JKAU: Eng. Sci., **Vol 17** No. 2, pp: 41 - 72 (2006A.D. /1427 A.H.)

A C# Algorithm for Creating Triangular Meshes of Highly-Irregular 2D Domains Using the Advancing Front Technique

Hassan S. Naji

Petroleum Geology and Sedimentology Department, Faculty of Earth Sciences, King Abdulaziz University, Jeddah, Saudi Arabia
hassan@petrobjects.com & petrobjects.com

Abstract. This paper describes a C# algorithm for creating efficient triangular meshes of highly-irregular 2D domains. The algorithm, which is based on the advancing front technique, requires boundary nodes as the only input. Basic shapes such as lines, curves, rectangles, polygons, circles, and/or ellipses are used to construct the domain. Shapes are interactively added to the domain in a sequential order. Whenever a shape is added, however, it is directly exploded to a set of nodes appended to the end of the domain. Nodes must be continuous and must not cross one another. Inside openings of the domain are implemented via connector lines; which have a two-way trip; one from the boundary to the starting point of the opening and the other from the ending point of the opening back to the boundary. Nodes are interactively moved or deleted; which allows a variable node density to be created easily and which optimizes the final shape of the domain.

The algorithm produces well-conditioned (close-to-equilateral) triangular elements. An additional smoothing procedure, however, is performed by shifting each interior node to the center of the surrounding polygon. Numbering of nodes has a definite influence on the *band width* of the coefficient matrix associated with the mesh. The smaller the band width, the less storage and amount of computation required. The Cuthill-McKee algorithm for renumbering mesh nodes is applied. The implementation of the algorithm using the C# object-oriented language allows flexibility in programming and increases the efficiency in the construction of complex highly-irregular two-dimensional domains. Examples of created domains along with their generated meshes in both simply and multiply connected domains are presented.

Keywords: Advancing front technique; finite element triangulation; 2D mesh generation.

Introduction

Numerical models are extensively used in engineering problems. The differential equations of such models are numerically solved using either the finite difference or finite element methods. The industry's current modeling interest has revealed a trend towards more flexible meshing techniques. The finite element method provides this flexibility, in addition to the higher accuracy involved in the method compared to finite differences. The first step of the solution method, however, requires creating a mesh. The conditioning of mesh elements and the numbering scheme of mesh nodes highly influence the efficiency of the method.

Many algorithms have been described in literature for generating finite element meshes [1-3, 5-7, 8-14, 15-17]. Briefly, the mesh generators may be classified into two broad categories: *structured* and *unstructured*. The unstructured have now superseded the structured methods due to their ease and flexibility to generate meshes of highly-irregular domains. One of the well-established unstructured triangulation methods is the *advancing front* method [6, 11, 14, 16]. In this method, nodes and elements are created one by one until the domain is completed. When this method is combined with element-smoothing and node-renumbering, it produces high-quality, close-to-equilateral triangles [14]. Numbering of mesh nodes has a definite influence on the *band width* of the coefficient matrix associated with the mesh. The smaller the band width, the less storage and amount of computation required [4].

The main disadvantage of the method, however, lies with its efficiency, where checking the intersection of edges/overlapping of elements takes considerable amount of time. In addition, the method may fail in cases where front nodes are not fed correctly. Experimenting with the method, however, indicated that manual input of front nodes frequently results in poor meshes that in many cases require a make-up stage to repair. In addition, there is always an *optimal representation of the initial front* of complex boundaries. In some cases where the method has failed with a specific front, little editing, e.g., shifting or deleting some nodes, while maintaining the same shape of the domain boundary, may be required for the advancing front triangulation method to succeed. Furthermore, the quality of the generated triangles also improves.

This paper describes a triangulation algorithm that incorporates the above requirements at minimal effort possible. The algorithm is implemented using the object-oriented C# programming language,

which provides substantial computing and programming advantages and allows the mesh generation algorithm to become truly free from human interference. Another important aspect of this algorithm is that a discussion of the C# implementation, to the best of my knowledge, has not yet been made on mesh generation for finite element applications. So far, most mesh generation codes have been developed using the traditional FORTRAN language, which is a natural choice from the view point of continuity in downstream data processing. Such a natural choice may not necessarily be the optimal choice. In fact, the use of an object-oriented language is more desirable for the improved mesh scheme, because the paradigm of object-oriented programming allows the different parts constituting the mesh to be described easily and naturally as if they were real world objects [1].

The C# programming language is an evolution of the C and C++. It has been designed taking into account many of the best features of both languages, while cleaning up their problems. Developing applications using C# is much simpler than using C or C++. All OOP features such as encapsulation, inheritance, and polymorphism are included in C#. The more advanced features of C and C++ that access and manipulate the computer memory through pointers may also be carried out using C# code marked as *unsafe*. These advanced capabilities in C/C++ are potentially dangerous because it is possible to overwrite system-critical blocks of memory.

Algorithm Description

The triangulation process of a *closed* 2D region using the advancing front technique is summarized by the following steps [14]: Representation of Domain Boundary, Elements' Generation, Grid Smoothing, and Nodes Renumbering.

1. Representation of Domain Boundary

A two-dimensional domain is constructed using a sequence of connected basic shapes. A basic shape may be a line, curve, rectangle, polygon, circle, or ellipse. It may appear on the domain boundary or inside the domain as an opening as shown in Fig. 1. To link the internal openings to the exterior boundary, however, connector lines are used. This forces the internal openings to become part of the exterior boundary, and incorporates them into the generation front. Once a basic shape is added to the domain, it is directly exploded to a set of points

(with predefined spacing, ratio, and direction) appended to the end of the domain boundary.

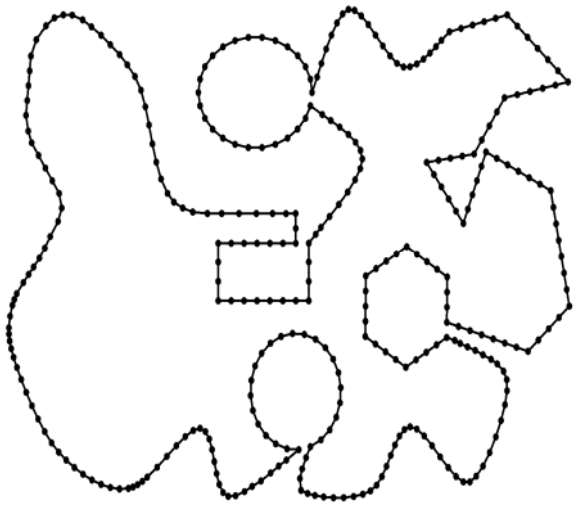

Fig. 1. A sequence of connected basic shapes forming a 2D domain.

Nodal x- and y-coordinates of the domain are read in a clockwise or counterclockwise direction. Nodal points spacing controls the boundary smoothness and the density and quality of the generated grid. The boundary shown in Fig. 2 will be referred to as the *front*. We now have an array of connected line segments, S_1 , S_2 , S_3 , ... , S_N, arranged in a counter clock wise direction as shown in Fig. 2. Lengths of line segments, S_1 , S_2 , S_3 , ... , S_N , and nodal angles, $_1$, $_2$, $_3$, ..., $_N$, formed by consecutive line segments, are calculated and sorted out, according to their magnitude, in an ascending order [14].

After the initial construction of the domain, nodes may be moved or deleted. On one hand, this helps create flexible domains in minimal time. On the other hand, the advancing front method may fail with an initial front, so little editing, e.g. shifting or deleting some nodes, while maintaining the same shape, may be required for the advancing front method to succeed. For example, Fig. 3 displays the same domain of Fig. 1 with some front nodes moved or deleted.

