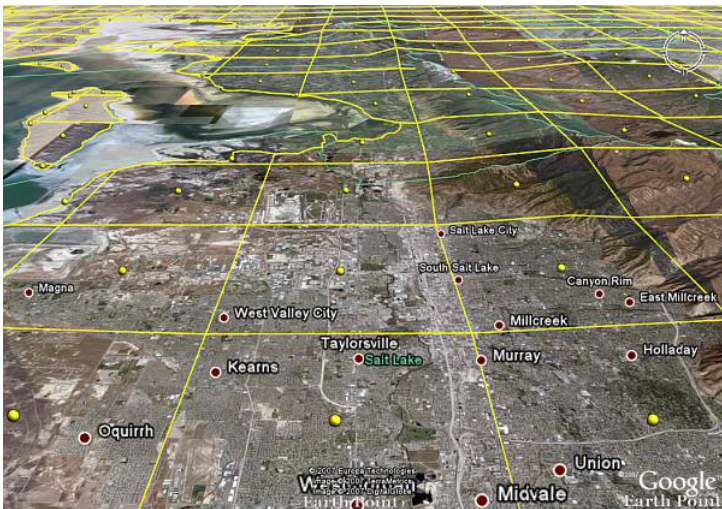
- **Meshing**
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- Conclusion

The finite element method begins by looking at a complicated region, and thinking of it as a mesh of smaller, simpler *subregions*.

The subregions are simple, (perhaps triangles) so we understand their geometry; they are small because when we approximate the differential equations, our errors will be related to the size of the subregions. More, smaller subregions usually mean less total error.

After we compute our solution, it is described in terms of the mesh. The simplest description uses piecewise linear functions, which we might expect to be a crude approximation. However, excellent results can be obtained as long as the mesh is small enough in places where the solution changes rapidly.

Thus, even though the hard part of the finite element method involves considering abstract approximation spaces, sequences of approximating functions, the issue of boundary conditions, weak forms and so on, ...it all starts with a <u>very simple idea</u>:

Given a geometric shape, break it into smaller, simpler shapes; fit the boundary, and be small in some places.

Since this is such a simple idea, you might think there's no reason to worry about it much!

Indeed, if we start by thinking of a 1D problem, such as modeling the temperature along a thin strand of wire that extends from **A** to **B**, our meshing problem is trivial:
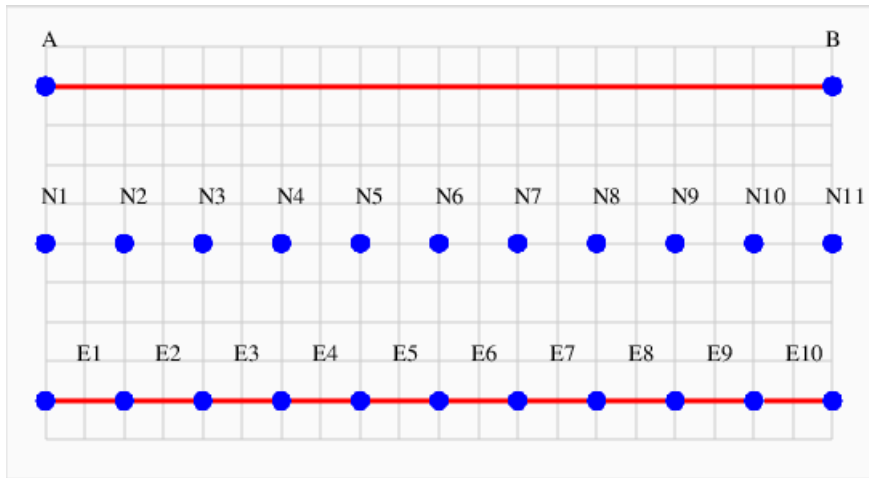
- Choose **N**, the number of subregions or <u>elements</u>;
- Insert **N-1** equally spaced *nodes* between **A** and **B**;
- Create **N** *elements*, the intervals between successive nodes.

For this problem, we can write down formulas for the location of each node, the location of each element, the indices of the pair of nodes **I** and **J** that form element **K**, and the indices of the elements **L** and **M** that are immediate neighbors to element **K**.

MESHING: Nodes and Elements in 1D

From 2 vertices, we define 11 nodes, and 10 elements.

It might seem that the 2D world is going to be just as easy! We just take our rectangular region, defined by four corners, place nodes along each side and then put nodes at intersection points, and then, because we prefer triangles, we split each of the resulting squares into two triangular elements.

Again, we can write down, fairly easily, the location of every node, the nodes that form each triangle, and the triangles that neighbor each triangle.

MESHING: The "ELL" Problem

For our basic 2D example, we'll consider an L-shaped region, and show how to go through the basic meshing steps.

When it's time to talk about programs for doing the meshing for us, we will come back to this same problem, so keep its simple shape in mind!

It's simply a square of dimension 2x2 units, from which a 1x1 unit square in the northeast has been removed.
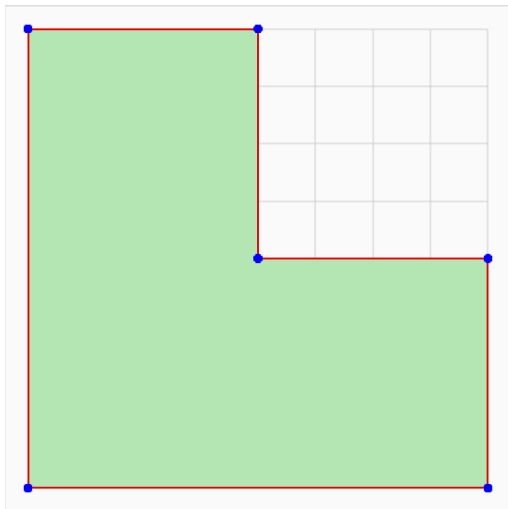
Interesting features might include:

