COMPUTING OBJECTS

CM2 TriaMesh® Iso/Aniso
CM2 QuadMesh® Iso/Aniso

Version 5.6

tutorials

Revision February 2025.

https://www.computing-objects.com

Forewords

This manual is a tutorial for the 2-D mesh generators of the **CM2 MeshTools®** SDK:

- The isotropic meshers **CM2 TriaMesh® Iso** and **CM2 QuadMesh® Iso,**
- The anisotropic meshers **CM2 TriaMesh® Aniso** and **CM2 QuadMesh® Aniso**.

All these mesh generators are constrained unstructured meshers: the boundary mesh (contour mesh) as well as the internal hard edges and hard points (embedded) are kept unmodified in the final mesh.

Based on a fast and robust hybrid "Advancing-Front and Delaunay" algorithm, they generate high quality elements with smooth grading sizes according to the length of the boundary edges or to the user-specified sizes. The speed is near independent of the number of the elements to be generated.

Setting switches can be used to adapt the meshers to the various needs of the user concerning mesh generation, refinement and optimization (they can also be used as optimizer-only of some already existing meshes).

The quad meshers can generate all-quad meshes (the default) or mixed quad-dominant meshes.

Many data concerning the mesh are available upon exit: shape and size qualities histograms, matrix of the neighbors, number of sub-domains, area...

Like many other meshers of the library, **CM2 TriaMesh® Iso/Aniso** and **CM2 QuadMesh® Iso/Aniso** are multi-threaded (you can select in the settings the maximum number of threads the generator can use).
The generated meshes are reproducible (same mesh with same input data and same mesh with any number of threads).

Data are exchanged with the CM2 mesh generators through vector and matrix objects (no file). Beginners should start by reading the **CM2 Math1® - overview** manual to get first views on these mathematical containers.

For a complete description of the data and settings structures used with these meshers please refer to the **CM2 TriaMesh & CM2 QuadMesh - reference manual**.

The source code of the **CM2 MeshTools®** SDK (full library) has been registered with the APP under Inter Deposit number IDDN.FR.001.260002.00.R.P.1998.000.20700 (22/06/1998) and IDDN.FR.001.480030.006.S.P.2001.000.20700 (23/05/2019) is regularly deposited since then.

The source code specific to **CM2 TriaMesh® Iso/Aniso**, together with this manual, has been registered with the APP under Inter Deposit number IDDN.FR.001.440021.000.R.P.2008.000.20700 (31/10/2008) and is regularly deposited since then.

The source code specific to **CM2 QuadMesh® Iso/Aniso**, together with this manual, has been registered with the APP under Inter Deposit number IDDN.FR.001.440020.000.R.P.2008.000.20700 (31/10/2008) and is regularly deposited since then.

Table of contents

Forewords 2

1. Getting started – a simple square 5

Some declarations 6

Authorization of the library 6

Contour mesh 7

2. Square with an internal line 14

3. Square with internal hole 17

4. Quadratic elements & high-order nodes 19

5. Square with grading mesh size 21

6. Square with an internal hard node 23

7. Multiple meshes 25

8. Shared boundaries 28

9. Background mesh 32

10. Anisotropic meshes 38

11. 3-D surface meshes (aniso meshers only) 43

Before meshing a 2-D domain, the first step is to generate a 1-D mesh of the external contour. This chapter mostly details cases where the boundary mesh is obtained using some simple **CM2 MeshTools** functions. One example (Section 7) illustrates the case where the boundary mesh has been generated by other means and is simply read from a file.

Each example starts with including the file `stdafx.h` (can be a pre-compiled header) giving access to the classes and the functions of the library (API).

The general namespace `cm2` has nested namespaces such as `cm2::vecscal`, `cm2::vecvec`, `cm2::meshtools` or `cm2::triamesh_iso`. The user can add a `using namespace cm2` directive in this `stdafx.h` file. Keeping namespaces in the user's source code can however be useful to improve the legibility and to avoid name conflicts. In the rest of this document we assume such a `using namespace cm2` directive.

File `stdafx.h`[1]:

```
// CM2 MESHTOOLS
#include "meshtools.h"        // General purpose mesh routines
#include "meshtools1d.h"      // To generate 1D meshes
#include "triamesh_iso.h"     // CM2 TriaMesh Iso
#include "quadmesh_iso.h"     // CM2 QuadMesh Iso
#include "triamesh_aniso.h"   // CM2 TriaMesh Aniso (Section 10 only)
#include "quadmesh_aniso.h"   // CM2 QuadMesh  Aniso (Section 10 only)

using namespace cm2;          // Main cm2 namespace can now be omitted.
```

Required libraries[2]:

- `cm2math1`
- `cm2misc`
- `cm2meshtools`
- `cm2meshtools1d`
- `cm2meshtools2d`
- `cm2triamesh_iso`
- `cm2quadmesh_iso`
- `cm2triamesh_aniso` (Section 10 only)
- `cm2quadmesh_aniso` (Section 10 only)

[1] If neither meshtools nor **CM2 QuadMesh Iso** nor the aniso meshers is used, the file `stdafx.h` can reduce to:`#include "triamesh_iso.h"` and link only with `cm2math1`, `cm2misc`, `cm2meshtools`, `cm2meshtools2d` and `cm2triamesh_iso`.

[2] The lib names end with `_($platform)_($ver)`. For instance `cm2math1_x64_56.dll`. On Windows, file extensions for the libraries are `.lib` and `.dll`. On Linux/Unix/macOS platforms, file extensions are usually .a (static archive), .so or .dylib (dynamic lib).

1. Getting started – a simple square

This first example is a regular mesh of a square. The four boundary segments are equally discretized with 10 elements.

```
#include "stdafx.h"
#include <iostream>

// Simple optional display handler.
static void display_hdl (void*, unsigned, const char* msg) { std::cout << msg; }


int main()
{
   const double       L(10.);
   const unsigned     N(10);
   const DoubleVec2   P0(0., 0.), P1(L, 0.), P2(L, L), P3(0., L);
   DoubleMat          pos;
   UIntVec            indices;
   UIntMat            connectB;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);
   meshtools1d::mesh_straight(pos, 0, 1, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, N, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);

   // THE 2D MESH.
   triamesh_iso::mesher               the_mesher;
   triamesh_iso::mesher::data_type    data(pos, connectB);
   the_mesher.run(data);

   // SOME OUTPUT INFO (OPTIONAL).
   data.print_info(&display_hdl);

   // VISUALISATION (OPTIONAL).
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

The resulting mesh is shown Figure 1.

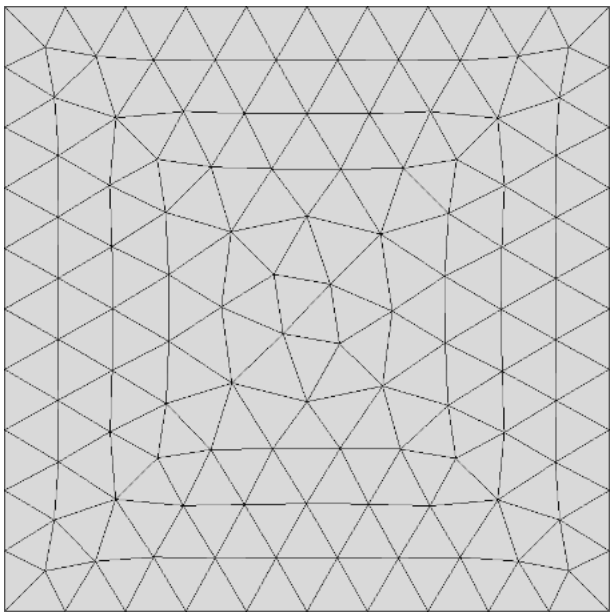


Figure 1 – Triangle mesh of a square.

Let us explain this program line by line.

Some declarations

Matrix `pos` is a `DoubleMat` (variable-sized matrix of doubles)[3] and the connectivity matrix `connectB` is a `UIntMat`. `connectB(i, j)` shall store the ith local node of the jth element. This integer refers to the column number in matrix `pos` where the coordinates of this node can be found4.

`indices` is a temporary vector.

Authorization of the library

The library `triamesh_iso` (resp. `quadmesh_iso`) is protected and need to be unlocked with a call to `triamesh_iso::registration` (resp. `quadmesh_iso::registration`). Two strings must be provided for each library: the name of your company or organization that has acquired the license and a secret code[5]. Note that both strings are case sensitive and the registration call must be made each time the library is loaded into memory and before the first run of the mesher.

```
triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");
```

[3] See manual **CM2 Math1 - overview**.

[4] Recall that array indices are zero based (from 0 to N-1).

[5] Contact `license@computing-objects.com` for any licensing inquiry.

Contour mesh

This is usually the heaviest part of the work for the user. In this example, we only use routines from the **CM2 MeshTools** SDK, but the user is free to generate the contour mesh with any other tool or even to read it form a file[6]. Anyway, the 2-D meshers need this contour mesh as a couple of matrices: the matrix `pos` containing the points' coordinates and the connectivity matrix `connectB` of the boundary edges.

First, the corners of the square are created as four pair of coordinates in the `pos` matrix:

```
pos.push_back(P0);
pos.push_back(P1);
pos.push_back(P2);
pos.push_back(P3);
```

The `push_back` function appends a new column at the end of a matrix. The size of the column must match the current number of rows of the matrix. If the matrix is empty, the first vector sets this number of rows.

After these four push-backs, the dimensions of the `pos` matrix are 2 x 4.

```
meshtools1d::mesh_straight(pos, 0, 1, N, indices); indices.pop_back();
meshtools1d::mesh_straight(pos, 1, 2, N, indices); indices.pop_back();
meshtools1d::mesh_straight(pos, 2, 3, N, indices); indices.pop_back();
meshtools1d::mesh_straight(pos, 3, 0, N, indices);
```

Now that the four corners are present, we can create the points in between and the associated edges:

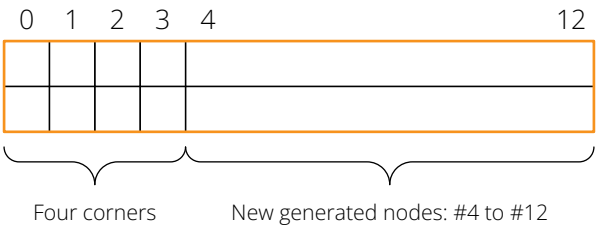
The `mesh_straight` routine of the `meshtools1d` library generates N - 1 new points equally spaced into new appended columns in the `pos` matrix:

```
meshtools1d::mesh_straight
   (DoubleMat& pos, unsigned i0, unsigned i1, unsigned N, UIntVec& indices);
```

The index of each point, i.e. the column in matrix `pos`, is also appended to the vector `indices`.
With i0 = 0 and i1 = 1, this vector contains upon exit of this function:

```
[0 4 5 6 7 8 9 10 11 12 1]
```

And the matrix `pos` is now of size 2 x 13:



[6] See Section 5, "Square with Grading Mesh Size".

The last value in the `indices` vector, i.e. value 1, must be suppressed to avoid having it twice:

```
indices.pop_back();
```

The second call to `mesh_straight` with i0 = 1 and i1 = 2 sets the `indices` vector to:

```
[0 4 5 6 7 8 9 10 11 12 1 13 14 15 16 17 18 19 20 21 2]
```

After the four line meshes, the matrix `pos` is of size 2 x 40 and the indices vector has 41 values - the last index equals to the first, here zero, to close the contour[7].

The vector of `indices` is used to create the connectivity matrix (2-node edges) of the boundary mesh:

```
meshtools1d::indices_to_connectE2(indices, connectB);
```

The `connectB` matrix has now dimensions 2 x 40:

```
2x40 [0 4 5 6 7 ... 39
      4 5 6 7 8 ...  0]
```

Now that we have done the boundary mesh, all we have to do is to call the 2-D mesher. This done by creating a data structure holding this 1-D mesh and make the mesher run on it:

```
triamesh_iso::mesher::data_type    data(pos, connectB);
the_mesher.run(data);
```

This constructs the `data` structure with shallow-copies of the matrices `pos` and of `connectB` into `data.pos` and `data.connectB`. Upon exit, the matrix `data.pos` is bigger and contains all the new points generated inside the square by the 2-D mesher. These new points are appended to the original matrix. The initial 40 points are left untouched in the first 40 columns.

The connectivity of the final mesh is stored in the matrix `data.connectM`, each column storing the indices of the nodes for an element[8]. `connectM(i, j)` is the i^{th} local node of the j^{th} element.

[7] The same result could have been achieved with:
```
UIntVec    hard_nodes(5);
hard_nodes[0] = 0;
hard_nodes[1] = 1;
hard_nodes[2] = 2;
hard_nodes[3] = 3;
hard_nodes[4] = 0;
meshtools1d::mesh_straight(pos, hard_nodes, 4*N, indices);
```
This variant of `mesh_straight` meshes a polyline going through some constrained points (`hard_nodes`).

[8] The elements are always oriented counter-clock wise (normal up with the right-hand thumb rule).

Printed information about the generated mesh and a MEDIT[9] output file are obtained with:

```
data.print_info(&display_hdl);
meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);
```

Here is the output given by `data.print_info(&display_hdl)`:

```
****************************************
*    CM2 TriaMesh(R) Iso (5.6.0.0)     *
****************************************
Hard nodes      : 40/40
Hard edges      : 40/40
Nodes           : 136
Triangles       : 230
Subdomains      : 1
Area            : 1.600000E+01
Qmin            : 8.348321E-01
Front time      : 0.00 s.
Refine time      : 0.00 s.
Optim time      : 0.00 s.
Total time      : 0.00 s. (114994.63 t/s.)

************ HISTOGRAM QS ************
Total number of bins     :           11
Total number of counts   :          230
Number of larger values  :            0
Number of smaller values :            0
V max                    : 1.000000E+00
V mean                   : 9.466349E-01
V min                    : 8.348321E-01

Bin number       -- Bin boundaries --          Hits

   10            0.90            1.00           177
    9            0.80            0.90            53
    8            0.70            0.80             0
    7            0.60            0.70             0
    6            0.50            0.60             0
    5            0.40            0.50             0
    4            0.30            0.40             0
    3            0.20            0.30             0
    2            0.10            0.20             0
    1            0.01            0.10             0
    0            0.00            0.01             0
```

Figure 2 – Output info for the square example.

The generated mesh has 136 nodes and 230 triangles for an initial contour mesh of 40 nodes and 40 edges (hard nodes and hard edges). The times spent in the three steps of the meshing process (front, refine, optimize) are given in seconds[10]. The front mesh is the triangulation mesh with only the boundary hard nodes. In the second step new nodes are generated inside the domain to get elements with good shape and size. Finally, the last step is for geometrical and topological optimizations to improve the quality of the elements.

[9] MEDIT is a free visualization program. Other output formats are: Ensight, FEMAP (neutral), Nastran, STL (ASCII or binary), VTK and Wavefront OBJ.

[10] Here the times are below 0.01 s. All runs are done with x64 CM2 libs (VS 2010 MD build) on Windows® 8.1 x64 with Intel® Xeon® E3-1270 V2 3.5 GHz (turbo boost disabled). The typical speed with default settings on such a platform ranges from 5 000 quads / s. (CM2 QuadMesh Aniso) to more than 100 000 triangles / s. (CM2 TriaMesh Iso).

The formula used to compute the shape quality of a triangle writes:

$$Q_s = 4\sqrt{3}\,\frac{S}{L_{\max}\,P}$$

with:

S	Area of the triangle.
L_{max}	Length of the longest edge of the triangle.
P	Perimeter of the triangle.

This quality measure ranges from 0 for a degenerated triangle, to 1 for an equilateral triangle.
On the square example, the worst shape quality is 0.83 and the average is 0.94.

The size quality is also an important parameter to take into account. The size quality of an edge is a measure based upon its actual length and the target size values defined at its two vertices. A size quality of 1 indicates that the edge has the optimal length. A too short edge has a size quality lesser than 1 - but always positive -, and a too long edge has a size quality greater than 1. For instance an edge with a quality of 2 is twice as long as it should be (and should have been split).

The formula used to compute the length quality of an edge AB writes:

$$Q_h^{AB} = L_{AB}\,\frac{\ln\left(\frac{h_A}{h_B}\right)}{h_A - h_B}$$

with:

L_{AB}	Actual length of edge AB.
h_A	Target size at node A (expected edge length at A).
h_B	Target size at node B (expected edge length at B).

Let's introduce also at this point the h-shock measure of an edge:

$$hs^{AB} = \min\left(\frac{h_A}{h_B}, \frac{h_B}{h_A}\right)^{\frac{1}{Q_h^{AB}}} - 1$$

These two measures are dimensionless and positive.

When $h_A = h_B$ the h-shock is null and the length quality simply writes $Q_h^{AB} = \frac{L_{AB}}{h_A}$.

When $Q_h^{AB} = 1$ edge AB is considered having optimal length with respect to its target mesh sizes h_A and h_B.

To optimize a mesh we need to improve simultaneously both the shape quality of the elements and the size quality of the edges. On top of these, the h-shock should be kept lower than a maximum threshold to ensure smooth gradations and all the prescribed entities (hard edges and hard nodes) must be honored. All this makes the job of the optimizer difficult and heuristics must be used.

The histogram of the size qualities can be computed either inside the mesher by raising the flag `settings.compute_Qh_flag`[11] before meshing or with *a posteriori* call to the auxiliary function `cm2::meshtools::edge_qualities`.

[11] See **CM2 TriaMesh Iso/Aniso & CM2 QuadMesh Iso/Aniso - reference manual** for full description of the meshers options.

On the square example, the size qualities are well centered on the value 1 with a small variance:

