hard copy ISSN 1974-3041
on-line ISSN 1974-305X

La Matematica
e le sue Applicazioni
n. 13, Luglio 2008

BBTR: an unstructured triangular mesh generator

A. M. A. Barbera, S. Berrone

Quaderni del
Dipartimento di Matematica
Politecnico di Torino
Corso Duca degli Abruzzi, 24 – 10129 Torino – Italia

Edizioni C.L.U.T. - Torino
Corso Duca degli Abruzzi, 24
10129 Torino
Tel. 011 564 79 80 - Fax. 011 54 21 92

La Matematica e le sue Applicazioni
hard copy ISSN 1974-3041
on-line ISSN 1974-305X
Direttore: Claudio Canuto
Comitato editoriale: N. Bellomo, C. Canuto, G. Casnati, M. Gasparini, R. Monaco,
G. Monegato, L. Pandolfi, G. Pistone, S. Salamon, E. Serra, A. Tabacco

Esemplare fuori commercio

BBTR: an unstructured triangular mesh generator

Andrea M.A. Barbera, Stefano Berrone

Abstract

In this paper we give a detailed description of the use of a `Matlab`® triangular mesh generator called BBTR. This code offers many features for an implementation of a finite element code. Particular attention has been paid to the generality of the polygonal domains that the triangulator can deal with. General non convex polygonal domains, as well as domains with holes, internal constrained segments and sharp angles of the boundary, can be successfully processed by BBTR. Moreover, suitable algorithms have been implemented to ensure a good quality of the mesh, controlling the angles of the elements. Suitable data structures produced by the mesh generator allow to set general boundary conditions.

The code BBTR results to be a quite general and efficient triangular mesh generator.

Key words. Mesh generation, Delaunay triangulations, constrained Delaunay triangulations, Planar Linear Straight Graphs.

AMS subject classifications. 65M30, 65N50, 97U30, 68-00

1 Introduction

Mesh generation is a fundamental issue in scientific computing and in the simulation of all the phenomena described by partial differential equations. Finite elements are very often used in the numerical solutions of elliptic and parabolic problems. Due to their flexibility in the discretization of complex domains and in giving strongly non uniform meshes, triangular grids are very common in the discretization of 2D problems. Moreover, 2D problems are the simplest "real world problems" that can be shown to students of courses of numerical methods for partial differential equations and scientific computing. To our knowledge, a triangular mesh generator freely available to the students and designed to be used within the widely spread software `Matlab`® was missing. Despite teaching targets stimulated the project of developing this code, we paid much attention to efficiency, in order to construct an instrument useful also to the community of researchers and engineers, who possibly can get some advantages from a simple and fast experimentation of their models and methods within `Matlab`®.

Our code, BBTR, can construct a constrained Delaunay triangulation of general polygonal domains, not necessarily convex. Indeed the code can triangulate domains with holes and can satisfy some additional constraints like having some fixed segments in the triangulation. The use within a finite element discretization requires two additional properties for a mesh generator: the first is the quality of the produced mesh: all the triangles must have internal angles not too small (or equivalently not too large); the second is the need to prescribe some element-size and to require some possible local refinement to suitably adapt the mesh to the solution.

The description of the polygonal geometry is given to the code BBTR by fixing the vertices and the boundaries as straight borders between the vertices. Moreover, to allow the finite element codes to impose suitable boundary conditions, vertices and borders are marked distinguishing between Dirichlet boundary conditions (no unknown values on the boundary) and Neumann (or Robin) boundary conditions. It is worth mentioning that these markers can be used to address any kind of feasible boundary condition: one can deal with constant boundary conditions directly using the data structures provided by the triangulator, whereas more general boundary conditions can be treated within the finite element code.

The data structures given by the code BBTR as output are planned to make the implementation of a space discretization with linear finite elements an easy and direct task. Nevertheless, with very small changes the given data structures can be extended to quadratic elements. Further, the data structures provided by the code BBTR give a description of the relations between the geometrical objects of the triangulation and the neighboring objects (triangles, edges, points) to make possible not only an implementation of classical finite elements, but also the evaluation of *a posteriori* error estimators within adaptive algorithms and the implementation of discontinuous Galerkin methods that are recently attracting a large research effort.

The code BBTR is available at the URL:

`http://calvino.polito.it/~sberrone/Faculty/Tesi/Barbera`

At the moment we are extending the code BBTR to the 3D case and a detailed description of the implemented algorithms and coding solutions is in preparation.

In the following of this manual we sometimes address the straight borders of the domain by the word *side*, we reserve the name *edge* for the edges of the triangles and we use the word *segment* for the internal segments constraining the triangulation. We remark that according to our convention, a side or a segment can contain several edges after the triangulation is built.

2 Inputs

In this section we describe the data structures (or variables) to be set in order to successfully run BBTR. Three input variables must be considered:

- `Domain`: used to describe the domain;
- `BC`: used to set boundary conditions on the borders of the domain;
- `RefiningOptions`: used to describe the properties that the mesh built by BBTR must fit.

Among these variables `Domain` is the main one, but also other two inputs must generally be carefully considered. We recall that `Matlab`® is a case-sensitive environment, so problems may arise if the user does not pay attention to this issue.

2.1 The variable `Domain`

The typical domain that BBTR may triangulate is a so called *Planar Linear Straight Graph* (or *PLSG*, see [8]). PLSGs are planar domains bounded by *straight borders* (*sides*) that may have holes (bounded again by straight borders) and *segments* in their inner region. Straight borders and segments must end in *vertices*. An example of PLSG is shown in Figure 1. An example of meshed PLSG is shown in Figure 2.

The elements of an admissible PLSG may also form small angles, both in the boundary of the domain or in the boundary of holes inside the domain. The presence of these small angles may require suitable settings of some parameters to successfully run BBTR (see Section 4).

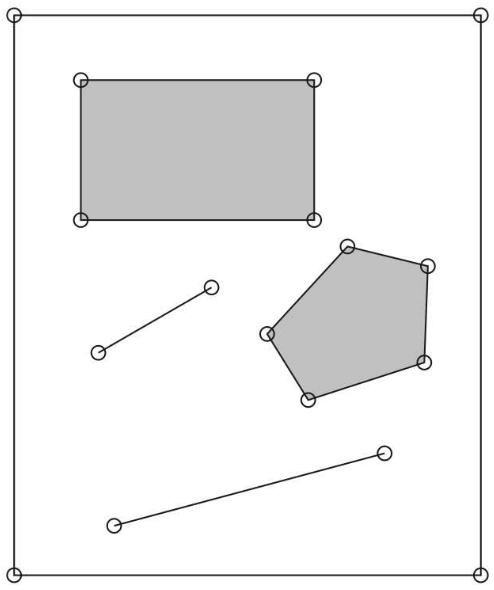


Figure 1: Example of PLSG. The shaded regions are holes of the domain.

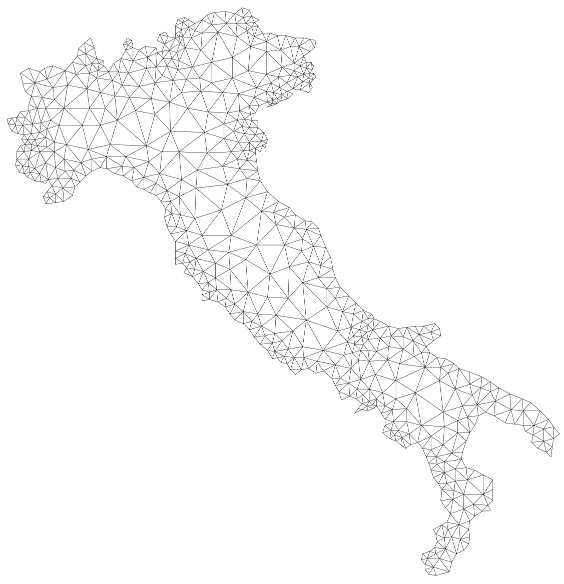


Figure 2: Example of mesh generated by BBTR starting from a complex PLSG.

The definition of a domain is given by setting the vertices of the domain and a description of some straight borders connecting these vertices. Holes and internal segments that the triangulation must fit are defined in the same way. Curves bounding the domain must be approximated by some piecewise straight borders, adding as many vertices as needed in order to obtain a sufficient description of the shape of the domain. Analogously, curves bounding holes and curved lines inside the domain must be realized as piecewise straight lines. An example of PLSG approximation of a curved domain in shown in Figure 3.

The structure `Domain` is composed by the following fields:

- `Domain.InputVertex`;
- `Domain.Boundary`;
- `Domain.Holes`;
- `Domain.Segments`.

