Triangle
A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.
Version 1.3

Copyright 1996 Jonathan Richard Shewchuk (bugs/comments to jrs@cs.cmu.edu)
School of Computer Science / Carnegie Mellon University
5000 Forbes Avenue / Pittsburgh, Pennsylvania 15213-3891
Created as part of the Archimedes project (tools for parallel FEM).
Supported in part by NSF Grant CMS-9318163 and an NSERC 1967 Scholarship.

This executable is compiled for double precision arithmetic.

Triangle generates exact Delaunay triangulations, constrained Delaunay triangulations, and quality conforming Delaunay triangulations. The latter can be generated with no small angles, and are thus suitable for finite element analysis. If no command line switches are specified, your .node input file will be read, and the Delaunay triangulation will be returned in .node and .ele output files. The command syntax is:

```
triangle [-prq__a__AcevngBPNEIOXzo_YS__lQVh] input_file
```

Underscores indicate that numbers may optionally follow certain switches; do not leave any space between a switch and its numeric parameter. input_file must be a file with extension .node, or extension .poly if the -p switch is used. If -r is used, you must supply .node and .ele files, and possibly a .poly file and .area file as well. The formats of these files are described below.

Command Line Switches:

-p Reads a Planar Straight Line Graph (.poly file), which can specify points, segments, holes, and regional attributes and area constraints. Will generate a constrained Delaunay triangulation fitting the input; or, if -s, -q, or -a is used, a conforming Delaunay triangulation. If -p is not used, Triangle reads a .node file by default.

-r Refines a previously generated mesh. The mesh is read from a .node file and an .ele file. If -p is also used, a .poly file is read and used to constrain edges in the mesh. Further details on refinement are given below.

-q Quality mesh generation by Jim Ruppert's Delaunay refinement algorithm. Adds points to the mesh to ensure that no angles smaller than 20 degrees occur. An alternative minimum angle may be specified after the `q'. If the minimum angle is 20.7 degrees or smaller, the triangulation algorithm is theoretically guaranteed to terminate (assuming infinite precision arithmetic - Triangle may fail to terminate if you run out of precision). In practice, the algorithm often succeeds for minimum angles up to 33.8 degrees. For highly refined meshes, however, it may be necessary to reduce the minimum angle to well below 20 to avoid problems associated with insufficient floating-point precision. The specified angle may include a decimal point.

