Meshing for the Finite Element Method

Summer Seminar
ISC5939

....

John Burkardt
Department of Scientific Computing
Florida State University
http://people.sc.fsu.edu/∼jburkardt/presentations/. . .
. . . mesh_2012_fsu.pdf

10/12 July 2012

FEM Meshing

- **Meshing**
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$D Problems
- 3D Problems
- Conclusion

MESHING:

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.

MESHING:

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 very simple idea:

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!

MESHING:

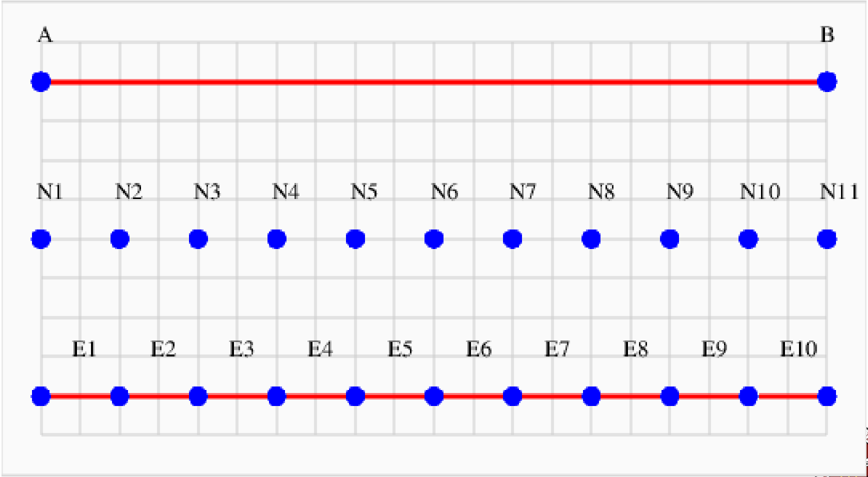
Indeed, if we start by thinking of a 1D problem, such as modeling the temperature along a thin strand of wire that extends from **A** to **B**, our meshing problem is trivial:

- Choose **N**, the number of subregions or elements;
- 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.



MESHING: Nodes and Elements in 1D

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

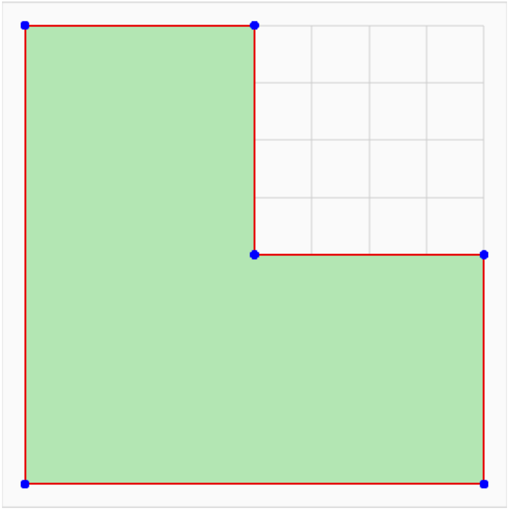
For our basic 2D example, we’ll consider an L-shaped region, and show how to go through the basic meshing steps.

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.

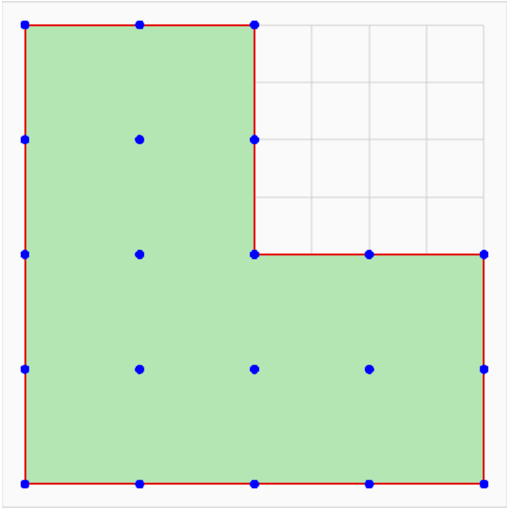
MESHING: Nodes and Elements in 2D

6 vertices define the shape.



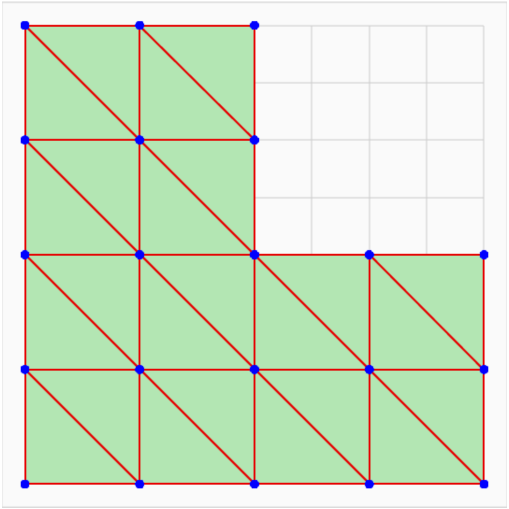
MESHING: Nodes and Elements in 2D

21 nodes will be used for the internal mesh.



MESHING: Nodes and Elements in 2D

24 triangular elements constitute the mesh.



MESHING: Rectangular Regions Are Not Good Enough!

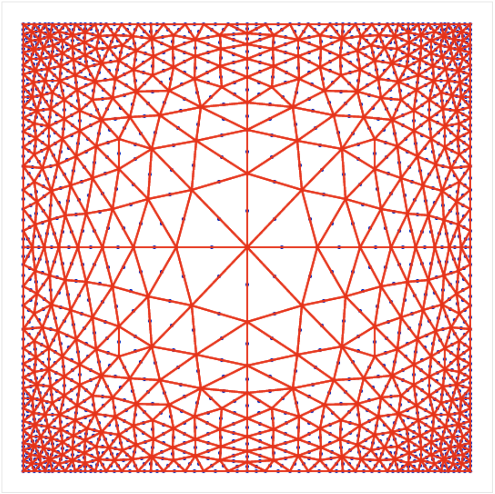
While a mathematician or academic computing person might regard our L-shaped region as wildly irregular, a person who actually needs to use the finite element method will regard the use of purely rectangular regions as unrealistic and much too limited to be useful.

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?

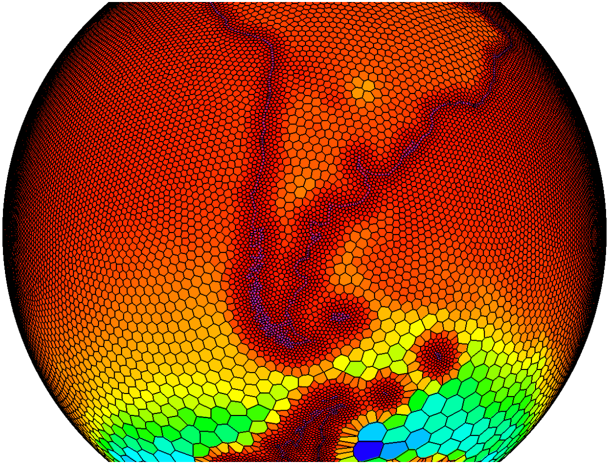
MESHING: We Want Small Elements Near Boundaries!

We need meshes that automatically vary in density.