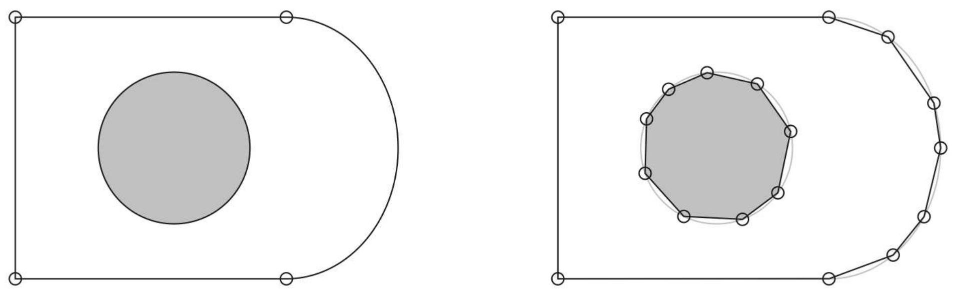


Figure 3: Example of PLSG discretization: new vertices are inserted in order to approximate the domain.

Each one of the above mentioned fields must be assigned in `Domain` in order to successfully run BBTR on a complex PLSG. In the following subsections we will describe in details these fields.

2.1.1 `Domain.InputVertex`

The field `Domain.InputVertex` is a $[n_V \times 2]$ matrix containing the Cartesian coordinates of all the n_V vertices used to describe the domain. The n–th row represents the n–th input vertex. The first component of n–th row contains its x-coordinate and the second component contains the y-coordinate.

Example 2.1 Let us define the four vertices of a squared domain:

```
Domain.InputVertex = [0 0
  5 0
  5 5
  0 5];
```

In this example the first vertex is $(0, 0)$, the second is $(5, 0)$, the third is $(5, 5)$ and the fourth is $(0, 5)$.

We remark that the ordering of the vertices in `Domain.InputVertex` is not relevant: any permutation of the vertices is equivalently admissible. The only important data are the coordinates of the vertices.

2.1.2 `Domain.Boundary`

The structure `Domain.Boundary` contains the field `Values`. The field `Domain.Boundary.Values` is an array defining the straight borders connecting the vertices whose coordinates are given in `Domain.InputVertex`.

`Domain.Boundary.Values` is a sorted list of integers, all contained in the range between 1 and n_V. Each couple of adjacent numbers i_1 and i_2 in this field corresponds to a straight line connecting vertices V_{i_1} and V_{i_2}. The vertices in `Domain.Boundary.Values` are

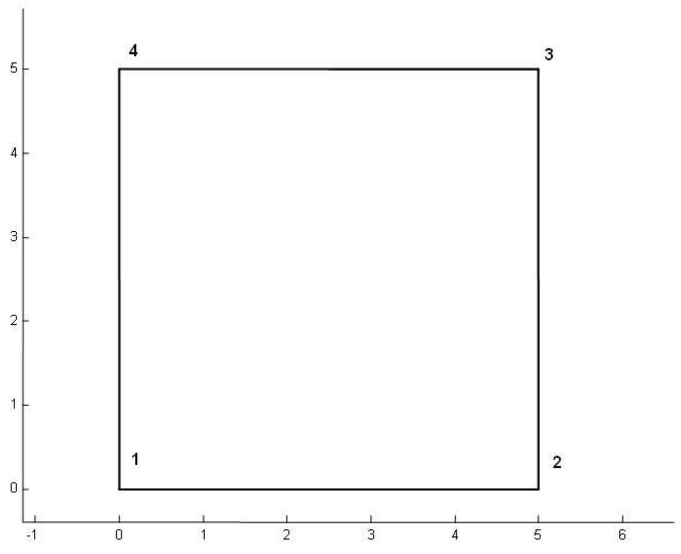


Figure 4: Example of squared domain.

denoted by their position in `Domain.InputVertex`. There is also a straight line between the last and the first vertices listed in `Domain.Boundary.Values`.

We remark that possible inner holes and inner segments are not described in the structure `Domain.Boundary`. These features of the domain may be described by setting the variables `Domain.Holes` and `Domain.Segments`, (see Sections 2.1.3 and 2.1.4).

Let us stress that `Domain.Boundary.Values` must contain an open and not a closed list: BBTR already considers that a side is present between the last and the first element of `Domain.Boundary.Values`: if the user forces the insertion of this side, this will result in an error.

Example 2.2 The assignment

```
Domain.Boundary.Values = [1 2 3 4];
```

describes a domain whose boundary is given by a straight line (side) between the first two vertices of `Domain.InputVertex`, a side between the second and the third, a side between the third and the fourth and a side between the fourth and the first. The first vertex has coordinates given in the first row of `Domain.InputVertex`, the second vertex in the second row and so on.

Example 2.3 Let us consider again the domain introduced in Examples 2.1 and 2.2 and illustrated in Figure 4. Referring to the variable `Domain.InputVertex`, the domain may be described in different ways. For example, we may set:

```
Domain.InputVertex = [0 0
  5 0
  5 5
  0 5];
Domain.Boundary.Values = [1 4 3 2];
```

or equivalently

```
Domain.InputVertex = [0 0; 5 0; 5 5; 0 5];
Domain.Boundary.Values = [3 4 1 2];
```

but not

```
Domain.InputVertex = [0 0
  5 0
  5 5
  0 5];
Domain.Boundary.Values = [1 3 4 2];
```

because there are no borders between vertices 1 and 3 or between vertices 2 and 4. As it is clear from this example, the boundaries can be described both clockwise and anticlockwise.

2.1.3 `Domain.Holes`

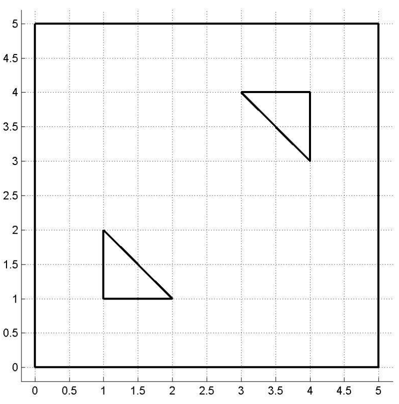
The structure `Domain.Holes` describes possible holes in the domain. This structure contains `Hole` as a field: `Domain.Holes.Hole` is an array whose dimensions are equal to the number of holes present in the domain. Every element of `Domain.Holes.Hole` is a structure itself with a field called `Values`.

`Domain.Holes.Hole(n).Values` is a sorted list of references to vertices, whose coordinates are given in `Domain.InputVertex`. It defines the straight borders which bound the n–th hole of the domain. The definition of a hole in `Domain.Holes.Hole(n).Values` is similar to the definition of the variable `Domain.Boundary.Values`: in the n–th hole of the domain there is a straight line between any two adjacent vertices whose position in `Domain.InputVertex` is the element of `Domain.Holes.Hole(n).Values` and between its first and last vertex.

The same conventions used in `Domain.Boundary.Values` apply, and, in particular, these sorted list are open and not closed.

If the domain has no holes, `Domain.Holes.Hole` must be an empty array:

```
Domain.Holes.Hole = [ ];
```



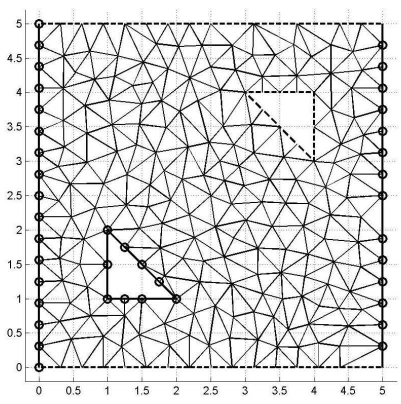


Figure 5: Squared domain with holes described in Example 2.4.

Figure 6: Mesh generate by BBTR starting from the domain described in Example 2.4.

Example 2.4 Let us consider the squared domain of Figure 5, containing two triangular holes inside the domain.

```
Domain.InputVertex = [ 0 0
  5 0
  5 5
  0 5
  1 1
  1 2
  2 1
  3 4
  4 3
  4 4];
Domain.Boundary.Values = [1 2 3 4];
Domain.Holes.Hole(1).Values = [5 6 7];
Domain.Holes.Hole(2).Values = [8 9 10];
```

The first hole, the lowest one in the Figure 5, is defined through straight borders by the assignment `Domain.Holes.Hole(1).Values = [5 6 7]`. This hole is realized with a straight border between the fifth and sixth vertices, with a straight border between the sixth and the seventh vertices and with a straight border between seventh and fifth vertices.
The second hole is defined in an analogous way.

2.1.4 `Domain.Segments`

The structure `Domain.Segments` is used in order to describe the segments in the domain that the mesh must fit. Its definition is very similar to the one of `Domain.Holes`. We point out that, in order to allow the user to insert segments more quickly, piecewise straight lines (a connected sequence of segments) may be inserted as a unique object.

The structure `Domain.Segments` contains the field `Segment`: `Domain.Segments.Segment` is an array whose dimension is equal to the number of piecewise straight lines present in the domain. Every element of `Domain.Segments.Segment` has, itself, a field called `Values`.

The array `Domain.Segments.Segment(n).Values` is a sorted list of references to vertices whose coordinates are given in `Domain.InputVertex`. It defines the segments forming the n–th piecewise straight line inside the domain. The use is the same as the one explained for `Domain.Boundary.Values` and `Domain.Holes.Hole(n).Values` (there is a segment between any two adjacent elements of `Domain.Segments.Segment(n).Values`), except for the fact that now there is not a segment automatically included between the last and the first vertices of `Domain.Segments.Segment(n).Values`.