-a Imposes a maximum triangle area. If a number follows the `a', no triangle will be generated whose area is larger than that number. If no number is specified, an .area file (if -r is used) or .poly file (if -r is not used) specifies a number of maximum area

```
    constraints.  An .area file contains a separate area constraint for
    each triangle, and is useful for refining a finite element mesh
    based on a posteriori error estimates.  A .poly file can optionally
    contain an area constraint for each segment-bounded region, thereby
    enforcing triangle densities in a first triangulation.  You can
    impose both a fixed area constraint and a varying area constraint
    by invoking the -a switch twice, once with and once without a
    number following.  Each area specified may include a decimal point.
-A  Assigns an additional attribute to each triangle that identifies
    what segment-bounded region each triangle belongs to.  Attributes
    are assigned to regions by the .poly file.  If a region is not
    explicitly marked by the .poly file, triangles in that region are
    assigned an attribute of zero.  The -A switch has an effect only
    when the -p switch is used and the -r switch is not.
-c  Creates segments on the convex hull of the triangulation.  If you
    are triangulating a point set, this switch causes a .poly file to
    be written, containing all edges in the convex hull.  (By default,
    a .poly file is written only if a .poly file is read.)  If you are
    triangulating a PSLG, this switch specifies that the interior of
    the convex hull of the PSLG should be triangulated.  If you do not
    use this switch when triangulating a PSLG, it is assumed that you
    have identified the region to be triangulated by surrounding it
    with segments of the input PSLG.  Beware:  if you are not careful,
    this switch can cause the introduction of an extremely thin angle
    between a PSLG segment and a convex hull segment, which can cause
    overrefinement or failure if Triangle runs out of precision.  If
    you are refining a mesh, the -c switch works differently; it
    generates the set of boundary edges of the mesh, rather than the
    convex hull.
-e  Outputs (to an .edge file) a list of edges of the triangulation.
-v  Outputs the Voronoi diagram associated with the triangulation.
    Does not attempt to detect degeneracies.
-n  Outputs (to a .neigh file) a list of triangles neighboring each
    triangle.
-g  Outputs the mesh to an Object File Format (.off) file, suitable for
    viewing with the Geometry Center's Geomview package.
-B  No boundary markers in the output .node, .poly, and .edge output
    files.  See the detailed discussion of boundary markers below.
-P  No output .poly file.  Saves disk space, but you lose the ability
    to impose segment constraints on later refinements of the mesh.
-N  No output .node file.
-E  No output .ele file.
-I  No iteration numbers.  Suppresses the output of .node and .poly
    files, so your input files won't be overwritten.  (If your input is
    a .poly file only, a .node file will be written.)  Cannot be used
    with the -r switch, because that would overwrite your input .ele
    file.  Shouldn't be used with the -s, -q, or -a switch if you are
    using a .node file for input, because no .node file will be
    written, so there will be no record of any added points.
-O  No holes.  Ignores the holes in the .poly file.
-X  No exact arithmetic.  Normally, Triangle uses exact floating-point
    arithmetic for certain tests if it thinks the inexact tests are not
    accurate enough.  Exact arithmetic ensures the robustness of the
    triangulation algorithms, despite floating-point roundoff error.
    Disabling exact arithmetic with the -X switch will cause a small
    improvement in speed and create the possibility (albeit small) that
    Triangle will fail to produce a valid mesh.  Not recommended.
    -z  Numbers all items starting from zero (rather than one).  Note that
        this switch is normally overrided by the value used to number the
        first point of the input .node or .poly file.  However, this switch
        is useful when calling Triangle from another program.
    -o2 Generates second-order subparametric elements with six nodes each.
    -Y  No new points on the boundary.  This switch is useful when the mesh
        boundary must be preserved so that it conforms to some adjacent
        mesh.  Be forewarned that you will probably sacrifice some of the
        quality of the mesh; Triangle will try, but the resulting mesh may
        contain triangles of poor aspect ratio.  Works well if all the
        boundary points are closely spaced.  Specify this switch twice
        (`-YY') to prevent all segment splitting, including internal
        boundaries.
    -S  Specifies the maximum number of Steiner points (points that are not
        in the input, but are added to meet the constraints of minimum
        angle and maximum area).  The default is to allow an unlimited
        number.  If you specify this switch with no number after it,
        the limit is set to zero.  Triangle always adds points at segment
        intersections, even if it needs to use more points than the limit
        you set.  When Triangle inserts segments by splitting (-s), it
        always adds enough points to ensure that all the segments appear in
        the triangulation, again ignoring the limit.  Be forewarned that
        the -S switch may result in a conforming triangulation that is not
        truly Delaunay, because Triangle may be forced to stop adding
        points when the mesh is in a state where a segment is non-Delaunay
        and needs to be split.  If so, Triangle will print a warning.
    -i  Uses an incremental rather than divide-and-conquer algorithm to
        form a Delaunay triangulation.  Try it if the divide-and-conquer
        algorithm fails.
    -F  Uses Steven Fortune's sweepline algorithm to form a Delaunay
        triangulation.  Warning:  does not use exact arithmetic for all
        calculations.  An exact result is not guaranteed.
    -l  Uses only vertical cuts in the divide-and-conquer algorithm.  By
        default, Triangle uses alternating vertical and horizontal cuts,
        which usually improve the speed except with point sets that are
        small or short and wide.  This switch is primarily of theoretical
        interest.
    -s  Specifies that segments should be forced into the triangulation by
        recursively splitting them at their midpoints, rather than by
        generating a constrained Delaunay triangulation.  Segment splitting
        is true to Ruppert's original algorithm, but can create needlessly
        small triangles near external small features.
    -C  Check the consistency of the final mesh.  Uses exact arithmetic for
        checking, even if the -X switch is used.  Useful if you suspect
        Triangle is buggy.
    -Q  Quiet: Suppresses all explanation of what Triangle is doing, unless
        an error occurs.
    -V  Verbose: Gives detailed information about what Triangle is doing.
        Add more `V's for increasing amount of detail.  `-V' gives
        information on algorithmic progress and more detailed statistics.
        `-VV' gives point-by-point details, and will print so much that
        Triangle will run much more slowly.  `-VVV' gives information only
        a debugger could love.
    -h  Help:  Displays these instructions.

Definitions:
```

A Delaunay triangulation of a point set is a triangulation whose vertices are the point set, having the property that no point in the point set falls in the interior of the circumcircle (circle that passes through all three vertices) of any triangle in the triangulation.