A C# program was written for the Microsoft Windows platform with a GUI interface to construct such domains interactively as shown in

Fig. 4. Implementation using C# entails creating various objects (classes); one for each basic shape and one for the completed domain. An overall description of each object is presented next. Furthermore, the

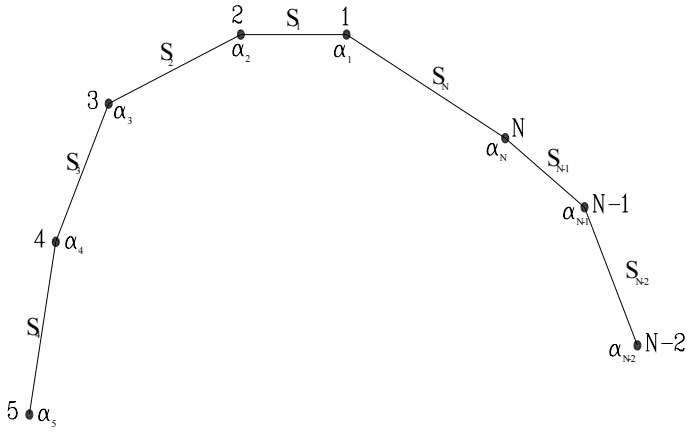

Fig. 2. A portion of a closed 2D domain showing the connected line segments and the nodal angle.

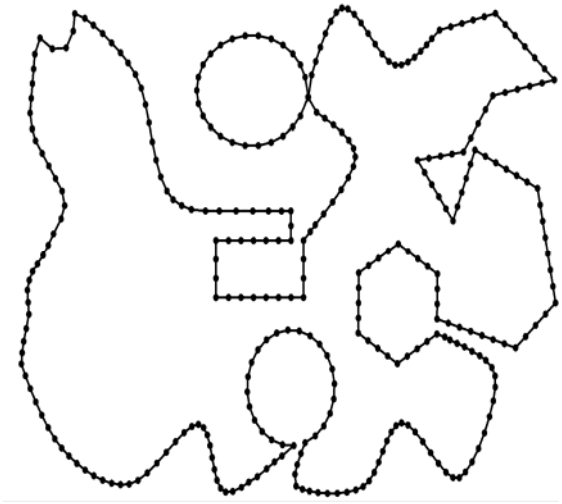

Fig. 3. The same domain of Fig. 1 after editing some boundary points.

C# code listing of a sample object (Line object) is given in Appendix A. Other objects follow exactly the same concept. A fully-functional

version of the program, written for MS Windows platform, may be downloaded from the author's website.

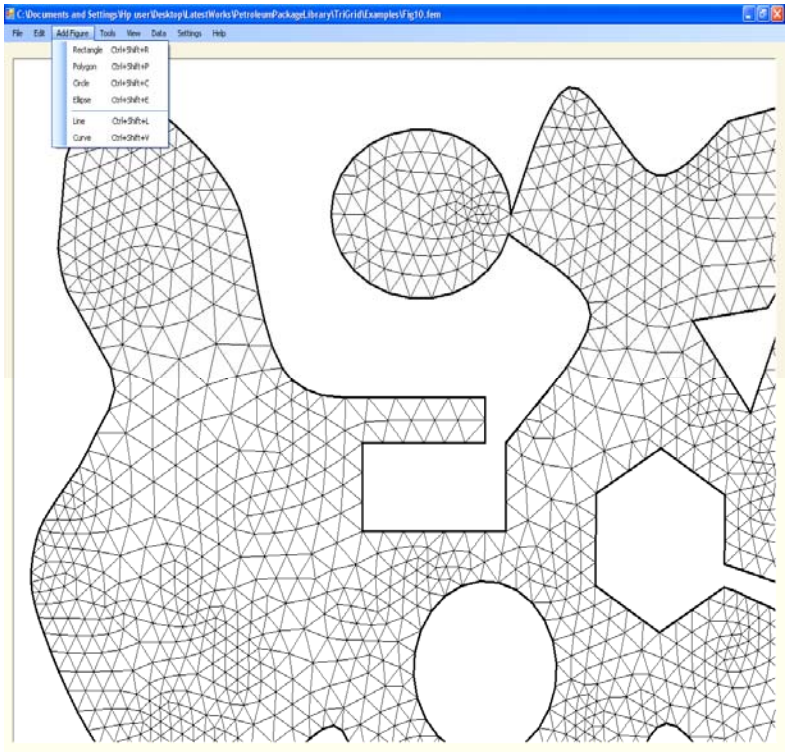

Fig. 4. The C# Windows program displaying the basic shapes menu.

The Line Object

The data structure of the Line object has the following properties, (see Appendix A):

- begin point of the line,
- end point of the line,
- requested point spacing on the line,
- ratio of last/first point spacing on the line.

The Rectangle Object

The data structure of the rectangle object has the following properties:

- Begin corner of the rectangle that connects the rectangle to the domain,
- end corner of the rectangle,
- requested point spacing on the rectangle perimeter,
- ratio of last/first point spacing on each line of the rectangle,
- explode direction, clockwise or anticlockwise.

The Polygon Object

The data structure of the polygon object has the following properties:

- The first point on polygon perimeter that connects the polygon to the domain,
- center of polygon,
- radius of polygon,
- number and set of polygon vertices,
- requested point spacing on polygon perimeter,
- ratio of last/first point spacing on polygon perimeter,
- explode direction, clockwise or anticlockwise.

The Circle Object

The data structure of the circle object has the following properties:

- The first point on circle perimeter that connects the circle to the domain,
- center of circle,
- radius of circle,
- requested point spacing on circle perimeter,
- ratio of last/first point spacing on circle perimeter,
- explode direction, clockwise or anticlockwise.

The Ellipse Object

The data structure of the ellipse object has the following properties:

- The first point on ellipse perimeter that connects the ellipse to the domain,
- center of ellipse,
- x- and y- radii of ellipse,
- requested point spacing on ellipse perimeter,
- ratio of last/first point spacing on ellipse perimeter,
- explode direction, clockwise or anticlockwise.

The Lines Object

The data structure of the lines object has the following properties:

- requested point spacing on each line,
- ratio of last/first point spacing on each line,
- array of lines vertices.

The Curve Object

The data structure of the curve object has the following properties:

- requested point spacing on the curve,
- ratio of last/first point spacing on the curve,
- array of curve vertices.

All basic shapes have methods that control the operation and facilitate the insertion of the shape into the domain. These methods include, (see Appendix A):

- A mouse down event for recording the first and last points of the shape,
- A mouse move event for tracing the shape,
- Two overloads of the drawing event; one for interactive drawing and another for drawing the final shape.

- Three overloads for explode event: one with default parameters, the second with preset parameters, and the third with static parameters,
- A read method for reading shape properties,
- A write method for saving shape properties.

The Completed Domain (Path) Object

The data structure of the completed domain (Path) object has the following properties:

- array of domain vertices,
- array of domain points,
- an instance of each basic shape; *i.e.* rectangle, polygon, circle, ellipse, lines, curve.

Similarly the domain object has methods that control its operation. These methods include, (see Appendix A):

- A mouse down event for recording mouse down events of the basic shapes,
- A mouse move event for recording mouse move events of the basic shapes,
- Two overloads for drawing event; one for interactive drawing and another for drawing the final domain.
- Three overloads for explode event: one with default parameters, the second with preset parameters, and the third with static parameters,
- A read method for reading domain properties,
- A write method for saving domain properties.

2. Elements Generation

Elements' generation process [14] starts at the *smallest* angle, α, on the front. Depending on the value of α, there will be three cases:

A. $\alpha < \pi/2$

One triangular element is created by connecting the two line segments, S_2 and S_3 as shown in Fig. 5a.

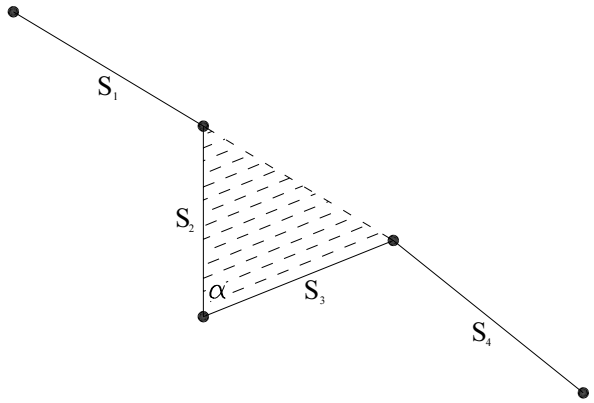

Fig. 5 (a). Creation of one triangular element by connecting the two line segments S_2 and S_3.