If the domain has no inner segments, `Domain.Segments.Segment` must be an empty array: `Domain.Segments.Segment = [ ];`

Applications of `Domain.Segments` are reported in Example 2.5.

Example 2.5 Let us consider the domain illustrated in Figure 7 and obtained with following assignments:

```
Domain.InputVertex = [ 0 0
   5.0 0.0
   5.0 5.0
   0.0 5.0
   0.8 1.0
   2.2 4.1
   4.1 3.2
   3.5 1.0
   2.1 1.0
   3.0 3.1];
Domain.Boundary.Values = [1 2 3 4] ;
Domain.Holes.Hole = [];
Domain.Segments.Segment(1).Values = [5 6 7 8];
Domain.Segments.Segment(2).Values = [9 10];
```

This domain contains two different piecewise straight lines, defined by `Domain.Segments.Segment(1).Values = [5 6 7 8];` and `Domain.Segments.Segment(2).Values = [9 10];`. These lines are composed respectively by three and one segments.
The assignment `Domain.Holes.Hole = [];` means that no holes are present in the domain. An example of mesh generated by BBTR is reported in Figure 8

2.1.5 Remarks on the vertices in `Domain.InputVertex`

Every vertex listed in `Domain.InputVertex` is inserted in the mesh in any way, even if it is not used to define the shape of the domain, a hole or a piecewise straight line.

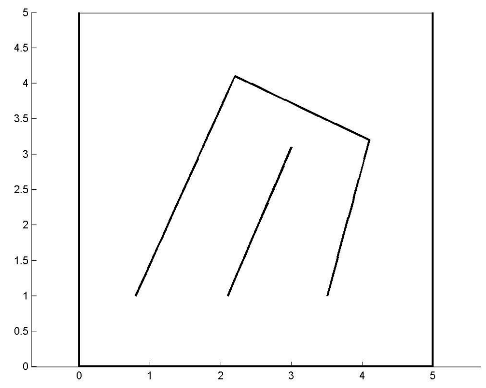


Figure 7: Domain with inner segments described in Example 2.5.

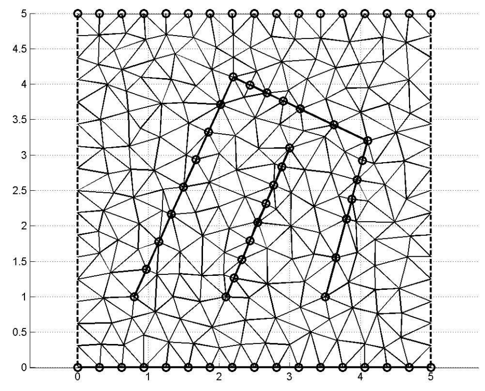


Figure 8: Mesh generated by BBTR starting from the domain described in Example 2.5.

A single vertex may however be used in order to define several geometrical objects, as for example several boundaries of the domain and/or different segments. On the contrary, inserting twice the same vertex will cause an error.

Finally, elements of the triangulations must be coherent and may not intersect each other, but may however touch themselves in a single point (so if for example a "cross" composed by two piecewise straight lines must be inserted in the domain, also the intersection point must be inserted in `InputVertex`).

2.2 The variable `BC`

The structure `BC` contains a description of the *boundary conditions* (or *BCs*). We have considered general Dirichlet and Neumann boundary conditions, but the data structure provided by BBTR allows the setting of Robin boundary conditions as well.

`BC` is a structure composed by different fields:

- `BC.InputVertexValues`
- `BC.Boundary`
- `BC.Holes`
- `BC.Segments`
- `BC.Values`

Each of the above mentioned fields must be included in the definition of `BC` to successfully run BBTR.

Markers are stored in most of these fields, and each marker is also a reference to a constant value set in `BC.Values`. This value can eventually be used to set the corresponding boundary conditions.

We use the following convention:

- odd markers refer to Dirichlet conditions;
- even markers refer to Neumann conditions;
- marker "0" means no boundary condition.

2.2.1 `BC.InputVertexValues`

`BC.InputVertexValues` associates a particular boundary condition to each vertex defined in `BC.InputVertex`. This structure is a sorted list of markers. These markers must be odd numbers, corresponding to Dirichlet boundary conditions, or zeros, corresponding to no boundary conditions, because it makes no sense imposing a Neumann condition in a vertex.

Example 2.6 An example of `BC.InputVertexValues` use is the following:

```
Domain.InputVertex = [0 0
  5 0
  5 5
  0 5];
BC.InputVertexValues = [0 1 1 3];
```

This means that on the vertex 1 with coordinates (0,0) there is no boundary condition set. Moreover, on the vertices 2, 3 and 4 of coordinates (5,0), (5,5) and (0,5) a Dirichlet boundary condition has been imposed.

A constant value of the boundary condition can be set in `BC.values` as described in the following. To set more general boundary conditions the use of the values in `BC.values` can be skipped, although some numbers must be set in the corresponding positions of `BC.values`, in any case.

2.2.2 `BC.Boundary`

The structure `BC.Boundary` specifies which boundary condition is set on each side defined in `Domain.Boundary` and contains `Values` as a field. The array `BC.Boundary.Values` contains a sorted list of markers defining which kind of boundary condition applies to a particular domain border. In particular, the marker stored in the position n of `BC.Boundary.Values` applies to the n–th segment defined in `Domain.Boundary`. As for the vertices, the values of the boundary conditions can be set in `BC.values` in the position corresponding to the marker.

An application of `BC.Boundary` is reported in Example 2.8.

2.2.3 `BC.Values`

`BC.Values` is an array which may be used in order to associate a numerical value to a particular maker. Coherently to the conventions previously defined, Dirichlet conditions must be stored in odd position of the array and Neumann conditions must be set in even positions of this array.

If a different number of Dirichlet and Neumann conditions are used, some conventional arbitrary values may be put to fill `BC.Values`.

Example 2.7 An example of `BC.Values` use is the following:

```
BC.Values = [ 3e-1 2e4 0 3e4 -1 0 -1 4e4];
```

In this case two Dirichlet condition (3e-1 in position 1 and 0 in position 3) and four Neumann conditions (2e4 in position 2, 3e4 in position 4, 0 in position 6 and 4e4 in position 8) are set for the markers 1, 2, 3, 4, 6 and 8, respectively.

Example 2.8 Let us consider the squared domain defined in Example 2.1, illustrated in Figure 4, and created with the following assignments:

```
Domain.InputVertex = [ 0 0
  5 0
  5 5
  0 5];
Domain.Boundary.Values = [1 2 3 4];
```

Let us consider the following boundary conditions:

```
BC.Boundary.Values = [3 3 2 4];
BC.Values = [ 3e-1 2e4 0 3e4 -1 0 -1 4e4];
```

Homogeneous Dirichlet boundary conditions are applied to the lower and the right borders (the first and the second defined in `Domain.Boundary.Values`), with marker set to 3, the upper border (the third one defined) with marker 2 has a Neumann boundary condition equal to 2e4 and the left border (the fourth one) with marker 4 has a Neumann boundary condition equal to 3e4.

2.2.4 `BC.Holes` and `BC.Segments`

The use of `BC.Holes` and `BC.Segments` is similar to the use of `BC.Boundary`.

If in the domain there are no holes or segments, both the arrays `Domain.Holes.Hole` and `Domain.Segments.Segment` must be empty arrays:

```
Domain.Holes.Hole = [ ];
Domain.Segments.Segment = [ ];
```

Example 2.10 shows how to use these structures.

Example 2.9 Let us consider the domain described in Example 2.4 and whose mesh is reported in Figure 6. In order to get that mesh with suitable boundary conditions the following options are used:

```
BC.InputVertexValues = [3 0 0 3 5 5 5 0 0 0];
BC.Boundary.Values = [2 1 4 3];
BC.Holes.Hole(1).Values = [5 5 5];
BC.Holes.Hole(2).Values = [6 6 6];
BC.Segments.Segment = [];
BC.Values =  [1e-2 25.6 1-5e-2 47.8 6e-2 74.2];
```

Referring to Figure 6, Dirichlet conditions are assigned on the vertices marked by small circles and Neumann conditions are assigned on the borders marked by broken lines.

Example 2.10 Let us consider the domain described in Example 2.10 and whose mesh is reported in Figure 8. In order to get that mesh with suitable boundary conditions the following options are used:

```
BC.InputVertexValues = [1 1 3 3 5 5 1 3 5 5];
BC.Boundary.Values = [1 4 3 2] ;
BC.Holes.Hole = [];
BC.Segments.Segment(1).Values = [5 5 1];
BC.Segments.Segment(2).Values = [5];
BC.Values= [7e-2 12 3e-2 24 6e-2]
```

Referring to Figure 8, Dirichlet conditions are assigned on the vertices marked by small circles and Neumann conditions are assigned on the borders marked by broken lines.

2.2.5 No boundary conditions

The structure `BC` must be created even if we are not interested in assigning boundary conditions.

If we do not want to set boundary conditions, it will be sufficient to set `BC.Values` equal to an arbitrary value and to create arrays of arbitrary values (for example one) as sub-fields of `BC` structure, with lengths coherent with the dimensions of the corresponding `Domain` structures, as shown in Example 2.11.