A Voronoi diagram of a point set is a subdivision of the plane into polygonal regions (some of which may be infinite), where each region is the set of points in the plane that are closer to some input point than to any other input point. (The Voronoi diagram is the geometric dual of the Delaunay triangulation.)

A Planar Straight Line Graph (PSLG) is a collection of points and segments. Segments are simply edges, whose endpoints are points in the PSLG. The file format for PSLGs (.poly files) is described below.

A constrained Delaunay triangulation of a PSLG is similar to a Delaunay triangulation, but each PSLG segment is present as a single edge in the triangulation. (A constrained Delaunay triangulation is not truly a Delaunay triangulation.)

A conforming Delaunay triangulation of a PSLG is a true Delaunay triangulation in which each PSLG segment may have been subdivided into several edges by the insertion of additional points. These inserted points are necessary to allow the segments to exist in the mesh while maintaining the Delaunay property.

File Formats:

All files may contain comments prefixed by the character '#'. Points, triangles, edges, holes, and maximum area constraints must be numbered consecutively, starting from either 1 or 0. Whichever you choose, all input files must be consistent; if the nodes are numbered from 1, so must be all other objects. Triangle automatically detects your choice while reading the .node (or .poly) file. (When calling Triangle from another program, use the -z switch if you wish to number objects from zero.) Examples of these file formats are given below.

.node files:

```
First line:  <# of points> <dimension (must be 2)> <# of attributes>
                                       <# of boundary markers (0 or 1)>
Remaining lines:  <point #> <x> <y> [attributes] [boundary marker]
```

The attributes, which are typically floating-point values of physical quantities (such as mass or conductivity) associated with the nodes of a finite element mesh, are copied unchanged to the output mesh. If -s, -q, or -a is selected, each new Steiner point added to the mesh will have attributes assigned to it by linear interpolation.

If the fourth entry of the first line is `1', the last column of the remainder of the file is assumed to contain boundary markers. Boundary markers are used to identify boundary points and points resting on PSLG segments; a complete description appears in a section below. The .node file produced by Triangle will contain boundary markers in the last column unless they are suppressed by the -B switch.

.ele files:

```
First line:  <# of triangles> <points per triangle> <# of attributes>
```

```
Remaining lines:  <triangle #> <point> <point> <point> ... [attributes]
```

Points are indices into the corresponding .node file. The first three points are the corners, and are listed in counterclockwise order around each triangle. (The remaining points, if any, depend on the type of finite element used.) The attributes are just like those of .node files. Because there is no simple mapping from input to output triangles, an attempt is made to interpolate attributes, which may result in a good deal of diffusion of attributes among nearby triangles as the triangulation is refined. Diffusion does not occur across segments, so attributes used to identify segment-bounded regions remain intact. In output .ele files, all triangles have three points each unless the -o2 switch is used, in which case they have six, and the fourth, fifth, and sixth points lie on the midpoints of the edges opposite the first, second, and third corners.

.poly files:

```
First line:  <# of points> <dimension (must be 2)> <# of attributes>
                                       <# of boundary markers (0 or 1)>
Following lines:  <point #> <x> <y> [attributes] [boundary marker]
One line:  <# of segments> <# of boundary markers (0 or 1)>
Following lines:  <segment #> <endpoint> <endpoint> [boundary marker]
One line:  <# of holes>
Following lines:  <hole #> <x> <y>
Optional line:  <# of regional attributes and/or area constraints>
Optional following lines:  <constraint #> <x> <y> <attrib> <max area>
```

A .poly file represents a PSLG, as well as some additional information. The first section lists all the points, and is identical to the format of .node files. <# of points> may be set to zero to indicate that the points are listed in a separate .node file; .poly files produced by Triangle always have this format. This has the advantage that a point set may easily be triangulated with or without segments. (The same effect can be achieved, albeit using more disk space, by making a copy of the .poly file with the extension .node; all sections of the file but the first are ignored.)

The second section lists the segments. Segments are edges whose presence in the triangulation is enforced. Each segment is specified by listing the indices of its two endpoints. This means that you must include its endpoints in the point list. If -s, -q, and -a are not selected, Triangle will produce a constrained Delaunay triangulation, in which each segment appears as a single edge in the triangulation. If -q or -a is selected, Triangle will produce a conforming Delaunay triangulation, in which segments may be subdivided into smaller edges. Each segment, like each point, may have a boundary marker.