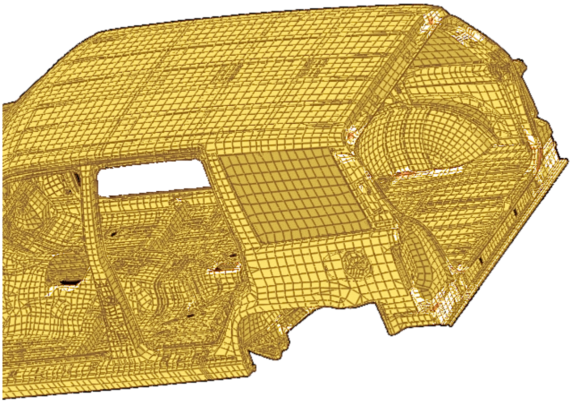
MESHING: We May Need Transition Zones!

We need the mesh to be able to get small near "sensitive spots".



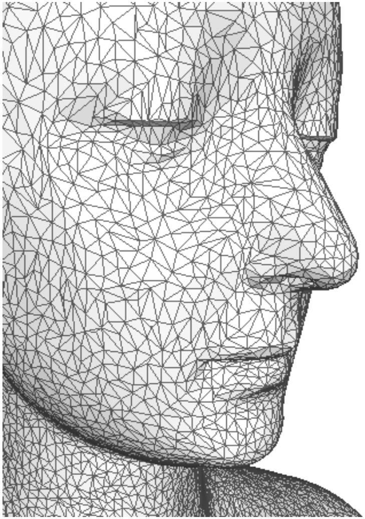
MESHING: We Want to Handle Holes (and Surfaces)!

We need to mesh surfaces that include holes and edges.



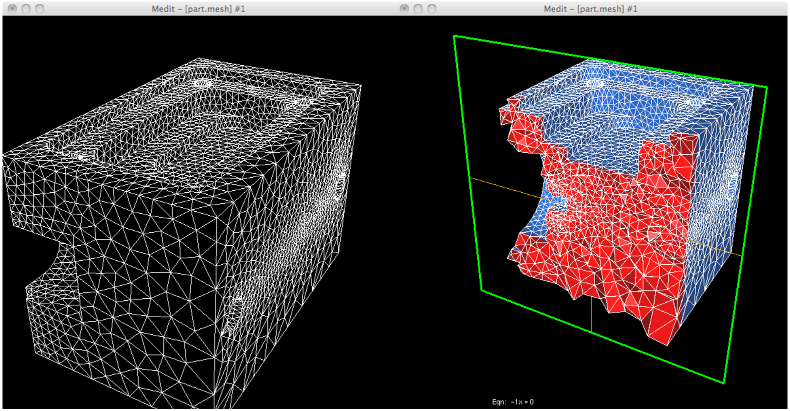
MESHING: On a Surface, We Must Match Curvature!

The mesh must notice and adapt to local features (here, curvature.)



MESHING: We May Need an Internal Mesh

For a true 3D problem, we need nodes and elements inside the surface



MESHING: Realistic Problems Can't Be Meshed by Hand!

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?

FEM Meshing

- Meshing
- **Computer Representations**
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$D Problems
- 3D Problems
- Conclusion

REP: Data and Files for Meshes

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.

REP: Define a Region by Boundary Vertices

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.

REP: Define a Region That is a Square Minus a Triangle

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.

REP: Draw A Vertex Boundary of Multiple Line Segments

```
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

REP: Nodes Fill the Region and the Vertex Boundary

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 )
```

REP: Triangles Are Formed From Nodes

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) )
```

REP: The Node Boundary is Edges Used Once

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.

REP: Triangle Neighbors

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.

FEM Meshing

- Meshing
- Computer Representations
- **The Delaunay Triangulation**
- TRIANGLE
- DISTMESH
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$D Problems
- 3D Problems
- Conclusion

DELAUNAY: Whence P and T?

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.

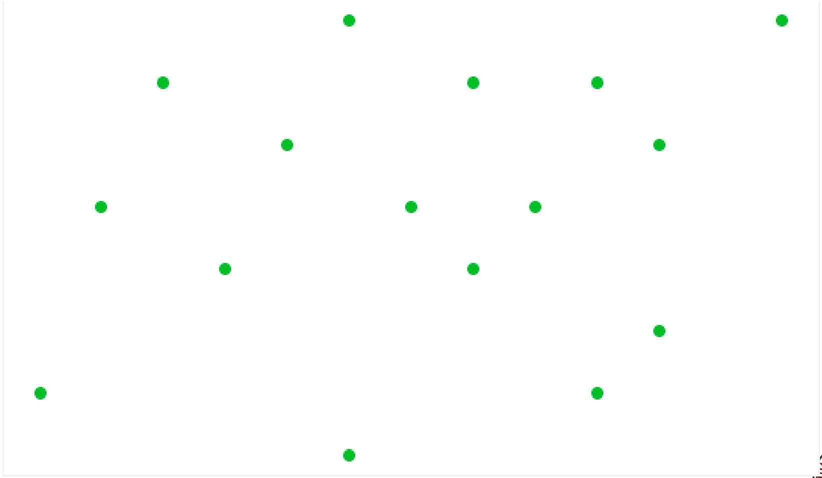
DELAUNAY: A Maximal Triangulation

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.

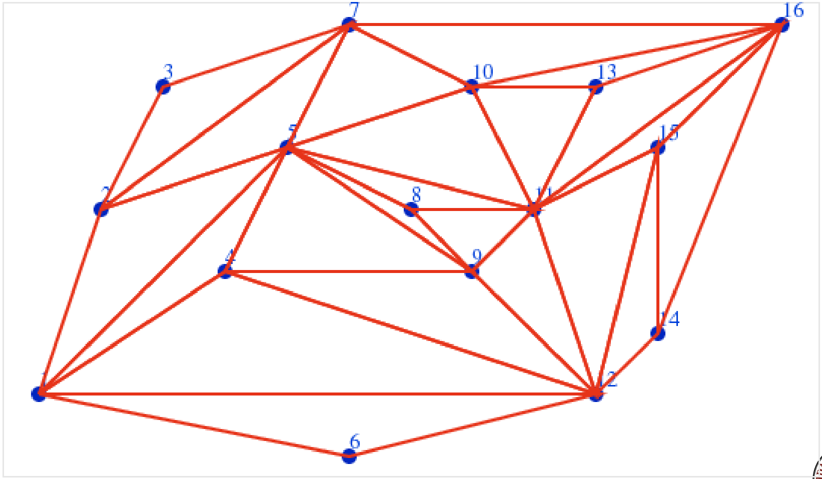
The process seems pretty arbitrary, and it fact there are many possible triangulations of a set of points. You may wonder how to automate this process; a natural way is to start by creating a giant triangle that encloses all the points you are going to use.

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.

DELAUNAY: Sixteen Nodes to Triangulate



DELAUNAY: A Triangulation of 16 Nodes



DELAUNAY: What is a “good” triangulation?

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.

DELAUNAY: What is a “good” triangulation?

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

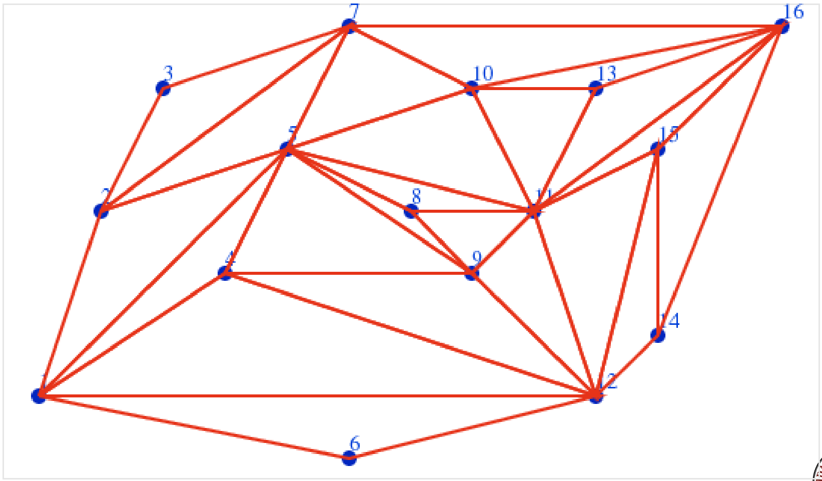
Strictly speaking, we consider all possible triangulations of a set of nodes. For each triangulation T, let $\theta(T)$ be the smallest angle that occurs in any triangle of that triangulation. Then a triangulation T^* is a Delaunay triangulation if

$$\theta(T) \leq \theta(T^*)$$

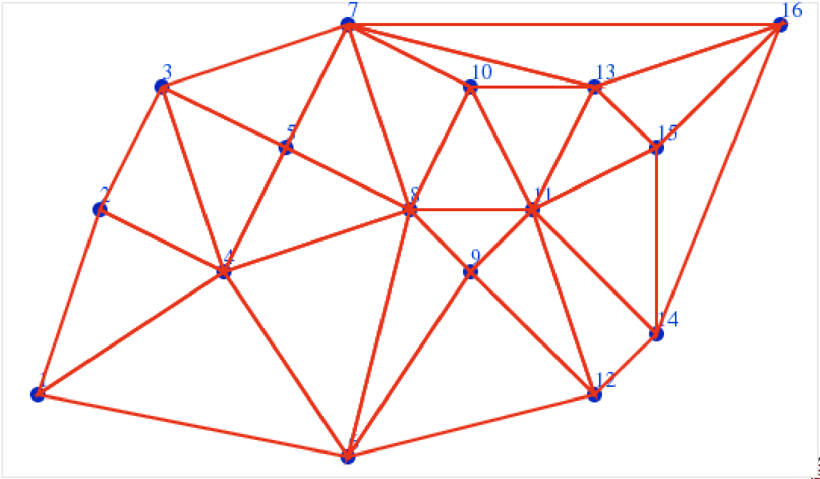
for all triangulations T.

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.

DELAUNAY: A Triangulation of 16 Points



DELAUNAY: A Delaunay Triangulation of 16 Points



DELAUNAY: A Delaunay Triangulation of 16 Points

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 come, 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!

DELAUNAY: An Algorithm

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...

DELAUNAY: MATLAB Calculation

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) )
```