```
************ HISTOGRAM QH ************
Total number of bins     :           20
Total number of counts   :          365
Number of larger values  :            0
Number of smaller values :            0
V max                    : 1.362359E+00
V mean                   : 1.004864E+00
V min                    : 7.529856E-01

Bin number       -- Bin boundaries --          Hits

   19           10.00            +INF            0
   18            5.00           10.00            0
   17            3.33            5.00            0
   16            2.50            3.33            0
   15            2.00            2.50            0
   14            1.67            2.00            0
   13            1.43            1.67            0
   12            1.25            1.43            1
   11            1.11            1.25           37
   10            1.00            1.11          156
    9            0.90            1.00          140
    8            0.80            0.90           23
    7            0.70            0.80            8
    6            0.60            0.70            0
    5            0.50            0.60            0
    4            0.40            0.50            0
    3            0.30            0.40            0
    2            0.20            0.30            0
    1            0.10            0.20            0
    0            0.00            0.10            0
```

Figure 3 – Histogram of the size-qualities of all the edges in the square example.

To mesh with quadrangles all is needed is to change the class of the mesher:

```
#include "stdafx.h"
#include <iostream>

// Simple optional display handler.
static void display_hdl (void*, unsigned, const char* msg) { std::cout << msg; }


int main()
{
   const double       L(10.);
   const unsigned     N(10);
   const DoubleVec2   P0(0., 0.), P1(L, 0.), P2(L, L), P3(0., L);
   DoubleMat          pos;
   UIntVec            indices;
   UIntMat            connectB;

   // UNLOCK THE DLL.
   quadmesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);
   meshtools1d::mesh_straight(pos, 0, 1, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, N, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);

   // THE 2D MESH.
   quadmesh_iso::mesher               the_mesher;
   quadmesh_iso::mesher::data_type    data(pos, connectB);
   the_mesher.run(data);
   data.print_info(&display_hdl);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACEQ4);

   return 0;
}   // main
```

In this particular case, the generated mesh is a perfectly structured quad mesh with all qualities equal to one[12].

[12] We could get the same structured Q4 mesh with `cm2::meshtools2d::mesh_struct_Q4`.

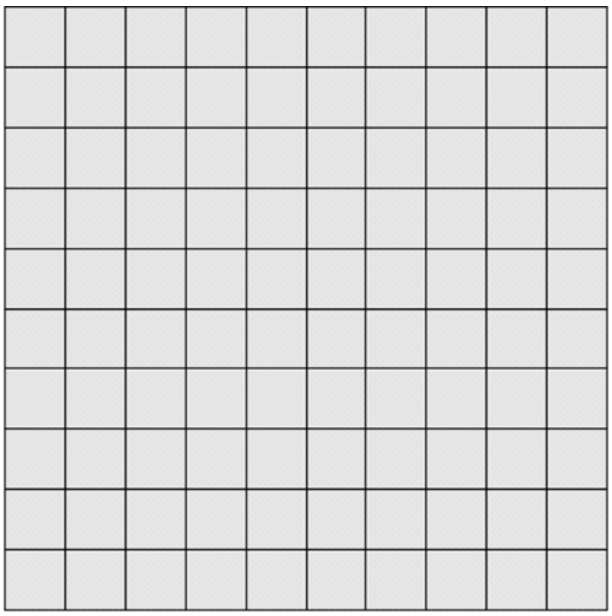


Figure 4 – Square meshed with quads.

For a plane quadrangle, we use the following measure of the shape quality:

$$Q_s = 8\sqrt{2}\,\frac{S_{\min}}{L_{\max}\,P}$$

with:

$S_{\min}$	Minimum area of the four triangles.
$L_{\max}$	Max length of the four sides and the two diagonals.
P	Perimeter of the quad.

This measure gives the maximal value 1 only for a square.

The size quality is given by the same measure as for the triangles (because it is based on edges only).

2. Square with an internal line

Starting from the previous example, we add a circle inside the square. Here is the program for a triangle mesh:

```
#include "stdafx.h"

int main()
{
   const double       L(10.), R(3.);
   const unsigned     N1(10), N2(20);
   const DoubleVec2   P0(0., 0.), P1(L, 0.), P2(L, L), P3(0., L);
   const DoubleVec2   P4(L/2 + R, L/2);
   DoubleMat          pos;
   UIntVec            indices;
   UIntMat            connectB;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);
   pos.push_back(P4);
   meshtools1d::mesh_straight(pos, 0, 1, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, N1, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);
   indices.clear();
   meshtools1d::extrude_rotate(pos, 4, DoubleVec2(L/2., L/2.), 2.*M_PI, N2, indices);
   indices.back() = indices.front();
   meshtools1d::indices_to_connectE2(indices, connectB);

   // THE 2D MESH.
   triamesh_iso::mesher               the_mesher;
   triamesh_iso::mesher::data_type    data(pos, connectB);
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

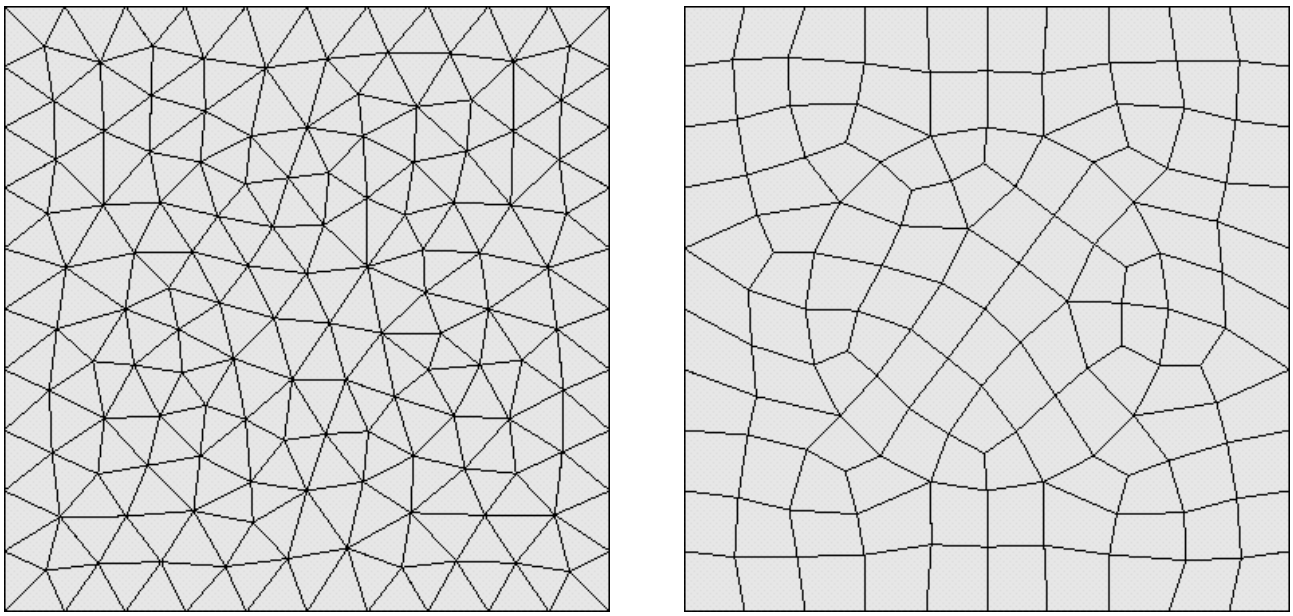


Figure 5 – Square with internal line (T3 and all-Q4).

The mesh of the circle is generated with the function `cm2::meshtools1d::extrude_rotate`. The rotation is defined by a center, here by the point `DoubleVec2(L/2, L/2)` and a rotation scalar around *Oz*, here 2 π.

The circular line is discretized using 20 elements[13] starting from point #4. Here, the last generated point - point #24 - is coincident with the first one - point #4. In order to close topologically the circle, it is important to replace value 24 with value 4 in the indices vector so that the first and the last point are identical not only coincident[17]:

```
indices.back() = indices.front();
```

As for the external contour, these indices are converted into edges with the `indices_to_connectE2` function and appended to the `connectB` matrix.

Again, to mesh with quads, we simply replace the `triamesh_iso` namespace with `quadmesh_iso`. Moreover, if we accept some triangles we can get a better mesh.

[13] Remember that **CM2 QuadMesh** needs an even number of edges on each line (external and internal lines) in all-quad mode.

[14] Note that the coordinates at column 24 in the `pos` matrix will remain unused.

Here with **CM2 QuadMesh® Iso** in quad-dominant mode:

```
#include "stdafx.h"

int main()
{
   const double       L(10), R(3.);
   const unsigned     N1(10), N2(20);
   DoubleMat          pos;
   const DoubleVec2   P0(0., 0.), P1(L, 0.), P2(L, L), P3(0., L);
   const DoubleVec2   P4(L/2+R, L/2);
   UIntVec            indices;
   UIntMat            connectB;

   // UNLOCK THE DLL.
   quadmesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);
   pos.push_back(P4);
   meshtools1d::mesh_straight(pos, 0, 1, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, N1, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);
   indices.clear();
   meshtools1d::extrude_rotate(pos, 4, DoubleVec2(L/2., L/2.), 2.*M_PI, N2, indices);
   indices.back() = indices.front();
   meshtools1d::indices_to_connectE2(indices, connectB);

   // THE 2D MESH.
   quadmesh_iso::mesher               the_mesher;
   quadmesh_iso::mesher::data_type    data(pos, connectB);
   the_mesher.settings.all_quad_flag = false;
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACE_MIX);

   return 0;
}   // main
```

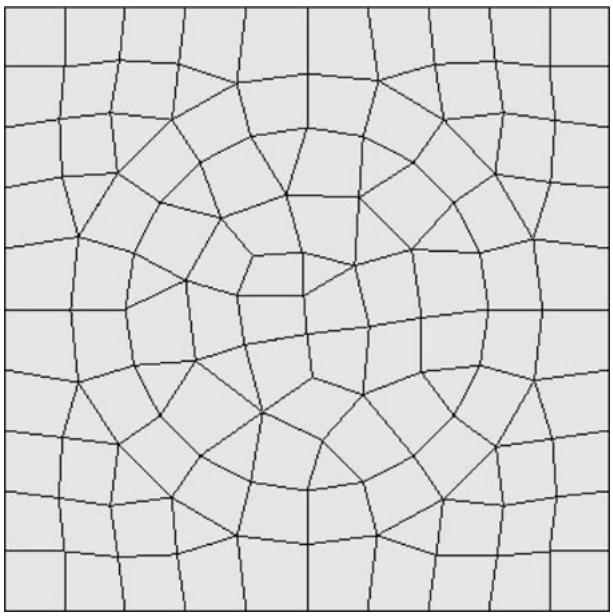


Figure 6 – Square with internal line (quad-dominant mode).

3. Square with internal hole

A hole is an internal closed contour with edges oriented the opposite way from the external contour. Note that this implies that all edges of the external contour should be oriented in a uniform way (either clockwise or counter-clockwise[15]). Based on the previous example, we simply change the sign of the rotation vector to revert the orientation of the internal edges and thus to remove the disk from the domain:

```
meshtools1d::extrude_rotate(pos, 4, DoubleVec3(L/2., L/2.), -2.*M_PI, N2, indices);
```

And the resulting meshes:

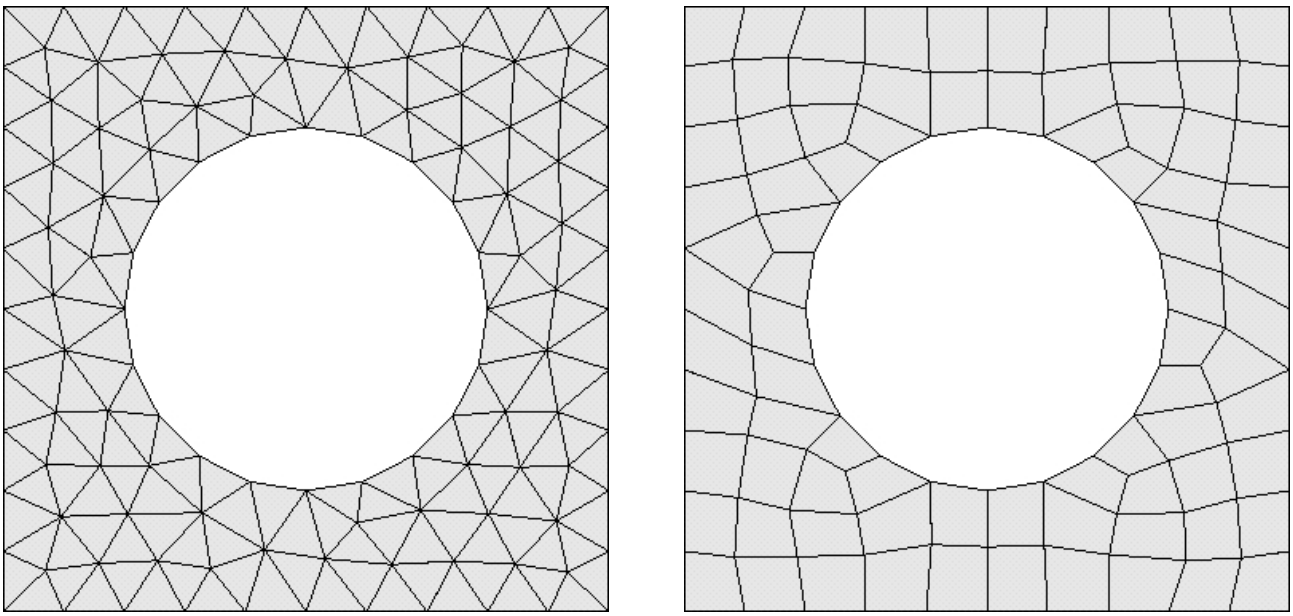


Figure 7 – Square with a circular hole (T3 and all-Q4).

[15] Without any closed internal hard line, the orientation of the external contour is irrelevant.