The third section lists holes (and concavities, if -c is selected) in the triangulation. Holes are specified by identifying a point inside each hole. After the triangulation is formed, Triangle creates holes by eating triangles, spreading out from each hole point until its progress is blocked by PSLG segments; you must be careful to enclose each hole in segments, or your whole triangulation may be eaten away. If the two triangles abutting a segment are eaten, the segment itself is also eaten. Do not place a hole directly on a segment; if you do, Triangle will choose one side of the segment arbitrarily.

The optional fourth section lists regional attributes (to be assigned to all triangles in a region) and regional constraints on the maximum triangle area. Triangle will read this section only if the -A switch is used or the -a switch is used without a number following it, and the -r switch is not used. Regional attributes and area constraints are propagated in the same manner as holes; you specify a point for each attribute and/or constraint, and the attribute and/or constraint will affect the whole region (bounded by segments) containing the point. If two values are written on a line after the x and y coordinate, the former is assumed to be a regional attribute (but will only be applied if the -A switch is selected), and the latter is assumed to be a regional area constraint (but will only be applied if the -a switch is selected). You may also specify just one value after the coordinates, which can serve as both an attribute and an area constraint, depending on the choice of switches. If you are using the -A and -a switches simultaneously and wish to assign an attribute to some region without imposing an area constraint, use a negative maximum area.

When a triangulation is created from a .poly file, you must either enclose the entire region to be triangulated in PSLG segments, or use the -c switch, which encloses the convex hull of the input point set. If you do not use the -c switch, Triangle will eat all triangles on the outer boundary that are not protected by segments; if you are not careful, your whole triangulation may be eaten away. If you do use the -c switch, you can still produce concavities by appropriate placement of holes just inside the convex hull.

An ideal PSLG has no intersecting segments, nor any points that lie upon segments (except, of course, the endpoints of each segment.) You aren't required to make your .poly files ideal, but you should be aware of what can go wrong. Segment intersections are relatively safe - Triangle will calculate the intersection points for you and add them to the triangulation - as long as your machine's floating-point precision doesn't become a problem. You are tempting the fates if you have three segments that cross at the same location, and expect Triangle to figure out where the intersection point is. Thanks to floating-point roundoff error, Triangle will probably decide that the three segments intersect at three different points, and you will find a minuscule triangle in your output - unless Triangle tries to refine the tiny triangle, uses up the last bit of machine precision, and fails to terminate at all. You're better off putting the intersection point in the input files, and manually breaking up each segment into two. Similarly, if you place a point at the middle of a segment, and hope that Triangle will break up the segment at that point, you might get lucky. On the other hand, Triangle might decide that the point doesn't lie precisely on the line, and you'll have a needle-sharp triangle in your output - or a lot of tiny triangles if you're generating a quality mesh.

When Triangle reads a .poly file, it also writes a .poly file, which includes all edges that are part of input segments. If the -c switch is used, the output .poly file will also include all of the edges on the convex hull. Hence, the output .poly file is useful for finding edges associated with input segments and setting boundary conditions in finite element simulations. More importantly, you will need it if you plan to refine the output mesh, and don't want segments to be missing in later triangulations.

.area files:

First line: <# of triangles>
Following lines: <triangle #> <maximum area>

An .area file associates with each triangle a maximum area that is used for mesh refinement. As with other file formats, every triangle must be represented, and they must be numbered consecutively. A triangle may be left unconstrained by assigning it a negative maximum area.

.edge files:

First line: <# of edges> <# of boundary markers (0 or 1)>
Following lines: <edge #> <endpoint> <endpoint> [boundary marker]

Endpoints are indices into the corresponding .node file. Triangle can produce .edge files (use the -e switch), but cannot read them. The optional column of boundary markers is suppressed by the -B switch.

In Voronoi diagrams, one also finds a special kind of edge that is an infinite ray with only one endpoint. For these edges, a different format is used:

```
    <edge #> <endpoint> -1 <direction x> <direction y>
```

The `direction' is a floating-point vector that indicates the direction of the infinite ray.

.neigh files:

First line: <# of triangles> <# of neighbors per triangle (always 3)>
Following lines: <triangle #> <neighbor> <neighbor> <neighbor>

Neighbors are indices into the corresponding .ele file. An index of -1 indicates a mesh boundary, and therefore no neighbor. Triangle can produce .neigh files (use the -n switch), but cannot read them.

The first neighbor of triangle i is opposite the first corner of triangle i, and so on.