B. $\pi/2 \leq \alpha \leq 5\pi/6$

Two triangular elements are created. This is done by creating an interior node as shown in Fig. 5(b). The position of the node is found in such away that is divided into two equal angles and the length of the dividing line segment, S, is given by:

$$S = \frac{1}{6}\left(S_1 + 2S_2 + 2S_3 + S_4\right) \tag{1}$$

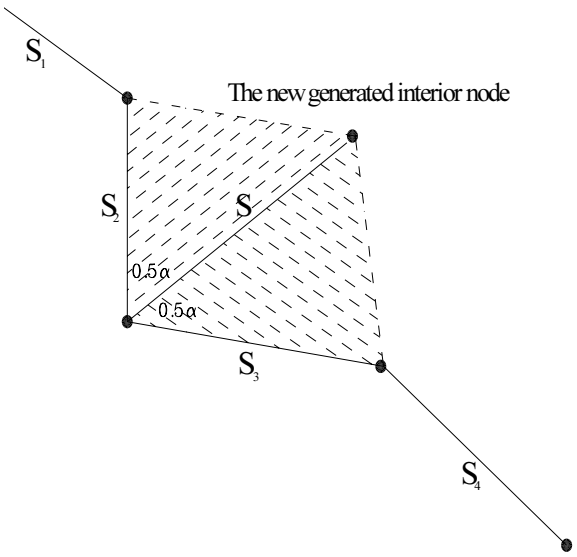

Fig. 5(b). Generation of two triangular elements by dividing the angle α into two equal angles.

C. $\alpha > 5\pi/6$

One triangular element is formed. This is done by creating an interior node opposite to the shorter line segment as shown in Fig. 5(c). The position of the node is found in such a way that the three sides, S_3, S_6, and S_5 are equal.

The number of nodes is increased by one each time an interior node is created. The position of the created node is checked to see if it is located outside the boundary or if it is close to an existing node. If that was the case, then the next smaller angle is taken as the departing candidate. The front is updated each time an element is created by suppressing old segments and adding the new ones. The elements' generation process is repeated until only four nodes are left on the front as shown in Fig. 5(d). This quadrilateral is divided into two triangles. Triangles 123 and 134 are formed, if $(\alpha_2+\alpha_4) \leq (\alpha_1+\alpha_3)$. Otherwise, triangles 124 and 234, are formed, see Fig. 5(d).

3. Grid Smoothing

The smoothing process of the generated grid is performed by shifting each interior node to the center of the surrounding polygon [14]. Let i be an interior node, N be the number of nodes surrounding node i, η_i be the set of nodes surrounding node i, x_o and y_o be the x- and y-coordinates of node i before smoothing as shown in Fig. 6. The x- and y-coordinates of node i after smoothing, x_n and y_n, are calculated as follows:

$$x_n = \frac{1}{N}\sum_{j\in\eta_i}^{N} x_j \tag{2}$$

$$y_n = \frac{1}{N}\sum_{j\in\eta_i}^{N} y_j \tag{3}$$

The distance between the old and new positions of the interior node i is given by:

$$r_i = \sqrt{(x_n - x_0)^2 + (y_n - y_0)^2} \tag{4}$$

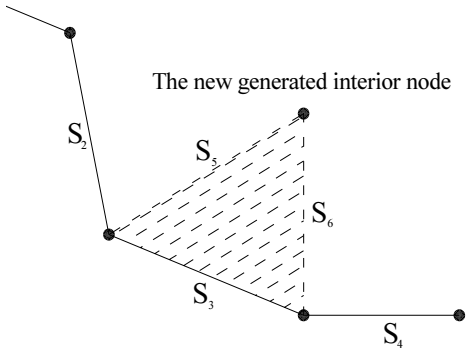

Fig. 5(c). Generation of one triangular element on the shorter line segment.

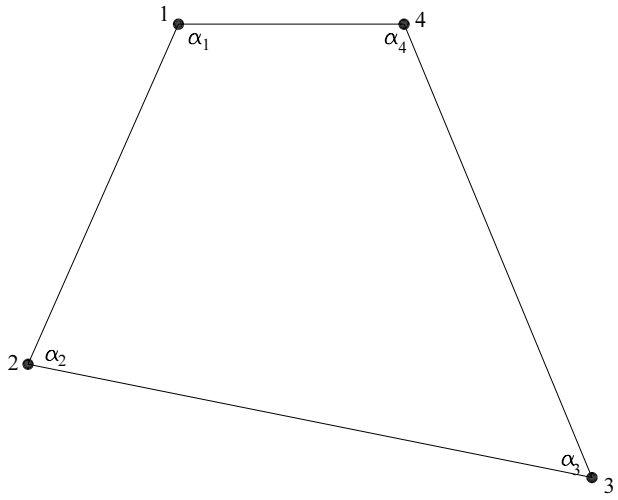

Fig. 5(d). The front with four nodes left on it.

On the other hand, the maximum distance, for all free nodes (interior nodes that are not fixed by other geometries), is given by:

$$r_{\max} = MAX\left(r_i\right) \tag{5}$$

Now, if r_{max} is less than a user-defined tolerance, ϵ, then, smoothing is done, else, the smoothing process is repeated for the new mesh until r_{max} is less than the desired tolerance, ϵ.

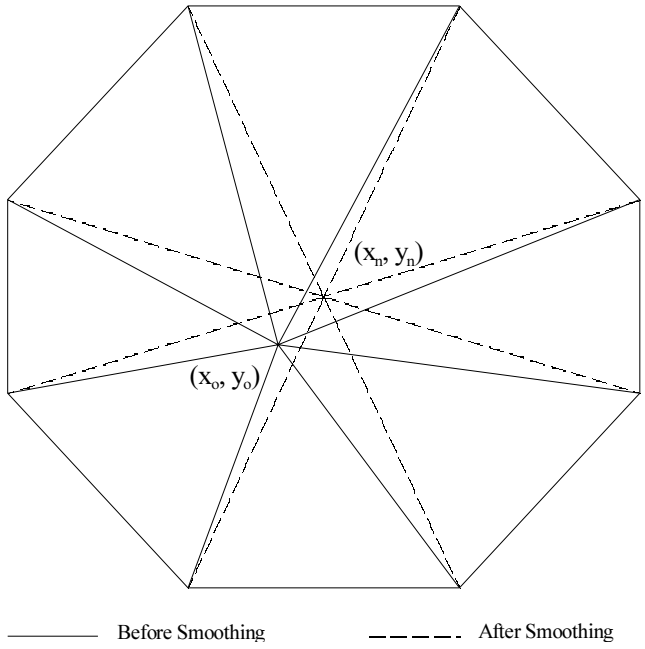

Fig. 6. Location of node i before and after smoothing.

4. Nodes Renumbering

Numbering of grid nodes has a definite influence on the *band width* of the coefficient matrix associated with the grid. The smaller the band width, the less storage and amount of computation required. Initially, each generated node is assigned the next node number in the grid. Therefore, by the end of the triangulation process, the grid nodes will be numbered arbitrarily in an unstructured manner. To find the band width of the coefficient matrix associated with a given grid, let: η_i be the set containing nodal numbers of node i and all surrounding nodes, $\tilde{\eta}_{min}$ be the minimum number in $\tilde{\eta}_i$, $\tilde{\eta}_{max}$ be the maximum number in $\tilde{\eta}_i$, then, p_i and q_i are defined as:

$$p_i = \eta_{\max} - i + 1 \tag{6}$$

$$q_i = i - \eta_{\min} + 1 \tag{7}$$

the maximum p and q, for all nodes in the grid, are given by:

$$p = MAX(p_i) \tag{8}$$

$$q = MAX(q_i) \tag{9}$$

and finally the *bandwidth*, ω, is calculated as:

$$\omega = p + q - 1 \tag{10}$$

Thus to reduce the band width of the coefficient matrix associated with a given grid, the difference $\tilde{\eta}_{max}$ - $\tilde{\eta}_{min}$ should be minimized. The Cuthill-McKee algorithm [4] minimizes the band width by renumbering grid nodes in such a way that the difference $\tilde{\eta}_{max}$ - $\tilde{\eta}_{min}$ be the minimum.