or

```
t = delaunayn ( p )
```

To display the triangulation,

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

FEM Meshing

- Meshing
- Computer Representations
- The Delaunay Triangulation
- **TRIANGLE**
- DISTMESH
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$D Problems
- 3D Problems
- Conclusion

TRIANGLE: The C Program "Triangle"

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

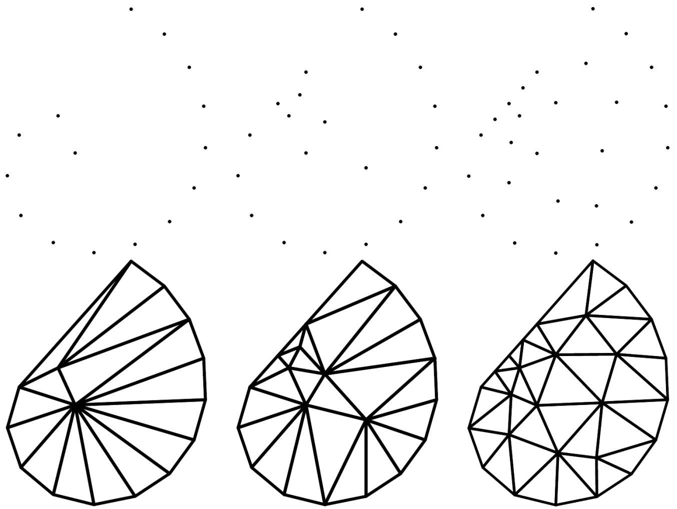
```
# spiral.node
15  2  0  0            <-- Point count, dimensions,
 1      0.00    0.00        attributes, boundary markers.
 2     -0.42    0.91
 3     -1.35    0.43
...     ...     ...
14      2.16    2.89
15      1.36    3.49
```

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
```

TRIANGLE: Angle Constraints

No New Points || Minimum Angle 20^o || Minimum Angle 32.5^o



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:

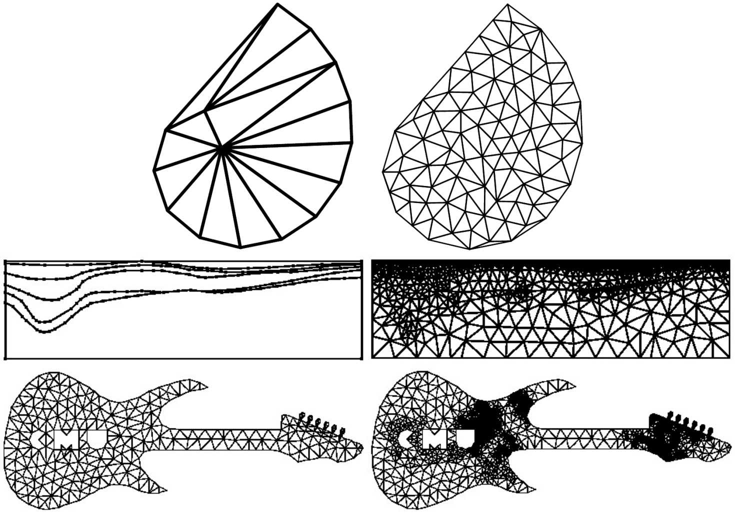
```
triangle -a0.2 spiral
```

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.

TRIANGLE: Area Constraints

Global Maximum || Subdomain Maximum || Pointwise Maximum



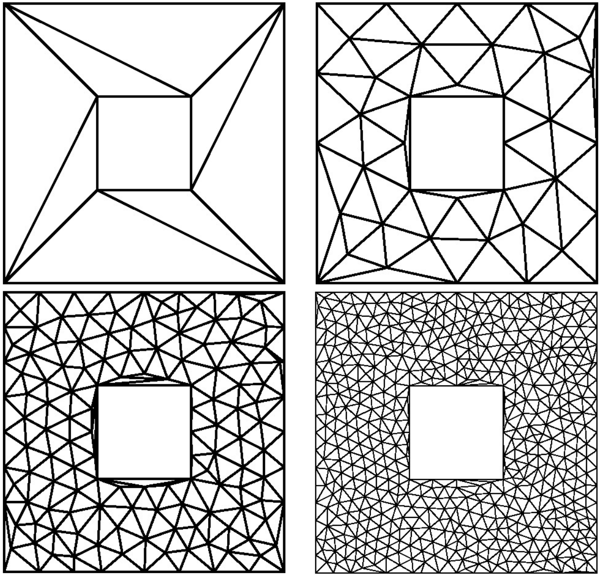
TRIANGLE: Refinement

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
```

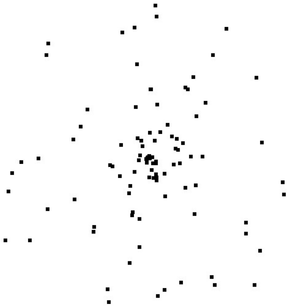
TRIANGLE: Refinement of the Box Mesh

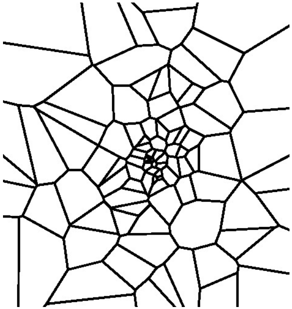


TRIANGLE: Voronoi Diagram

triangle can compute the Voronoi diagram of the nodes.