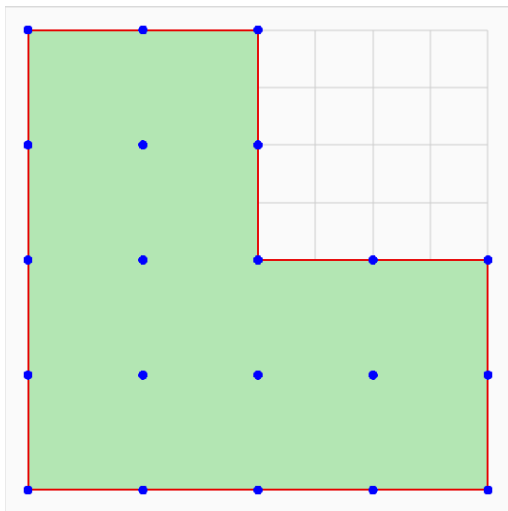- the vertices (points)
- the boundary (lines)
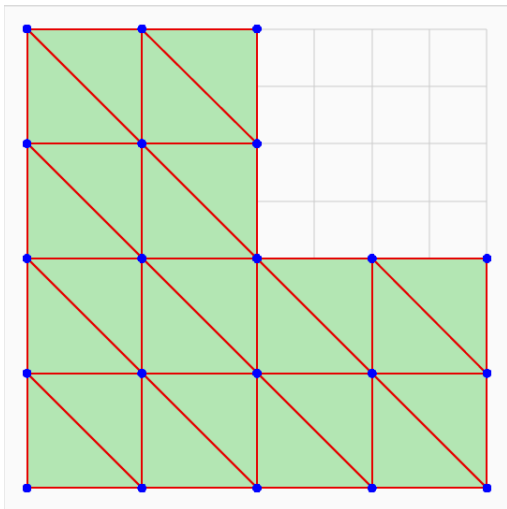- the region (area)

6 vertices define the shape.

21 nodes will be used for the internal mesh.

24 triangular elements constitute the mesh.

While a mathematician or academic computing person might regard our L-shaped region as wildly irregular, a person who actually needs to use the finite element method will regard the use of purely rectangular regions as unrealistic and much too limited to be useful.
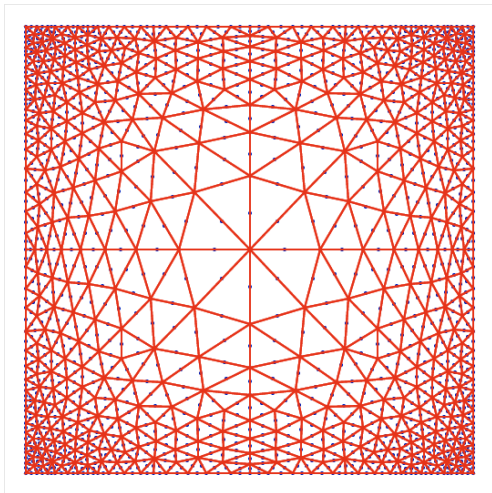
It's similar to trying to analyze a horse race by starting out with the assumption *"All horses can be regarded as perfect spheres."*

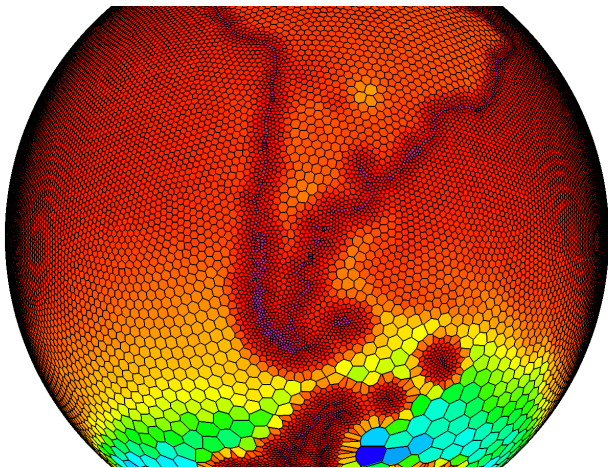Well, what kind of problems do we really need to be able to solve?

We need meshes that automatically vary in density.

We need the mesh to be small near interfaces.

We need to mesh surfaces that include holes and edges.
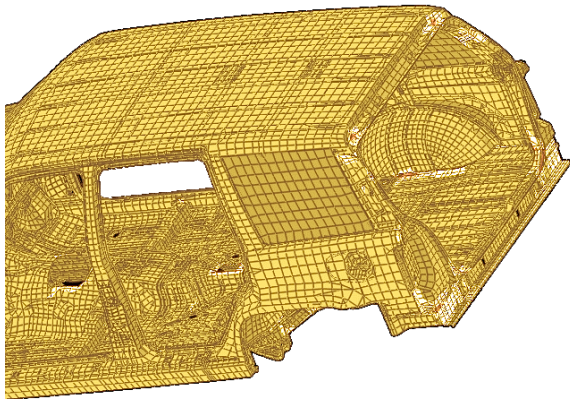
The mesh must notice and adapt to local features (here, curvature.)
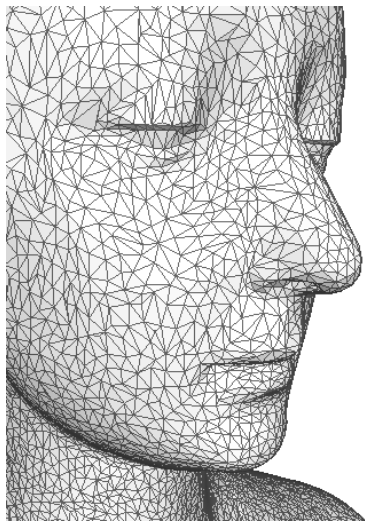
For a true 3D problem, we need nodes and elements inside the surface
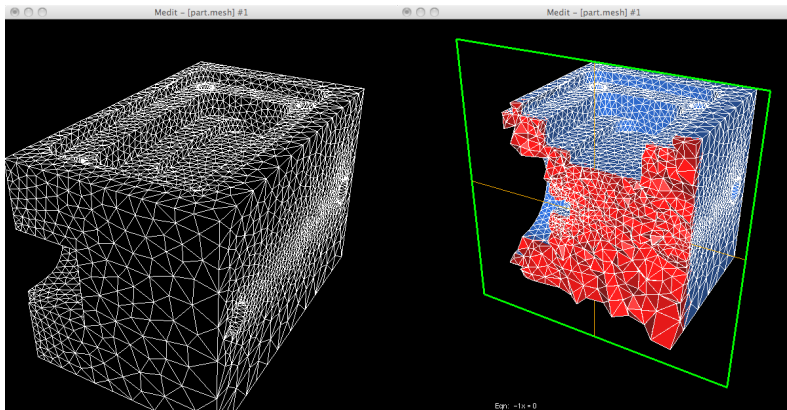
These regions are complicated and realistic and not rectangular. The meshes "respond" to the geometry. How is this done?