To begin with, the number of connections of each node will be referred to as the *degree* of that node. The Cuthill-McKee algorithm will be illustrated using the grid shown in Fig. 7(a), before renumbering, and the grid shown in Fig. 7(b), after renumbering. The algorithm can be s-ummarized in the following steps [4]:

1. A boundary node with the lowest degree becomes the starting node and is given the number 1 (node 5 in Fig. 7(a) becomes node 1 in Fig. 7(b)).

2. All surrounding nodes are numbered successively, in order of increasing degree. Nodes numbered in this step constitute the first level (dotted curve surrounding nodes 2, 3, and 4 in Fig. 7 (b)).

3. Move to the next level and start with the lowest number node; *i.e.*, node 2. Step 2 is performed for all nodes in this level. Nodes numbered in this level constitute the second level.

Step 3 is repeated until all nodes are numbered. When more than one node has the same degree, renumbering is done at each of them and the band width is calculated in each case. The node that gives the minimal band width is taken as the right candidate. Structure matrices before and after renumbering are shown in Fig. 7(c & d).

Meshed Cases

Many examples are presented to reveal the power and efficiency of the method. The examples include varieties of cases that were presented in the literature. The general domain example, however, has been designed for the purpose of this paper to check the algorithm's capability to handle all basic shapes simultaneously.

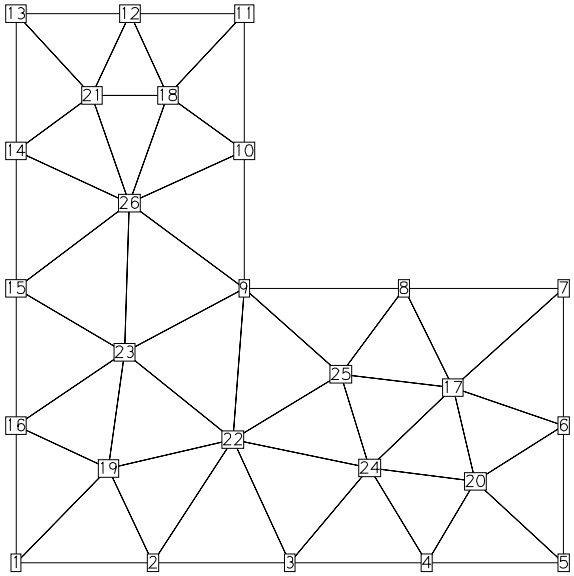

Fig. 7(a). Grid with initial numbering (NNM=26, NEM=34).

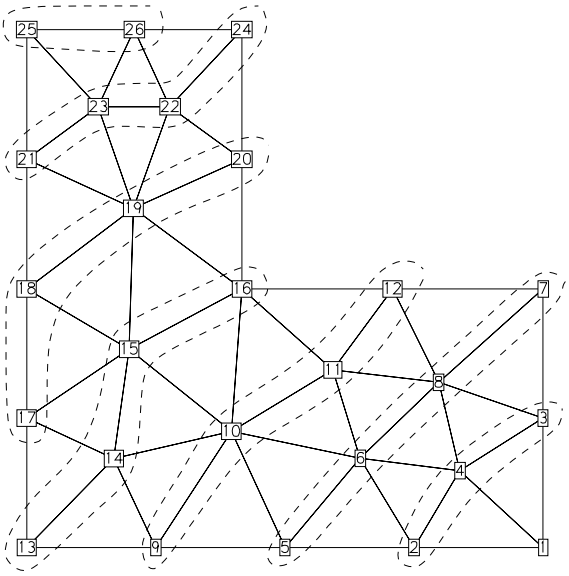

Fig. 7(b). Numbering of node points using the Cuthill-McKee algorithm (NNM=26, NEM=34, BW=13).

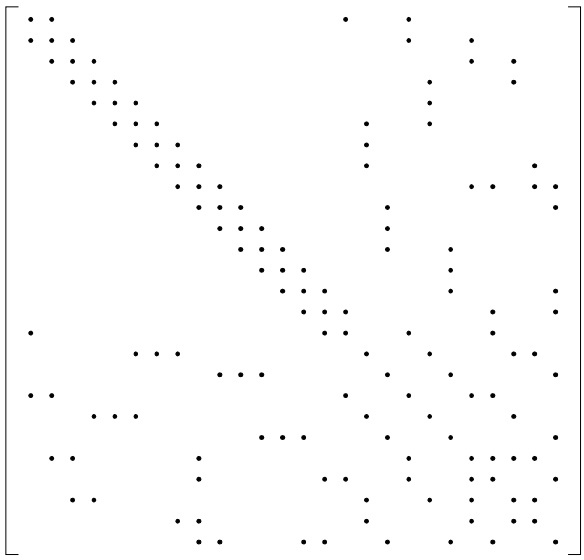

Fig. 7(c). Structure of the coefficient matrix associated with the grid shown in Fig. 7(a).

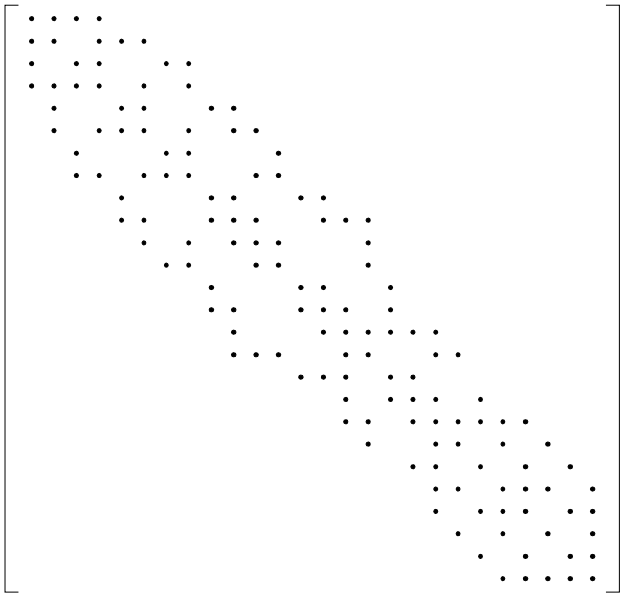

Fig. 7(d). Structure of the coefficient matrix associated with the grid shown in Fig. 7(b) (band width = 13).

General Domain

A general domain that consists of all basic shapes is considered as shown in Fig. 8(a). The geometry of this figure has been designed to test the algorithm's capability to handle all shapes simultaneously. The domain consists of successive lines, curves, a rectangle, polygons (triangle and hexagon), a circle, and an ellipse. The boundary nodes are shown as small donuts on the domain boundary. The generated mesh for this boundary is shown in Fig. 8(b). Note that a basic shape may be included in or excluded from the domain. This is simply accomplished by switching node direction, e.g. the ellipse is included in the triangulation process, whereas all other figures are excluded from the domain.

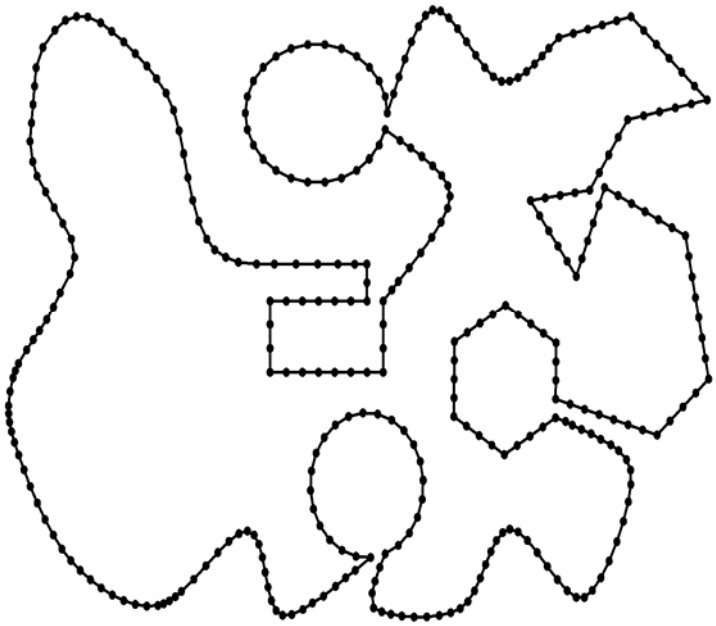

Fig. 8(a). Path representation of the domain to be triangulated.