One can nest alternatively positive and negative rotations:

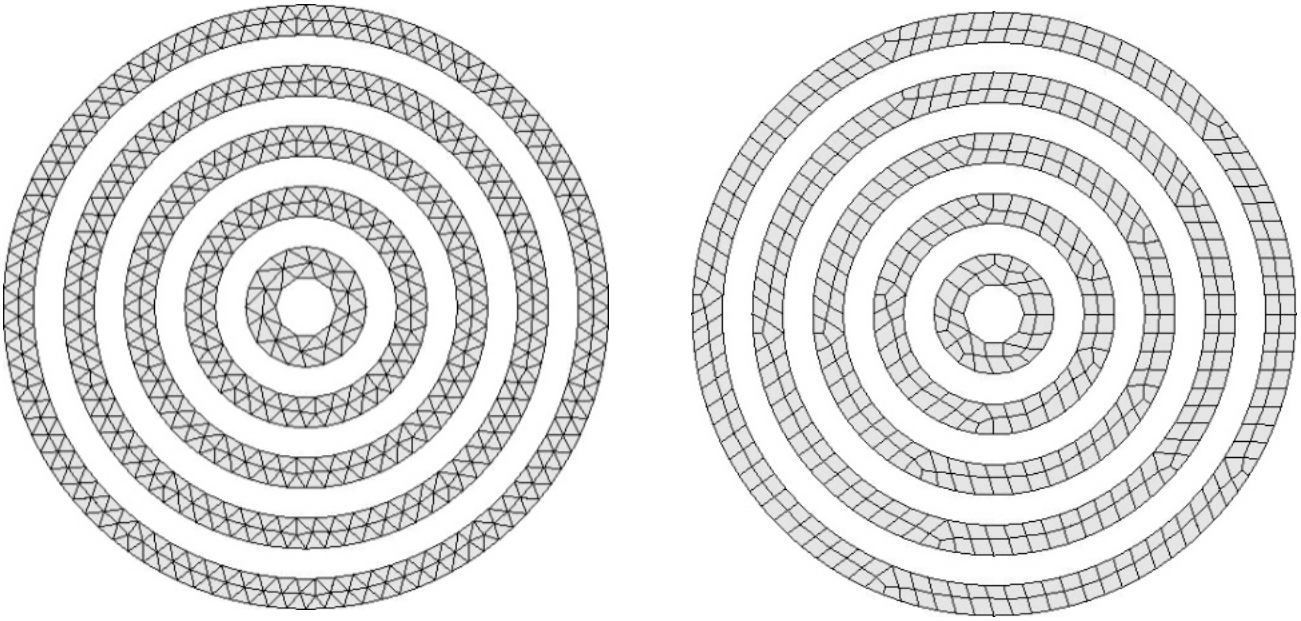


Figure 8 – Concentric circles with alternate orientation (T3 and all-Q4).

4. Quadratic elements & high-order nodes

Let us derive the example 3 “Square with internal hole” to generate quadratic T6 elements. In addition, we would like also the edges along the circular hole to be curved.

For that matter we use the conversion functions `cm2::meshtools1d::convert_into_quadratic, cm2::meshtools1d::convert_into_linear` and `cm2::meshtools2d::convert_into_quadratic`.

```
#include "stdafx.h"

int main()
{
   const double       L(10.), R(3.);
   const unsigned     N1(10), N2(20);
   const DoubleVec2   P0(0.), P1(L, 0.), P2(L, L), P3(0., L);
   const DoubleVec2   P4(L/2 + R, L/2);
   DoubleMat          pos;
   UIntVec            indices;
   UIntMat            connectB, connectE2;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);
   pos.push_back(P4);
   meshtools1d::mesh_straight(pos, 0, 1, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, N1, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, N1, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);
   meshtools1d::convert_into_quadratic(pos, connectB);
   indices.clear();
   meshtools1d::extrude_rotate(pos, 4, DoubleVec2(L/2, L/2), -2 * M_PI, 2 * N2, indices);
   indices.back() = indices.front();
   meshtools1d::indices_to_connectE3(indices, connectB);
   connectE2.copy(connectB);
   meshtools1d::convert_into_linear(connectE2);

   // THE 2D MESH.
   triamesh_iso::mesher               the_mesher;
   triamesh_iso::mesher::data_type    data(pos, connectE2);   // Linear edges here.
   the_mesher.run(data);
   meshtools2d::convert_into_quadratic(data.pos, data.connectM, connectB);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET6);

   return 0;
}   // main
```

To keep the boundary edges straight (not curved) we convert T3 into T6[16] without the `connectB` parameter (this will create new high-order nodes, different from those in `connectB`):

```
meshtools2d::convert_into_quadratic(data.pos, data.connectM);
```

[16] A more general function is available to convert into any type of high-order elements:. `cm2::meshtools2d::convert_into_high_order.` Refer to the HTML reference manual for detailed information.

This would create and push new (high-order) nodes into matrix `data.pos` and convert the T3 connectivity matrix `data.connectM` into a T6 connectivity matrix gaining three new rows. The new nodes being linearly interpolated between the initial vertices, all the edges would remain straight.

This is not what we want here (we want the edges along the circle to be curved). Moreover, the connectivity matrix of the boundary edges (or some of them) is usually required later to setup boundary conditions (Dirichlet, Neumann...)

So, for the outer square we use `meshtools2d::convert_to_quadratic` to convert linear edges into 3-node edges creating and pushing new high-order nodes into matrix `pos` (created at the centers of the edges):

```
meshtools1d::convert_into_quadratic(pos, connectB);
```

The connectivity matrix `connectB` gains one new row (the new high-order nodes). The first two rows are unchanged. A matrix view to these first two rows is equivalent to the initial connectivity matrix (linear edges[17]).

linear nodes {	0	1	2	3	4	6	8	10
	1	2	3	4	5	7	9	11
	12	13	14	15	16	17	18	19

Figure 9 – Example of connectivity matrix for quadratic edges and view to linear edges.

For the inner circle, we create quadratic edges directly with `indice_to_connectE3` to transform a sequence of node indices into a quadratic connectivity matrix, as illustrated by the second set of edges (along the circle). This is possible because we have generated along the circle twice as much nodes as in the previous example (`extrude_rotate` with 2 N2).

Now we have quadratic edges all along the boundaries. Straight edges along the square, curved edges along the circle.

However, the mesh generators accept only linear edges upon entry and give only linear face elements upon exit. Hence, we have to feed the mesher with the linear view of the `connectB` edge connectivity matrix (called `connectE2` in the example). For that matter, we duplicate `connectB` and transform the copy back into linear edges with `convert_into_linear`.

After the surface meshing, to transform the linear T3 faces into T6 faces and to reuse the quadratic nodes along the boundaries (and then keeping curved edges along the circle), we pass the quadratic edge connectivity matrix `connectB` created before:

```
meshtools2d::convert_into_quadratic(data.pos, data.connectM, connectB);
```

This forces `convert_into_quadratic` to use the high-order nodes of `connectB` wherever edges match.

[17] Note that the mid-side node is local node #2 after the linear nodes #0 and #1 though geometrically placed between them.

5. Square with grading mesh size

There are two ways to get a graded size in a mesh. First, you can simply generate edges with varying size along the boundary (or interior lines). The mesher computes a default size value on each hard node[18], interpolates these values inside the domain and generates elements accordingly.

To illustrate this, let us use again the example of the square. Instead of meshing regularly the four segments of the contour we specify different mesh sizes on each four vertices:

```
#include "stdafx.h"

int main()
{
   const double       L(10.);
   const unsigned     N(10);
   const DoubleVec2   P0(0.), P1(L, 0.), P2(L, L), P3(0., L);
   DoubleMat          pos;
   UIntVec            indices, hard_nodes(5);
   DoubleVec          sizes(5);
   UIntMat            connectB;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);
   hard_nodes[0] = 0;  sizes[0] = 0.1*L/N;
   hard_nodes[1] = 1;  sizes[1] = 2.0*L/N;
   hard_nodes[2] = 2;  sizes[2] = 0.1*L/N;
   hard_nodes[3] = 3;  sizes[3] = 2.0*L/N;
   hard_nodes[4] = 0;  sizes[4] = 0.1*L/N;
   meshtools1d::mesh_straight(pos, hard_nodes, sizes, true, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);

   // THE 2D MESH.
   triamesh_iso::mesher               the_mesher;
   triamesh_iso::mesher::data_type    data(pos, connectB);
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

This variant of the `cm2::meshtools1d::mesh_straight` function uses a vector of hard nodes (a polygonal line) and a vector of target size values, one value for each hard node. The contour mesh is generated to fit best the target values on the four corners.

These target sizes are not used by the 2-D mesher. Only the resulting edge lengths of the contour will be used to compute the 2-D size map.

[18] By averaging the lengths of the adjacent edges to each hard node.

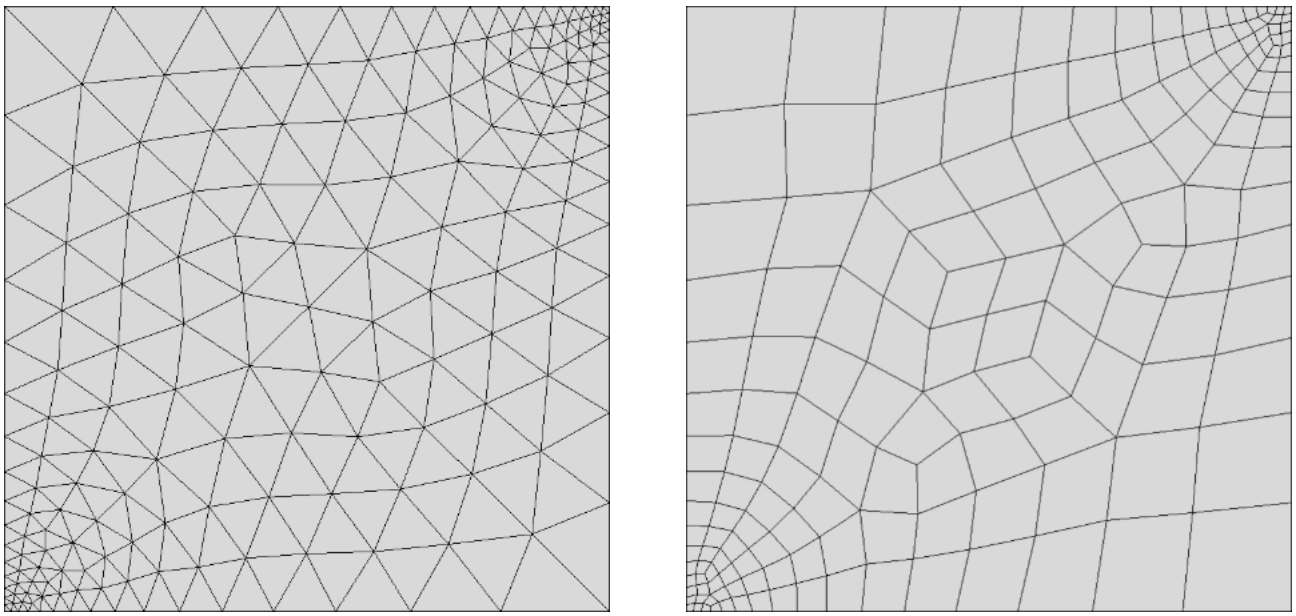


Figure 10 – Meshes with grading size (T3 and all-Q4).

The second way to get grading sizes is to specify in the data of the 2-D mesher the target size values on some hard nodes. This is explained in the next section.

6. Square with an internal hard node

So far, we have seen only three fields of the structure used to exchange data with the mesher:

- The `pos` matrix for the coordinates of the points.
- The `connectB` matrix for the connectivity of the hard edges.
- The `connectM` matrix for the connectivity of the 2-D mesh.

In this example we add an isolated hard node at the center of the square and specify a target size on it. This will be done using the two new fields `isolated_nodes` and `metrics`:

```
#include "stdafx.h"

int main()
{
   const double       L(10.);
   const unsigned     N(10);
   const DoubleVec2   P0(0., 0.), P1(L, 0.), P2(L, L), P3(0., L), P4(L/2., L/2.);
   DoubleMat          pos;
   UIntVec            indices;
   UIntMat            connectB;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);
   pos.push_back(P4);
   meshtools1d::mesh_straight(pos, 0, 1, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, N, indices); indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, N, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);

   // THE 2D MESH.
   triamesh_iso::mesher               the_mesher;
   triamesh_iso::mesher::data_type    data(pos, connectB);
   data.isolated_nodes.push_back(4);
   data.metrics.resize(5, 0.0);
   data.metrics[4] = 0.1*L/N;
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

We have created a new point at the centre with coordinates placed in column #4 of matrix `pos`. Its index (4) is pushed into the vector `data.isolated_nodes`. This new field stores the isolated nodes that must be honored in the final mesh.

The vector `data.metrics` stores the user-specified target sizes. If the value for a node is zero -or negative or not present- a default value will be used instead[19].

In our example the vector is resized to 5 with all values set to zero except for point #4 where we ask for a 10 times finer mesh around it.

[19] For an isolated node, the default computed size is based on the size value of the nearest nodes.

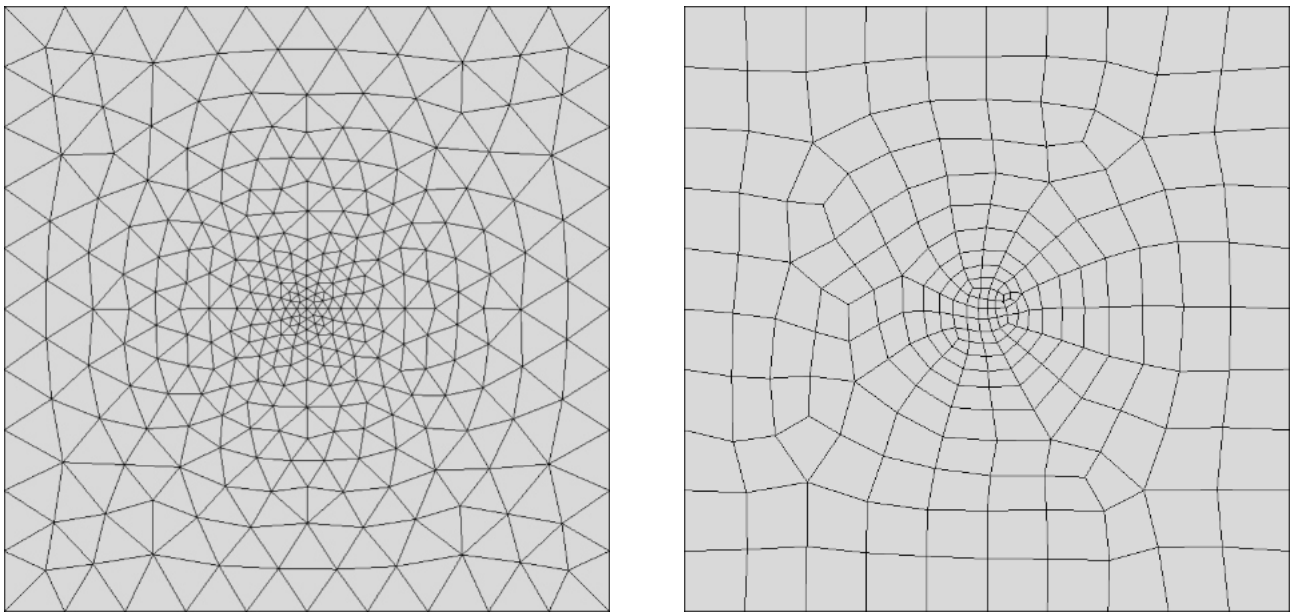


Figure 11 – Mesh concentration near a hard node (T3 and all-Q4).

7. Multiple meshes

The matrix `connectB` can contain internal lines. It can also contain several external disconnected contours (to mesh several disconnected domains simultaneously). Some care must be taken however in the orientation of these contours. For multiple domains, the edges of all external contours must be oriented the same way, for instance counter-clockwise (the so-called positive orientation). In addition, these contours must not intersect each other.