Given the boundaries of the region, and perhaps a desired mesh density at every point, how can we:

- describe the input information to a computer?
- use the input information to choose nodes?
- use the nodes to construct elements?
- handle boundaries, internal holes, internal walls?
- construct all the arrays of connectivity information?

What if we have 1,000,000 nodes? What if our problem is 3D?

MESHING: Reference

One good web site is Robert Schneider's *"Mesh Generation & Grid Generation on the Web"*.

```
www.robertschneiders.de/meshgeneration/meshgeneration.html
```

including

- People and research groups
- List of people
- Latest news
- Software
- Conferences
- Literature
- Open positions
- Information on related topics

How to Make a Mesh

- Meshing
- **Computer Representations**
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- Conclusion

The objects we are talking about must somehow be represented on a computer. It may be helpful to go over how these objects might be represented, and in some cases, suggest how one object can be computed from another.

The fundamental object, of course, is the region. Let's keep things simple and assume we're simply dealing with a subset of the plane, such as a circle, square, perhaps an irregular polygon, or possible an arbitrary curvy closed loop.

This region might have sharp corners, stretches with a fixed curvature, interior holes or barriers. In the most general case, this is a hard object to describe.

We will assume that the region can be defined by one or more closed curves, approximated using straight line segments. Even if our region is a circle, we specify it by a sequence of straight lines.

A circle might be specified by 24 evenly spaced vertices **V**.

A region is really specified by the curve suggested by the vertices, so we should be more careful and either insist that the 24 vertices are connected one after another, or else we should include an additional set of information, namely, the order in which the given vertices should be connected to bound the region.

The advantage of the second approach is that, if I always specify such a curve in counterclockwise order, then it is easy to describe regions with multiple parts, or with holes.

Here is a region defined by a square with a triangular hole.

```
Vertices V: { (0,0), (5,0), (5,5), (0,5),
              (4,2), (2,1), (2,4) }

Boundary Indices BI: { 1, 2, 3, 4, 1, 5, 6, 7, 5 }
```

This describes a square from which a triangle has been removed. The region is on the "inside" of both curves, that is, points that lie on the left hand side as you follow each curve.

MATLAB could plot this data by starting with the first index (and remembering it!), drawing to the next one, until it returns to the start. Then it should jump to the next index and start a new line segment. We assume **V** is stored as a **V_NUM** by 2 array.

```
hold on
next = 1;
s = bi(1);
t2 = s;
draw = 1;
while ( next < length ( bi ) )
  t1 = t2;
  next = next + 1;
  t2 = bi(next);
  if ( draw )
    line ( [ v(t1,1), v(t2,1) ], [ v(t1,2), v(t2,2) ] );
    if ( t2 == s )
      draw = 0;
    end
  else
    s = t2;
    draw = 1;
  end
end
hold off
```

http://people.sc.fsu.edu/~jburkardt/m_src/fem_meshing/boundary_display.m

The vertices outline the boundary of the region, but we need to fill up the region (and the vertex boundary) with what we have called **nodes**. These nodes will be used to define our elements, and the basis functions. If our region isn't rectangular, it might not be obvious how to produce them, but once we have them, we'll think of them as a list **P** of (X,Y) coordinates.

```
Nodes P: { (0.0,0.0), (0.5,0.0), (1.0,0.0), (1.5,0.0),
  (2.0,0.0) ... (1.0,2.0) }
```

It is very likely that some or all of the vertices **V** will be included in the list **P**. If we've stored the P data as a **P_NUM** by 2 array, then MATLAB can plot the nodes:

```
plot ( p(:,1), p(:,2), 'r.', 'MarkerSize', 5 )
```

Even if we can't compute the triangles, we can imagine how to store them. A triangle is formed by three nodes. We can store the collection **T** of triangles as a **T_NUM** by 3 array of node indices:

```
Triangles T: { (1, 2, 3), (6,1,4), (5,6,8), ...
        ... (89,43,27) }
```

When listing triangles, we choose the counterclockwise ordering. This means that every interior edge will be listed twice, while boundary edges will all be listed once. *In other words, the "logical sum" of all the triangles is an outline of the original region!*

MATLAB can plot a triangulation:

```
trimesh ( t, p(:,1), p(:,2) )
```

One way to compute the node boundary takes all the edges and drops the duplicates. The node boundary can be stored as a **B_NUM** by 2 list of pairs of node indices:

```
Boundary Edges: { (1, 2), (7,18), (4,63), ... (82,14) }
```

Simply having a collection of boundary edges is different than actually having the edges in sequence. If you need that, you start with one edge, find a connecting edge, keep looking until you get back to where you started, and then check to see whether you have more edges to work on.

We seem to have discussed the boundary twice. First was the *vertex boundary*, which only involved vertices. The *node boundary*, includes short line segments between nodes added to the boundary between the vertices.

The standard finite element method doesn't need to know element neighbors; however, there are many times when dealing with a mesh when this is necessary. For example, there's a fast algorithm to find a random point hidden in one of 1,000,000 elements that will take, on average, 500 trials, rather than 500,000, but it requires being able to move from one triangle to its neighbor.

All the information for determining triangle neighbors is available. Two triangles are neighbors if they share an edge. That is, one triangle uses nodes 5 and 17, in that order, the other uses 17 and 5. There are ways to efficiently examine all the edges, find these pairs of matching data, and indicate that two triangles are neighbors. Some triangles don't have a neighbor on a particular side, because they are on the boundary, so that neighbor is -1.

How to Make a Mesh

- Meshing
- Computer Representations
- **The Delaunay Triangulation**
- TRIANGLE
- DISTMESH
- MESH2D
- Conclusion

A pair of mysteries remain:

- where does the set of nodes **P** come from?
- how are these nodes arranged into triangles **T**?

The answer to both questions involves the Delaunay triangulation, which can compute a "good" triangulation of any set of nodes **P**.

That explains **T**, but what about **P**? Well, it turns out that we can start with an arbitrary or random set of nodes **P**, and use information from the Delaunay triangulation that will rearrange the nodes to better fill the region, either uniformly or in accordance with some density function we specify. By iterating on this process, we get good nodes and good triangles.