Boundary Markers:

Boundary markers are tags used mainly to identify which output points and edges are associated with which PSLG segment, and to identify which points and edges occur on a boundary of the triangulation. A common use is to determine where boundary conditions should be applied to a finite element mesh. You can prevent boundary markers from being written into files produced by Triangle by using the -B switch.

The boundary marker associated with each segment in an output .poly file or edge in an output .edge file is chosen as follows:
- If an output edge is part or all of a PSLG segment with a nonzero boundary marker, then the edge is assigned the same marker.
- Otherwise, if the edge occurs on a boundary of the triangulation (including boundaries of holes), then the edge is assigned the marker one (1).
- Otherwise, the edge is assigned the marker zero (0).

The boundary marker associated with each point in an output .node file is chosen as follows:
- If a point is assigned a nonzero boundary marker in the input file,

```
      then it is assigned the same marker in the output .node file.
    - Otherwise, if the point lies on a PSLG segment (including the
      segment's endpoints) with a nonzero boundary marker, then the point
      is assigned the same marker.  If the point lies on several such
      segments, one of the markers is chosen arbitrarily.
    - Otherwise, if the point occurs on a boundary of the triangulation,
      then the point is assigned the marker one (1).
    - Otherwise, the point is assigned the marker zero (0).

  If you want Triangle to determine for you which points and edges are on
  the boundary, assign them the boundary marker zero (or use no markers at
  all) in your input files.  Alternatively, you can mark some of them and
  leave others marked zero, allowing Triangle to label them.

Triangulation Iteration Numbers:

  Because Triangle can read and refine its own triangulations, input
  and output files have iteration numbers.  For instance, Triangle might
  read the files mesh.3.node, mesh.3.ele, and mesh.3.poly, refine the
  triangulation, and output the files mesh.4.node, mesh.4.ele, and
  mesh.4.poly.  Files with no iteration number are treated as if
  their iteration number is zero; hence, Triangle might read the file
  points.node, triangulate it, and produce the files points.1.node and
  points.1.ele.

  Iteration numbers allow you to create a sequence of successively finer
  meshes suitable for multigrid methods.  They also allow you to produce a
  sequence of meshes using error estimate-driven mesh refinement.

  If you're not using refinement or quality meshing, and you don't like
  iteration numbers, use the -I switch to disable them.  This switch will
  also disable output of .node and .poly files to prevent your input files
  from being overwritten.  (If the input is a .poly file that contains its
  own points, a .node file will be written.)

Examples of How to Use Triangle:

  `triangle dots' will read points from dots.node, and write their Delaunay
  triangulation to dots.1.node and dots.1.ele.  (dots.1.node will be
  identical to dots.node.)  `triangle -I dots' writes the triangulation to
  dots.ele instead.  (No additional .node file is needed, so none is
  written.)

  `triangle -pe object.1' will read a PSLG from object.1.poly (and possibly
  object.1.node, if the points are omitted from object.1.poly) and write
  their constrained Delaunay triangulation to object.2.node and
  object.2.ele.  The segments will be copied to object.2.poly, and all
  edges will be written to object.2.edge.

  `triangle -pq31.5a.1 object' will read a PSLG from object.poly (and
  possibly object.node), generate a mesh whose angles are all greater than
  31.5 degrees and whose triangles all have area smaller than 0.1, and
  write the mesh to object.1.node and object.1.ele.  Each segment may have
  been broken up into multiple edges; the resulting constrained edges are
  written to object.1.poly.

  Here is a sample file `box.poly' describing a square with a square hole:
```

```
# A box with eight points in 2D, no attributes, one boundary marker.
8 2 0 1
# Outer box has these vertices:
 1   0 0   0
 2   0 3   0
 3   3 0   0
 4   3 3   33     # A special marker for this point.
# Inner square has these vertices:
 5   1 1   0
 6   1 2   0
 7   2 1   0
 8   2 2   0
# Five segments with boundary markers.
5 1
 1   1 2   5      # Left side of outer box.
 2   5 7   0      # Segments 2 through 5 enclose the hole.
 3   7 8   0
 4   8 6   10
 5   6 5   0
# One hole in the middle of the inner square.
1
 1   1.5 1.5
```