Example 2.11 Let us consider the domain introduced in Example 2.4, whose complete definition is given by the following assignments:

```
Domain.InputVertex = [ 0 0
  5 0
  5 5
  0 5
  1 1
  1 2
  2 1
  3 4
  4 3
  4 4];
Domain.Boundary.Values = [1 2 3 4];
Domain.Holes.Hole(1).Values = [5 6 7];
Domain.Holes.Hole(2).Values = [8 9 10];
Domain.Segments.Segment = [];
```

If we are not interested in boundary conditions, we may set:

```
BC.InputVertexValues = ones (size(Domain.InputVertex),1);
BC.Boundary.Values = ones (size(Domain.Boundary.Values),1);
BC.Holes.Hole(1) = ones (size(Domain.Holes.Hole(1)),1);
BC.Holes.Hole(2) = ones (size(Domain.Holes.Hole(2)),1);
BC.Segments.Segment = [];
BC.Values = 1;
```

2.3 The variable `RefiningOptions`

The code BBTR generates first a coarse constrained Delaunay triangulation that fits the domain boundaries and the internal holes and segments. Then BBTR refines the mesh according to parameters chosen by the users, inserting new vertices in the best positions in order to ensure a good quality of the mesh. To set these parameters the user must consider the `RefiningOptions` structure, whose fields are:

- `RefiningOptions.CheckArea`
- `RefiningOptions.AreaValue`
- `RefiningOptions.CheckAngle`
- `RefiningOptions.AngleValue`
- `RefiningOptions.Subregions`

Each of the abovementioned fields must be included in `RefiningOptions` to successfully run BBTR.

2.3.1 `RefiningOptions.CheckArea` and `RefiningOptions.AreaValue`

The first, coarse, mesh generated may be refined as finely as the users want.

If `RefiningOptions.CheckArea` (a character variable) assumes value `’y’` or `’Y’`, BBTR checks the area of each triangle and applies a local refinement if the area of a triangle is larger than `RefiningOption.AreaValue`. If the users are not interested in controlling the size of the triangles, it will be sufficient to impose `RefiningOptions.CheckArea` equal to `’n’` or `’N’`, and to set a dummy value to `RefiningOptions.AreaValue`.

2.3.2 `RefiningOptions.CheckAngle` and `RefiningOptions.AngleValue`

BBTR can also take care about the quality of the mesh produced. If the mesh is used, for example, in a Finite Elements discretization of partial differential equations, the triangles have not to be too thin. This means that their angles have not to be too small or too large.

If `RefiningOptions.CheckAngle` (a character variable) assumes value `’y’` or `’Y’` BBTR checks the angles of every triangle and applies a local refinement if the minimum angle of a triangle is smaller than `RefiningOption.AngleValue`, given in degrees. If the user is not interested in the quality of the triangles, it will be sufficient to impose `RefiningOptions.CheckAngle` equal to `’n’` or `’N’`, and to set a dummy value to the variable `RefiningOptions.AngleValue`.

The parameter `RefiningOptions.AngleValue` must be less then or equal to $30°$ to guarantee that the refinement process converges [8].

A simple example of the use of the structure `RefiningOptions` is described in Example 2.12.

Example 2.12 Let us consider the squared domain of Figure 4, whose complete definition is given by assignments:

```
Domain.InputVertex = [0 0
  5 0
  5 5
  0 5];
Domain.Boundary.Values= [1 2 3 4];
Domain.Holes.Hole = [];
Domain.Segments.Segment = [];

BC.InputVertexValues = [1 1 1 1];
BC.Boundary.Values = [1 1 1 1];
BC.Values = 1;
```

We are not interested in boundary conditions, but we need to set some dummy value to the `BC` structure in order to successfully run BBTR. Let us consider the following refinement options:

```
RefiningOptions.CheckArea = 'Y';
RefiningOptions.CheckAngle = 'n';
RefiningOptions.AreaValue = 0.2;
RefiningOptions.AngleValue = [];
RefiningOptions.Subregions.Subregion = [];
```

The mesh generated by these assignment, plotted in Figure 9, is homogeneously refined imposing that all the triangles in the domain have an area less than 0.2. No bound is imposed on the quality of the triangles.

2.3.3 `RefiningOptions.Subregions`

It is possible to improve the quality of the mesh or decrease the area of the triangles in some portions of the domain called *subregions*. The users can define any number of subregions. The array `Subregion` is the only field of the structure `RefiningOptions.Subregions`. The length of this array is equal to the number of the subregions. Here follow the fields of the variable `RefiningOptions.Subregions.Subregion(n)`:

- `RefiningOptions.Subregions.Subregion(n).CheckArea`
- `RefiningOptions.Subregions.Subregion(n).AreaValue`
- `RefiningOptions.Subregions.Subregion(n).CheckAngle`
- `RefiningOptions.Subregions.Subregion(n).AngleValue`
- `RefiningOptions.Subregions.Subregion(n).Type`
- `RefiningOptions.Subregions.Subregion(n).Descriptors`

The fields named `CheckAngle`, `CheckArea`, `AngleValue` and `AreaValue` of the array `RefiningOptions.Subregions` can be used in the same way as already described in sections 2.3.1 and 2.3.2, with the difference that they apply only to the n–th subregion. Two kind of subregions are already defined in the current versions of BBTR:

- **rectangular subregion**. The sides of the rectangle must be parallel to the axes of the Cartesian coordinate system used in the description of the domain. The dimensions and the position of the rectangle may be defined by the users.
- **elliptic subregion**. The axes of the ellipse must be again parallel to the axes of the Cartesian coordinate system. The dimensions and the position of the ellipse may be defined by the users.

The n–th subregion is identified by the fields `Type` and `Descriptors`. `Type` is a string which describes the shape of the subregion: in BBTR the words 'rectangular' or 'elliptic' characterize the homonymous subregions.

`Descriptors` is an array containing the parameters needed to define the subregions. For the rectangular subregion `Descriptors` assumes the value $[x_{max}\ x_{min}\ y_{max}\ x_{min}]$, the maximum and minimum coordinates of two vertices of the rectangle. For the elliptic subregion `Descriptors` assumes the value $[C_x\ C_y\ a\ b]$, where C_x and C_y are the coordinates of the centre of the ellipse and a and b the horizontal and vertical semi-axes, respectively.

The barycenter of a triangle determines if it belongs or not to a subregion and thus if it must be considered for the refinement or not.

Example 2.13 Referring to Example 2.12, other refinement options may be defined by the following assignments:

```
RefiningOptions.CheckArea = 'Y';
RefiningOptions.CheckAngle = 'n';
RefiningOptions.AreaValue = 0.2;
RefiningOptions.AngleValue = [];

RefiningOptions.Subregions.Subregion(1).Type = 'rectangular';
RefiningOptions.Subregions.Subregion(1).Descriptors = [3 2 4 1];
RefiningOptions.Subregions.Subregion(1).CheckArea = 'y';
RefiningOptions.Subregions.Subregion(1).CheckAngle = 'N';
RefiningOptions.Subregions.Subregion(1).AreaValue = 0.01;
RefiningOptions.Subregions.Subregion(1).AngleValue = [];
```

These assignments first refine the mesh as in Example 2.12, and then refine a rectangular subregion imposing that all the triangles in the subregion have an area smaller than 0.01. The resulting mesh is plotted in Figure 10.

It is also possible for the user to create his own subregions, building a proper function in the subdirectory `/methods`. This function must obey to particular rules. As an example let us describe in details the definition of an elliptic region as given in BBTR. Let the following be a heading of a valid function:

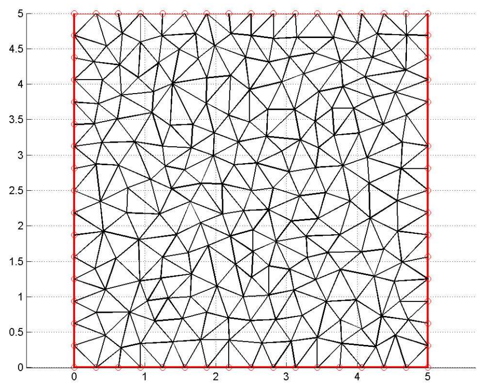


Figure 9: First, coarse, refining of a squared domain.

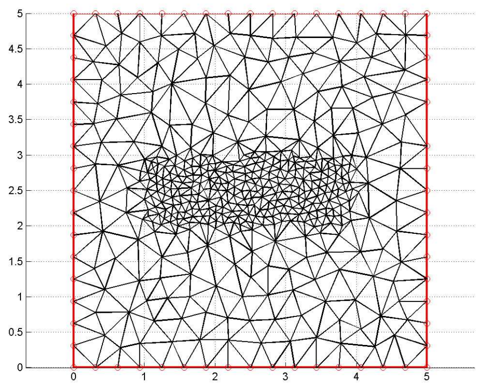


Figure 10: Fine refining in a rectangular subregion.