Suppose we generate a random set of nodes **P** within our problem region. We can then connect as many pairs of nodes as possible without ever crossing a previous line. The result is a (maximal) triangulation of the nodes.

The process seems pretty arbitrary, and in fact there are many possible triangulations of a set of points. You may wonder how to automate this process; a natural way is to start by creating a giant triangle that encloses all the points you are going to use.
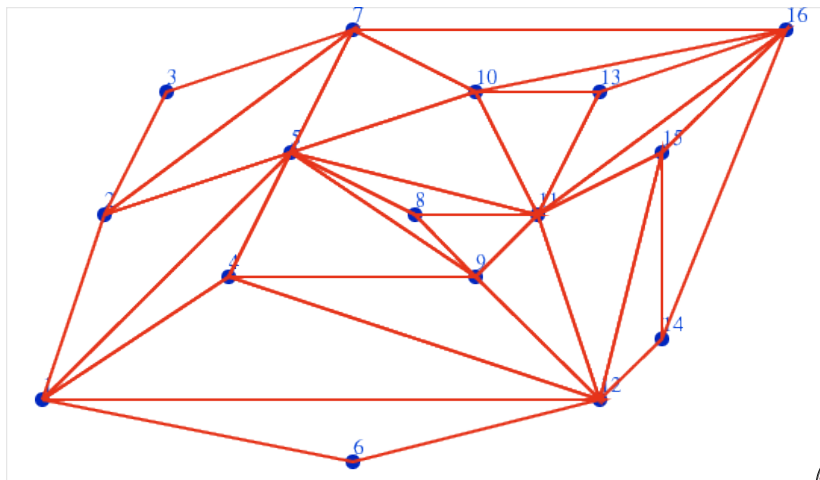
Then add the first node. Connect it to each vertex of the enclosing triangle, and you've got a maximal triangulation. Add the second node. It falls into one of the triangles you already created, so you subdivide that triangle. Keep going. At the end, remove the enclosing triangle, and any edges that connect to it, and you have a maximal triangulation of the nodes.
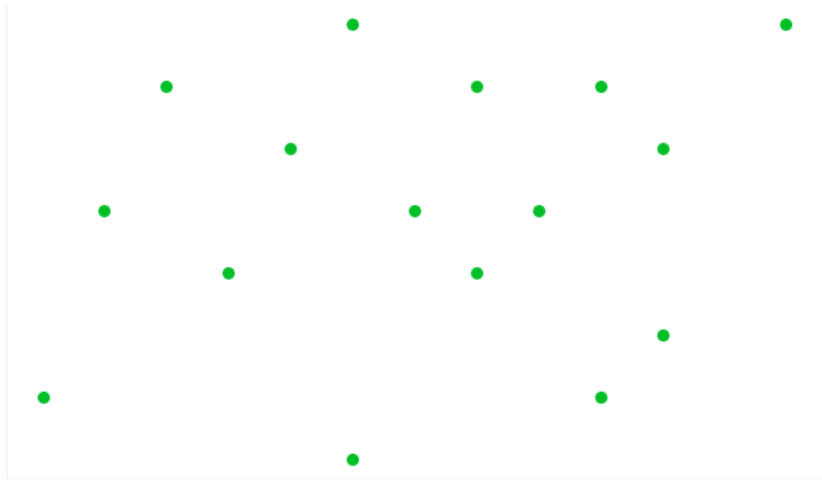
We drew the lines of our triangulation at random. If we tried a second time, we'd get a different picture. There are actually many ways to triangulate a set of points in the plane. Given that fact, it's likely that some triangulations are "better" than others, but that depends on what we want to do with our triangulations!

If we think about the connecting lines as "roads", we might prefer a triangulation that uses the shortest total length.

If we think about the triangles as representing patches of territory, we might dislike triangles that have a very small angle.

For graphics applications, and for many computational purposes, *the avoidance of small angles* is a very common criterion.

The Delaunay triangulation of a set of points is the (usually unique) triangulation which does the best job of avoiding small angles.

Strictly speaking, we consider all possible triangulations of a set of nodes. For each triangulation T, let $\theta(T)$ be the smallest angle that occurs in any triangle of that triangulation. Then a triangulation T^* is a Delaunay triangulation if
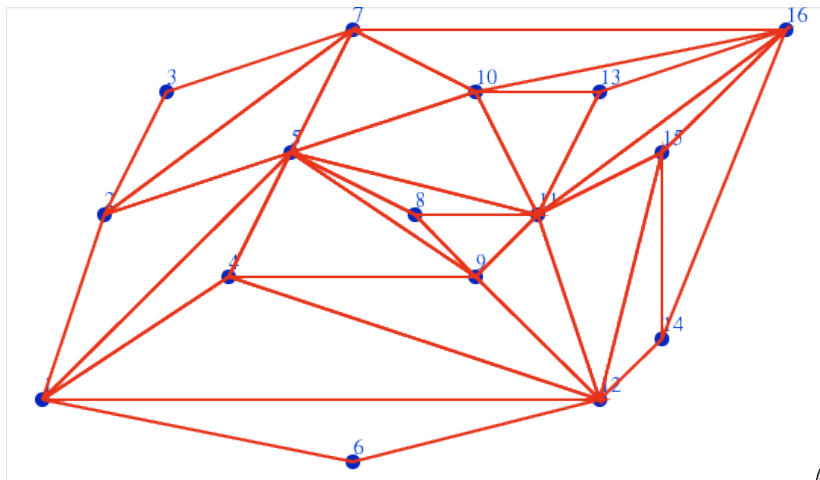
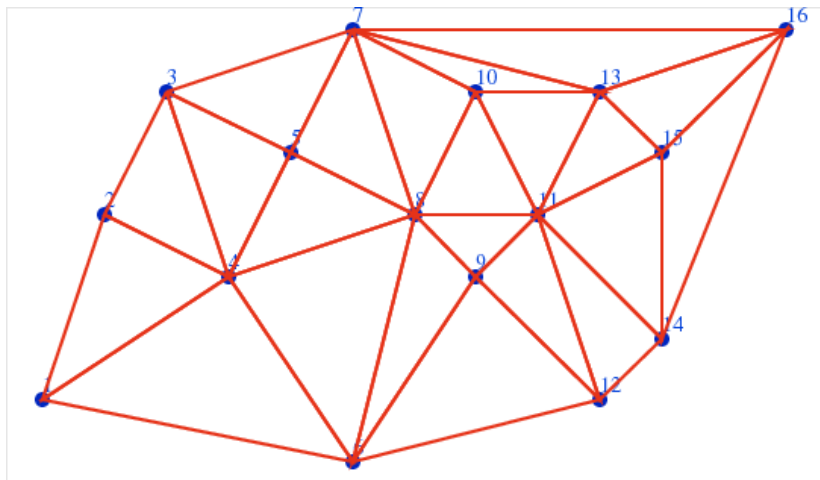$$\theta(T) \leq \theta(T^*)$$

for all triangulations T.