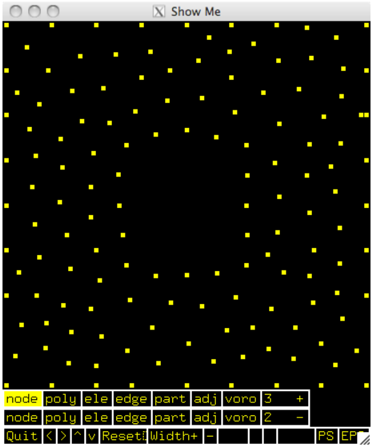
```
triangle -v dots
```

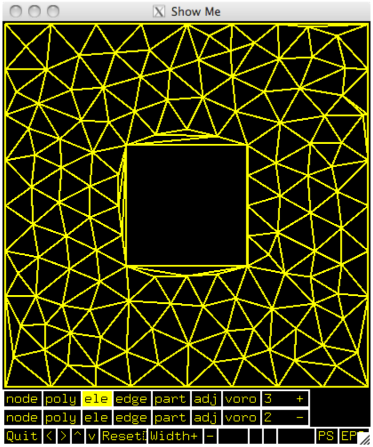




TRIANGLE: The SHOWME Graphics Interface

triangle includes a graphics program called **showme**, which can display the nodes, edges, triangulation, or Voronoi diagram.


Show Me
node poly ele edge part adj voro 3 +
node poly ele edge part adj voro 2 -
Quit < > ^ v Reset Width+ - PS EP


Show Me
node poly ele edge part adj voro 3 +
node poly ele edge part adj voro 2 -
Quit < > ^ v Reset Width+ - PS EP

TRIANGLE: Comments

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.

FEM Meshing

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- **DISTMESH**
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$D Problems
- 3D Problems
- Conclusion

DISTMESH: Using T to Improve P

So any set of nodes **P** defines a Delaunay triangulation **T**. How can we use **T** to improve **P**?

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.

DISTMESH: Using T to Improve P

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.

DISTMESH: Usage

```
[ 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.

DISTMESH: Region Defined by Signed Distance

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!).

DISTMESH: The ELL Region

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!

Remind me to sketch the L-shaped distance function now!

DISTMESH: Distance function for the L Region

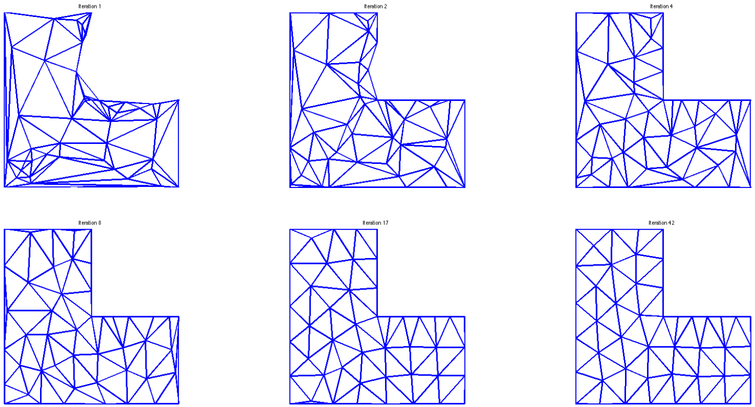
distmesh() supplies some basic functions that make it easier to construct distance functions:

```
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
```

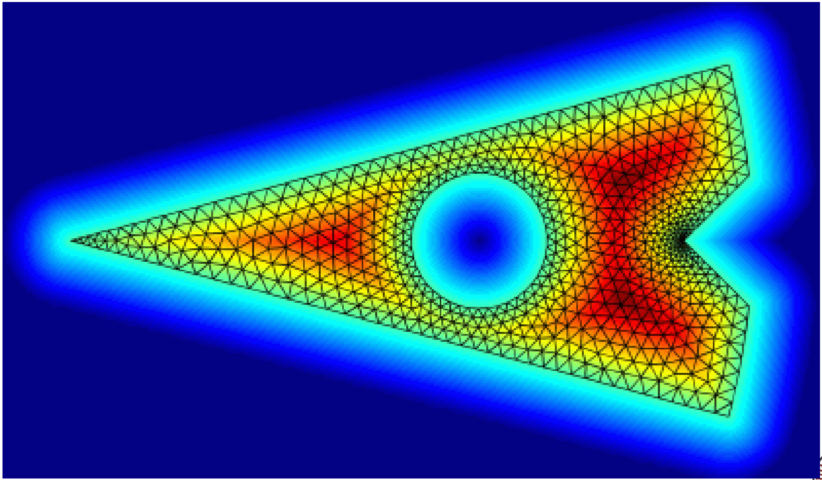
---

http://people.sc.fsu.edu/∼jburkardt/m_src/distmesh/p11_fd.m

DISTMESH: First Iterates for the L Region



DISTMESH: Distance function and Mesh for “Holey Pie”



DISTMESH: Reference

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`

FEM Meshing

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- **MESH2D**
- Files and Graphics
- $2\frac{1}{2}$D Problems
- 3D Problems
- Conclusion

MESH2D: Region Defined by Vertices

Darren Engwirda has adapted some of the ideas from **distmesh()** and added some new features that offer a second powerful and flexible MATLAB meshing program called **mesh2d()**.

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

`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.

MESH2D: Usage

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.

MESH2D: Simple ELL Mesh

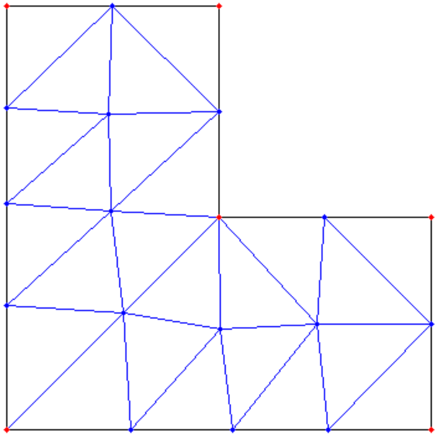
As examples of mesh2d usage, we can start with variations of the L-shaped problem:

```
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

MESH2D: Simple ELL Mesh



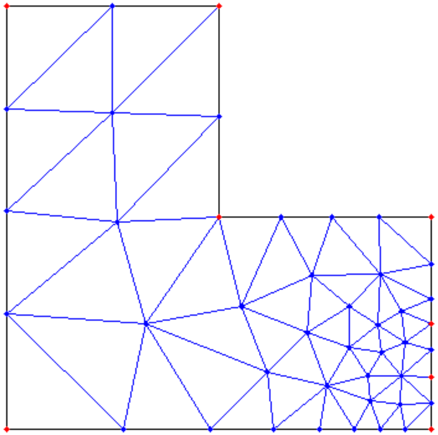
MESH2D: Two Short Boundary Segments

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 );
```

MESH2D: Two Short Boundary Segments

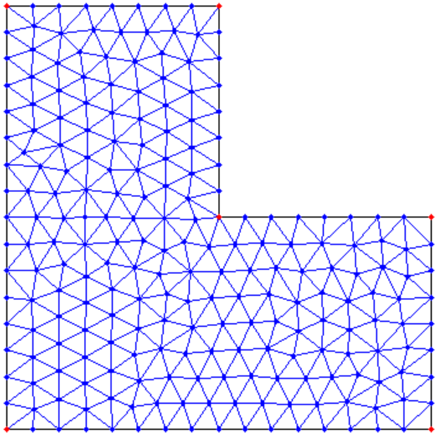


MESH2D: Set Maximum Element Size

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 );
```

MESH2D: Set Maximum Element Size



MESH2D: Use a Density Function

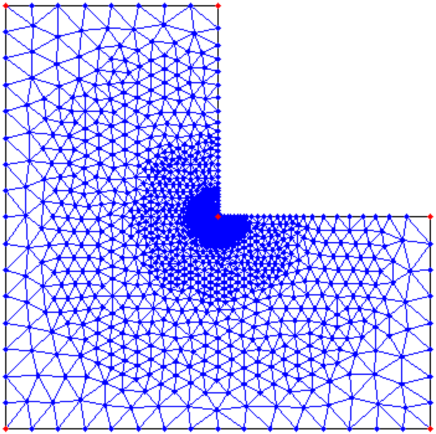
Go back to the original problem, but specify a density function so elements are small near the reentrant corner:

```
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 );
```

MESH2D: Use a Density Function

```
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
```

MESH2D: Use a Density Function

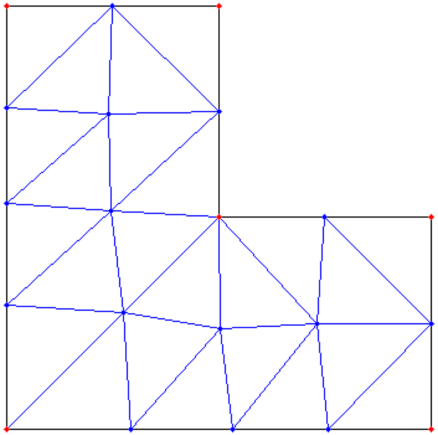


MESH2D: Refine Example 1

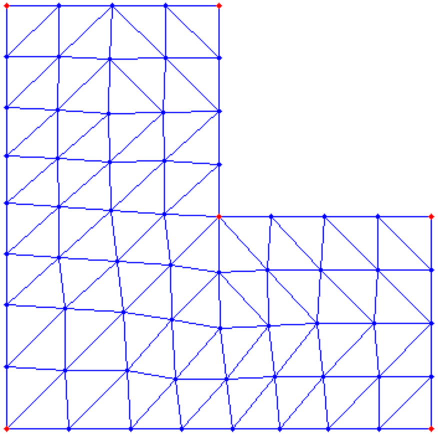
Go back to the original problem, then refine the mesh:

```
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 );
```

MESH2D: Refine Example 1



MESH2D: Refine Example 1



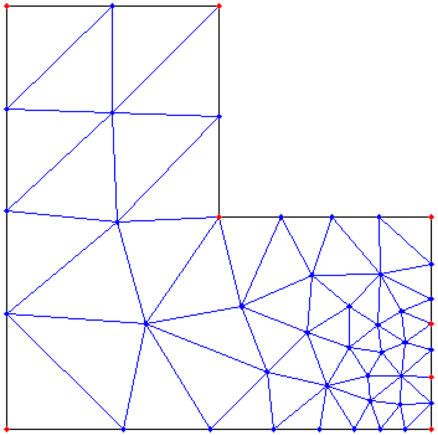
MESH2D: Smooth Example 2

Go back to problem 2, but smooth the mesh:

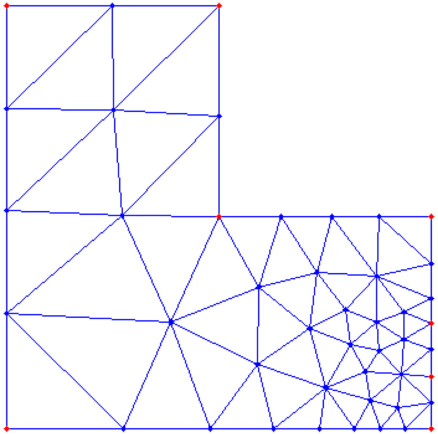
```
v = [ 0.0, 0.0; 2.0, 0.0; 2.0, 0.25; 2.0, 0.5; ...
      2.0, 1.0; 1.0, 1.0; 1.0, 2.0; 0.0, 2.0 ];
```

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

MESH2D: Smooth Example 2



MESH2D: Smooth Example 2



FEM Meshing

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- **Files and Graphics**
- $2\frac{1}{2}$D Problems
- 3D Problems
- Conclusion

FILES: Compute Your Mesh Ahead of Time!

A person writing a finite element program does not need to do the mesh generation inside the program! It is far better to take advantage of good software written by others.

The easiest way to handle this issue is to create the mesh ahead of time, and write it to a file.

The **triangle** program automatically creates files of output.

distmesh() and **mesh2d()** can be convinced to do so.

Such files can be easily read back into a finite element program written in C, FORTRAN, PYTHON, or any appropriate language.

FILES: What a Node File Might Look Like

The fundamental mesh quantities are the arrays **P** and **T**, and if we understand them, we can handle other items, such as the triangle neighbor list, or the boundary node list.

When dealing with communication between programs, the best idea is to keep things simple. So we will create one file for each array. Text files are bigger, but easier to read than binary files. Since **P** is an array of **P_NUM** rows by 2 columns, our file will contain that many rows and columns of data.

The **P** file for the *ell* problem should look something like this:

```
0.0  0.0
1.0  0.0
2.0  0.0
...  ...
2.0  4.0
```

FILES: What a Triangle File Might Look Like

Similarly there should be a separate **T** file, and because it contains integers, we want to read and write it with an integer format. *(Note that the MATLAB* **save** *command writes integers with a real number format that can cause problems later.)*

The **T** file for the *ell* problem should look something like this:

```
  1   2   6
  7   6   2
  2   3   7
  8   7   3
 ..  ..  ..
 17  18  20
 21  20  18
```

Any programming language should be able to read such files and store the corresponding data.

FILES: A MATLAB Function To Write Integer Data

```
function triangle_write ( outfile, m, n, table )

  outunit = fopen ( outfile, 'wt' );

  for j = 1 : n
    for i = 1 : m
      fprintf ( outunit, '  %12d', round ( table(i,j) ) );
    end
    fprintf ( outunit, '\n' );
  end

  fclose ( outunit );

  return
end
```

---

http://people.sc.fsu.edu/∼jburkardt/m_src/fem_meshing/triangle_write.m

FILES: The FEM Program Needs to Allocate Space

It's a good idea to try to let the FEM program figure out the size of the array simply by reading the file. That way, the same program can solve problems of different sizes without needing to be modified and recompiled.

It is not too difficult to write functions that will count the number of lines in a file, and the number of data items on a single line. This gives you the number of rows and columns you need to allocate for your array.

But if you don't like my idea, you can always put the number of rows and columns as the first line of the file!

Once your FEM program knows how big the array is that is described by the file, it can allocate the necessary space, and read the actual data.