Figure 8(c) depicts the same domain of Fig. 8(a) after moving and deleting some of the boundary points. Note that the ellipse has been disconnected from the rest of the shape by only moving one point to overlay the other. The program accommodates this by eliminating the point with the highest serial number. As a quality check of the generated mesh, the connecting point of the two shapes is shown in Fig. 8(d) (node

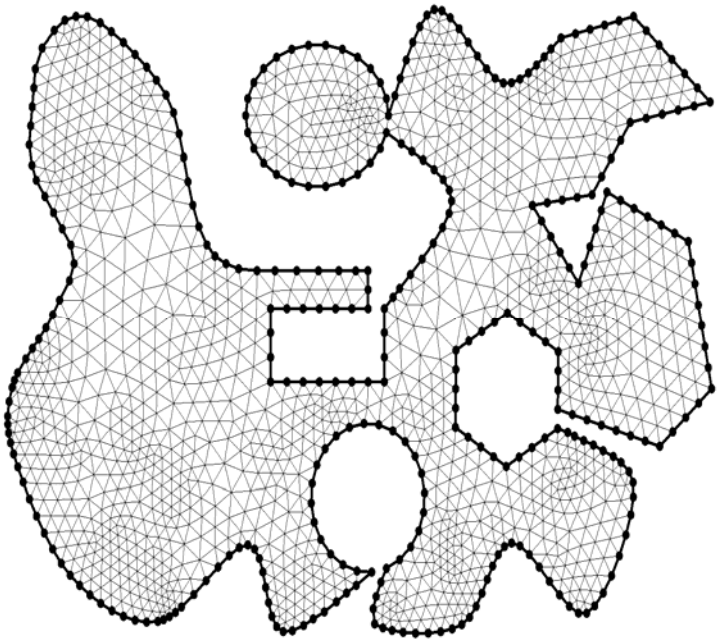

Fig. 8(b). Triangulated domain of Fig. 8(a).

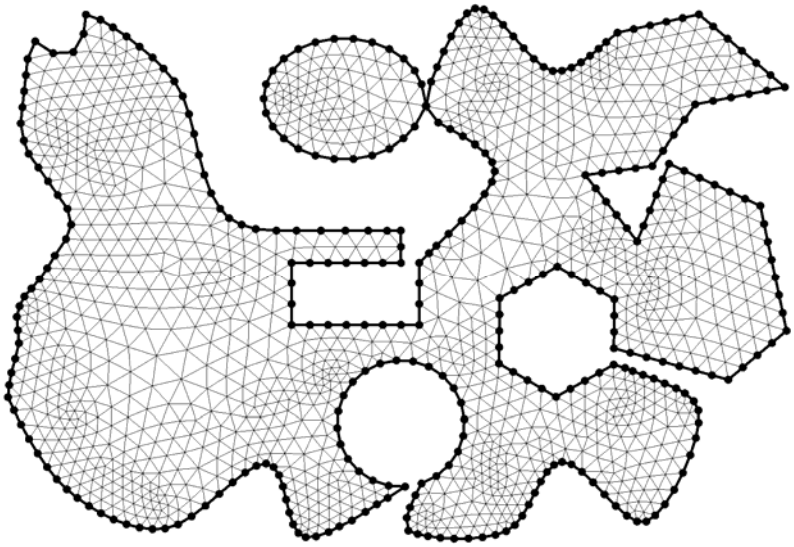

Fig. 8(c). Triangulated domain after editing.

number 325). Next the points on the upper-left corner of the domain were moved downward. One more place on the left side of the domain where some points were deleted and the rest were moved a little bit to the right.

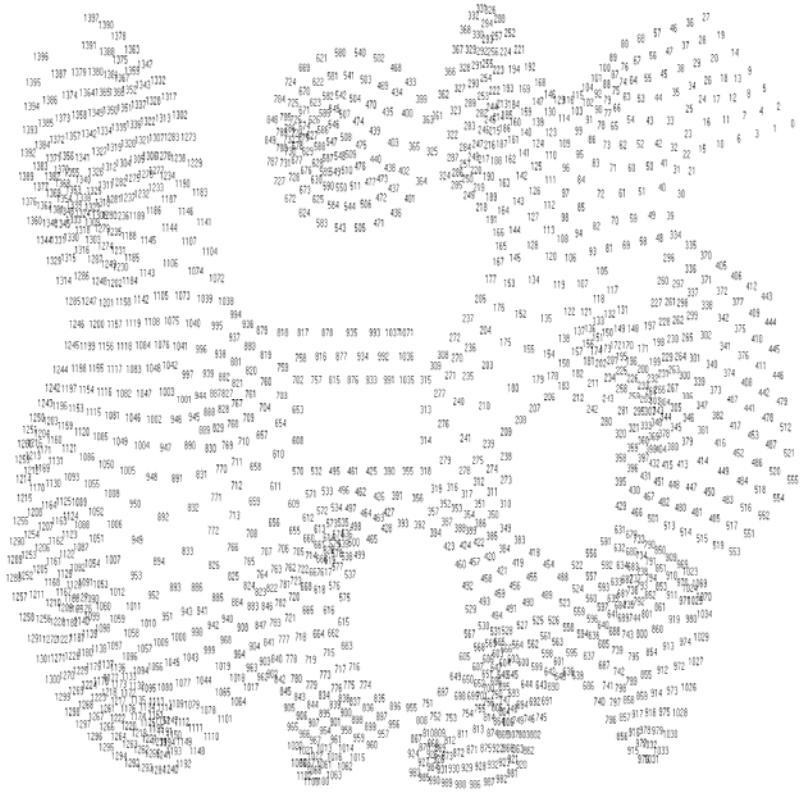

Fig. 8(d). Node numbers of the triangulated domain after editing.

El-Hamalawi Example

The following example was selected from El-Hamalawi work [5]. As he stated that the geometry could be a machine part with narrow edges and two holes of different shapes. Figure 9(a) shows the geometry with boundary nodes shown as small donuts. Initially a uniform mesh was produced as shown in Fig. 9(b), which consisted of 563 nodes and 966 elements.

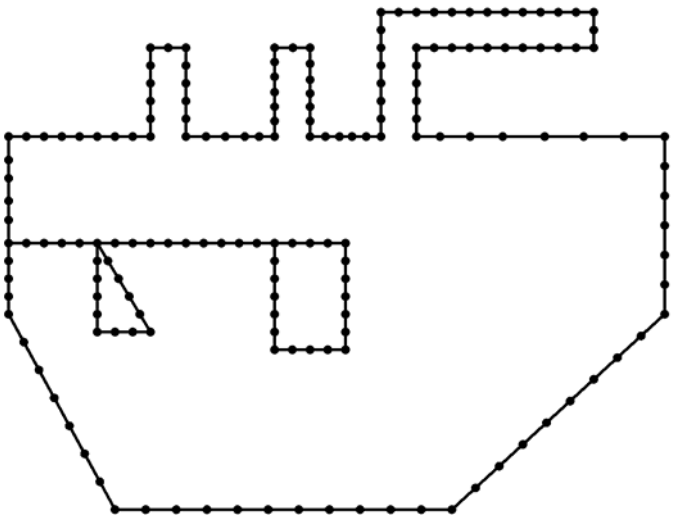

Fig. 9(a). The geometry of El-Hamalawi example.