Since there are only finitely many possible triangulations, the Delaunay triangulation must exist, and if we had no other way, we could find it by computing and comparing every triangulation.

Although we chose the Delaunay triangulation based on an angle consideration, comparing the two pictures suggests that the Delaunay triangulation also does a better job of connecting nearby nodes rather than far-away ones, avoiding long triangle sides, and creating triangles that have a more uniform shape.

The *convergence* of the finite element method comes, in part, from ensuring that all the elements get small. The *accuracy* of the finite element calculations within a triangle depend, in part, on the triangle having a relatively equilateral shape. The *smoothness* of the approximation depends somewhat on having relatively short triangle sides.

So the Delaunay triangulation has much to recommend it!

Even though we will end up calling a piece of software to take care of all the details for us, it's important to understand that there are simple ways to compute a Delaunay triangulation.

For instance, a triangulation is Delaunay if each triangle is "locally Delaunay". A triangle is locally Delaunay if we can't improve the (local) minimum angle by merging with a neighbor triangle and flipping the edge.

So we check each triangle, and if an edge swap improves the local minimum angle situation, we take it. We keep doing this until no more improvement is possible.

It's not magic, it's an algorithm...

To compute the triangles that form a Delaunay triangulation of a set of data points, use the MATLAB command

```
t = delaunay ( p(:,1), p(:,2) )
```

To display the triangulation,

```
triplot ( t, p(:,1), p(:,2) )
```

MATLAB includes a Delaunay function that can handle 3D or higher:

```
t = delaunayn ( p )   (P contains 2, 3, or N-dimensional points)
                      (T contains triangles, tetrahedrons, or simplexes)
```

MATLAB can make a 3D plot of tetrahedral meshes:

```
tetramesh ( t, p )
```

For a discussion of triangulations, the Delaunay triangulation, and Voronoi diagrams as implemented in MATLAB, see Chapter 5, "Computational Geometry", in the following free PDF book.

```
MATLAB Mathematics
www.mathworks.com/help/pdf_doc/matlab/math.pdf
```

- Meshing
- Computer Representations
- The Delaunay Triangulation
- **TRIANGLE**
- DISTMESH
- MESH2D
- Conclusion

Jonathan Shewchuk's **triangle** can start from a node file:

```
# spiral.node
15  2  0  0              <-- Point count, dimensions,
 1      0.00    0.00         attributes, boundary markers.
 2     -0.42    0.91
 3     -1.35    0.43
...     ...     ...
14      2.16    2.89
15      1.36    3.49
```

We can triangulate the nodes we are given, or add nodes to increase the minimum angle.

```
triangle spiral          <-- Triangulate the nodes
triangle -q spiral       <-- Minimum angle 20 deg
triangle -q32.5 spiral   <-- Minimum angle 32.5 deg
```

http://www.cs.cmu.edu/~quake/triangle.html

No New Points || Minimum Angle 20º || Minimum Angle 32.5º
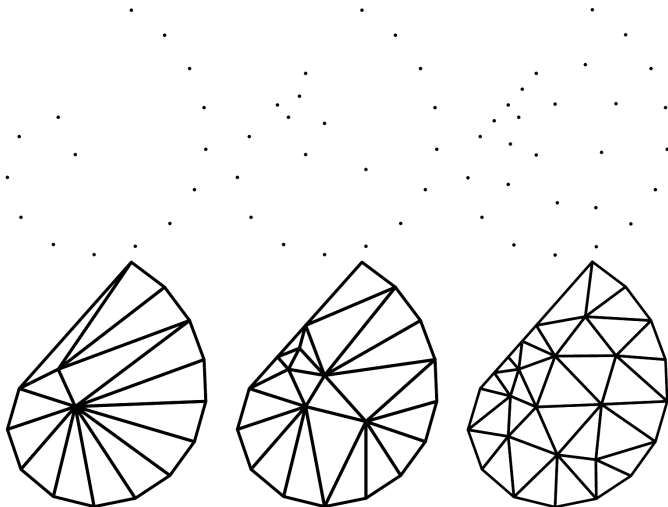
TRIANGLE: Area Constraints

In finite element calculations, one of the crucial quantities to control is the area of the elements. Sometimes we simply want all the elements to be smaller than some tolerance. Other times, we only need elements to be small in places where the solution changes rapidly, or has low differentiability.

The "-a" switch sets a global maximum for the area of all elements:

```
triangle -a0.2 spiral
```

If you can decompose your domain, you can specify a separate maximum area for each subdomain.

You can also determine a mesh density function which is defined pointwise. This might come from error estimators determined from a previous finite element mesh.

Global Maximum || Subdomain Maximum || Pointwise Maximum
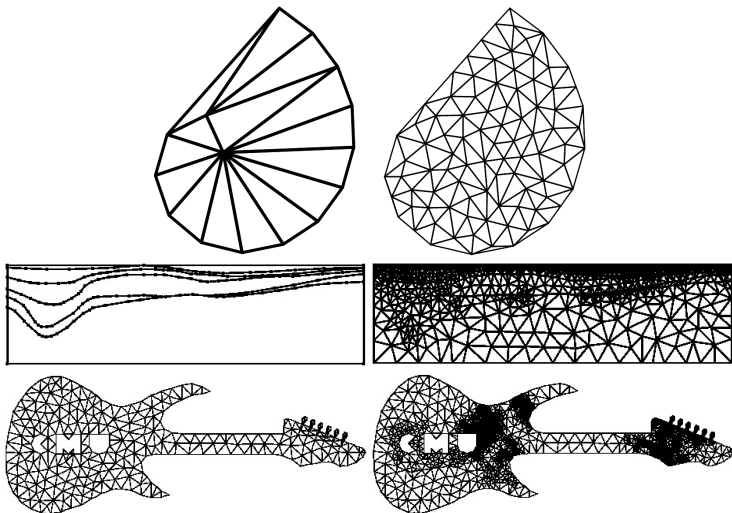
triangle helps you make a sequence of refined meshes, including all points from the current mesh.