FILES: C++ Reads an MxN Integer Array

```
int **triangle_read ( string infile, int m, int n )
{
  ifstream inunit;
  int i, j;
  int **t;

  inunit.open ( infile.c_str ( ) );

  t = i4mat_new ( m, n );  <-- Set up a two dimensional array t[][]

  for ( i = 0; i < m; i++ )
  {
    for ( j = 0; j < n; j++ )
    {
      inunit >> t[i][j];
    }
  }

  inunit.close ( );

  return table;
}
```

---

http://people.sc.fsu.edu/∼jburkardt/m_src/fem_meshing/triangle_read_example.cpp

FILES: FORTRAN Reads an MxN Integer Array

```
subroutine triangle_read ( infile, m, n, t )

  integer m, n
  integer i
  character ( len = * ) infile
  integer t(m,n)

  open ( unit = 1, file = infile, status = 'old' )
  do i = 1, m
    read ( 1, * ) t(i,1:n)
  end do
  close ( unit = 1 )

  return
end
```

---

http://people.sc.fsu.edu/∼jburkardt/m_src/fem_meshing/triangle_read_example.f90

FILES: PYTHON Reads an MxN Integer Array

```
define triangle_read ( filename )

input = open ( filename, 'r' )

t = []

for line in input.readlines():
  x = line.split ( )
  t.append ( [ int ( xi ) for xi in x ] )

input.close ( )

return t
```

---

http://people.sc.fsu.edu/∼jburkardt/m_src/fem_meshing/triangle_read_example.py

FILES: Advantages to Using Files

When you let a program like **mesh2d()** create your finite element mesh, you get the advantage of being able to handle general regions, variable size meshing, and so on.

But by storing your meshes as files, you also make it possible to

- think about your FEM program independently of the mesh, so that it can be written to solve any size or shape problem;
- feed the mesh to different FEM programs, compare results;
- compute a complicate mesh once, use it many times;
- plot the mesh any time, without running the FEM program;
- add/delete elements or nodes by (carefully) modifying files;
- compute related quantities (such as the triangle neighbor list) by working directly on the files.

FILES: Visualizing a Mesh

If you can store your mesh information as **p** and **t** data, then you can save it to a file and pass it to a graphics program for visualization. One reason is simply to look at your mesh and make sure it corresponds to your geometry.

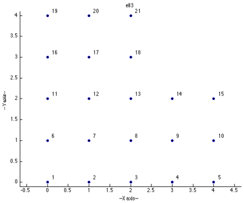
- Triangle's **showme** program displays nodes or elements; otherwise, the **.node** and **.ele** files contain **p** and **t**;
- MESH2D displays the mesh as you go, and returns **p** and **t**;
- DISTMESH returns the **p** and **t** arrays;

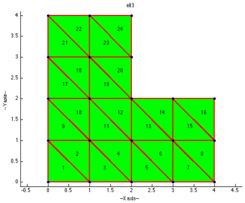
If you have **p** and **t** available, you can display the mesh with MATLAB commands like:

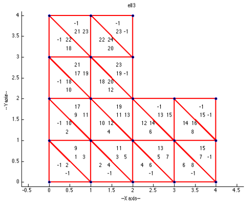
```
trimesh ( t, p(:,1), p(:,2), zeros(N,1) )
```

FILES: Numeric Displays for Debugging

Sometimes you want to debug a mesh, and see the numeric labels for the nodes, elements, or element neighbors:







http://people.sc.fsu.edu/∼jburkardt/m_src/triangulation_display/triangulation_display.m

FILES: For $2\frac{1}{2}$D and 3D

For the 2D case we have looked at so far, there are not many standard file formats, perhaps because the problem is pretty simple.

For $2\frac{1}{2}$D and 3D, however, there is a lot more information to store (more nodes, more connectivity, more element choices), and there are many applications (biomedical scan analysis, computer graphics, geographic information, computer-aided design) so many file formats have been created which add features such as texture, color, surface normals, and so on.

For these more complicated problems, you will probably want to use a standard format, especially if you can find a good program to display and modify your mesh. We will see a few examples shortly.

FEM Meshing

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$**D Problems**
- 3D Problems
- Conclusion

$2\frac{1}{2}$D:

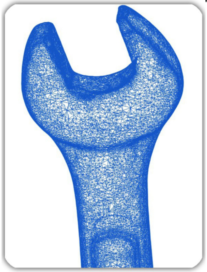
Since we don't have X-ray vision, we can't see inside objects. We can't even see the backs of objects. That means that if we want to make a computer graphics image of a 3D region, we can simplify out job by only modeling the surfaces.

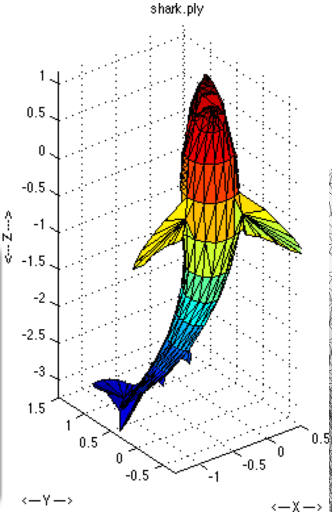
Our data may come from a 3D scanner, in which case we only have surface data.

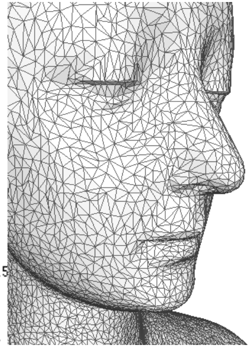
Sometimes the region we're looking at really is essentially 2D, although it "lives" on a curved 3D surface. You may have noticed this about the earth, for instance. A simple model of weather requires a "flat" mesh on the surface of the sphere.

Even if we are interested in a true 3D problem, we might start by working on the surface, since a good representation of the boundary is enough for a meshing program to fill in the interior.

$2\frac{1}{2}$D: Approximating Surfaces







$2\frac{1}{2}$D: Data Description

For a simple problem, it's easy to see how a surface can be described by a 3D mesh of 2D triangles. We could imagine flexible paper, printed with triangles, that we can use to wrap around a shape, such as a sphere or a teapot. We'd have to stretch the paper, and cut it, and glue it, but it would still be essentially a sheet of paper that's been mildly distorted.

That means the geometry can again be defined by a set of points **p** and triangles **t**, just like in the 2D case, except that:

- the points have 3D coordinates;
- the connectivity can become very complicated.
- the Delaunay criterion is difficult to apply.

$2\frac{1}{2}$D: Scanner Data

What tools are there for creating a "good" 3D mesh of triangles?

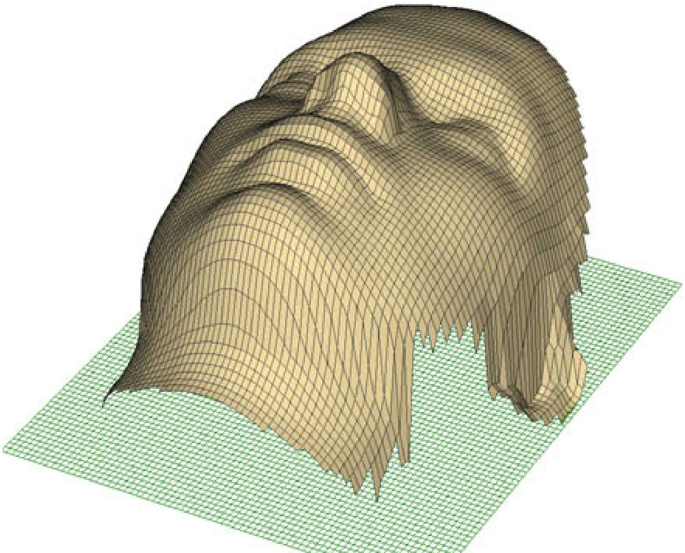
If your data is coming from a 3D scanner, then you simply have to collect the point data **p**.