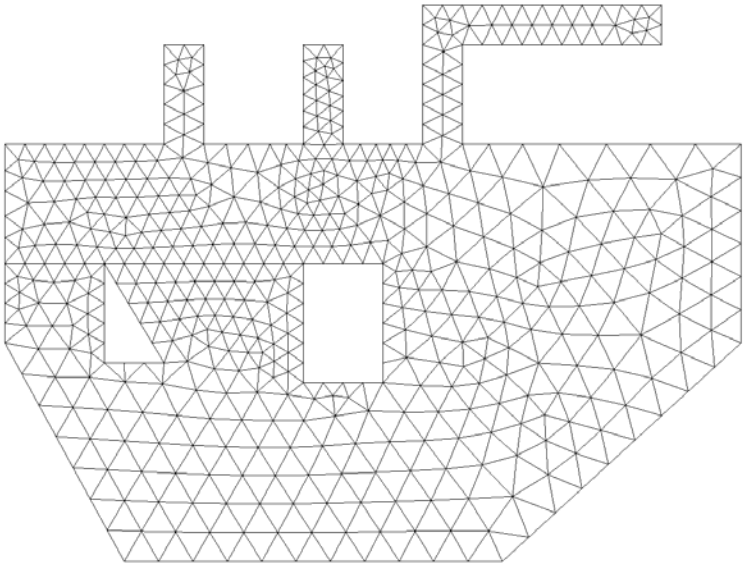

Fig. 9(b). The generated mesh of El-Hamalawi geometry.

Next a higher node density has been specified on the narrow edges of the domain. These are usually problematic when abrupt changes in element sizes occur. Figure 9(c) displays boundary node numbers which may not be visibly well in the denser area. Figure 9(d) shows the domain after triangulation, which consisted of 1445 nodes and 2585 elements.

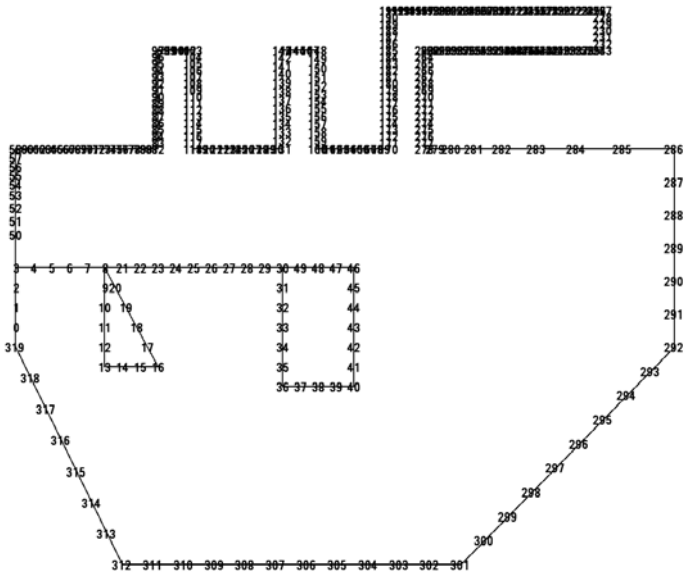

Fig. 9(c). El-Hamalawi geometry with higher node density at the narrow edges.

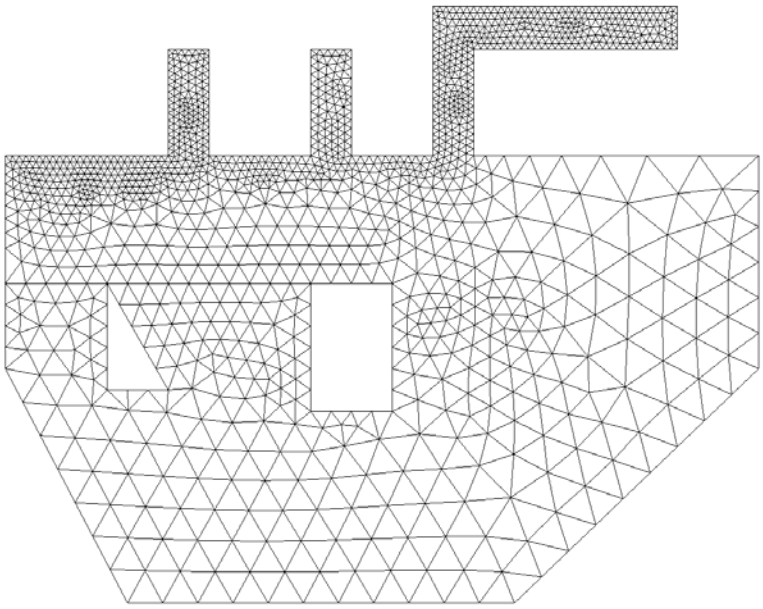

Fig. 9(d). El-Hamalawi geometry after triangulation.

Finally, for El-Hamalawi example, we required lesser node density at the right side of the domain as shown in Fig. 9(e). A total of 307 nodes were used to describe the front. The generated mesh consisted of 1198 nodes and 2104 elements. The bandwidth of the mesh coefficient matrix before node renumbering is 2181 and after renumbering is only 87 as shown in Fig. 9(f).

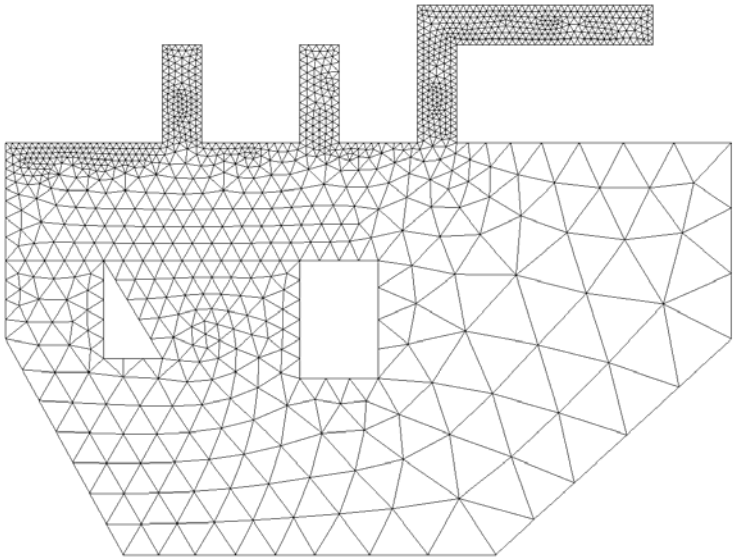

Fig. 9(e). El-Hamalawi example with lesser node density at the right side of the domain.

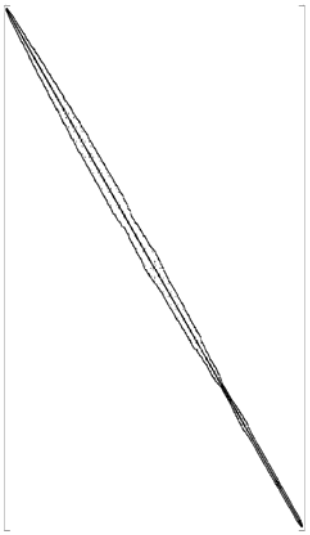

Fig. 9(f). The coefficient matrix of El-Hamalawi example as shown in Fig. 9(e).

Owen's Model

Figure 10(a) is an example of applying this algorithm to Owen's model as described in Ref. [8]; which consists of an external rectangle with a square and circle inside. The Fig. shows boundary nodes as small donuts on the boundary. Figure 10(b) displays the model after triangulation is completed with higher node density around the circle perimeter.

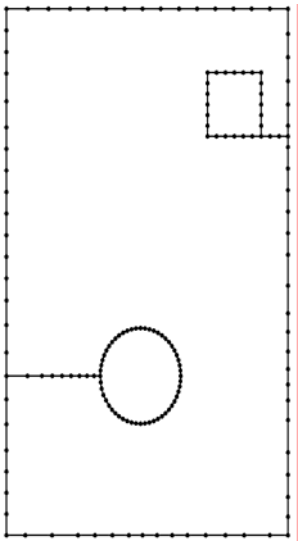

Fig. 10(a). Owen's model as presented in Ref.

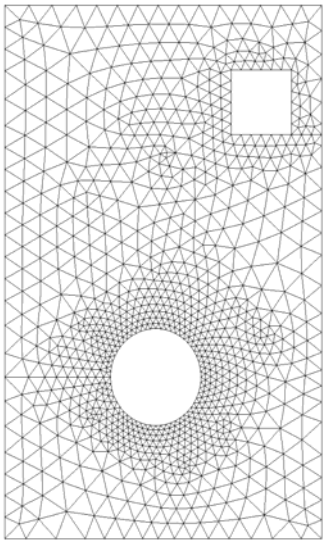

Fig. 10(b). Owen's model after triangulation.