The "box.poly" file contains a square with a square hole. We can compute a triangulation, and a series of refinements, as follows:

```
triangle box                 Creates "box.1" mesh
triangle -rpa0.2 box.1       Creates "box.2" mesh, and so on
triangle -rpa0.05 box.2
triangle -rpa0.0125 box.3
```
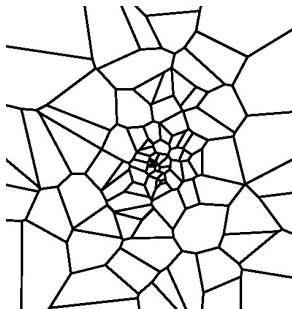
triangle can compute the Voronoi diagram of the nodes.

```
triangle -v dots
```

TRIANGLE: The SHOWME Graphics Interface

triangle includes a graphics program called **showme**, which can display the nodes, edges, triangulation, or Voronoi diagram.
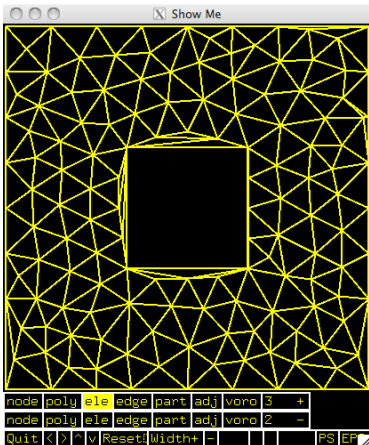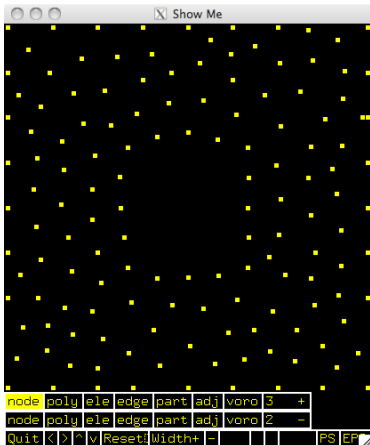
triangle is also available as a compiled library, which means a C program you write can use **triangle** directly as it is running.

Web page:

    www.cs.cmu.edu/~quake/triangle.html

Reference:

- Jonathan Shewchuk,
  Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator,
  in Applied Computational Geometry: Towards Geometric Engineering, edited by Ming Lin, Dinesh Manocha,
  Lecture Notes in Computer Science, Volume 1148,
  Springer, 1996.

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- **DISTMESH**
- MESH2D
- Conclusion

A picture is worth 1,000 words.

An animation is therefore worth 1,000,000 words $\approx$ 8 Megabytes?

```
meshdemo_2d ( )
```

---

http://persson.berkeley.edu/distmesh/
http://people.sc.fsu.edu/~jburkardt/m_src/distmesh/meshdemo_2d.m

Given any set of nodes **P**, we can compute a Delaunay triangulation **T**.

Can we use **T** to "improve" **P**? (whatever that means!)

The meshing program **distmesh()**, by Persson and Strang, uses the idea that, in the typical case, we'd like each node to be roughly the same distance from all its neighbors. The Delaunay triangulation connects a node to its neighbors (but not to far away nodes!). We can imagine each of these connections to be a little spring, which exerts a force if it is too long or too short.

So **distmesh()** actually sets up a linear system for the forces in a differential equation, and then takes a small time step, that is, it lets each node respond to the force by moving in the appropriate direction.

Once the nodes have been allowed to move, it is necessary to recalculate the spring forces, and take another step. By repeating this process carefully, a good result can be obtained.

Nodes that try to cross the boundary are pushed back in.

The result is a mesh of nodes that is well-spaced internally, and adapts to the shape of the boundary.

Moreover, if the user wants nodes to be denser in some areas than others, this information is easily used to make the springs "stiffer" in some regions and "looser" in others, again creating a mesh that smoothly varies in density according to the user's request.

```
[ p, t ] = distmesh ( @fd, @fh, h, box, itmax, fixed );
```

where:

- **@fd**, the name of a distance function defining the region;
- **@fh**, the name of a mesh density function;
- **h**, the nominal mesh spacing;
- **box**, a box that contains the region;
- **itmax**, the maximum number of iterations;
- **fixed**, a list of points which must be included;
- **p**, node coordinates;
- **t**, triangles defined by node indices.

A peculiar input to **distmesh()** is the distance function **fd()**. This is the way the program expects the region to be defined. The function returns a signed distance **d** from any point **(x,y)** to the boundary of the region, with the distance being negative if the point is actually inside the region.

This makes it wonderfully easy to describe mathematical regions such as a circle of radius **r**, because in that case

$$d = \sqrt{x^2 + y^2} - r$$

However, for complicated geometries, it can be difficult to write down a good formula, and inefficient for MATLAB to evaluate it millions of times (which it must do!).

Although the L-shaped region is defined by straight line segments, the true distance function is actually pretty complicated!

That is because exterior corners of the shape create curved level sets of distance, while interior corners create sharp bends.

For convenience, **distmesh()** allows the user to define a distance function that is only approximate, but both the true distance function and the approximation can cause some odd behaviors in the mesh near corners.

And trying to write an exactly correct distance function, even for the L-shaped region, is surprisingly tricky!