`function [Flag] = elliptic (Descriptors, P)`
Input and output variables are:

- `elliptic` is the name of the function. To use this function `RefeiningOptions` must assume the following value:

```
RefiningOptions.Subregions.Subregion(n).Type = 'elliptic';
```

- `Descriptors` is the array of parameters necessary in order to characterize the subregion. This array is runtime assigned by user.

- `P` is a 2 components array containing the coordinates of a generic point.

- `Flag` is a boolean value. The function must be written in such a way that this variable becomes true if the point `P` belongs to the subregion and false elsewhere.

Usually, the field `Type` contains the name of the corresponding function to be called and via this function defines the shape of the subregion; dimensions and positions are usually contained in `Descriptors`, but this is only a rule of thumb. Users are invited to create their own functions to refine particular subregions, remembering that, in any case, `Descriptors` must be included in the function, even if not used.

BBTR uses this function to check if the barycenter of each triangle falls inside the subregion. If it does, then this triangle may be refined to satisfy the `RefiningOptions` given for this subregion.

Example 2.14 describes the definition of the elliptic subregion given in the code.

Example 2.14 This is `elliptic`, a sample of refining subregion.

```
function [Flag] = elliptic (Descriptors, P)
%
% Descriptors: specifies the limits of the ellipse, in the form:
%    Descriptors = [xC, yC, a, b]
%       c = [xC yC] is the center of the ellipse
%       a is the semiaxis in x-direction
%       b is the semiaxis in y-direction
%
%  (The ellipse equation used is (xP-xC)^2/a^2 + (yP-yC)^2/b^2 = 1)
%
% P: the generic control point (P = [xP,yP]) that may belong or
%        not belong to the subregion
%
% Flag: the boolean that is true if P belongs to the subregion

if ( (P(1) - Descriptors(1)) / Descriptors(3) )^2 + ...
   ( (P(2) - Descriptors(2)) / Descriptors(4) )^2 <= 1
    Flag = true;
else
    Flag = false;
end

return
```

3 Output

The full description of a triangulation built by BBTR is given through the definition of the different elements of the mesh: triangles, edges of triangles, vertices, and with their reciprocal relationship of neighbourhood. BBTR has a single output variable, `geom`. This variable is a structure that keeps information of every feature of the mesh.

Let n_V be the number of vertices in the mesh, n_E be the number of edges and n_T the number of triangles in the mesh. Fields of `geom` are specified below:

- **geom.elements.coordinates:** `geom.elements.coordinates` is a $[n_V \times 2]$ matrix which contains the list of coordinates of every vertex. So, the n–th row of the matrix contains $[V_x\ V_y]$, the x–coordinate and the y–coordinate of the n–th vertex. The numbering of input vertices in `Domain.InputVertex` may not coincide with their numbering in `geom.elements.coordinates`, but the vertices in `Domain.InputVertex` certainly appear in this field.

  In the fields which follows, reference to a vertex is obtained via its position in this array.

- **geom.elements.triangles:** `geom.elements.triangles` is a $[n_T \times 3]$ matrix which contains the link between every triangle and its vertices. So, the n–th row of the matrix contains $[V_1\ V_2\ V_3]$, the integers referring to the three vertices defining the n–th triangle. Vertices are stored anticlockwise.

- **geom.elements.borders:** `geom.elements.borders` is a $[n_E \times 4]$ matrix that contains the link between each edge of the triangulation and its vertices. Furthermore it contains, for each edge, information about the triangles which surround it. So, the n–th row of the matrix contains $[V_1\ V_2\ T_1\ T_2]$: V_1 and V_2 are the two vertices which belong to the n–th edge, T_1 and T_2 are the two triangles which surround the n–th edge. If an edge is adjacent to a region out of domain, the value in `geom.elements.borders` in the corresponding position is conventionally -1.

- **geom.elements.neighbourhood:** this data structure is a $[n_T \times 9]$ matrix which contains the link between every triangle and the three adjacent triangles and edges.

  The n–th row of the matrix contains $[T_1\ T_2\ T_3\ E_1\ E_2\ E_3\ R_1\ R_2\ R_3]$: T_1, T_2 and T_3 are the three triangles which are adjacent to the n–th triangle, E_1 is the edge between the n–th triangle and T_1, E_2 is the edge between the n–th triangle and T_2, E_3 is the edge between the n–th triangle and T_3.

  R_i may generically assume values from 1 to 3. Let us consider again the n–th row of the matrix, the element R_i is the column-index in which we find a reference to the triangle n in the row T_i, $\forall i = 1, 2, 3$.

  If an edge is adjacent to a region which has not been meshed, the reference in `geom.elements.neighbourhood` is conventionally -1. In this case also the corresponding values in columns between 4 and 9 assume value -1.

  An example of this data structure may be found in Figure 11.

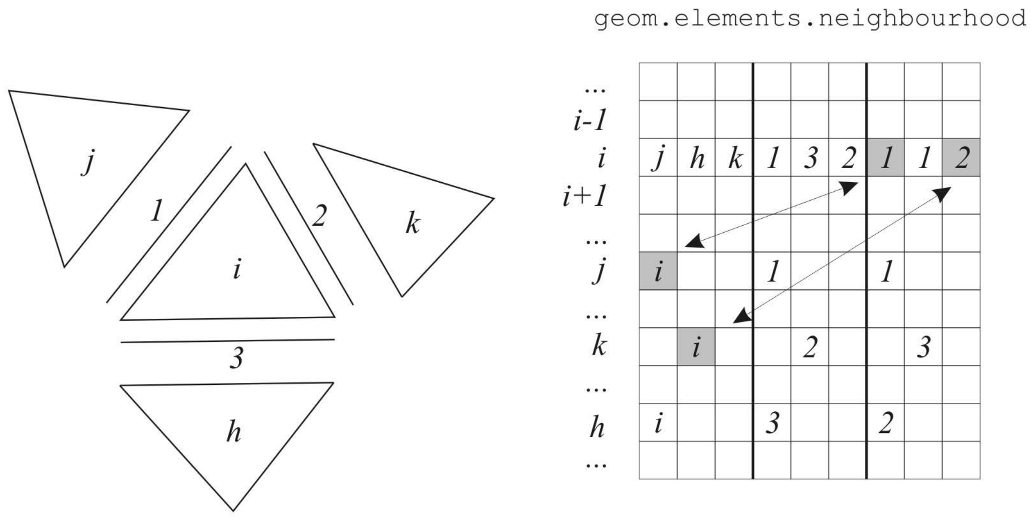


Figure 11: Application of data structure **geom.elements.neighbourhood**.

- **geom.elements.vertexesneighbourhood:** this is an array of n_V elements. Its i–th element contains a number and two arrays:

  $$\texttt{geom.elements.vertexesneighbourhood(i).n} = N,$$
  $$\texttt{geom.elements.vertexesneighbourhood(i).B} = [B_1, B_2, \dots, B_N],$$
  $$\texttt{geom.elements.vertexesneighbourhood(i).V} = [V_1, V_2, \dots, V_N].$$

  The number N is the number of vertices and edges directly connected with the i–th vertex. The two arrays `B` (edges) and `V` (vertices) contain these edges and vertices (via a reference to `geom.elements.borders` and `geom.elements.coordinates`, respectively).

  The edge `geom.elements.vertexesneighbourhood(i).B(j)` is the edge connecting the vertex i with the vertex `geom.elements.vertexesneighbourhood(i).V(j)`.

- **geom.nelements.nTriangles:** `geom.nelements.nTriangles` contains n_T, the number of triangles in the mesh.

- **geom.nelements.nBorders:** `geom.nelements.nBorders` contains n_E, the number of edges in the mesh.

- **geom.nelements.nVertexes:** `geom.nelements.nVertexes` contains n_V, the number of vertices in the mesh.

- **geom.pivot.nodelist:** `geom.pivot.nodelist` is an array whose length is n_V. This array contains information about boundary conditions set on each vertex: if the n–th component of `geom.pivot.nodelist` is equal to zero no Dirichlet boundary condition is set for the n–th vertex, otherwise the n–th component assumes an odd value that is a marker referring to `BC.Values` (see 2.2.3).

- **geom.pivot.Di:** `geom.pivot.Di` is a $[n_{Di} \times 2]$ matrix which contains the list of vertices on which a Dirichlet boundary condition has been assigned. n_{Di} is the number of these vertices present in the mesh. So, the n–th row of the matrix contains $[i_{Di}\ BC_v]$, where i_{Di} is a vertex in `geom.elements.vertexes` and BC_v is a marker referring to a Dirichlet boundary condition, as already explained in section 2.2.

- **geom.pivot.Ne:** `geom.pivot.Ne` is a $[n_{Ne} \times 2]$ matrix which contains the list of edges on which a Neumann boundary condition has been assigned. The value n_{Ne} is the number of these edges with Neumann boundary conditions present in the mesh. So, the n–th row of the matrix contains $[i_{Ne}\ BC_v]$, where i_{Ne} is an edge in `geom.elements.borders` and BC_v is a marker referring to a Neumann boundary condition, as already explained in section 2.2.