Bastian Modified Example

The following modified example was selected from Bastian [1]. It considers a case where a small hole has been added off center from a large ellipse. Higher node density has been specified on the right side of the domain as shown in Fig. 11a. The final mesh after smoothing is shown in Fig. 11b.

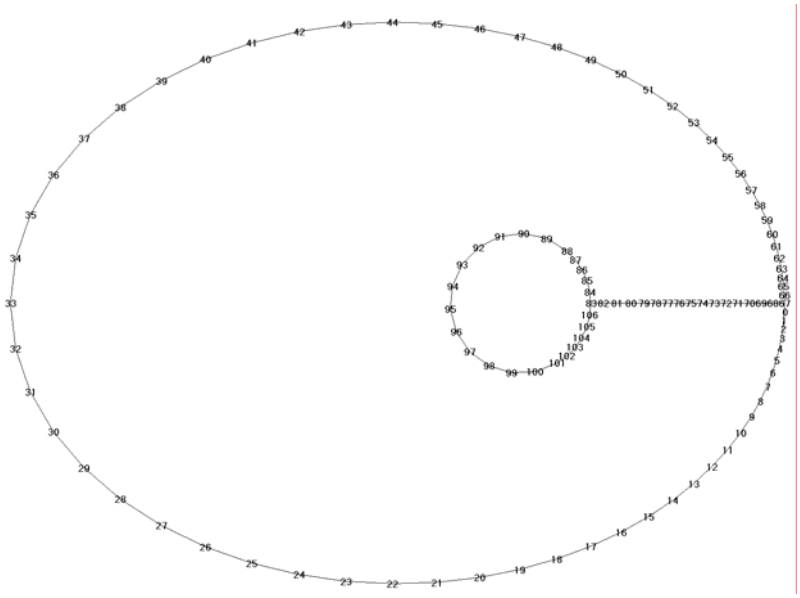

Fig. 11(a). An elliptical domain with an off circle inside.

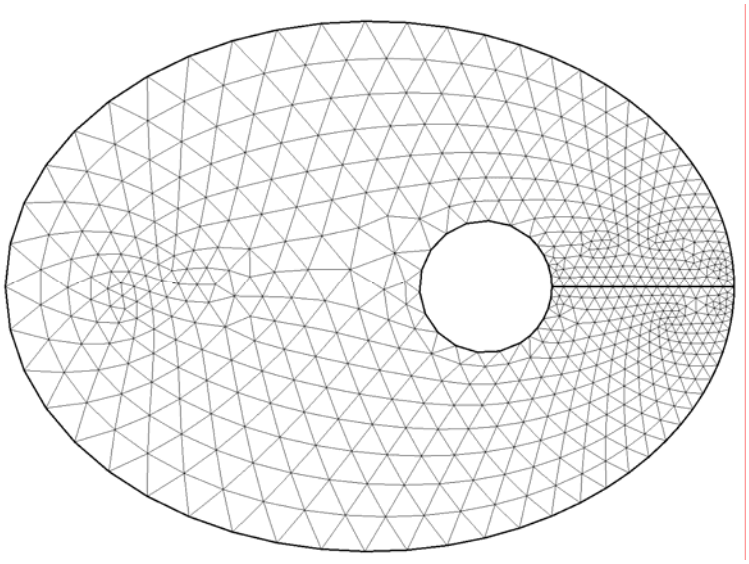

Fig. 11(b). Triangulated elliptical domain.

Highly-Irregular (Hand) Example

Figure 12(a) shows a hand example which represents a highly-curved region. Cubic-spline sets of curves were used to trace the hand. The hand is scanned in a clockwise direction. A cubic spline interpolation routine was used to explode the curve into a set of boundary nodes. Figure 12(b) depicts the triangulated domain. Note how the triangles are denser at the finger tips and pits.

Fig. 12(a). Hand domain to be triangulated.

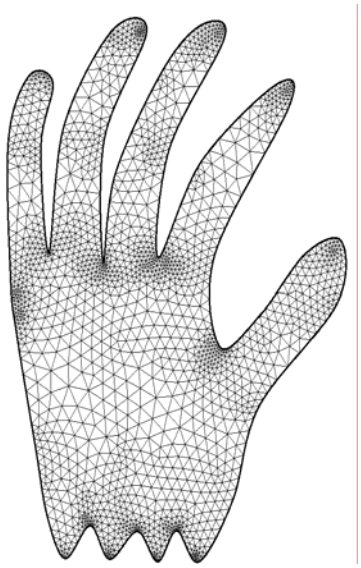

Fig. 12(b). Triangulated hand domain.

Conclusions

A method for creating efficient finite element grids was illustrated. The method is simple, fast, and requires minimal input. The method is an improved version of the advancing front technique as described in Ref. [14]. Examples were presented to illustrate the simplicity and efficiency of the method.

From an efficiency point of view, the mesh generation process is divided into two parts: *path representation* and *domain triangulation*. The path representation step may be thoroughly managed using object-oriented programming languages. The C# language offers this capability. The algorithm presented in this paper is developed using the C# object-oriented language. It facilitates the creation of initial fronts for the advancing front triangulation schemes. It almost eliminates user interference. Furthermore, it allows for any combination of basic shapes in any order.

The ability to start with a primitive shape (rectangle, polygon, circle, or ellipse) and switch to a path (a set of lines and/or curves) makes it flexible to trace highly-irregular and complex shapes. The path object outputs the domain boundary as a set of points (nodes) arranged successively in a clockwise or anticlockwise direction. This output becomes an input to the mesh generation object. The way a region path is traced has a very strong influence on the quality of the generated mesh.

References

[1] **Bastian, M.** and **Li, B. Q.,** An efficient automatic mesh generator for quadrilateral elements implemented using C++, *Finite Elements in Analysis and Design*, **39**: 905-930 (2003).

[2] **Bykat, A.**, "Automatic Generation of Triangular Grid: I-Subdivision of a General Polygon into Convex Subregions. II-Triangulation of Convex Polygons," *Int. J. Numer. Methods Eng.*, **10**: 1329-1342 (1976).

[3] **Cavendish, J. C.**, "Automatic Triangulation of Arbitrary Planar Domains for the Finite Element Method," *Int. J. Numer. Methods Eng*., **8**: 679-696 (1974).

[4] **Cuthill, E.** and **McKee, J.,** "Reducing the Bandwidth of Sparse Symmetric Matrices, " *Proc. 24rh Nat. Conf. Assoc. Comput. Mach*., pp: 157-172 (1969).

[5] **El-Hamalawi, A.**, "A 2D combined advancing front-Delaunay mesh generation scheme," *Finite Elements in Analysis and Design*, **40**: 967-989 (2004).

[6] **George, P. L.**, "*Automatic Mesh Generation: Application to Finite Element Methods*," John Wiley & Sons, pp: 137-147 (1991).

[7] **Hales, H. B.**, "A Method for Creating 2-D Orthogonal Grids Which Conform to Irregular Shapes," *SPE*, **35273** (1996).

[8] **Lee**, **Kyu-Yeul, Kim, In-I1, Cho, Doo-Yeoun** and **Kim, Tae-Wan**, "An algorithm for automatic 2D quadrilateral mesh generation with line constraints," *Computer-Aided Design*, **35**: 1055-1068 (2003).

[9] **Lämmer, L.** and **Burghardt, M.**, "Parallel generation of triangular and quadrilateral meshes," *Advances in Engineering Software*, **31:** 929-936 (2000).

[10] **Lee, C. K.** and **Hobbs, R. E.**, "Automatic adaptive finite element mesh generation over rational B-spline surfaces," *Computers and Structures*, **69**: 577-608 (1998).

[11] **Lee, C. K.** and **Hobbs, R. E.**, "Automatic adaptive finite element mesh generation over arbitrary two-dimensional domains using advancing front technique," *Computers and Structures*, **71**: 9-34 (1999).

[12] **Lo, S. H.**, "A New Mesh Generation Scheme For Arbitrary Planar Domains", *Int. J. Numer. Methods Eng.*, **21**: 1403-1426 (1985).

[13] **Lo, S.H.** and **Wang, W. X.**, "Generation of finite element mesh with variable size over an unbounded 2D domains," *Comput. Methods Appl. Mech. Engrg*., **194:** 4668-4684 (2005).