It represents one coordinate, say z, as a function of (x, y), and samples on a regularly spaced m by n grid of points (x_i, y_j). Thus, your data is "logically" a rectangular array. Diagonally slicing each rectangle gives you triangles, and your **t** array of connectivity.

However, you probably need to do some processing to eliminate data points where the scanner did not detect the object, and hence measured the background.

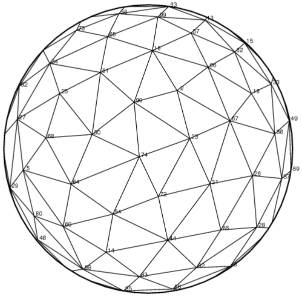
If your object has folds, or levels, or hollows, these will not be detected by the scanner. And the scanner won't give you the hidden back side of the object.

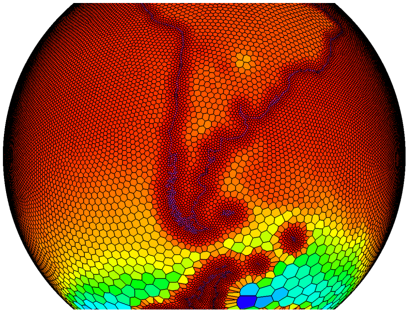
$2\frac{1}{2}$D: Scanned Surface



$2\frac{1}{2}$D: Grids on a Sphere

If the surface you are studying is regular, then it may be possible to lay out a smoothly varying grid, and to construct a Delaunay triangulation. The classic example of this involves meshes on a sphere, especially when used to model climate on the surface of the earth.





$2\frac{1}{2}$D: File Formats

A lot of information is stored as surface grids, and for this reason many formats have arisen for organizing this information. Using a common format allows your information to be recognized and used by a variety of programs for computation or graphical display.

These formats are recognized by their filename extensions:

- **.mesh**: medit mesh file format
- **.obj**: wavefront object format
- **.off**: geomview object file format;
- **.ply**: polygon file format/Stanford triangle format
- **.poly + .node**: 3D version of Triangle files
- **.smesh**: medit surface mesh file format
- **.stl**: stereolithography

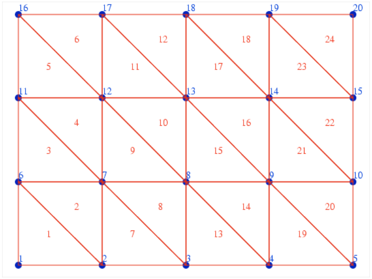
$2\frac{1}{2}$D: Example OBJ File

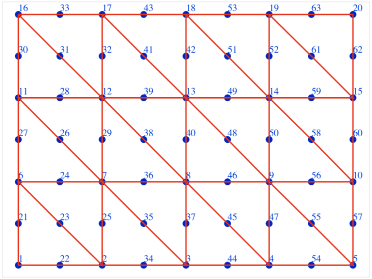
```
g Octahedron          <-- begin object

v   1.0   0.0   0.0   <-- vertex 1 coordinates
v   0.0  -1.0   0.0
v  -1.0   0.0   0.0
v   0.0   1.0   0.0
v   0.0   0.0   1.0
v   0.0   0.0  -1.0

f  2  1  5  <-- face 1 uses vertices 2, 1, 5
f  3  2  5
f  4  3  5
f  1  4  5
f  1  2  6
f  2  3  6
f  3  4  6
f  4  1  6
```

$2\frac{1}{2}$D: Finite Element Applications

Although many of the file formats were developed for computer graphics, they usually support triangular elements, and thus have the node and element information you would need to describe a piecewise linear finite element model on your surface.

If you wanted quadratic triangles, you might simply refine each linear triangle by computing the locations of the nodes at the midpoint of each side and adding these to the triangulation.





$2\frac{1}{2}$D: Graphics

MeshLab is an open source, portable, and extensible system for the processing and editing of unstructured triangular meshes in 3D.

MeshLab is aimed to help the processing of the typical not-so-small unstructured models arising in 3D scanning, providing a set of tools for editing, cleaning, healing, inspecting, rendering and converting this kind of mesh.

MeshLab can visualize your mesh, but not your finite element solution. If you want contours of scalars, or vector flow fields, you need to consider working in MATLAB, or try sophisticated graphics package such as ParaView or Visit.

---

http://meshlab.sourceforge.net

FEM Meshing

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$D Problems
- **3D Problems**
- Conclusion

3D: Problems Are Harder, But Important

The finite element method works the same way in 3D as in 2D.

If we used triangles in 2D, it is natural to go to tetrahedrons in 3D. The same Delaunay principles can be used to construct elements that are good because they avoid small angles.

The elements depend on the choice of nodes, so we also need a way to place nodes in the region, well separated, and perhaps distributed according to a mesh density specified by the user.

The treatment of the boundary can become more difficult, since the boundary is now a surface, rather than a curve.

We'll hope that if we can describe the geometry and mesh density of our region in a simple, mathematical way, that the meshing software will take care of treating the boundary, filling the region with nodes, and producing the elements.

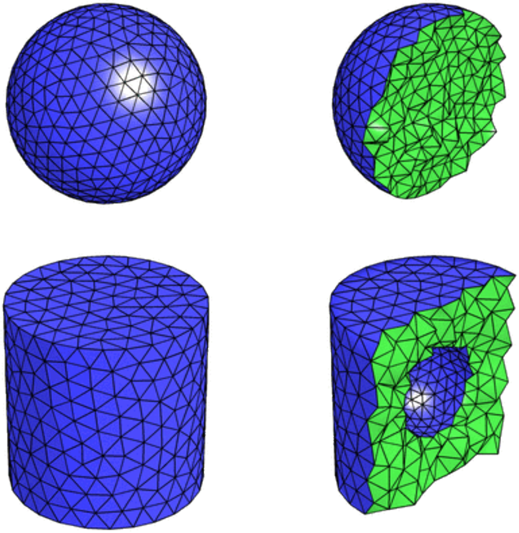
3D: DISTMESH

One of the beautiful things about the DISTMESH approach is that the 3D problem works the same as the 2D problem.

The algorithm has the same logic: imagine a set of points in 3D. The 3D Delaunay "triangulation" identifies nodes that are neighbors. Assume a force between neighbors, inversely proportional to distance. Move all the nodes a small amount, in accordance with the forces. Nodes that move outside the region must be pushed back into the region. Repeat until the mesh “settles down”.

The user simply has to produce a distance computation for the 3D shape instead of 2D. This does not have to be exact; it is easy to handle a region that is the logical sum or difference of simple geometric shapes.

3D: DISTMESH



3D: Start with a PLY File

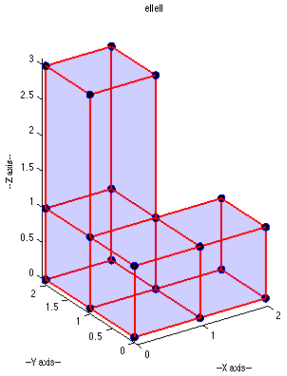
An alternative approach starts by describing the bounding surface and looking for a program that will fill in the interior.