Note that some segments are missing from the outer square, so one must use the `-c' switch. After `triangle -pqc box.poly', here is the output file `box.1.node', with twelve points. The last four points were added to meet the angle constraint. Points 1, 2, and 9 have markers from segment 1. Points 6 and 8 have markers from segment 4. All the other points but 4 have been marked to indicate that they lie on a boundary.

```
12  2  0  1
   1    0   0      5
   2    0   3      5
   3    3   0      1
   4    3   3     33
   5    1   1      1
   6    1   2     10
   7    2   1      1
   8    2   2     10
   9    0   1.5    5
  10    1.5   0    1
  11    3   1.5    1
  12    1.5   3    1
# Generated by triangle -pqc box.poly
```

Here is the output file `box.1.ele', with twelve triangles.

```
12  3  0
   1     5   6   9
   2    10   3   7
   3     6   8  12
   4     9   1   5
   5     6   2   9
   6     7   3  11
   7    11   4   8
   8     7   5  10
```

```
    9    12   2   6
   10     8   7  11
   11     5   1  10
   12     8   4  12
  # Generated by triangle -pqc box.poly
```

Here is the output file `box.1.poly'. Note that segments have been added to represent the convex hull, and some segments have been split by newly added points. Note also that <# of points> is set to zero to indicate that the points should be read from the .node file.

```
  0  2  0  1
  12  1
     1     1   9     5
     2     5   7     1
     3     8   7     1
     4     6   8    10
     5     5   6     1
     6     3  10     1
     7     4  11     1
     8     2  12     1
     9     9   2     5
    10    10   1     1
    11    11   3     1
    12    12   4     1
  1
     1   1.5 1.5
  # Generated by triangle -pqc box.poly
```

Refinement and Area Constraints:

The -r switch causes a mesh (.node and .ele files) to be read and refined. If the -p switch is also used, a .poly file is read and used to specify edges that are constrained and cannot be eliminated (although they can be divided into smaller edges) by the refinement process.

When you refine a mesh, you generally want to impose tighter quality constraints. One way to accomplish this is to use -q with a larger angle, or -a followed by a smaller area than you used to generate the mesh you are refining. Another way to do this is to create an .area file, which specifies a maximum area for each triangle, and use the -a switch (without a number following). Each triangle's area constraint is applied to that triangle. Area constraints tend to diffuse as the mesh is refined, so if there are large variations in area constraint between adjacent triangles, you may not get the results you want.

If you are refining a mesh composed of linear (three-node) elements, the output mesh will contain all the nodes present in the input mesh, in the same order, with new nodes added at the end of the .node file. However, there is no guarantee that each output element is contained in a single input element. Often, output elements will overlap two input elements, and input edges are not present in the output mesh. Hence, a sequence of refined meshes will form a hierarchy of nodes, but not a hierarchy of elements. If you a refining a mesh of higher-order elements, the hierarchical property applies only to the nodes at the corners of an element; other nodes may not be present in the refined mesh.

It is important to understand that maximum area constraints in .poly files are handled differently from those in .area files. A maximum area in a .poly file applies to the whole (segment-bounded) region in which a point falls, whereas a maximum area in an .area file applies to only one triangle. Area constraints in .poly files are used only when a mesh is first generated, whereas area constraints in .area files are used only to refine an existing mesh, and are typically based on a posteriori error estimates resulting from a finite element simulation on that mesh.

`triangle -rq25 object.1' will read object.1.node and object.1.ele, then refine the triangulation to enforce a 25 degree minimum angle, and then write the refined triangulation to object.2.node and object.2.ele.

`triangle -rpaa6.2 z.3' will read z.3.node, z.3.ele, z.3.poly, and z.3.area. After reconstructing the mesh and its segments, Triangle will refine the mesh so that no triangle has area greater than 6.2, and furthermore the triangles satisfy the maximum area constraints in z.3.area. The output is written to z.4.node, z.4.ele, and z.4.poly.

The sequence `triangle -qa1 x', `triangle -rqa.3 x.1', `triangle -rqa.1 x.2' creates a sequence of successively finer meshes x.1, x.2, and x.3, suitable for multigrid.

Convex Hulls and Mesh Boundaries:

If the input is a point set (rather than a PSLG), Triangle produces its convex hull as a by-product in the output .poly file if you use the -c switch. There are faster algorithms for finding a two-dimensional convex hull than triangulation, of course, but this one comes for free. If the input is an unconstrained mesh (you are using the -r switch but not the -p switch), Triangle produces a list of its boundary edges (including hole boundaries) as a by-product if you use the -c switch.