- **geom.support.TInfo:** `geom.support.TInfo` is an array whose length is n_T and which contains information about every triangle. The n–th element of this data structure is composed by:

  – `geom.support.TInfo(n).Area`: the area of the n–th triangle;

  - `geom.support.TInfo(n).B`: a measure of the quality of n–th triangle. Denoting by α the smallest angle of the n–th triangle, B is given by:

$$B = \frac{1}{2 \sin \alpha}.$$

  - `geom.support.TInfo(n).Circumcenter`: a 2 components array composed by $[C_x \; C_y]$ which contains the coordinates of the circumcenter (the center of the triangle's circumcircle) of n–th triangle.
  - `geom.support.TInfo(n).Circumradius`: the square of the circumradius (the radius of the circumcircle) of the n–th triangle.

- **geom.support.BInfo:** `geom.support.BInfo` is a $[n_E \times 3]$ matrix which contains information about every edge. So, the n–th row of the matrix contains $[I_1 \; I_2 \; I_3]$:
  - I_1 specifies if the n–th edge belongs to the domain border (value = 1), to a hole border (value = 2), to a segment (value = 3) or if it is a edge inner to the mesh (value = 0).
  - I_2 specifies if the n–th edge is a border of the triangulation (value = 1) or not (value = 0). (This column will be removed in a future release for saving memory, being this information contained in I_1).
  - I_3 specifies if the n–th edge has some boundary condition assigned (value = marker previously defined) or not (value = 0).

- **geom.support.BCircle:** `geom.support.BCircle` is an array whose length is n_E. This array contains information about the edge *diametral circle*, i.e. the circle having the edge as diameter. The n–th element of `geom.support.BCircle` is composed by:
  - `geom.support.BCircle(n).Center`: this structure is a 2 components array composed by $[C_x \; C_y]$, containing the coordinates of the centre of the diametral circle of the n–th edge.
  - `geom.support.TInfo(n).r2`: the squared radius of the diametral circle of the n–th edge.

- **geom.support.EB:** `geom.support.EB` contains a structure necessary to BBTR to refine the mesh.

- **geom.input.Domain:** `geom.input.Domain` contains the `Domain` input structure that has been used to create the mesh.

- **geom.input.BC:** `geom.input.BC` contains the `BC` input structure that has been used to create the mesh.

4 Parameters

There are many parameters that are used by BBTR to generate the mesh. These parameters are stored in a file called `global_parameter`. Users may modify these parameters in order to solve special problems or to speed up the mesh generation.

There are two groups of parameters: a first group will be most probably modified by the users because the modifications may improve the running of mesh generator. The second group of parameters may be modified, but should not led to any improvements when running the BBTR code. Changing these parameters may require a deep knowledge of the algorithms implemented.

4.1 Parameters that will most probably be modified by the users

- **tolerance:** (default value `1e-12`) running BBTR, two numbers are considered equal if their difference is less than `tolerance`.
- **SafeMeshing:** if `SafeMeshing` (default value `true`) is activated, the program becomes slower, but can triangulate also domains that otherwise would let BBTR crash. The parameter `SafeMeshing` must be activated if there are boundary borders very close to each other.
- **RecursiveEncroachingLimit:** (default value `4`) the algorithms used by BBTR could enter in a loop, infinitely refining triangles near a small angle of the boundary. If this infinite procedure begins, BBTR quits the loop after a number of iteration equal to `RecursiveEncroachingLimit`. If the domain has small input angles, the higher this value is, the safer and slower BBTR becomes.
- **RefiningFolderName:** `RefiningFolderName` (default value `'methods'`) is the name of the directory that contains the `functions` defining the refining subregions (see section 2.3.3).

4.2 Parameters that will less probably be modified by the users

- **SplittingBordersLimits:** (default value `10`) the algorithms used by BBTR to refine a mesh divide the domain in many sudomains (called Encroachable subregions). Each of these subdomains can contain no more than `SplittingBordersLimits` edges on the domain borders. Exceeding this limit forces BBTR to create new subdomains.
- **XBoxEnlarge, YBoxEnlarge:** (default values `0.5`) to generate a first, coarse mesh, BBTR creates a rectangle which contains every input vertex. Let be x_{min}, y_{min}, x_{max} and y_{max} the minimum and maximum coordinates of the input vertices, moreover, let be $\Delta_x = x_{max} - x_{min}$ and $\Delta_y = y_{max} - y_{min}$. The rectancle created by BBTR containing every input vertex is: $[x_{min} - \texttt{XBoxEnlarge}\Delta_x, x_{max} + \texttt{XBoxEnlarge}\Delta_x] \times [y_{min} - \texttt{YBoxEnlarge}\Delta_y, y_{max} + \texttt{YBoxEnlarge}\Delta_y]$.
- **GenerationUntilResearch:** (default value `10`) when BBTR must identify which triangle contains a point, it first tries to search it in a triangle which is supposed to contain the point. It the search fails, BBTR checks the three neighbouring triangles.

In case of failure, BBTR checks the neghbouring triangles of the neighbouring triangles. If a triangle containig the point is not found before, this research is repeated up to `GenerationUntilResearch` times.

- **FirstExtimateTmultiplier, FirstExtimateBmultiplier, FirstExtimateVmultiplier:** `FirstExtimateTmultiplier` (default value `8`), `FirstExtimateBmultiplier` (default value `8`) and `FirstExtimateVmultiplier` (default value `6`) are multipliers used to make a first estimate of the number of elements of the mesh.
- **FirstMaxRefiningRoof :** `FirstMaxRefiningRoof` (default value `5`) is a multiplier used to give a first estimate of the maximum number of triangles that BBTR tries to refine at every step.
- **RecursionStackLimit:** `RecursionStackLimit` (default value `1000`) is the maximum recursion stack limit that is set in `Matlab`®.

5 Appendix

In this appendix some examples are given for a detailed description of the inputs and outputs of BBTR.

5.1 Full Example: data structures

This example is useful to understand the data structure exploited and produced by BBTR. Let us consider the simple mesh generated with the following script:

```
Domain.InputVertex = [ 0 0
    1 0
    1 1
    0 1];
Domain.Boundary.Values = [1 2 3 4] ;
Domain.Holes.Hole = [];
Domain.Segments.Segment = [];

BC.Boundary.Values = [1 2 4 3];
BC.InputVertexValues = [5 7 0 1];
BC.Values = [0.0 20.0 0.12 0.0 0.654 -1 0.33]
BC.Holes.Hole = [];
BC.Segments.Segment = [];

RefiningOptions.CheckArea = 'Y';
RefiningOptions.CheckAngle = 'N';
RefiningOptions.AreaValue = 0.2;
RefiningOptions.AngleValue = [];
RefiningOptions.Subregions = [];

[geom] = bbtr30(Domain,BC,RefiningOptions);
draw_grid(geom,1);
```

The instruction

```
draw_grid(geom,1)
```

plots the mesh. In Figure 12 a plot of the mesh is reported and every element has been associated to the label used by BBTR to identify it. These correspondences may be verified typing the names of the objects.

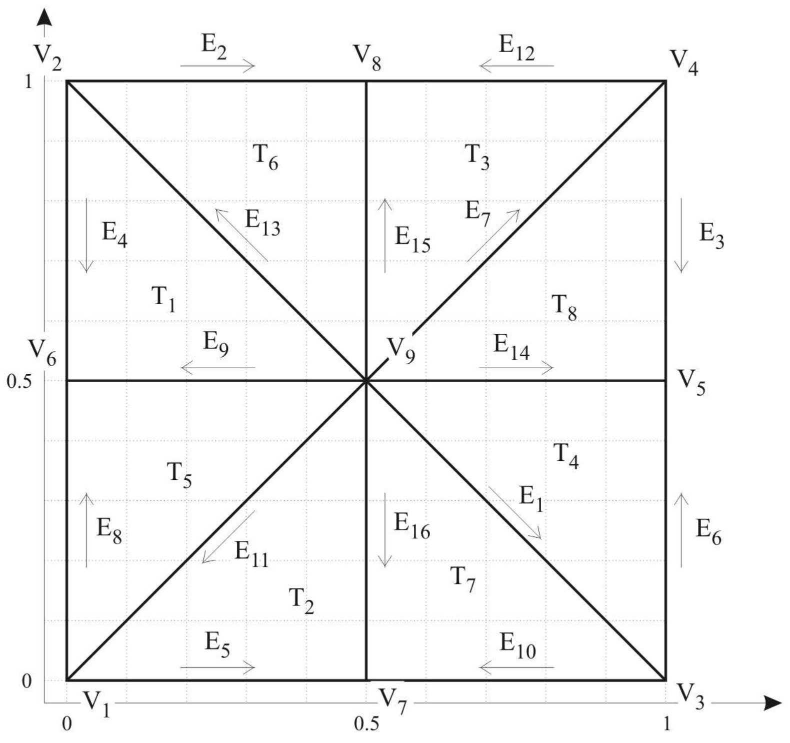


Figure 12: Example 5.1.V_i are the labels of the vertices, E_i are the labels of the edges and T_i are the labels of triangles. The versus of the arrow related to each edge denotes which vertex belonging to an edge has been stored in second column of `geom.elements.borders`.