Can we sketch the distance function for the L-shaped region?

distmesh() supplies some basic functions that make it easier to construct distance functions:

```
function d = p11_fd ( p )

% The L shaped region is the union of two rectangles.

  g1 = drectangle ( p, 0.0, 1.0, 0.0, 0.5 );
  g2 = drectangle ( p, 0.0, 0.5, 0.0, 1.0 );

  d = dunion ( g1, g2 );

  return
end
```
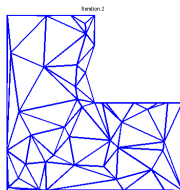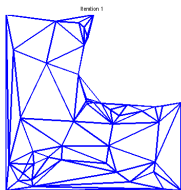
http://people.sc.fsu.edu/~jburkardt/m_src/distmesh/p11_fd.m
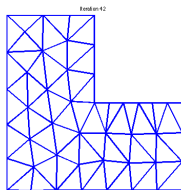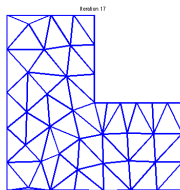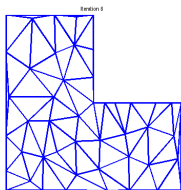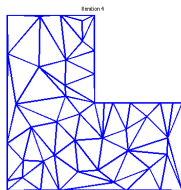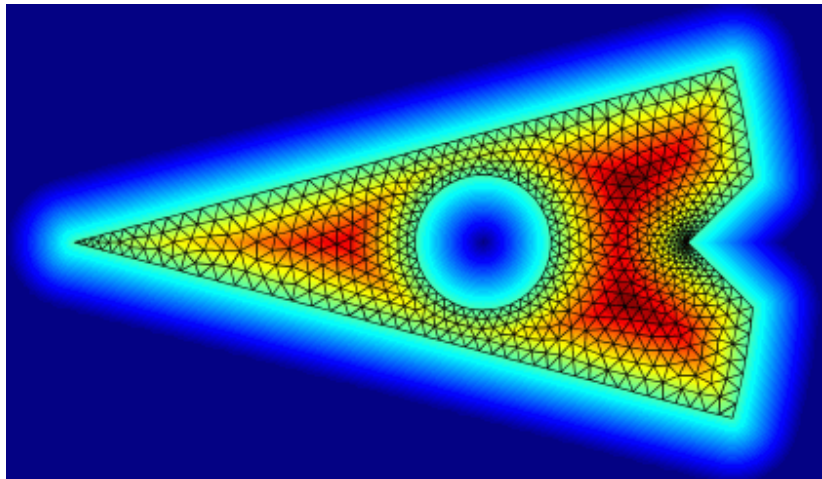
The source code for **distmesh()** is freely available at

`http://persson.berkeley.edu/distmesh/`

and a very readable and useful reference is available:

```
Per-Olof Persson, Gilbert Strang,
A Simple Mesh Generator in MATLAB,
SIAM Review,
Volume 46, Number 2, June 2004, pages 329-345.
```

`http://persson.berkeley.edu/distmesh/persson04mesh.pdf`

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- **MESH2D**
- Conclusion

mesh2d is a program which starts with some of the computational features of **distmesh()** and adds features that offer the user some important options.

In particular, rather than trying to describe the geometry with a Matlab function, we are simply going to give it as a list of points.

This means that if we want to deal with a circle, we're going to have to accept a polygonal approximation to a circle...I guess that's OK.

But can we do regions with holes? Can we use a density? Can we refine the mesh?

---

http://www.mathworks.com/matlabcentral/fileexchange/25555-mesh2d-automatic-mesh-generation

MESH2D: Usage

```
[ p, t ] = mesh2d ( vertices, edge, hdata, options );
```

where:

- *vertices*, a **V** by 2 list of boundary vertex coordinates;
- *edge*, (optional input), lists pairs of vertex indices that form the boundary;
- *hdata*, (optional input), a structure containing element size information;
- *options*, (optional input), allows the user to modify the default behavior of the solver .
- *p*, the coordinates of nodes generated by the program;
- *t*, the triangulation of the nodes.

The **mesh2d** program has some nice features:

- a very short call **[p,t]=mesh2d(v)** is possible;
- short boundary segments result in small interior elements;
- the region is described by vertices and the program is optimized for this case; this means it's actually pretty easy to triangulate a map, diagram, or CAD outline;
- the output is "clean"; duplicate and unused nodes and small elements are discarded, elements are in counterclockwise order.
- a **refine()** function can refine a mesh.
- a **smoothmesh()** function will smooth a mesh.

As examples of mesh2d usage, we can start with variations of the L-shaped problem:

```
v = [ 0.0, 0.0; 2.0, 0.0; 2.0, 1.0; 1.0, 1.0; ...
      1.0, 2.0; 0.0, 2.0 ];

[ p, t ] = mesh2d ( v );
```

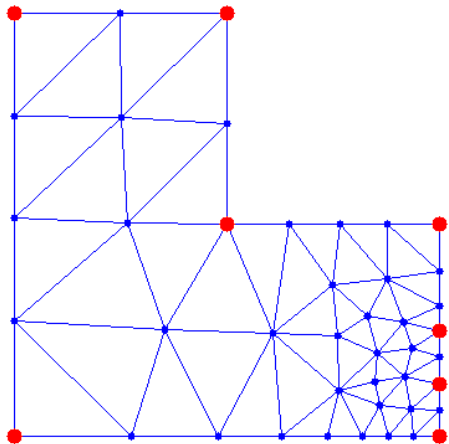http://people.sc.fsu.edu/~jburkardt/m_src/mesh2d/ell_demo.m

Suppose we add two extra boundary vertices:

```
v = [ 0.0, 0.0; 2.0, 0.0; 2.0, 0.25; 2.0, 0.5; ...
      2.0, 1.0; 1.0, 1.0; 1.0, 2.0; 0.0, 2.0 ];

[ p, t ] = mesh2d ( v );
```