Voronoi Diagrams:

The -v switch produces a Voronoi diagram, in files suffixed .v.node and .v.edge. For example, `triangle -v points' will read points.node, produce its Delaunay triangulation in points.1.node and points.1.ele, and produce its Voronoi diagram in points.1.v.node and points.1.v.edge. The .v.node file contains a list of all Voronoi vertices, and the .v.edge file contains a list of all Voronoi edges, some of which may be infinite rays. (The choice of filenames makes it easy to run the set of Voronoi vertices through Triangle, if so desired.)

This implementation does not use exact arithmetic to compute the Voronoi vertices, and does not check whether neighboring vertices are identical. Be forewarned that if the Delaunay triangulation is degenerate or near-degenerate, the Voronoi diagram may have duplicate points, crossing edges, or infinite rays whose direction vector is zero. Also, if you generate a constrained (as opposed to conforming) Delaunay triangulation, or if the triangulation has holes, the corresponding Voronoi diagram is likely to have crossing edges and unlikely to make sense.

Mesh Topology:

You may wish to know which triangles are adjacent to a certain Delaunay edge in an .edge file, which Voronoi regions are adjacent to a certain

Voronoi edge in a .v.edge file, or which Voronoi regions are adjacent to each other. All of this information can be found by cross-referencing output files with the recollection that the Delaunay triangulation and the Voronoi diagrams are planar duals.

Specifically, edge i of an .edge file is the dual of Voronoi edge i of the corresponding .v.edge file, and is rotated 90 degrees counterclockwise from the Voronoi edge. Triangle j of an .ele file is the dual of vertex j of the corresponding .v.node file; and Voronoi region k is the dual of point k of the corresponding .node file.

Hence, to find the triangles adjacent to a Delaunay edge, look at the vertices of the corresponding Voronoi edge; their dual triangles are on the left and right of the Delaunay edge, respectively. To find the Voronoi regions adjacent to a Voronoi edge, look at the endpoints of the corresponding Delaunay edge; their dual regions are on the right and left of the Voronoi edge, respectively. To find which Voronoi regions are adjacent to each other, just read the list of Delaunay edges.

Statistics:

After generating a mesh, Triangle prints a count of the number of points, triangles, edges, boundary edges, and segments in the output mesh. If you've forgotten the statistics for an existing mesh, the -rNEP switches (or -rpNEP if you've got a .poly file for the existing mesh) will regenerate these statistics without writing any output.

The -V switch produces extended statistics, including a rough estimate of memory use and a histogram of triangle aspect ratios and angles in the mesh.

Exact Arithmetic:

Triangle uses adaptive exact arithmetic to perform what computational geometers call the `orientation' and `incircle' tests. If the floating-point arithmetic of your machine conforms to the IEEE 754 standard (as most workstations do), and does not use extended precision internal registers, then your output is guaranteed to be an absolutely true Delaunay or conforming Delaunay triangulation, roundoff error notwithstanding. The word `adaptive' implies that these arithmetic routines compute the result only to the precision necessary to guarantee correctness, so they are usually nearly as fast as their approximate counterparts. The exact tests can be disabled with the -X switch. On most inputs, this switch will reduce the computation time by about eight percent - it's not worth the risk. There are rare difficult inputs (having many collinear and cocircular points), however, for which the difference could be a factor of two. These are precisely the inputs most likely to cause errors if you use the -X switch.

Unfortunately, these routines don't solve every numerical problem. Exact arithmetic is not used to compute the positions of points, because the bit complexity of point coordinates would grow without bound. Hence, segment intersections aren't computed exactly; in very unusual cases, roundoff error in computing an intersection point might actually lead to an inverted triangle and an invalid triangulation. (This is one reason to compute your own intersection points in your .poly files.) Similarly, exact arithmetic is not used to compute the vertices of the Voronoi diagram.

Underflow and overflow can also cause difficulties; the exact arithmetic routines do not ameliorate out-of-bounds exponents, which can arise during the orientation and incircle tests. As a rule of thumb, you should ensure that your input values are within a range such that their third powers can be taken without underflow or overflow. Underflow can silently prevent the tests from being performed exactly, while overflow will typically cause a floating exception.

Calling Triangle from Another Program:

Read the file triangle.h for details.

Troubleshooting:

Please read this section before mailing me bugs.

`My output mesh has no triangles!'

If you're using a PSLG, you've probably failed to specify a proper set of bounding segments, or forgotten to use the -c switch. Or you may have placed a hole badly. To test these possibilities, try again with the -c and -O switches. Alternatively, all your input points may be collinear, in which case you can hardly expect to triangulate them.