For example, typing `geom` we get the first level of fields memorized in that variables:

```
>> geom
geom =
     elements: [1x1 struct]
    nelements: [1x1 struct]
        pivot: [1x1 struct]
      support: [1x1 struct]
        input: [1x1 struct]
```

Typing `geom.nelements` we get the total number of elements in the mesh:

```
>> geom.nelements
ans =
    nTriangles: 8
      nBorders: 16
     nVertexes: 9
```

Table 1: Example 5.1. Data structure `geom.elements.coordinates`.

$\mathbf{V_i}$	$\mathbf{V_x}$	$\mathbf{V_y}$
V_1	0	0
V_2	0	1.0000
V_3	1.0000	0
V_4	1.0000	1.0000
V_5	1.0000	0.5000
V_6	0	0.5000
V_7	0.5000	0
V_8	0.5000	1.0000
V_9	0.5000	0.5000

Table 2: Example 5.1. Data structure `geom.elements.triangles`.

$\mathbf{T_i}$	$\mathbf{V_1}$	$\mathbf{V_2}$	$\mathbf{V_3}$
T_1	9	2	6
T_2	9	1	7
T_3	9	4	8
T_4	9	3	5
T_5	9	6	1
T_6	9	8	2
T_7	9	7	3
T_8	9	5	4

Typing `geom.elements` we get the field that contains the structures which describe the mesh:

```
>> geom.elements
ans =
                triangles: [22x3 double]
              coordinates: [24x2 double]
                  borders: [41x4 double]
            neighbourhood: [22x9 double]
    vertexesneighbourhood: [1x20 struct]
```

Typing `geom.elements.coordinates` we get the data structure which defines the coordinates of the vertices. Its first lines are:

```
>> geom.elements.coordinates
ans =
         0         0
         0    1.0000
```

The complete data structure is reported in Table 1

Typing `geom.elements.triangles` we get the data structure which defines the triangles. Its first lines are:

```
>> geom.elements.triangles
ans =
     9     2     6
     9     1     7
```

The complete data structure is reported in Table 2

Typing `geom.elements.borders` we get the data structure which defines the edges and its neighbouring triangles. Its first lines are:

```
>> geom.elements.borders
ans =
     9     3     7     4
     2     8    -1     6
```

Table 3: Example 5.1. Data structure `geom.elements.borders`. The reported V_i are the two vertices belonging to each edge, and the reported T_i are the triangles adjacent to each edge.

$\mathbf{E_i}$	$\mathbf{V_1}$	$\mathbf{V_2}$	$\mathbf{T_1}$	$\mathbf{T_2}$
E_1	9	3	7	4
E_2	2	8	-1	6
E_3	4	5	-1	8
E_4	2	6	-1	1
E_5	1	7	-1	2
E_6	3	5	-1	4
E_7	9	4	8	3
E_8	1	6	-1	5
E_9	9	6	1	5
E_{10}	3	7	-1	7
E_{11}	9	1	5	2
E_{12}	4	8	-1	3
E_{13}	9	2	6	1
E_{14}	9	5	4	8
E_{15}	9	8	6	3
E_{16}	9	7	2	7

The complete data structure is reported in Table 3

Typing `geom.elements.neighbourhood` we get the data structure which defines the edges and its neighbouring triangles. Its first lines are:

```
>> geom.elements.neighbourhood
ans =
     5    -1     6     9     4    13     1    -1     3
     7    -1     5    16     5    11     1    -1     3
```

The complete data structure is reported in Table 4

Typing `geom.elements.vertexesneighbourhood` we get the data structure which defines the edges and its neighbouring triangles:

```
>> geom.elements.vertexesneighbourhood
ans =
1x20 struct array with fields:
    n
    B
    V
```

To understand this data structure we may explore it, vertex by vertex, typing for example:

```
>> geom.elements.vertexesneighbourhood(1)
ans =
    n: 3
    B: [5 8 11]
    V: [7 6 9]
```

Table 4: Example 5.1. Data structure `geom.elements.neighbourhood`, which contains information of the neighbourhood of each triangle. The T_i and E_i reported in first six columns are the three triangles and respectively edges adjacent to each triangle. The reported R_i are explained in section 3.

$\mathbf{T_i}$	$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathbf{T_3}$	$\mathbf{E_1}$	$\mathbf{E_2}$	$\mathbf{E_3}$	$\mathbf{R_1}$	$\mathbf{R_2}$	$\mathbf{R_3}$
T_1	5	-1	6	9	4	13	1	-1	3
T_2	7	-1	5	16	5	11	1	-1	3
T_3	6	-1	8	15	12	7	1	-1	3
T_4	8	-1	7	14	6	1	1	-1	3
T_5	1	-1	2	9	8	11	1	-1	3
T_6	3	-1	1	15	2	13	1	-1	3
T_7	2	-1	4	16	10	1	1	-1	3
T_8	4	-1	3	14	3	7	1	-1	3

or

```
>> geom.elements.vertexesneighbourhood(9)
ans =
    n: 8
    B: [15 9 16 14 13 11 1 7]
    V: [8 6 7 5 2 1 3 4]
```

The complete data structure is reported in Table 5

Typing `geom.pivot` we get the data structure which specify the boundary conditions applied to the vertices and edges in the mesh:

```
>> geom.pivot
ans =
    nodelist: [9x1 double]
          Di: [5x2 double]
          Ne: [4x2 double]
```

Typing `geom.pivot.nodelist` we get the data structure which defines the edges and its neighbouring triangles. Its first lines are:

```
>> geom.pivot.nodelist
ans =
     5
     1
```

The complete data structure is reported in Table 6

Typing `geom.pivot.pivot.Di` we get the data structure which defines which vertices have a Dirichlet boundary condition applied on them, and which condition it is. Its first lines are:

```
>> geom.pivot.Di
ans =
     2     1
     6     3
```

Table 5: Example 5.1. Data structure `geom.elements.vertexesneighbourhood`. The E_j reported in second column represent all the edges which contain a given vertex and the V_j represent the other vertex belonging to that edges.

$\mathbf{V_i}$	$\mathbf{E_j}$, $\mathbf{V_j}$								
V_1	E_j :	5	8	11					
	V_j :	7	6	9					
V_2	E_j :	4	2	13					
	V_j :	6	8	9					
V_3	E_j :	10	6	1					
	V_j :	7	5	9					
V_4	E_j :	3	12	7					
	V_j :	5	8	9					
V_5	E_j :	3	6	14					
	V_j :	4	3	9					
V_6	E_j :	4	8	9					
	V_j :	2	1	9					
V_7	E_j :	5	10	16					
	V_j :	1	3	9					
V_8	E_j :	2	12	15					
	V_j :	2	4	9					
V_9	E_j :	15	9	16	14	13	11	1	7
	V_j :	8	6	7	5	2	1	3	4

The first line means that the second vertex has a boundary condition marked by 1 and the first value in `BC.Values` can be used to set this boundary condition. The complete data structure is reported in Table 7

Typing `geom.pivot.Ne` we get the data structure which defines which edges have a Neumann boundary condition applied on them, and which condition it is. Its first lines are:

```
>> geom.pivot.Ne
ans =
     2     4
     3     2
```

The first line means that the second edge has a boundary condition marked by 4 and the fourth value in `BC.Values` can be used to set this boundary condition. The complete data structure is reported in Table 8

5.2 Full Example: squared domain with holes and inner constraints

This is a sample of triangulation of a complex domain, with two holes, different segments and different BC assigned to different elements. This is the script necessary to generate the mesh:

Table 6: Example 5.1. Data structure `geom.pivot.nodelist`. The value Marker is a reference to the input variable `BC.Value`.

$\mathbf{V_i}$	**Marker**	**Meaning**
V_1	5	on this vertex the boundary condition marked by 5 is applied
V_2	1	on this vertex the boundary condition marked by 1 is applied
V_3	7	on this vertex the boundary condition marked by 7 is applied
V_4	0	no boundary condition is applied on this vertex
V_5	0	no boundary condition is applied on this vertex
V_6	3	on this vertex the boundary condition marked by 3 is applied
V_7	1	on this vertex the boundary condition marked by 1 is applied
V_8	0	no boundary condition is applied on this vertex
V_9	0	no boundary condition is applied on this vertex

Table 7: Example 5.1. Data structure `geom.pivot.n.Di`.

$\mathbf{V_i}$	**Marker**
V_2	1
V_6	3
V_1	5
V_7	1
V_3	7

Table 8: Example 5.1. Data structure `geom.pivot.n.Ne`.