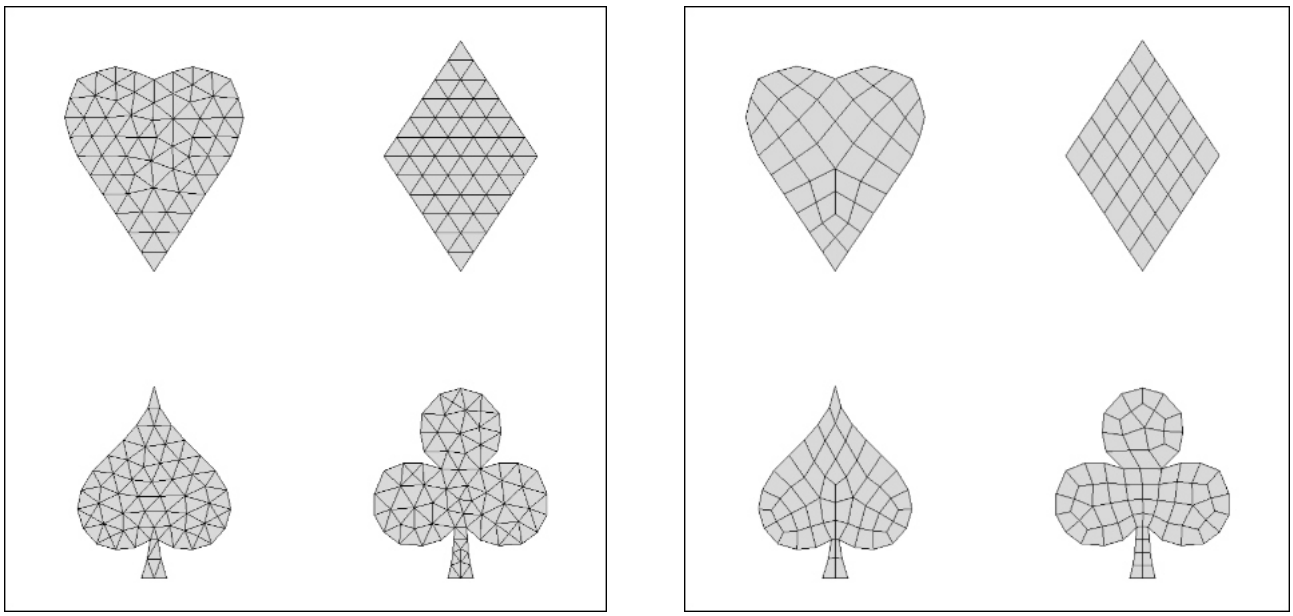


Figure 12 – Multiple meshes (T3 and all-Q4).
The four sub-domains are meshed simultaneously.

In this example, the coordinates matrix and the connectivity of the contour meshes are read from a file[20]:

```
#include "stdafx.h"
#include <fstream>

int main()
{
   std::ifstream                    istrm("cards.dat");
   triamesh_iso::mesher             the_mesher;
   triamesh_iso::mesher::data_type  data;

   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   matio::read(istrm, data.pos);
   matio::read(istrm, data.connectB);

   the_mesher.run(data);
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

[20] We could also have used the function `cm2::meshtools1d::mesh_spline` which generate 1-D meshes along splines.

The input boundary meshes are read from an ASCII file with `cm2::matio::read`[21].

The format for the matrices is:

```
n X m [
d0,0   d0,1   d0,2  ... d0,m-1
d1,0   d1,1   d1,2  ... d1,m-1
...
dn-1,0 dn-1,1 dn-1,2... dn-1,m-1 ]
```

The format for each component of the matrix is free.

For instance a 2 x 4 `DoubleMat` can be stored as:

```
2 X 4 [
0 0.5 1  2.0
0  1  1  2.E-1]
```

Notes:

- We can see in this example that the meshes may not always be symmetric even with a symmetric contour.
- We can set the flag `multi_structured_flag = true` to force any rectangle-like (or diamond-like) sub-domain to be meshed in a structured manner.

[21] A similar `cm2::matio::transpose_read` function can read a matrix and transpose it on the fly.
This can be more useful because it is usually more convenient to store the transposed matrices in the ASCII files.

As an exercise we can get the same result by making four successive meshes and concatenating the results:

```
#include "stdafx.h"
#include <fstream>

int main()
{
   std::ifstream                istrm
   UIntMat                      connectM;
   DoubleMat                    pos;

   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   triamesh_iso::mesher              the_mesher;
   triamesh_iso::mesher::data_type   data;

   istrm.open("heart.dat");
   matio::read(istrm, data.pos);
   matio::read(istrm, data.connectB);
   the_mesher.run(data);
   pos.push_back(data.pos);
   connectM.push_back(data.connectM);

   istrm.open("spade.dat");
   matio::read(istrm, data.pos);
   matio::read(istrm, data.connectB);
   the_mesher.run(data);
   matscal::add(pos.cols(), data.connectM);   // Shift indices.
   pos.push_back(data.pos);
   connectM.push_back(data.connectM);

   istrm.open("diamond.dat");
   matio::read(istrm, data.pos);
   matio::read(istrm, data.connectB);
   the_mesher.run(data);
   matscal::add(pos.cols(), data.connectM);   // Shift indices.
   pos.push_back(data.pos);
   connectM.push_back(data.connectM);

   istrm.open("club.dat");
   matio::read(istrm, data.pos);
   matio::read(istrm, data.connectB);
   the_mesher.run(data);
   matscal::add(pos.cols(), data.connectM);   // Shift indices.
   pos.push_back(data.pos);
   connectM.push_back(data.connectM);

   meshtools::medit_output("out.mesh", pos, connectM, CM2_FACET3);

   return 0;
}   // main
```

8. Shared boundaries

Edges can be shared between some contours and lines. In this case some edges are defined several times (usually twice) in the `connectB` matrix but with different orientation. In addition it is sometimes more convenient for the user to generate the 1-D meshes of the contours independently from each other. That usually implies duplicated nodes on the shared contours.

The following example deals with such a case.

Consider three sub-domains all oriented counter-clockwise as defined below. Several edges are shared between sub-domains but with different orientation. We also want to mesh the contours of the sub-domains independently from each other but without any duplicated nodes.

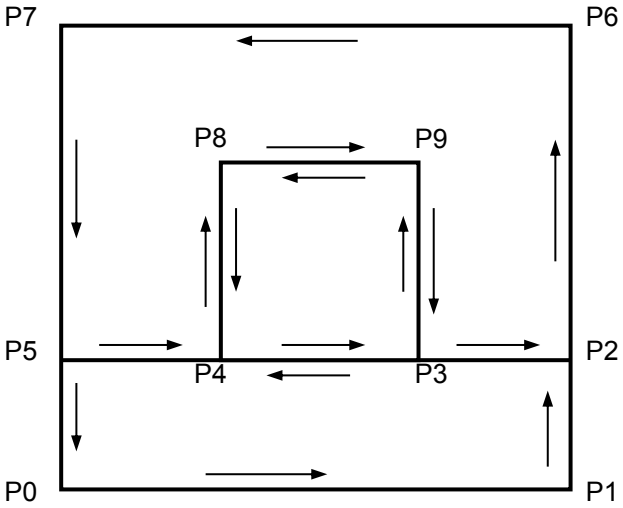


Figure 13 – Domain composed of three attached sub-domains.

The shared edges are no problem for the mesher. The duplicate nodes however must be avoided. In non-strict mode (see **CM2 TriaMesh® Iso/Aniso and CM2 QuadMesh® Iso/Aniso - reference manual**), duplicated nodes can be discarded indeed but that implies also that the associated edges cannot be enforced. As a side effect, the mesher may not be able to tell the sign of the inner square, and that can lead to a hole.

The solution consists in merging the nodes after the meshing of the edges before the 2-D meshing:

```
#include "stdafx.h"

static void mesh_segment
  (DoubleMat& pos, UIntMat& connectB,
   unsigned start_index, unsigned stop_index, unsigned num_edges)
{
   UIntVec            indices;
   meshtools1d::mesh_straight(pos, start_index, stop_index, num_edges, indices);
   meshtools1d::indices_to_connectE2(indices, connectB);
}

int main()
{
   const DoubleVec2   P0(0., 0.), P1(10., 0.), P2(10., 2.), P3(8., 2.);
   const DoubleVec2   P4(2., 2.), P5(0., 2.), P6(10., 10.), P7(0., 10.);
   const DoubleVec2   P8(2., 8.), P9(8., 8.);
   const unsigned     N(4);
   DoubleMat          pos;
   UIntMat            connectB;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES POINTS.
   pos.push_back(P0); pos.push_back(P1); pos.push_back(P2); pos.push_back(P3);
   pos.push_back(P4); pos.push_back(P5); pos.push_back(P6); pos.push_back(P7);
   pos.push_back(P8); pos.push_back(P9);

   // BOTTOM RECTANGLE POSITIVE (I.E. COUNTER-CLOCKWISE).
   mesh_segment(pos, connectB, 0, 1, N);
   mesh_segment(pos, connectB, 1, 2, N);
   mesh_segment(pos, connectB, 2, 3, N);
   mesh_segment(pos, connectB, 3, 4, N);
   mesh_segment(pos, connectB, 4, 5, N);
   mesh_segment(pos, connectB, 5, 0, N);

   // TOP HORSE-SHOE POSITIVE (I.E. COUNTER-CLOCKWISE).
   mesh_segment(pos, connectB, 2, 6, N);
   mesh_segment(pos, connectB, 6, 7, N);
   mesh_segment(pos, connectB, 7, 5, N);
   mesh_segment(pos, connectB, 5, 4, N);
   mesh_segment(pos, connectB, 4, 8, N);
   mesh_segment(pos, connectB, 8, 9, N);
   mesh_segment(pos, connectB, 9, 3, N);
   mesh_segment(pos, connectB, 3, 2, N);

   // INNER SQUARE POSITIVE (I.E. COUNTER-CLOCKWISE).
   mesh_segment(pos, connectB, 3, 9, N);
   mesh_segment(pos, connectB, 9, 8, N);
   mesh_segment(pos, connectB, 8, 4, N);
   mesh_segment(pos, connectB, 4, 3, N);

   // MERGE TOGETHER DUPLICATED NODES.
   meshtools::merge(pos, connectB, /*tol=>*/ 1E-6, /*merge_type=>*/ 0);

   // THE 2D MESH.
   triamesh_iso::mesher               the_mesher;
   triamesh_iso::mesher::data_type    data(pos, connectB);
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

Note that this solution works because the shared edges are discretized similarly and the nodes are (almost) coincident.

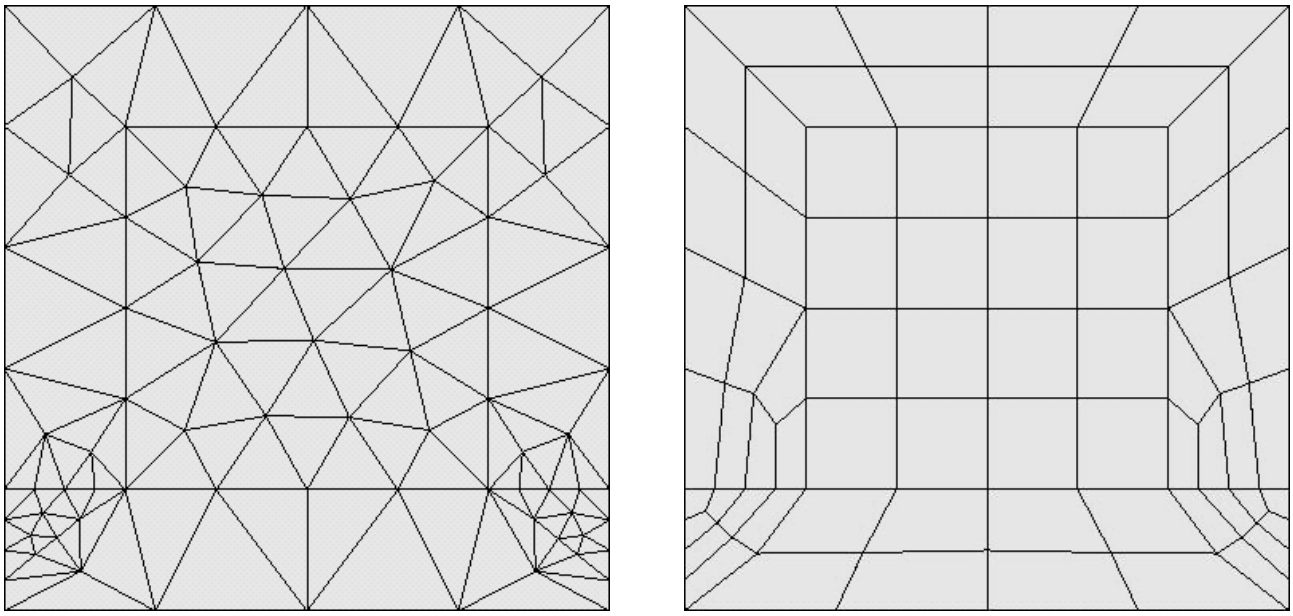


Figure 14 – Example with shared boundaries (T3 and all-Q4).

Note that the contour of the inner square is oriented completely both ways (positive and negative). In such a case, the mesher favors the positive orientation and keeps the inner sub-domain.

A similar case occurs when an inner contour is not properly oriented (see figure below). The mesher considers the inner domain to have the same status as the "most external domain" adjacent to it. Here the most external domain adjacent to the inner square is the outer square. Hence, the inner square will be meshed (i.e. no hole).

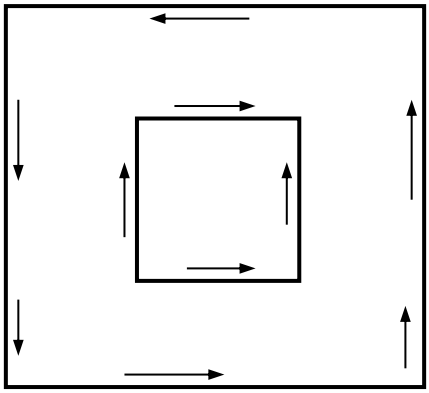


Figure 15 – Example of ambiguous orientation of an inner contour.

Here is another example where a hole is adjacent to the external contour. In this case, the most external domain adjacent to the inner square is the outside void. Hence, the inner square will not be meshed (i.e. hole).

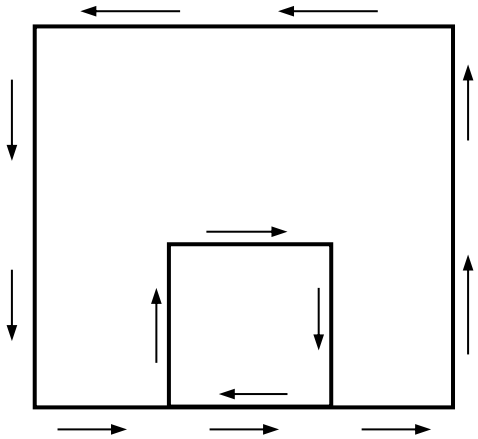


Figure 16 – Hole adjacent to the external contour.

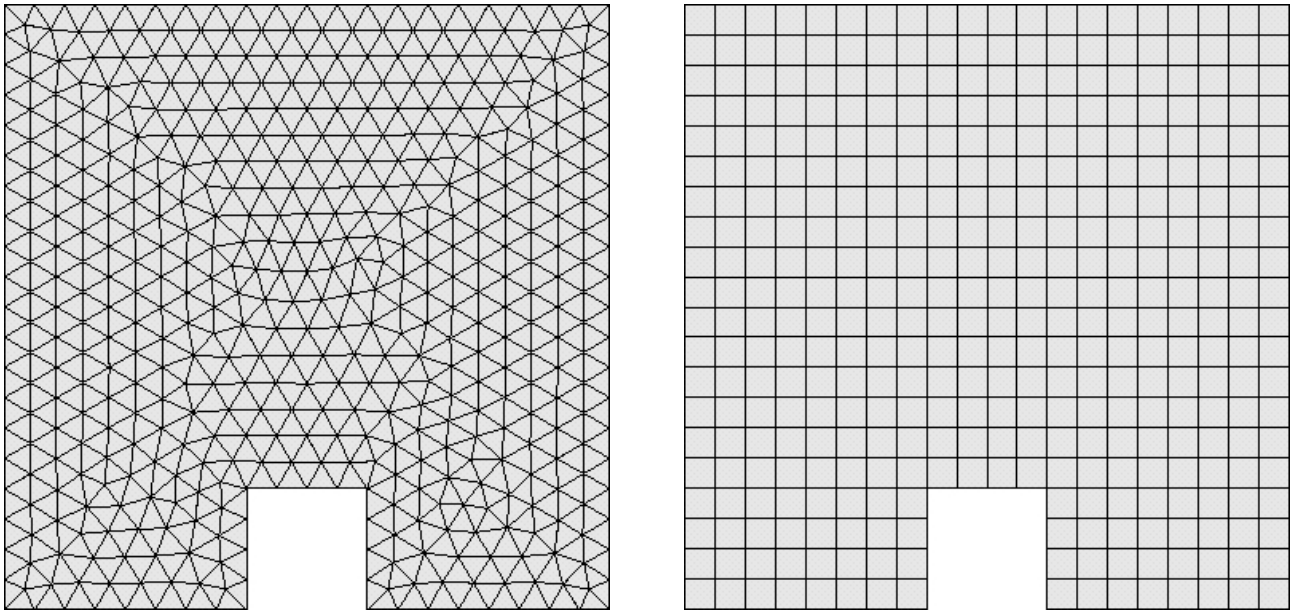


Figure 17 – Example of ambiguous orientation of an inner contour.