[14] **Naji, H.**, "An Improved advancing front algorithm for triangulating arbitrary two-dimensional regions," *The 17th National Computer Conference*, *April 2004*, pp: 505-518, King Abdulaziz University, Jeddah, Saudi Arabia.

[15] **O'Bara, R. M.**, "Adaptive mesh generation for curved domains," *Applied Numerical Mathematics*, **52:** 251-271(2005).

[16] **Sadek, E. A.**, "A Scheme for the Automatic Generation of Triangular Finite Elements", *Int*. J. Numer. Methods Eng., **15**:1813-1822 (1980).

[17] **Secchi, S.** and **Simon, L.**, "An improved procedure for 2D unstructured Delaunay mesh generation," *Advances in Engineering Software*, **34**: 217-234 (2003).

Appendix (A)
A listing of the Line class

```
     public class Line
     {
     // User-input fields
      public PointF b;     // begin point of line
      public PointF e;     // end point of line
      public float  s;     // requested point spacing on line
      public float  r;     // ratio of last/first point spacing
on line

      // Class-required fields
      public byte nClickLMB;     // click number of left mouse
button
      public PointF pt;          // tracking point
      public bool isFinished;    // drawing status

      public Line( )
      {
            Clear();
      }

      public void Clear( )
      {
            this.b              = PointF.Empty;
            this.e              = PointF.Empty;
            this.s              = 0;
            this.r              = 0;
            this.nClickLMB      = 0;
            this.pt             = PointF.Empty;
            this.isFinished     = false;
      }

      public void MouseDown(object sender,
System.Windows.Forms.MouseEventArgs mea)
      {
            if(mea.Button ==
System.Windows.Forms.MouseButtons.Left)
            {
                  this.nClickLMB++;
                  if(this.nClickLMB == 1)
                  {
                        this.b = new PointF(mea.X, mea.Y);
                        this.isFinished = false;
                  }
                  else if(this.nClickLMB == 2)
                  {
                        this.e = new PointF(mea.X, mea.Y);
                        this.isFinished = true;
                  }
            }
      }
      public void MouseMove(object sender,
System.Windows.Forms.MouseEventArgs mea)
```

```
        {
                if(this.nClickLMB == 1)
                {
                        System.Windows.Forms.Panel panel =
                                (System.Windows.Forms.Panel) sender;
                        Graphics grfx = panel.CreateGraphics();
                        Draw(grfx, panel.BackColor, this.pt);
                        this.pt = new PointF(mea.X, mea.Y);
                        Draw(grfx, panel.ForeColor, this.pt);
                        grfx.Dispose();
                }
        }

        public void Draw(Graphics grfx, Color clr, PointF pt)
        {
                this.e = pt;
                Pen pen = new Pen(clr);
                grfx.DrawLine(pen, this.b, this.e);
                pen.Dispose();
        }

        public void Draw(Graphics grfx, Pen pen)
        {
                grfx.DrawLine(pen, this.b, this.e);
        }

        public PointF[] Explode( )
        {
                return Explode(this.b, this.e, this.s, this.r);
        }

        public PointF[] Explode(float s, float r)
        {
                return Explode(this.b, this.e, s, r);
        }

        public static PointF[] Explode(PointF b, PointF e, float
es, float r)
        {
                // This function explodes a line into a set of
points
                decimal l = (decimal)
TriGrid.PathComponent.Point.Distance(b, e);
                decimal s = (decimal) es;
                int n     = (int) (l/s);
                PointF[] points = new PointF[n+1];
                points[0] = b;
                points[n] = e;
                if(n <= 1) return points;
                decimal Sum = 0.0m;
                for(int c = 0; c < n; c++)
                        Sum += (1 + c*((decimal) r- 1.0m)/(n-1));
                decimal ss = l/Sum;
                decimal[] sl = new decimal[n];
                for(int c = 0; c < n; c++)
```

```
                    sl[c] = ss + c*((decimal) r - 1.0m)*ss/(n-
1);
             PointF p = new PointF(e.X-b.X, e.Y-b.Y);
             float a  = TriGrid.PathComponent.Point.XAngle(p);
             for(int c = 1; c < n; c++)
             {
                    points[c].X = points[c-1].X + ((float)
(sl[c-1]*((decimal)
     System.Math.Cos(a))));
                    points[c].Y = points[c-1].Y + ((float)
(sl[c-1]*((decimal)
     System.Math.Sin(a))));
             }
             return points;
       }

       public void Write(StreamWriter sw)
       {
             sw.WriteLine(this.b.X.ToString());
             sw.WriteLine(this.b.Y.ToString());
             sw.WriteLine(this.e.X.ToString());
             sw.WriteLine(this.e.Y.ToString());
             sw.WriteLine(this.s.ToString());
             sw.WriteLine(this.r.ToString());
       }

       public void Read(StreamReader sr)
       {
             this.b = new PointF(float.Parse( sr.ReadLine() ),
                    float.Parse( sr.ReadLine() ));
             this.e = new PointF(float.Parse( sr.ReadLine() ),
                    float.Parse( sr.ReadLine() ));
             this.s = float.Parse( sr.ReadLine() );
             this.r = float.Parse( sr.ReadLine() );
       }
     }
```

برنامج بلغة C# لتقسيم المناطق غير المنتظمة ثنائية البعد إلى مثلثات باستخدام طريقة الواجهة المتقدمة

حسن سليمان ناجي

قسم جيولوجيا البترول والترسبات، كلية علوم الأرض، جامعة الملك عبد العزيز

جدة، المملكة العربية السعودية

hassan@petrobjects.com & petrobjects.com

المستخلص. يصف هذا البحث إنشاء برنامج بلغة C# لتقسيم أي منطقه غير منتظمة ثنائية البعد إلى مثلثات لاستخدامها في حل المعادلات غير الخطية بطريقة العنصر المحدود (Finite Element Method). هذه الطريقة، والتي تعتمد على طريقة الواجهة المتقدمة في تقسيم المناطق، تتطلب حدود المنطقة فقط كمدخل وحيد. في هذه الطريقة، يتم استخدام الأشكال الأساسية مثل الخطوط المستقيمة، والمنحنيات، والمستطيلات، ومتعددات الأضلاع والدوائر، والقطوع الناقصة لإنشاء الحد الخارجي للمنطقة، حيث يمكن إضافة هذه الأشكال بطريقة تفاعليه وسهلة في ترتيب تسلسلي، وكلما أضيف شكل ما يتم تقسيمه مباشرة إلى مجموعة نقاط تحكّم تضاف إلى حدود المنطقة. هذه النقاط لابد أن تكون مستمرة ولا ينبغي أن يقطع بعضها البعض، بالإضافة إلى أن هذه النقاط يمكن تحريكها أو حذفها بأوامر في غاية السهولة وهذا يمكِن بدوره من إنشاء أشكال في غاية التعقيد بالإضافة إلى إخراج الشكل النهائي على الوجه المطلوب.

إن الفتحات الداخلية والتي قد توجد في المناطق ثنائية البعد يمكن إضافتها عن طريق خطوط مستقيمة تأخذ اتجاهين من وإلى الحد الخارجي للشكل. وينتج عن هذه الطريقة مثلثات متساوية الأضلاع كلما أمكن، بالإضافة إلى وجود إجراء تحسيني يتم بتحريك النقاط الداخلية إلى مراكز متعددة الأضلاع. وحيث إن لترقيم نقاط التحكم تأثيراً مباشراً على حجم مصفوفة المعاملات (Coefficient Matrix) المرافقة (Finite Element Mesh)، فكلما كان عرض شريط المصفوفة أقل كلما قلت السعة المطلوبة للحل من ذاكرة الحاسب. ولقد استخدمت طريقة Cuthill-Mckee في إعادة ترقيم مصفوفة المعاملات. إن استخدام لغة C# قد أضاف

مرونة كبيرة في برمجة هذه الطريقة وأضاف فعالية أكبر في إنشاء أعقد الأشكال. وقد تمت الاستعانة بالعديد من الأمثلة من المجلات العالميـــة ذات العلاقـــة والتـــي توضح بساطة وقوة هذه الطريقة.