$\mathbf{E_i}$	**Marker**
E_2	4
E_3	2
E_6	2
E_{12}	4

```
Domain.InputVertex = [ 0 0
    5 0
    5 5
    0 5
    1 1
    4 1.25
    1 1.5
    4 4
    4 3.5
    1 3.75];
Domain.Boundary.Values = 1:4 ;
Domain.Holes.Hole(1).Values = 5:7;
Domain.Holes.Hole(2).Values = 8:10;
Domain.Segments.Segment(1).Values = [4 10];
Domain.Segments.Segment(2).Values = [10 6];
Domain.Segments.Segment(3).Values = [6 2];

BC.Values = [2 4 6 8 10 12];
BC.Boundary.Values = [1 2 1 2];
BC.Holes.Hole(1).Values = [3 3 3];
BC.Holes.Hole(2).Values = [4 4 4];
BC.Segments.Segment(1).Values = [5];
BC.Segments.Segment(2).Values = [6];
BC.Segments.Segment(3).Values = [7];
BC.InputVertexValues = [1 1 0 0 3 3 3 0 0 0];


RefiningOptions.CheckArea = 'Y';
RefiningOptions.CheckAngle = 'N';
RefiningOptions.AreaValue = 0.1;
RefiningOptions.AngleValue = [ ];
RefiningOptions.Subregions = [ ];


[geom] = bbtr30(Domain,BC,RefiningOptions);
draw_grid (geom,1);
```

The result is reported in Figure 13. Dirichlet conditions are assigned on the vertices marked by small circles, Neumann conditions are assigned on the borders marked by broken lines.

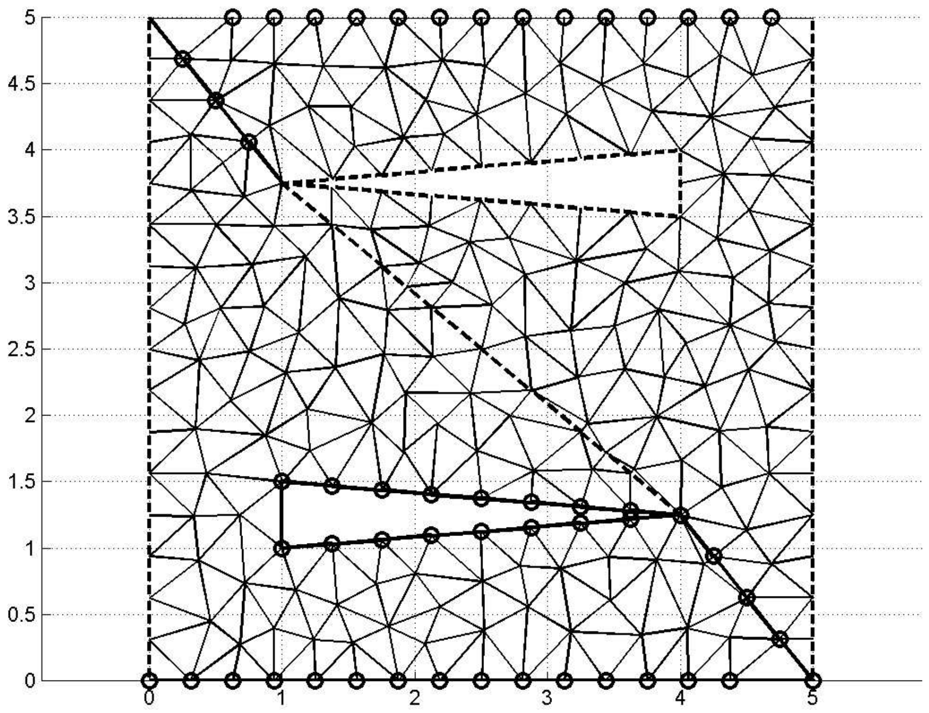


Figure 13: Domain described in Full Example 5.2.

5.3 Full Example: Refining a squared mesh with subregions

In this example we show how to get a triangulation with many output vertices on a single border and fewer in the other borders. Two refining subregions are set, an elliptic one inside the domain and a degenerate rectangular one at the left side.

```
for i = 0: 99
    Domain.InputVertex(i+1,:) = [i/10 0];
end
Domain.InputVertex(101:106,:) = [ 10 0
    10 10
    6.2 10
    6 10
    5.8 10
    0 10];
Domain.Boundary.Values = 1:106;
Domain.Holes.Hole = [ ];
Domain.Segments.Segment = [ ];

BC.Values = 1;
BC.Boundary.Values = ones(1,106);
BC.Holes.Hole = [ ];
BC.Segments.Segment = [ ];
BC.InputVertexValues = ones(1,106);

RefiningOptions.CheckArea = 'N';
RefiningOptions.CheckAngle = 'Y';
RefiningOptions.AreaValue = 0.1;
RefiningOptions.AngleValue = 5;

RefiningOptions.Subregions.Subregion(1).Type = 'rectangular';
RefiningOptions.Subregions.Subregion(1).Descriptors = [2 0 10 0];
RefiningOptions.Subregions.Subregion(1).CheckArea = 'N';
RefiningOptions.Subregions.Subregion(1).CheckAngle = 'Y';
RefiningOptions.Subregions.Subregion(1).AreaValue = [ ];
RefiningOptions.Subregions.Subregion(1).AngleValue = 30;


RefiningOptions.Subregions.Subregion(2).Type = 'elliptic';
RefiningOptions.Subregions.Subregion(2).Descriptors = [6 6 3 2];
RefiningOptions.Subregions.Subregion(2).CheckArea = 'Y';
RefiningOptions.Subregions.Subregion(2).CheckAngle = 'N';
RefiningOptions.Subregions.Subregion(2).AreaValue = 0.01;
RefiningOptions.Subregions.Subregion(2).AngleValue = 30;


[geom] = bbtr30(Domain,BC,RefiningOptions);
draw_grid (geom,1);
```

and the result is reported in Figure 14.

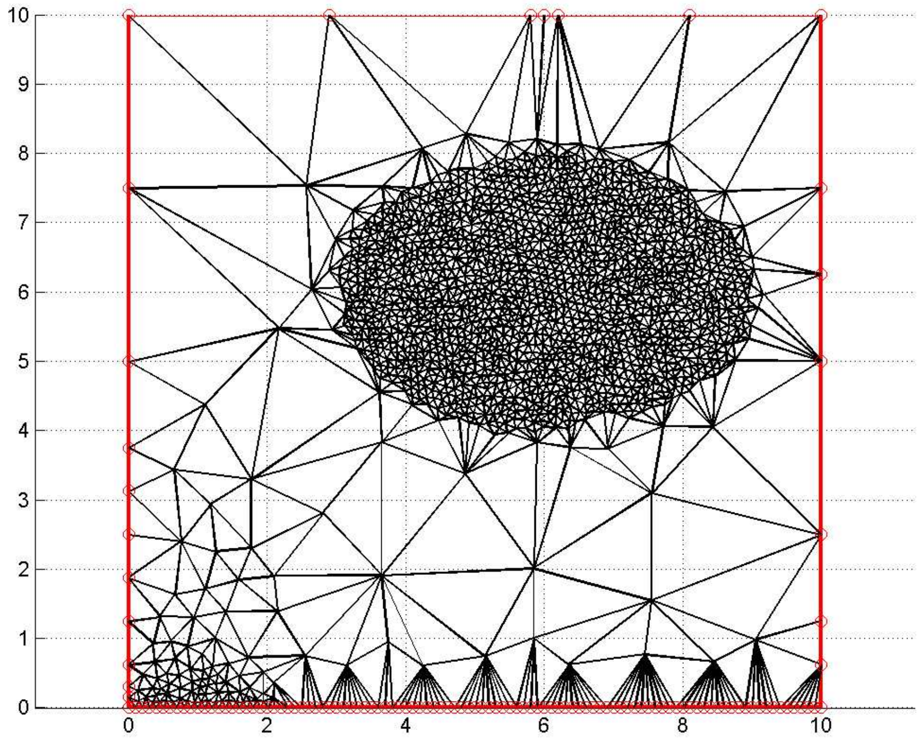


Figure 14: Domain described in Full Example 5.3.

References

[1] Adrian Bowyer. Computing dirichlet tessellations. *Computer Journal*, 24(2):162–166, 1981.

[2] D. Braess. *Finite Elements – Theory, fast solvers and applications in solid mechanics.* Cambridge University Press, 2007.

[3] L. Paul Chew. Guaranteed-quality triangular meshes. *Technical Report TR-89-983, Department of Computer Science, Cornell University*, 1989.

[4] L. Paul Chew. Guaranteed-quality triangular mesh generation for curved surfaces. In Association for Computing Machinery, editor, *Proceedings of the Ninth Annual Symposium of Computational Geometry (San Diego, California)*, pages 274–280, may 1993.

[5] Boris N. Delaunay. Sur la sphére vide. *Izvestia Akademia Nauk, SSSR, VII Seria*, 7:793–800, 1934.

[6] Claes Johnson. *Numerical solution of partial differential equations by the finite element method.* Cambridge University Press, Studenlitteratur, 1987.

[7] Jonathan R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editor, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, may 1996. From the First ACM Workshop on Applied Computational Geometry.

[8] Jonathan R. Shewchuk. *Delaunay Refinement Mesh Generation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, may 1997. Available as Technical Report CMU-CS-97-137.

[9] Jonathan R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1–3):21–74, may 2002.

[10] V. T. Rajan. Optimality of the delaunay triangulations in $\mathbb{R}^d$. In *Proceedings of the Seventh Annual Symposium on computational Geometry*, pages 357–363, 1991.

[11] Jim Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In Association for Computing Machinery, editor, *Proceedings of the Fourth Annual Symposium of Discrete Algorithms*, january 1993.

[12] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585, may 1995.

[13] David F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *Computer Journal*, 24(2):167–171, 1981.