Our plan is:

- Describe our bounding surface using polygons
- Store this information in a file
- Find a program can use the file and fill in nodes, and construct tetrahedral elements

3D: Our 3D Region “LL”

Let’s take a simple region, which we might imagine is an office building, with an L-shaped base and an L-shaped profile. Counting the corners, we find we need 20 vertices. We subdivide the surface into 18 rectangular faces.



3D: The PLY File

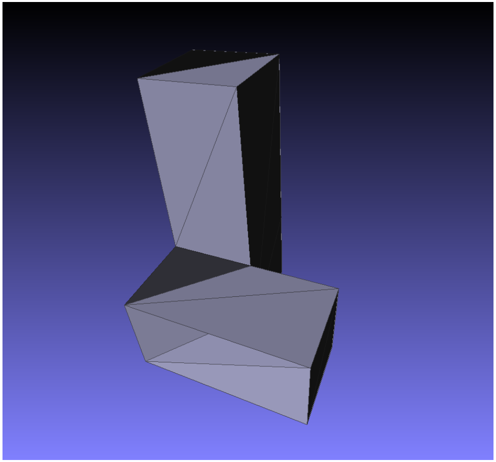
```
ply
format ascii 1.0
element vertex 20
property float32 x
property float32 y
property float32 z
element face 18
property list uint8 int32 vertex_index
end_header
0  0  0              <-- coordinates of node 0
1  0  0              <-- coordinates of node 1
2  0  0
0  1  0
1  1  0
2  1  0
0  2  0
1  2  0
...
0  2  3              <-- coordinates of node 19

4  0  3  4  1        <-- 4 nodes make up face 0
4  1  4  5  2        <-- 4 nodes make up face 1
4  3  6  7  4
4  8  9 12 11
...
4 14 15 19 18        <-- 4 nodes make up face 17
```

http://people.sc.fsu.edu/∼jburkardt/m_src/xyzf_display/xyzf_display.html

3D: MESHLAB Can Display Our Data

MESHLAB can display our PLY file data:

```
meshlab ellell.ply
```



3D: TETGEN can “fill in” the interior mesh

tetgen is a C++ program which can generate a Delaunay tetrahedral mesh that fills a region specified by the user.

The mesh creation command might have the form:

```
tetgen -p ellell.ply
```

The **.ply** format is not the only input choice to **tetgen**:

- **.mesh**: medit mesh file format
- **.off**: geomview object file format;
- **.ply**: polygon file format/Stanford triangle format
- **.poly + .node**: 3D version of Triangle files
- **.smesh**: medit surface mesh file format
- **.stl**: stereolithography

http://tetgen.berlios.de/

3D: TETGEN's Output

The mesh information that **tetgen** creates is stored as three files:

If the input file was *ellell.ply*, then the mesh will be stored in:

- *ellell.1.node*: the mesh node coordinates
- *ellell.1.ele*: the mesh elements;
- *ellell.1.face*: the mesh faces

The mesh can be view with **tetview**:

```
tetview ellell.1
```

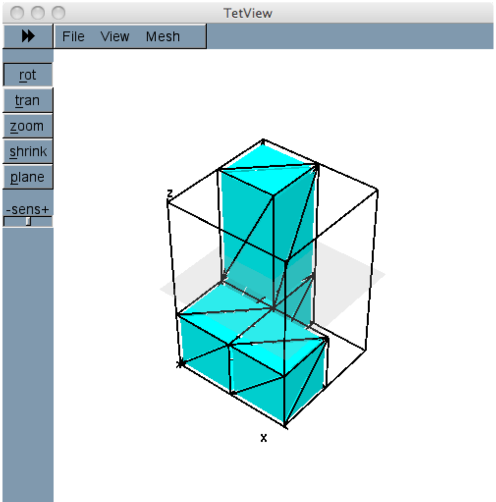
---

http://tetgen.berlios.de/tetview.html

3D: TETGEN Can Mesh the Data

TETGEN can create the Delaunay mesh of 24 tetrahedrons and 66 triangular faces.

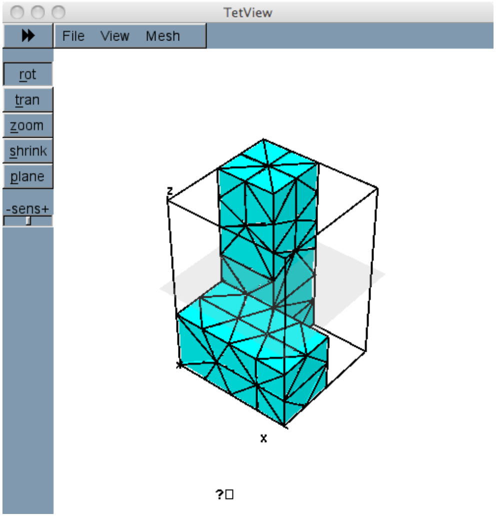
```
tetgen -pq ellell.ply
```



3D: TETGEN Can Refine the Mesh

TETGEN can refine the Delaunay mesh to 132 tetrahedrons:

```
tetgen -ra0.10 ellell.1
```



3D: TETGEN Files Used by FEM Code

The node, element and face files created by TETGEN can be read by a user program to define the mesh for a finite element calculation.

If a quadratic mesh is desired that uses 10-node tetrahedrons, the user can simply compute the midpoints of the six edges of each tetrahedron, and add these points to the mesh in the appropriate way. (In fact, TETGEN includes an option to automatically generate such a mesh, saving the user a lot of effort.)

TETGEN can also be used as a library, which means that a running program can add, move, or delete points, call TETGEN to update the mesh, and then continue computing.

FEM Meshing

- Meshing
- Computer Representations
- The Delaunay Triangulation
- TRIANGLE
- DISTMESH
- MESH2D
- Files and Graphics
- $2\frac{1}{2}$D Problems
- 3D Problems
- **Conclusion**

CONCLUSION: The Whole Talk in One Slide

I have suggested that computing a good mesh for a big, interesting region is possible, important, but too hard for the average programmer to worry about.

There is good software available to carry out this task.

The meshing can be done in advance of the finite calculation, and the mesh data stored as files in a simple way.

I've suggested some MATLAB software that is easy to use.

Since graphical output can be crucial for checking a mesh, I've outlined some procedures for displaying mesh data.

CONCLUSION: Things I Skipped

I concentrated on the MATLAB programs **distmesh** and **mesh2d** because they are accessible, powerful, usable, and easy to display graphically.

The **triangle** program is written in C, and its graphical interface program "showme" is somewhat difficult and awkward to use. However, a C program that you write can access **triangle** as a library, which means you can have an efficient code that generates meshes as part of a larger calculation.

For the 3D discussion, I concentrated on **tetgen** simply because it was free, included a simple graphics interface, and accepted file formats I was familiar with. If you are interested in 3D problems, there are many more graphics packages available, including **medit**, **Paraview**, and **VisIt**, and other choices for mesh generators.

CONCLUSION: Your Future in Meshing

While we have encountered meshing from a finite element approach, it's really a fundamental operation of computational science, coming up whenever a geometric object needs to be represented, analyzed, and manipulated.

This means understanding meshing gives you an opening into

- 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".

In other words, while it's possible that your future will involve working with finite elements, it's certain that you will be working with meshes. Learn to love them!