9. Background mesh

Sometimes it is not convenient to specify the target mesh sizes at some hard nodes. This is the case especially when automatic mesh adaptivity is involved. The *background mesh* option is the solution in this case.

The background mesh is an auxiliary mesh used by the mesher to find the target mesh size at any point inside the domain. It's represented by the connectivity matrix `background_mesh` in the data of the mesher.

As always the indices of the nodes refer to columns in the same `pos` matrix as all other connectivity matrices or vectors (such as `connectM` or `connectB`).

The nodes of the background mesh can share nodes with `connectB` or can all be different. They must all have a valid associated size value in the `metrics` array. The size map (also called metric map) is interpolated inside the background mesh.

In the following example, a regular structured background mesh is used to support a size map with a sinusoidal variation in the two directions. The domain to be meshed is a simple square regularly discretized along its boundaries[22].

[22] For a change, we use here the mesh_straight overload with the parameters for the sizes at the extremities.

```
#include "stdafx.h"

int main()
{
   const double      L(4.), h0(0.25), h1(0.05);
   DoubleMat         pos;
   UIntVec           indices;
   UIntMat           connectE2, connectT3, BGM;
   DoubleVec         sizes;
   unsigned          n;
   double            w, h;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES AND LINE MESHES.
   pos.push_back(DoubleVec2(-L/2, -L/2));
   pos.push_back(DoubleVec2 (+L/2, -L/2));
   pos.push_back(DoubleVec2 (+L/2, +L/2));
   pos.push_back(DoubleVec2 (-L/2, +L/2));
   meshtools1d::mesh_straight(pos, 0, 1, h0, h0, true, indices);
   indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, h0, h0, true, indices);
   indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, h0, h0, true, indices);
   indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, h0, h0, true, indices);
   meshtools1d::indices_to_connectE2(indices, connectE2);

   // THE BACKGROUND MESH.
   n = unsigned(L/h1);
   indices.clear();
   meshtools1d::mesh_straight(pos, 0, 1, n, indices);
   indices.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, n, indices);
   indices.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, n, indices);
   indices.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, n, indices);
   meshtools2d::mesh_struct_T3(pos, indices, n, true, BGM);

   // THE METRICS ON THE BACKGROUND MESH.
   indices.clear();
   meshtools::unique_indices(indices, BGM);
   sizes.resize(pos.cols(), 0.);       // Null value for nodes not in BGM.
   for (size_t i = 0; i < indices.size(); ++i)
   {
      n = indices[i];
      w = std::max(::fabs(pos(0, n)), ::fabs(pos(1, n)));
      h = ::cos(8.*M_PI*w/L) * (h0-h1)/2. + (h0+h1)/2.;
      sizes[n] = h;
   }

   // THE 2D MESH.
   triamesh_iso::mesher               the_mesher;
   triamesh_iso::mesher::data_type    data(pos, connectE2);
   data.background_mesh = BGM;
   data.metrics = sizes;
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

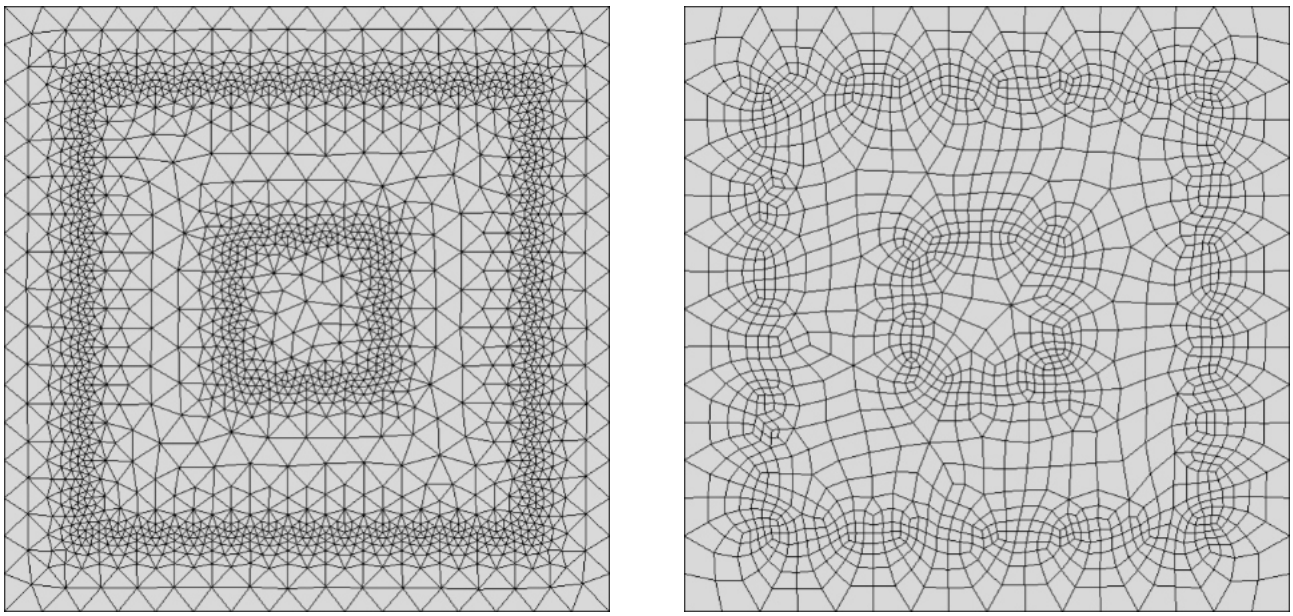


Figure 18 – Use of a background mesh to support a sizes map on the domain (T3 and all-Q4).

The background mesh is the same structured triangle mesh in both cases (here, covering all the domain):

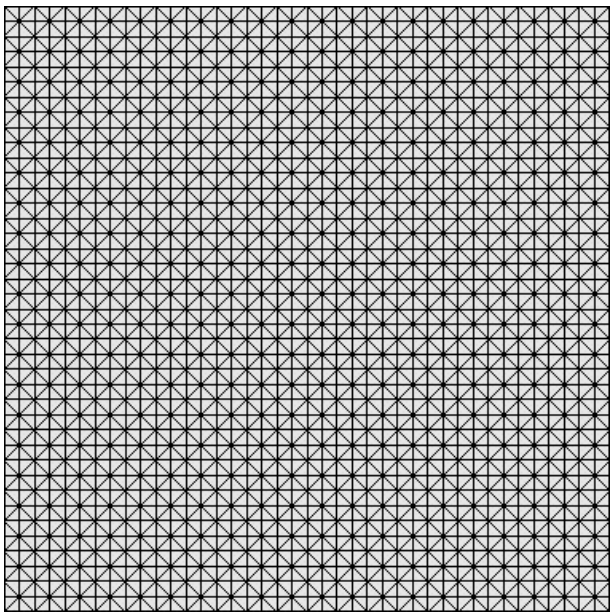


Figure 19 – The background mesh used in the previous examples.

The background mesh does not need to fit exactly the domain to be meshed. It can cover only a small part of it and/or be partially outside of the domain. In the areas not covered by the background mesh, the default size field based on hard edge length, specific sizes at the hard nodes and target size (if any of them) is used instead.

Here is an example where the domain is a disk and the background mesh is also a disk but with half the radius. We have set a uniform value for the sizes map on the background mesh to get a finer mesh in this area.

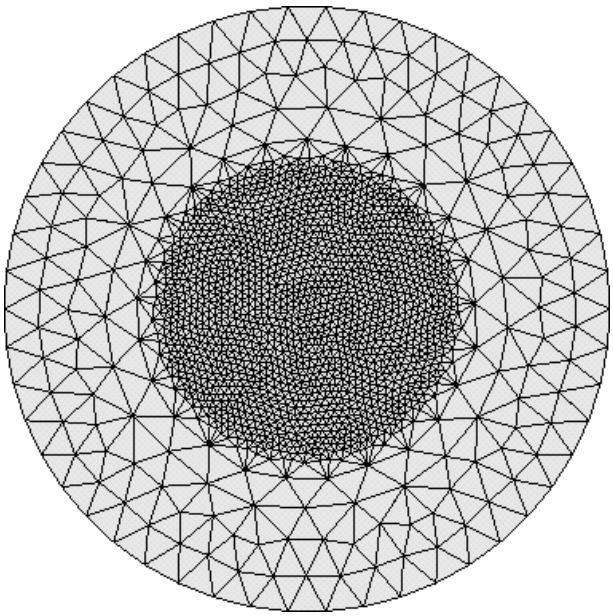


Figure 20 – Background mesh covering only a fraction of the domain.

We can also consider the case where the boundary mesh of the domain must also be governed by a background mesh. In addition to the 2D-background mesh we need also to discretize the boundary in order to support the sizes map on this line. Then the real boundary mesh is generated using this 1D "background mesh" and the associated sizes. The discretization for this 1D background mesh must be fine enough to represent accurately the geometry of the line.

An overload of the `cm2::meshtools1d::mesh_line` function is used for that[23]. Here we discretize a full circle with 200 nodes in `indices0` starting from node #1 and centered on point `CR`:

```
meshtools1d::extrude_rotate(pos, 1, CR, 2 * M_PI, 200, indices0);
indices0.back() = indices0.front();
```

Sizes are specified on the nodes of this circle and a new set of adapted nodes are generated:

```
vecvec::push_back(sizes, indices0, sizes0);   // Pick-up sizes for indices0.
meshtools1d::mesh_line(pos, indices0, sizes0, true, 1,
                       UINT_MAX, 0., indices, new_U, new_sizes);
meshtools1d::indices_to_connectE2(indices, connectE);
```

The parameters `true`, `1`, `UINT_MAX` and `0`. stand for: force even number of edges, minimum of 1 edge, maximum of `UINT_MAX` edges along the arc and no chordal control[21].

The `indices` vector now contains the nodes of the circle mesh adapted to the metrics.

`new_U` and `new_sizes` contain the parameter values along the circle and the interpolated metrics at these nodes but these vectors are not used in the rest of the example.

[23] Several overloads for `mesh_straight`, `mesh_spline` and `mesh_line` exist in the `meshtools1d` library.

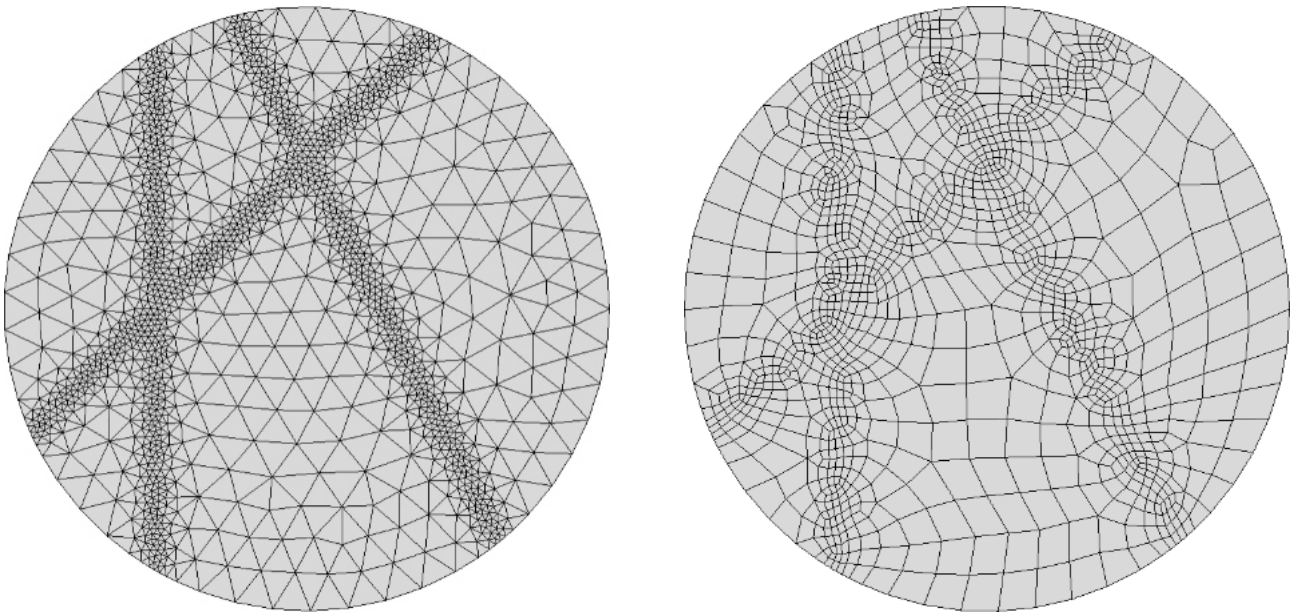


Figure 21 – Background meshes for both the boundary and the domain (T3 and all-Q4).

Note: A background mesh can be used also in `REGULARIZE_MODE` on a pre-existing mesh to adapt/optimize it to a changed metrics map.

[24] See HTML reference manual for more info on these parameters.

```
#include "stdafx.h"

int main()
{
   const double            R(4.);
   const double            h0(0.5), h1(0.05), sig(0.40);
   const DoubleVec2        CR(0., 0.), P0(R, 0.);
   DoubleMat               pos;
   DoubleVec               new_U, sizes, sizes0, new_sizes;
   UIntVec                 indices0, indices;
   UIntMat                 connectE, connectM, BGM;
   double                  x, y, w, w0, w1, w2;

   // UNLOCK THE DLL.
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   triamesh_iso::mesher        the_mesher;

   pos.push_back(CR);    // Point #0 (centre of the circle).
   pos.push_back(P0);    // Point #1 (start of the circle).

   // THE 2D BACKGROUND MESH
   n = unsigned(2.*M_PI*R / h1);
   meshtools1d::extrude_rotate(pos, 1, CR, 2 * M_PI, n, indices);
   indices.back() = indices.front();   // Close the circle.
   meshtools1d::indices_to_connectE2(indices, connectE);
   triamesh_iso::mesher::data_type    BGMdata(pos, connectE);
   the_mesher.run(BGMdata);
   BGMdata.extract(pos, BGM);

   // MESH THE GEOMETRIC SUPPORT OF THE BOUNDARY (1D BACKGROUND MESH).
   meshtools1d::extrude_rotate(pos, 1, CR, 2*M_PI, 200, indices0);
   indices0.back() = indices0.front();

   // THE METRICS ON THE BACKGROUND MESHES (1D AND 2D).
   indices.clear();
   meshtools::unique_indices(indices, BGM);
   indices.push_back(indices0);      // Also the nodes of the circle.
   sizes.resize(pos.cols(), 0.);
   for (size_t i = 0; i < indices.size(); ++i)
   {
      const size_t     n = indices[i];
      x = pos(0, n);
      y = pos(1, n);
      w0 = ::fabs(y + 2*x - R/2) / ::sqrt(5.);
      w1 = ::fabs(y - x - R/2) / ::sqrt(2.);
      w2 = ::fabs(x + R/2);
      w = std::min(w0, w1);
      w = std::min(w, w2) / sig;
      w = ::exp(-w*w);          // Gaussian variations.
      sizes[n] = = h0*(1.-w) + h1*w;
   }

   // Pick-up the sizes along the circle.
   sizes0.clear();
   vecvec::push_back(sizes, indices0, sizes0);

   // MESH THE CIRCLE ACCORDING TO THE METRICS.
   indices.clear();
   connectE.clear();
   meshtools1d::mesh_line(pos, indices0, sizes0, true, 1,
                          UINT_MAX, 0., indices, new_U, new_sizes);
   meshtools1d::indices_to_connectE2(indices, connectE);

   // THE 2D MESH ACCORDING TO THE METRICS ON THE BGM.
   triamesh_iso::mesher::data_type    data(pos, connectE);
   data.background_mesh = BGM;
   data.metrics = sizes;
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

10. Anisotropic meshes

CM2 TriaMesh Iso and **CM2 QuadMesh Iso** are isotropic unstructured meshers, that is, they tend to produce equilateral triangles and squares. It is sometimes useful however to have elements *stretched* in some specific directions. To deal with complex domains we still need an unstructured mesher. Here come the anisotropic unstructured meshers **CM2 TriaMesh Aniso** and **CM2 QuadMesh Aniso**. They are almost identical to their isotropic counterparts except for the `data.metrics` array that is now a matrix (`DoubleMat`). In the isotropic case we needed only a scalar at each node to define the target mesh size. Now the target mesh size is defined by a 2 x 2 symmetric matrix at each node, stored column-wise in the `metrics` array.