Go back to the original problem, but specify a maximum element size:

```
v = [ 0.0, 0.0; 2.0, 0.0; 2.0, 1.0; 1.0, 1.0; ...
      1.0, 2.0; 0.0, 2.0 ];


hdata = [];
hdata.hmax = 0.1;

[ p, t ] = mesh2d ( v, [], hdata );
```
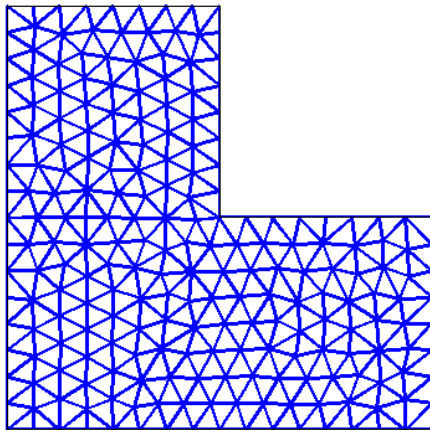
Go back to the original problem, but specify a density function so elements are small near the reentrant corner:

```
v = [ 0.0, 0.0; 2.0, 0.0; 2.0, 1.0; 1.0, 1.0; ...
      1.0, 2.0; 0.0, 2.0 ];


hdata = [];
hdata.fun = @hfun;

[ p, t ] = mesh2d ( v, [], hdata );
```

```
function h = hfun ( x, y )

%
%  Minimum size is 0.01, increasing as we move away
%  from ( 1.0, 1.0 ).
%
  h = 0.01 + 0.1 * sqrt ( ( x-1.0 ).^2  + ( y-1.0 ).^2 );

  return
end
```
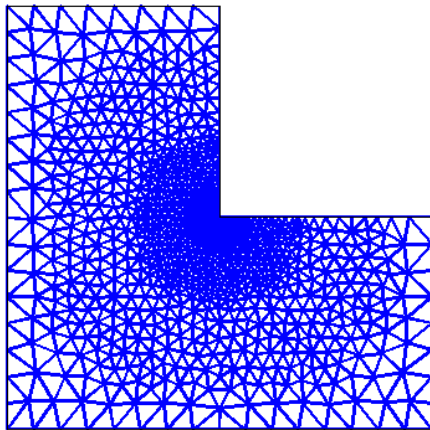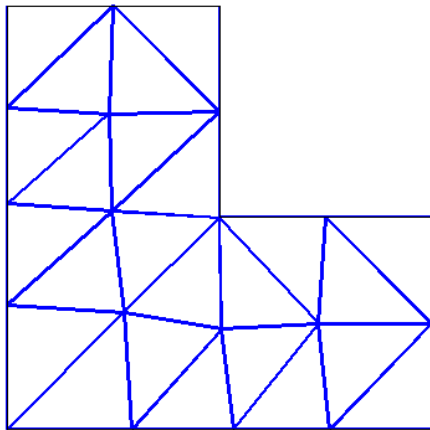
Go back to the original problem, then refine the mesh:

```
v = [ 0.0, 0.0; 2.0, 0.0; 2.0, 1.0; 1.0, 1.0; ...
        1.0, 2.0; 0.0, 2.0 ];

[ p, t ] = mesh2d ( v );
[ p, t ] = refine ( p, t );
```
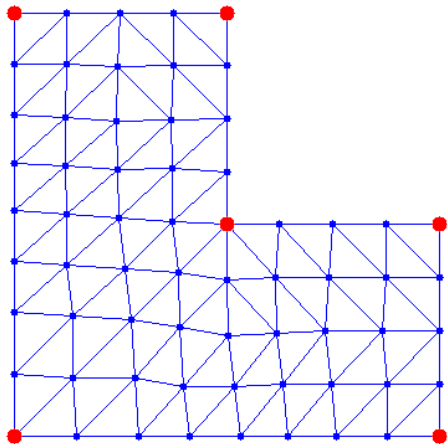
Go back to problem 2, but smooth the mesh:

```
v = [ 0.0, 0.0; 2.0, 0.0; 2.0, 0.25; 2.0, 0.5; ...
      2.0, 1.0; 1.0, 1.0; 1.0, 2.0; 0.0, 2.0 ];

[ p, t ] = mesh2d ( v );
[ p, t ] = smoothmesh ( p, t );
```
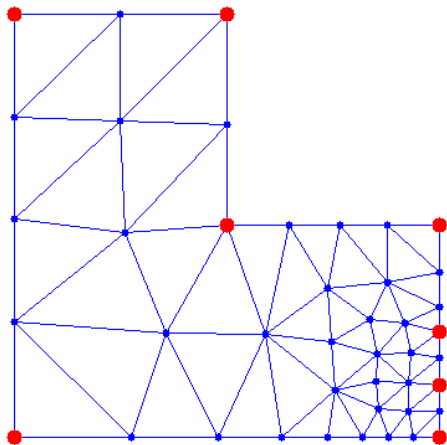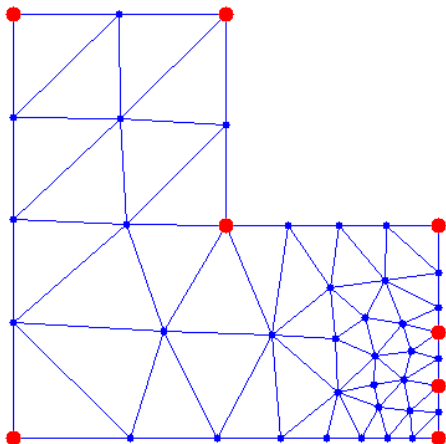
Initial mesh:

After 2 smoothing steps:

After 6 smoothing steps:

MESH2D: Reference
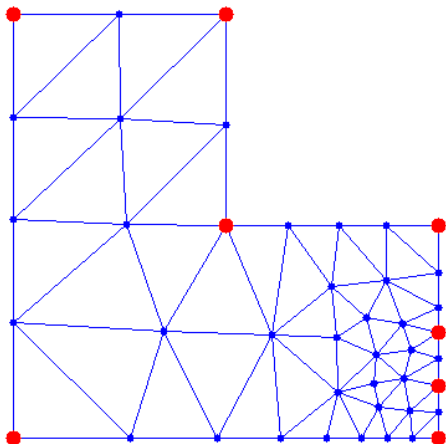
mesh2d was written by Darren Engwirda.

You can get a copy of **mesh2d** from the Matlab Central Exchange:

`http://www.mathworks.com/matlabcentral/fileexchange/...`
`25555-mesh2d-automatic-mesh-generation`

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- **Conclusion**

Do not try to write your own meshing package!

There is already excellent software available.

A mesh can be precomputed, stored in a file, and referred to later.

Graphical output enables your eye to improve or correct mistakes.

CONCLUSION: Things I Skipped

The MATLAB programs **distmesh** and **mesh2d** are accessible, powerful, usable, and graphical.

The **triangle** program is written in C, and its graphical program "showme" is awkward to use. It's easy to write your own C program that calls **triangle** as a runtime library to generate meshes.

Surface and volume meshing can be done by **Gmsh**, **medit**, **MeshLab**, **Paraview**, **tetgen** and **VisIt**.

CONCLUSION: Your Future in Meshing

Meshing creates a discrete model of reality for computers.

Meshing and grid generation arise in:

- computer graphics, 3D animation, gaming;
- computer geometry;
- facial recognition;
- GIS (geographic information systems);
- medical scan analysis;
- CAD/CAM, (computer-aided design and modeling);
- 3D "printers".

It's possible that your future will involve finite elements, but it's certain that you will work with meshes.

The concepts are simple, and there are powerful tools.