`Triangle doesn't terminate, or just crashes.'

Bad things can happen when triangles get so small that the distance between their vertices isn't much larger than the precision of your machine's arithmetic. If you've compiled Triangle for single-precision arithmetic, you might do better by recompiling it for double-precision. Then again, you might just have to settle for more lenient constraints on the minimum angle and the maximum area than you had planned.

You can minimize precision problems by ensuring that the origin lies inside your point set, or even inside the densest part of your mesh. On the other hand, if you're triangulating an object whose x coordinates all fall between 6247133 and 6247134, you're not leaving much floating-point precision for Triangle to work with.

Precision problems can occur covertly if the input PSLG contains two segments that meet (or intersect) at a very small angle, or if such an angle is introduced by the -c switch, which may occur if a point lies ever-so-slightly inside the convex hull, and is connected by a PSLG segment to a point on the convex hull. If you don't realize that a small angle is being formed, you might never discover why Triangle is crashing. To check for this possibility, use the -S switch (with an appropriate limit on the number of Steiner points, found by trial-and-error) to stop Triangle early, and view the output .poly file with Show Me (described below). Look carefully for small angles between segments; zoom in closely, as such segments might look like a single segment from a distance.

If some of the input values are too large, Triangle may suffer a floating exception due to overflow when attempting to perform an orientation or incircle test. (Read the section on exact arithmetic above.) Again, I recommend compiling Triangle for double (rather than single) precision arithmetic.

`The numbering of the output points doesn't match the input points.'

You may have eaten some of your input points with a hole, or by placing them outside the area enclosed by segments.

`Triangle executes without incident, but when I look at the resulting mesh, it has overlapping triangles or other geometric inconsistencies.'

If you select the -X switch, Triangle's divide-and-conquer Delaunay triangulation algorithm occasionally makes mistakes due to floating-point roundoff error. Although these errors are rare, don't use the -X switch. If you still have problems, please report the bug.

Strange things can happen if you've taken liberties with your PSLG. Do you have a point lying in the middle of a segment? Triangle sometimes copes poorly with that sort of thing. Do you want to lay out a collinear row of evenly spaced, segment-connected points? Have you simply defined one long segment connecting the leftmost point to the rightmost point, and a bunch of points lying along it? This method occasionally works, especially with horizontal and vertical lines, but often it doesn't, and you'll have to connect each adjacent pair of points with a separate segment. If you don't like it, tough.

Furthermore, if you have segments that intersect other than at their endpoints, try not to let the intersections fall extremely close to PSLG points or each other.

If you have problems refining a triangulation not produced by Triangle: Are you sure the triangulation is geometrically valid? Is it formatted correctly for Triangle? Are the triangles all listed so the first three points are their corners in counterclockwise order?

Show Me:

Triangle comes with a separate program named `Show Me', whose primary purpose is to draw meshes on your screen or in PostScript. Its secondary purpose is to check the validity of your input files, and do so more thoroughly than Triangle does. Show Me requires that you have the X Windows system. If you didn't receive Show Me with Triangle, complain to whomever you obtained Triangle from, then send me mail.

Triangle on the Web:

To see an illustrated, updated version of these instructions, check out

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

A Brief Plea:

If you use Triangle, and especially if you use it to accomplish real work, I would like very much to hear from you. A short letter or email (to jrs@cs.cmu.edu) describing how you use Triangle will mean a lot to me. The more people I know are using this program, the more easily I can justify spending time on improvements and on the three-dimensional successor to Triangle, which in turn will benefit you. Also, I can put you on a list to receive email whenever a new version of Triangle is available.

If you use a mesh generated by Triangle in a publication, please include an acknowledgment as well.

Research credit:

Of course, I can take credit for only a fraction of the ideas that made this mesh generator possible. Triangle owes its existence to the efforts of many fine computational geometers and other researchers, including Marshall Bern, L. Paul Chew, Boris Delaunay, Rex A. Dwyer, David Eppstein, Steven Fortune, Leonidas J. Guibas, Donald E. Knuth, C. L. Lawson, Der-Tsai Lee, Ernst P. Mucke, Douglas M. Priest, Jim Ruppert, Isaac Saias, Bruce J. Schachter, Micha Sharir, Jorge Stolfi, Christopher J. Van Wyk, David F. Watson, and Binhai Zhu. See the comments at the beginning of the source code for references.