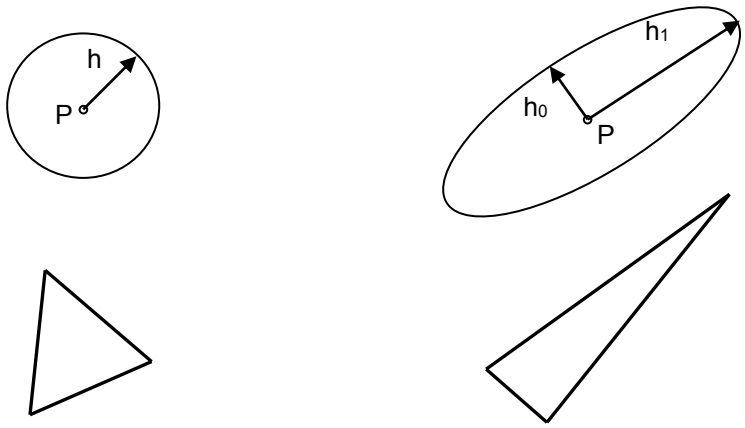


Figure 22 – A single scalar defines an isotropic metric (left).
A 2D-anisotropic metric needs two vectors (right).

$$M_j = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

$$with:$$

$$a > 0$$

$$ac - b^2 > 0$$

$$i.e.\ the\ two\ eigen\ values\ are > 0$$

$$data.metrics = \begin{bmatrix} . & . & . & a & . & . & . \\ . & . & . & b & . & . & . \\ . & . & . & c & . & . & . \end{bmatrix}$$

column #j

Figure 23 – Definition and storage of the 2-D anisotropic metrics.

Let (v_0, v_1) be the two ortho-normal vectors along the axes of the ellipse:

$$\|\mathbf{v}_0\| = \|\mathbf{v}_1\| = 1$$

$$\langle \mathbf{v}_0, \mathbf{v}_1 \rangle = 0$$

Then the metrics *Mj* writes:

$$M_j = \mathbf{B} \begin{bmatrix} \frac{1}{h_0^2} & 0 \\ 0 & \frac{1}{h_1^2} \end{bmatrix} {}^T\mathbf{B}$$

$$with:$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{v}_0 & \mathbf{v}_1 \end{bmatrix}$$

$$\text{stored column} - \text{wise}$$

The metric equivalent to an isotropic size of *h* writes:

$$M_j = \begin{bmatrix} \frac{1}{h^2} & 0 \\ 0 & \frac{1}{h^2} \end{bmatrix}$$

A null matrix would lead to infinite sizes in both directions (infinite circle).

When the user doesn't specify a metric, the mesher uses the default one which is equivalent to the isotropic default metrics we have seen before. For each hard node the default metric is based on the length of the adjacent edges. This leads to the same default behavior as their related isotropic counterparts. Take for instance examples 1, 2, 3 or 4 and replace:

```
triamesh_iso::mesher       the_mesher;
```

with:

```
triamesh_aniso::mesher     the_mesher;
```

and you get the same meshes[25].

[25] The anisotropic meshers are however much slower than their isotropic counterparts (about 4 times slower).

To benefit from the anisotropic feature the user must fill the `metrics` array with valid anisotropic matrices (i.e. positive-definite matrices). Some functions in `meshtools` and `meshtools1d` can help in computing these matrices as in the following example.

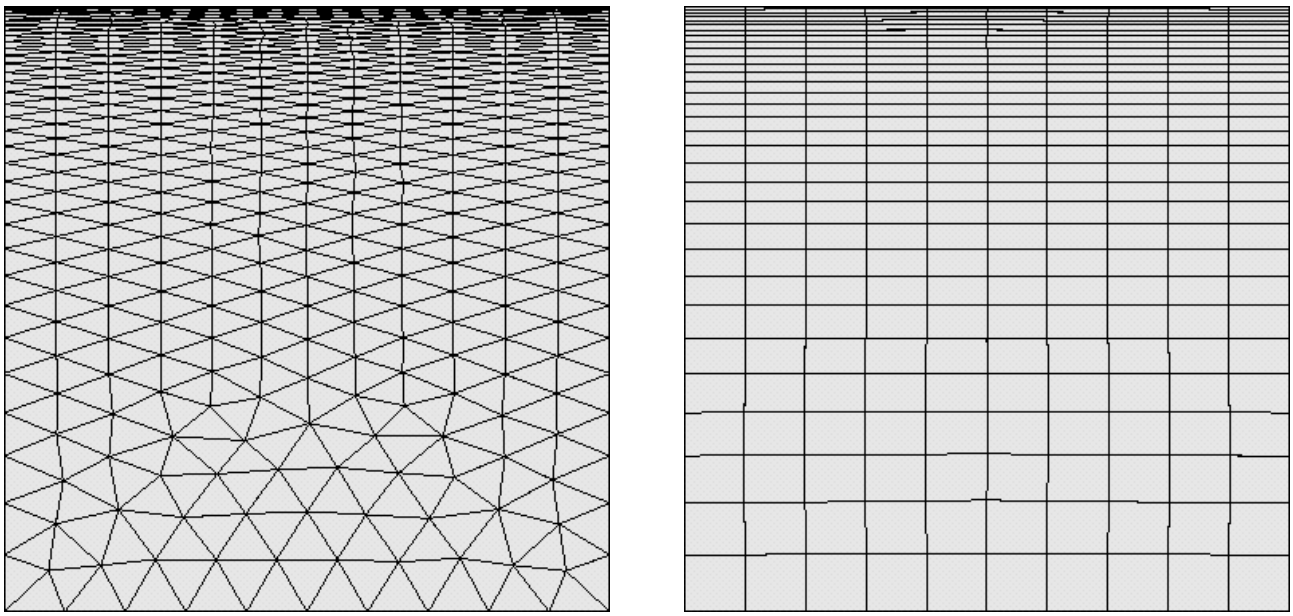


Figure 24 – Anisotropic meshes (T3 and Q4).

Here a square is meshed non-uniformly with the variant of `mesh_straight` we have already seen in the previous section[26]. This is not sufficient to get a 2-D anisotropic mesh. We need an anisotropic mesher. We need also specify that we want a different size along the normals than along the tangents of the boundary lines (along the tangents the default sizes, i.e. mean of the edges' lengths, suit us). This is the role of `cm2::meshtools1d::metrics_gen_aniso2d`. This function takes a 1D mesh and a size along the normal and generates a set of 2-D anisotropic metrics stored in an array `metrics` as depicted in Figure 23. At each node *Ni*, a metric *M(Ni, hn)* is computed. For instance, along the right vertical line we specify a constant size *hn* in the horizontal direction[27]:

```
meshtools1d::metrics_gen_aniso2d(pos, connect2, hn, metrics);
```

[26] We could obviously get about the same structured Q4 mesh with `cm2::meshtools2d::mesh_struct_Q4`.

[27] Note that the metrics parameter is not a pure output parameter. Indeed this function does not simply overwrite the existing columns in `metrics` but replace them with their intersection with the newly computed metric M(Ni, hn). If Mi in column #i already exists in metrics, Mi is replaced by intersection (Mi, M(Ni, hn)). Intersection (Mi, Mj) is the ellipse inscribed inside the two associated ellipses.
Note also that a null metric is equivalent to an infinite circle, and that intersection (Mi, 0) = Mi.
This property of the `metrics_gen_aniso2d` function is essential to make coherent the intersections of the generated metrics at the four summits of the square.

```
#include "stdafx.h"

int main()
{
   const double           L(10.);
   const double           hx(1.);
   const double           h0y(hx);           // Y size at bottom line.
   const double           h1y(hx / 20.);     // Y size at top line.
   const DoubleVec2       P0(0., 0.);
   const DoubleVec2       P1(L, 0.);
   const DoubleVec2       P2(L, L);
   const DoubleVec2       P3(0., L);
   DoubleMat              pos;
   UIntVec                indices;
   UIntMat                connect1, connect2, connect3, connect4, connectE;
   UIntMat                connectM;
   DoubleMat              metrics;

   // UNLOCK THE DLL.
   triamesh_aniso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   // VERTICES
   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);

   // BOTTOM LINE
   indices.clear();
   meshtools1d::mesh_straight(pos, 0, 1, hx, hx, true, indices);
   meshtools1d::indices_to_connectE2(indices, connect1);
   connectE.push_back(connect1);

   // RIGHT-SIDE LINE
   indices.clear();
   meshtools1d::mesh_straight(pos, 1, 2, h0y, h1y, true, indices);
   meshtools1d::indices_to_connectE2 (indices, connect2);
   connectE.push_back(connect2);

   // LEFT-SIDE LINE
   indices.clear();
   meshtools1d::mesh_straight(pos, 2, 3, hx, hx, true, indices);
   meshtools1d::indices_to_connectE2 (indices, connect3);
   connectE.push_back(connect3);

   // TOP LINE
   indices.clear();
   meshtools1d::mesh_straight(pos, 3, 0, h1y, h0y, true, indices);
   meshtools1d::indices_to_connectE2 (indices, connect4);
   connectE.push_back(connect4);

   // METRICS
   meshtools1d::metrics_gen_aniso2d(pos, connect1, /*hn=>*/ h0y, metrics);
   meshtools1d::metrics_gen_aniso2d(pos, connect2, /*hn=>*/  hx, metrics);
   meshtools1d::metrics_gen_aniso2d(pos, connect3, /*hn=>*/ h1y, metrics);
   meshtools1d::metrics_gen_aniso2d(pos, connect4, /*hn=>*/  hx, metrics);

   // 2D MESH
   triamesh_aniso::mesher               the_mesher;
   triamesh_aniso::mesher::data_type    data(pos, connectE2);
   data.metrics = metrics;
   the_mesher.run(data);

   // VISUALIZATION.
   meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

   return 0;
}   // main
```

As already stated, except for the `metrics` array, the anisotropic meshers have the very same options and parameters as their isotropic counterparts. They accept internal hard lines, isolated nodes, multiple domains, shared boundaries, background meshes...

The following example illustrates the internal hard line feature.
We have specified a normal size along the inner circle much smaller than the default tangent size (using again `meshtools1d::metrics_gen_aniso2d`).

For the external square, nothing was specified in the metrics array and the mesher used its default isotropic metrics based on the length of the adjacent edges.

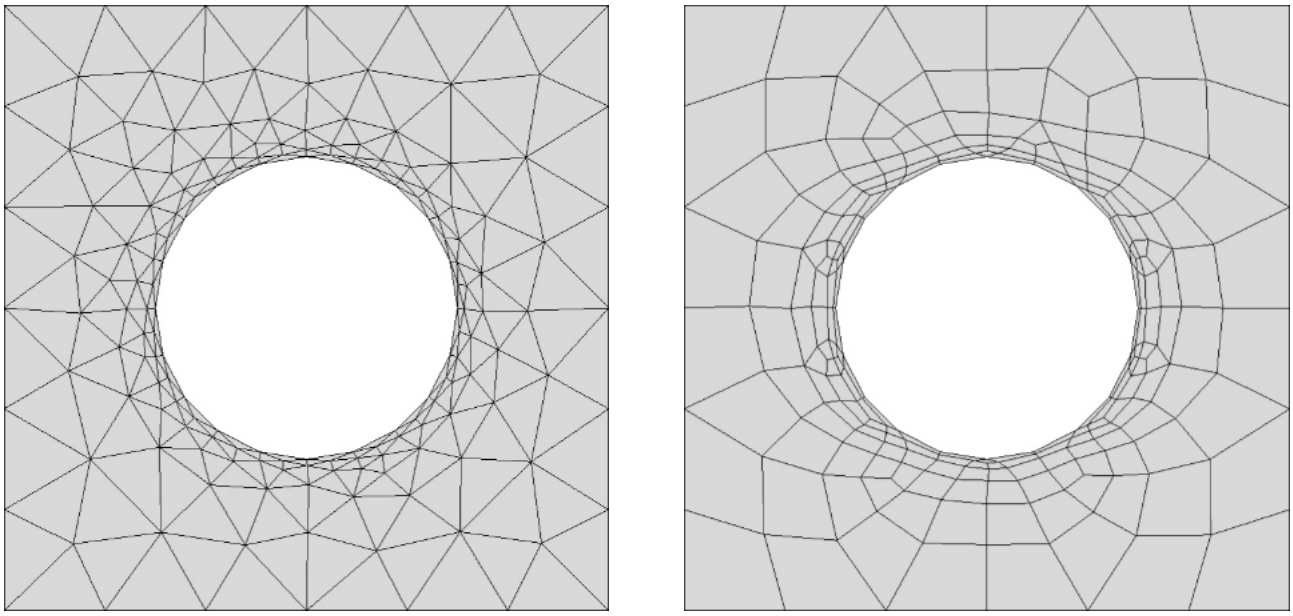


Figure 25 – 2-D anisotropic meshes (T3 and all-Q4).

The example below is mesh of Figure 21 revisited the anisotropic way.
Here we specify a small size in the directions normal to the three lines but a uniform size along the tangents.
The normal sizes follow the same kind of Gaussian variation.
All these metrics are specified at the nodes of the same uniform background mesh.

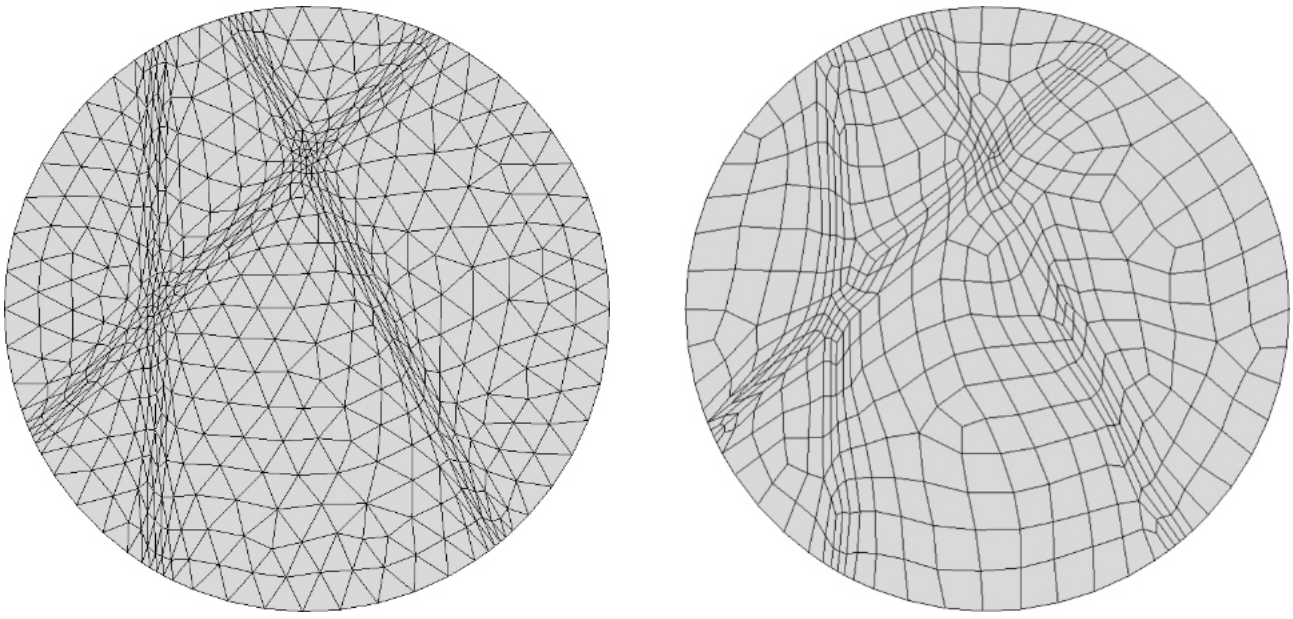


Figure 26 – 2-D anisotropic meshes (T3 and all-Q4).

11. 3-D surface meshes (aniso meshers only)

The four meshers **CM2 TriaMesh Iso**, **CM2 QuadMesh Iso** and their anisotropic versions **CM2 TriaMesh Aniso** and **CM2 QuadMesh Aniso** are plane 2-D meshers. They generate or optimize meshes in the Z = 0 plane only. To generate meshes on 3-D parametric surfaces, **CM2 MeshTools** offers a convenient solution by the way of a template function that pre- and post-process the data for a 2-D anisotropic mesher (**CM2 TriaMesh Aniso** or **CM2 QuadMesh Aniso**):

```
template <class Surface, class AnisoMesher, class AuxMesher>
int
meshtools2d::mesh_surface_param
   (const Surface& S, AnisoMesher& mesher2D,
    typename AnisoMesher::data_type& data3D, AuxMesher& aux_mesher,
    double max_chordal_error, double min_h, unsigned chordal_control_type,
    unsigned high_order_type = 0, unsigned high_order_mode = 2,
    double max_chordal_error_ratio = 0.10, bool dry_run_flag = false,
    unsigned max_bgm_remeshings = 4,
    bool recompute_Qs_flag = true, bool compute_area_flag = true);
```

This function can be used as in the following code sample:

```
triamesh_aniso::mesher              the_mesher;
triamesh_iso::mesher                aux_mesher;
triamesh_aniso::mesher::data_type   data(pos, connectE2);
surface_type                        S(some parameters);   // A parametric surface.

meshtools2d::mesh_surface_param(S, the_mesher, data, aux_mesher, -0.05, 0., 4);
data.extract(pos, connectM);
```

The class `Surface` for parameter `S` is a concept of parametric surface with members:

```
int get_3D_coordinates (const DoubleMat& pos2D, DoubleMat& pos3D) const;
int get_2D_coordinates (const DoubleMat& pos3D, const UIntVec& nodeIDs,
                        DoubleMat& pos2D) const;
int get_tangents (const DoubleMat& pos2D, DoubleMat& T) const;
int get_curvatures (const DoubleMat& pos2D, DoubleMat& C) const;
```

The `Surface::get_3D_coordinates` member should compute the 3-D coordinates of a set of 2-D points located on the reference plane. The 3-D coordinates of the point in column #j of `pos2D` must be returned in column #j of `pos3D`. This function should return zero when successful and a negative value (-1 for instance) when failed.

The `Surface::get_2D_coordinates` member is the reciprocal function of the previous one[28]. It should give the coordinates in the 2-D reference plane of a set of 3-D points. The reference coordinates of the point in column #j of `pos3D` must be returned in column #j of `pos2D`. This function should return zero when successful and a negative value (-1 for instance) when failed.

[28] For parametric surfaces such as Bezier surfaces or NURB surfaces, the computation of reference coordinates often involves a non-linear search. However, this function is called only for the nodes on the boundary mesh and for the isolated nodes (i.e. the hard nodes only). It is not called for the new nodes generated inside the surface by the mesher.

`nodeIDs` is an auxiliary vector that can be helpful for an effective implementation. It contains the global indices of the nodes for which the 2-D coordinates are required. These are the indices in the the global matrix `data3D.pos`. `nodeIDs[j]` is the node ID (i.e. column in `data.pos`) for the coordinates in column `j` of `pos3D`. This array can be used for fast 2-D coordinates retrieval if these coordinates have been computed before.

The `Surface::get_tangents` member should compute the two tangents $B_u = \frac{\partial P}{\partial u}$ and $B_v = \frac{\partial P}{\partial v}$ on the surface at a set of points given by their reference coordinates.

These tangents must not be normalized. They are the mere derivatives of the surface with respect to two reference parameters. The two tangents at the point in column #j of `pos2D` must be returned in column #j of `T` (dimension 6 x N). The first three values are for the first tangent (with respect to the first reference coordinate), then the next three are for the second tangent[29].

The `Surface::get_curvatures` function may compute the curvatures of the surface at a set of points given by their reference coordinates (optional).

The curvatures H are 2 x 2 symmetric matrices defined as:

- $H_{uu} = \left\langle \frac{\partial^2 P}{\partial u^2}, N \right\rangle$

  Dot product between the derivative of B_u (first local tangent) with respect to u, and the normal N to the surface.

- $H_{uv} = \left\langle \frac{\partial^2 P}{\partial u \, \partial v}, N \right\rangle$

  Dot product between the derivative of B_u (first local tangent) with respect to v, or derivative of B_v (second local tangent) with respect to u, and the normal N to the surface.

- $H_{vv} = \left\langle \frac{\partial^2 P}{\partial v^2}, N \right\rangle$

  Dot product between the derivative of B_v (second local tangent) with respect to v, and the normal N to the surface.

These three values must be stored column-wise in matrix `H:` H_{uu} on row 0, H_{uv} on row 1 and H_{vv} on row 2.

You can leave the implementation of this member empty (returning -1 for instance). In this case approximate curvatures computed from variations of the tangents will be used instead.

[29] This function should normally return in `T` only valid bases made of two non-null and non-colinear vectors. When the surface exhibits some singularities, the user can "correct" the deficient bases. As far as the mesher is concerned, the exactness of these tangents with respect to the true surface is not critical. More precisely, the tangent bases are used by the template function as transformation matrices to compute the target anisotropic 2-D `metrics` array. The template function checks for deficient aniso metrics (derived from deficient local bases) and replace them with a default one.

The template class `AnisoMesher` is a concept of triangle anisometric mesher with function:

```
void run (typename AnisoMesher::data_type& data) const;
```

The `mesh_surface_param` function is designed to work with one of the 2-D anisotropic meshers **CM2 TriaMesh Aniso** or **CM2 QuadMesh Aniso**.

The `data3D` parameter is the structure gathering all the input and output data, just like for any other unstructured mesher of the **CM2 MeshTools** SDK. The type of `data3D` is either `triamesh_aniso::mesher::data_type` or `quadmesh_aniso::mesher::data_type` depending on the type of anisotropic mesher used. The point is that the `pos` matrix is now a 3-D coordinates matrix and the `metrics` array contains 3-D anisotropic metrics (dimensions 6 x NODS).

3D-anisotropic metrics are defined as below:

$$M_j = \begin{bmatrix} a & b & d \\ b & c & e \\ d & e & f \end{bmatrix}$$

with:

$$a > 0$$
$$ac - b^2 > 0$$
$$Det(M_j) > 0$$
$$i.e.\ the\ three\ eigen\ values\ are > 0$$

$$data3D.metrics = \begin{bmatrix} . & . & . & a & . & . & . \\ . & . & . & b & . & . & . \\ . & . & . & c & . & . & . \\ . & . & . & d & . & . & . \\ . & . & . & e & . & . & . \\ . & . & . & f & . & . & . \end{bmatrix}$$

column #j

Figure 27 – Definition and storage of the 3-D anisotropic metrics.

Let $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2)$ be the three ortho-normal vectors along the axes of the ellipsoid:

$$\|\mathbf{v}_0\| = \|\mathbf{v}_1\| = \|\mathbf{v}_2\| = 1$$
$$\langle \mathbf{v}_0, \mathbf{v}_1 \rangle = 0$$
$$\langle \mathbf{v}_0, \mathbf{v}_2 \rangle = 0$$
$$\langle \mathbf{v}_1, \mathbf{v}_2 \rangle = 0$$
$$\langle \mathbf{v}_0 \times \mathbf{v}_1, \mathbf{v}_2 \rangle = 1$$

Then, the metrics M_j writes:

$$M_j = \mathbf{B} \begin{bmatrix} \frac{1}{h_0^2} & 0 & 0 \\ 0 & \frac{1}{h_1^2} & 0 \\ 0 & 0 & \frac{1}{h_2^2} \end{bmatrix} {}^T\mathbf{B} \quad \text{with:} \quad \mathbf{B} = [\mathbf{v}_0 \quad \mathbf{v}_1 \quad \mathbf{v}_2]$$
$$stored\ column-wise$$

The 3-D metric equivalent to an isotropic size of *h* writes:

$$M_j = \begin{bmatrix} \frac{1}{h^2} & 0 & 0 \\ 0 & \frac{1}{h^2} & 0 \\ 0 & 0 & \frac{1}{h^2} \end{bmatrix}$$

A null matrix would lead to infinite sizes in the three directions (infinite sphere).

The two parameters `max_chordal_error` and `chordal_control_type` are used to limit the chordal error between the mesh and the parametric surface. We don't use them in this tutorial (set to 0). Please refer to the HTML reference manual for more information on them.

This first example illustrates the use of the anisotropic mesh as the intermediate mesh. Here, the parametric surface to be meshed is plane but its boundaries are curved (sinusoidal). The parameters' range is the unit square [0 1] x [0 1].

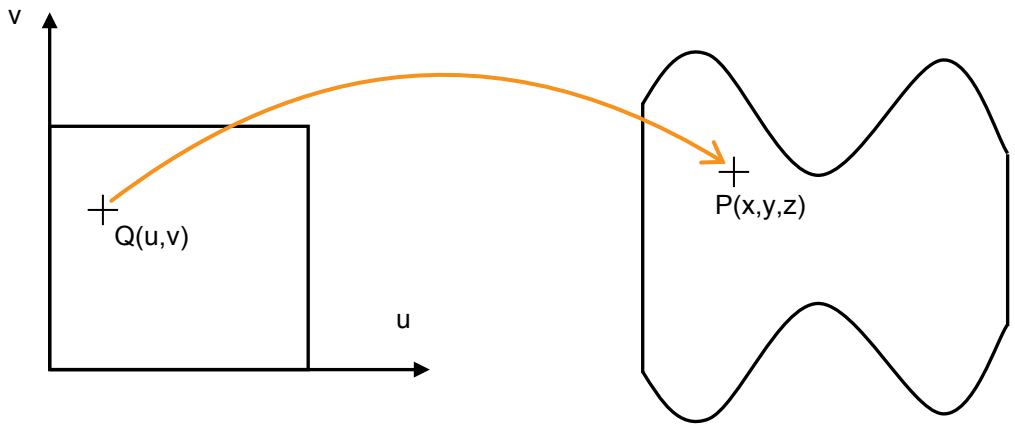


Figure 28 – Mapping between the reference space and the surface.

The source of this example is as follow:

```
#include "stdafx.h"

/*The Surface class implements the functions needed by mesh_surface_param*/
struct surface
{
// Constructor (parameters to define the surface should be passed here).
surface (double Lx, double Ly, double a = 0.5)
   : _Lx(Lx), _Ly(Ly), _a(a) { }

// Computes the 3D coordinates.
int get_3D_coordinates (const DoubleMat& pos2D, DoubleMat& pos3D) const
{
   const size_t      NODS(pos2D.cols());
   double            u, v, x, y, z;

   if (pos2D.rows() != 2) return -1;   // Error.

   if ((pos3D.rows() != 3) || (pos3D.cols() < NODS))
      pos3D.resize(3, NODS);

   for (size_t j = 0; j < NODS; ++j)
   {
      u = pos2D(0, j);
      v = pos2D(1, j);
      x = u * _Lx;
      y = v * _Ly * (1. + _a * ::sin(x));
      z = 0.;
      pos3D(0, j) = x;
      pos3D(1, j) = y;
      pos3D(2, j) = z;
   }
   return 0;      // OK.
}

// Computes the reference coordinates (UV). nodeIDs not used.
int get_2D_coordinates (const DoubleMat& pos3D, const UIntVec& nodeIDs,
                        DoubleMat& pos2D) const
{
   const size_t      NODS(pos3D.cols());
   double            u, v, x, y;

   if (pos3D.rows() != 3) return -1;   // Error.

   if ((pos2D.rows() != 2) || (pos2D.cols() < NODS))
      pos2D.resize(2, NODS);

   for (size_t j = 0; j < NODS; ++j)
   {
      x = pos3D(0, j);
      y = pos3D(1, j);
      u = x / _Lx;
      v = y / (_Ly * (1. + _a * ::sin(x)));
      pos2D(0, j) = u;
      pos2D(1, j) = v;
   }
   return 0;      // OK.
}

/// Computes the local tangents.
int get_tangents (const DoubleMat& pos2D, DoubleMat& T) const
{
   const size_t      NODS(pos2D.cols());
   double            u, v, x;

   if (pos2D.rows() < 2) return -1;   // Error.

   if ((T.rows() != 6) || (T.cols() < NODS))
      T.resize(6, NODS);

   for (size_t j = 0; j < NODS; ++j)
   {
      u = pos2D(0, j);
      v = pos2D(1, j);
      x = u * _Lx;
      T(0, j) = _Lx;
      T(1, j) = v * _Ly * _a * _Lx * ::cos(x);
```

```
      T(2, j) = 0.;
      T(3, j) = 0.;
      T(4, j) = _Ly * (1. + _a * ::sin(x));
      T(5, j) = 0.;
   }
   return 0;      // OK.
}

/// Computes the local curvatures.
int get_curvatures (const DoubleMat& pos2D, DoubleMat& H) const
{
   const size_t      NODS(pos2D.cols());

   if (pos2D.rows() < 2) return -1;   // Error.

   if ((H.rows() != 3) || (H.cols() < NODS))
      H.resize(3, NODS);

   H = 0.;      // Null curvatures here (the surface is plane).
   return 0;
}

/// Data members.
double      _Lx, _Ly, _a;

};   // surface


///
int main()
{
   const DoubleVec2        P0(0., -0.5);
   const DoubleVec2        P1(1., -0.5);
   const DoubleVec2        P2(1., +0.5);
   const DoubleVec2        P3(0., +0.5);
   DoubleMat               pos;
   UIntVec                 indicesG, indices;
   DoubleVec               Us;
   UIntMat                 connectE2, connectM;
   DoubleVec               sizesG, sizes;
   const double            Lx(10.);
   const double            Ly(6.0);
   const double            h0(0.25);
   surface                 S(Lx, Ly, 0.5);  // The parametric surface to be meshed.

   // UNLOCK THE DLLs.
   triamesh_aniso::registration("Licensed to SMART Inc.", "B657DA67QZ01");
   triamesh_iso::registration("Licensed to SMART Inc.", "F53EA108BCWX");

   pos.push_back(P0);
   pos.push_back(P1);
   pos.push_back(P2);
   pos.push_back(P3);

   // GEOMETRIC SUPPORT FOR THE EXTERNAL CONTOUR.
   meshtools1d::mesh_straight(pos, 0, 1, 1./100., 1./100., false, indicesG);
   indicesG.pop_back();
   meshtools1d::mesh_straight(pos, 1, 2, 1./100., 1./100., false, indicesG);
   indicesG.pop_back();
   meshtools1d::mesh_straight(pos, 2, 3, 1./100., 1./100., false, indicesG);
   indicesG.pop_back();
   meshtools1d::mesh_straight(pos, 3, 0, 1./100., 1./100., false, indicesG);
   S.get_3D_coordinates(pos, pos);        // Map UV -> XYZ (same node IDs).

   // MESH THE EXTERNAL CONTOUR WITH UNIFORM SIZE H0.
   sizesG.clear();
   sizesG.resize(indicesG.size(), h0);    // Uniform mesh size.
   meshtools1d::mesh_line(pos, indicesG, sizesG, true,
                          1, UINT_MAX, 0., 0., indices, Us, sizes);
   meshtools1d::indices_to_connectE2(indices, connectE2);

   // MESH THE SURFACE.
   triamesh_aniso::mesher               the_mesher, aux_mesher;
   triamesh_aniso::mesher::data_type    data(pos, connectE2);
   meshtools2d::mesh_surface_param(S, the_mesher, data, aux_mesher, 0., 0., 0);
   data.extract(pos, connectM);
```

```
    data.print_info(&display_hdl);

    // VISUALISATION.
    meshtools::medit_output("out.mesh", data.pos, data.connectM, CM2_FACET3);

    return 0;
}   // main
```

We present below the intermediate anisotropic meshes on the reference space (normally not shown) and the final meshes on the parametric surface.

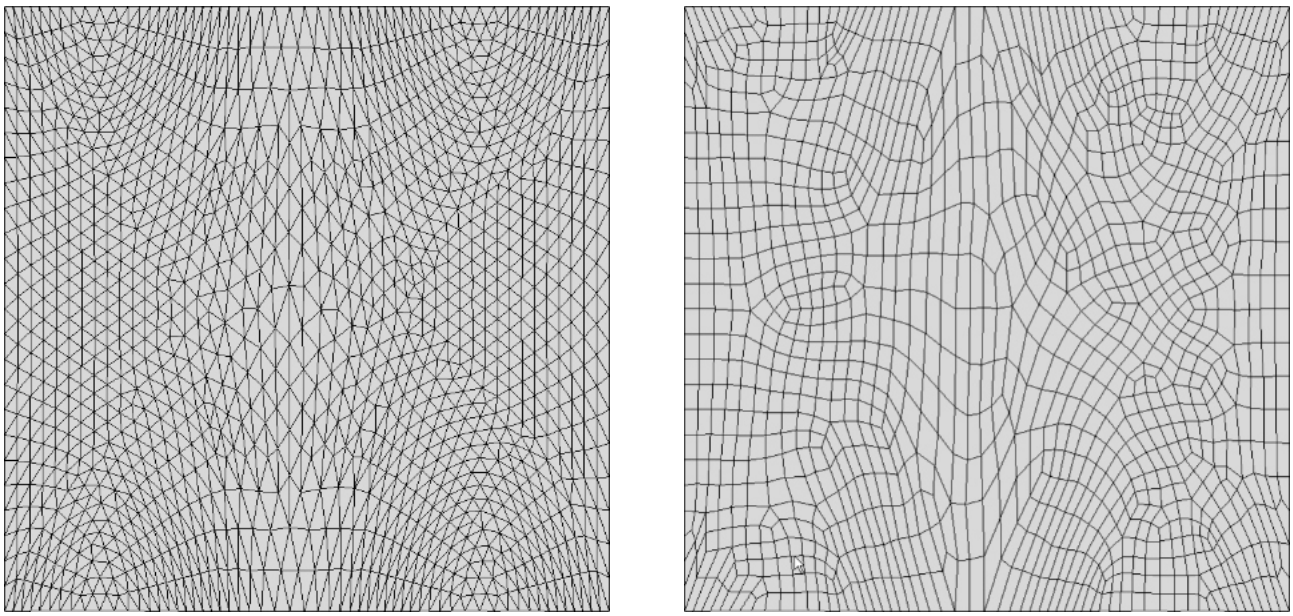


Figure 29 – 2-D anisotropic meshes in the reference space (UV).

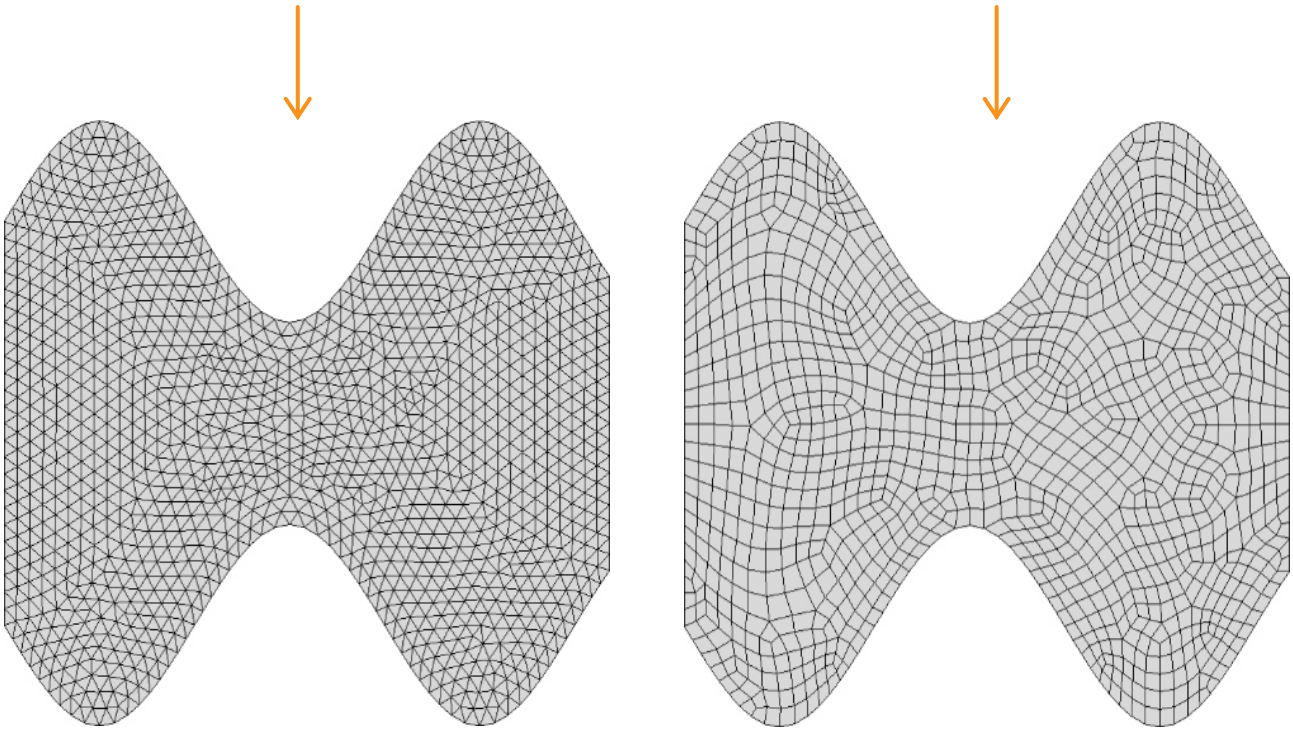


Figure 30 – Surface meshes (T3 and all-Q4) obtained via an anisotropic mesh in the reference space.

The next example is a true 3-D parametric surface (only the source code for the surface class is shown).

```
#include "stdafx.h"

/*The Surface class implements the functions needed by mesh_surface_param*/
struct surface
{
// Constructor.
surface (double L, double H) : _L(L), _H(H) { }

// Computes the 3D coordinates.
int get_3D_coordinates (const DoubleMat& pos2D, DoubleMat& pos3D) const
{
   const size_t      NODS(pos2D.cols());
   double            u, v, x, y, z;

   if (pos2D.rows() < 2) return -1;   // Error.

   if ((pos3D.rows() != 3) || (pos3D.cols() < NODS))
      pos3D.resize(3, NODS);

   for (size_t j = 0; j < NODS; ++j)
   {
      u = pos2D(0, j);
      v = pos2D(1, j);
            x = _L * u;
      y = _L * v;
      z = _H * ::cos(x) * ::cos(y);
      pos3D(0, j) = x;
      pos3D(1, j) = y ;
      pos3D(2, j) = z;
   }
   return 0;      // OK.
}

// Computes the reference coordinates (UV). nodeIDs not used.
int get_2D_coordinates (const DoubleMat& pos3D, const UIntVec& nodeIDs,
                        DoubleMat& pos2D) const
{
   const size_t      NODS(pos3D.cols());
   double            u, v, x, y;

   if (pos3D.rows() < 2) return -1;   // Error.

   if ((pos2D.rows() != 2) || (pos2D.cols() < NODS))
      pos2D.resize(2, NODS);

   for (size_t j = 0; j < NODS; ++j)
   {
      x = pos3D(0, j);
      y = pos3D(1, j);
      u = x / _L;
      v = y / _L;
      pos2D(0, j) = u;
      pos2D(1, j) = v;
   }
   return 0;      // OK.
}

/// Computes the local tangents.
int get_tangents (const DoubleMat& pos2D, DoubleMat& T) const
{
   const size_t      NODS(pos2D.cols());
   double            u, v, x, y;

   if (pos2D.rows() < 2) return -1;   // Error.

   if ((T.rows() != 6) || (T.cols() < NODS))
      T.resize(6, NODS);

   for (size_t j = 0; j < NODS; ++j)
   {
      u = pos2D(0, j);
      v = pos2D(1, j);
      x = _L * u;
      y = _L * v;
      T(0, j) = _L;
      T(1, j) = 0.;
      T(2, j) = -_H * _L * ::sin(x) * ::cos(y);
      T(3, j) = 0.;
      T(4, j) = _L;
      T(5, j) = -_H * _L * ::cos(x) * ::sin(y);
   }
   return 0;      // OK.
}
```

```
/// Computes the local curvatures.
int get_curvatures (const DoubleMat& pos2D, DoubleMat& H) const
{
   const size_t      NODS(pos2D.cols());
   double            u, v, x, y;

   if (pos2D.rows() < 2) return -1;   // Error.

   if ((H.rows() != 3) || (H.cols() < NODS))
      H.resize(3, NODS);

   for (size_t j = 0; j < NODS; ++j)
   {
      u = pos2D(0, j);
      v = pos2D(1, j);
      x = _L * u;
      y = _L * v;
      sx = ::sin(x);  cx = ::cos(x);
      sy = ::sin(y);  cy = ::cos(y);
      s = _H *_L*_L / ::sqrt(1. + _H*_H * (sx*sx*cy*cy + cx*cx*sy*sy));
      H(0, j) = - cx*cy * s;
      H(1, j) = + sx*sy * s;
      H(2, j) = - cx*cy * s;
   }
   return 0;
}

// Data members.
double      _L, _H;

};    // surface.
```

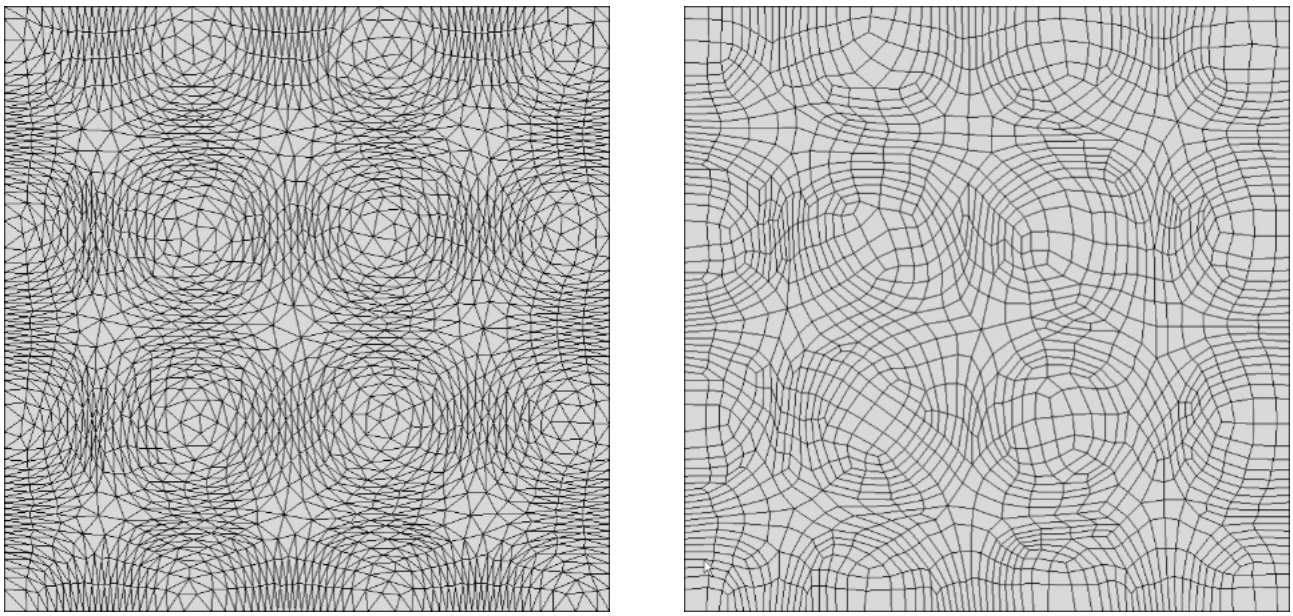


Figure 31 – 2-D anisotropic meshes in the reference space (UV).

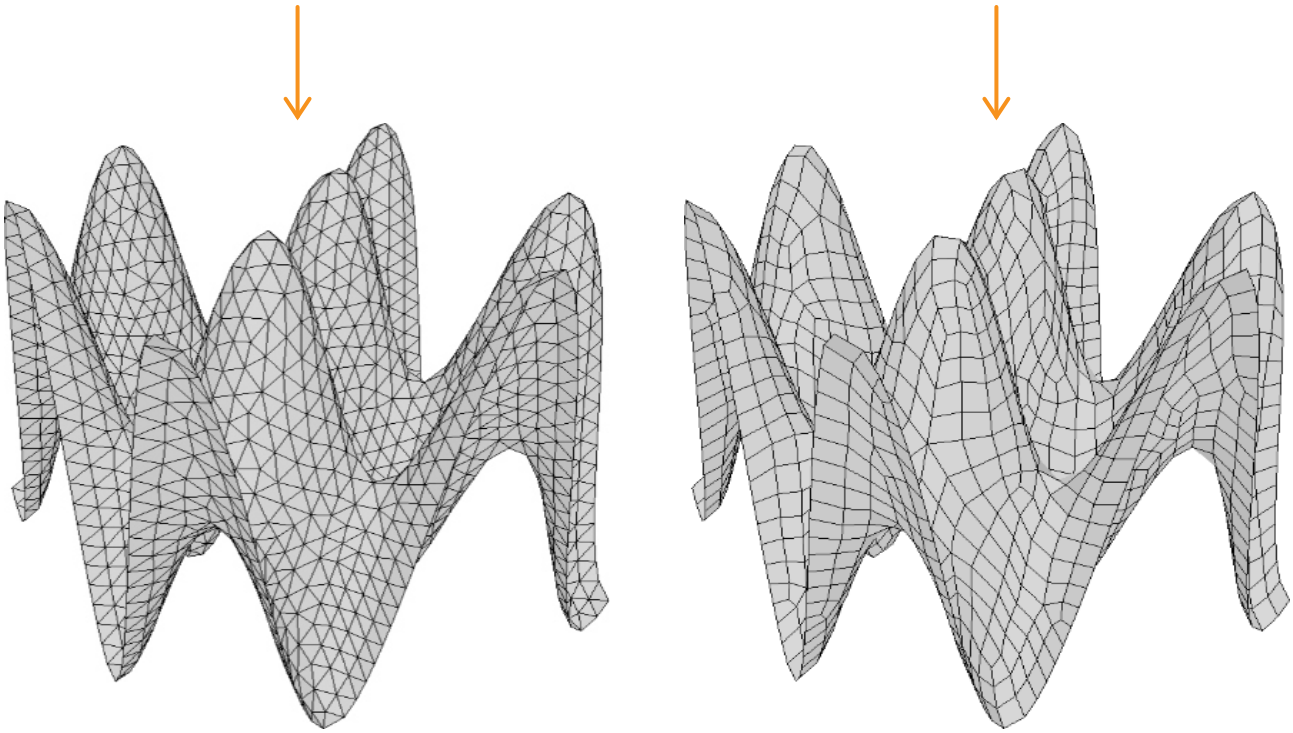


Figure 32 – 3-D surface meshes (T3 and all Q4).

Notes:

- This solution for 3-D surface meshing can be used only when a "mathematical" representation of the surface is available (through a CAD kernel for instance). This method is implemented in **CM2 SurfMesh® T3** and **CM2 SurfMesh® Q4** (based also on the OpenCascade® OCCT kernel).
  For more information, refer to **CM2 SurfMesh T3/Q4 - tutorials** and **reference manual**.
- When there is only a discrete representation of the surface available (such as a tessellated surface), a different method can be used: 3-D patch remeshing implemented in **CM2 SurfRemesh® T3** and **CM2 SurfRemesh® Q4**, two other components of the **CM2 MeshTools®** library.
  For more information, refer to **CM2 SurfRemesh T3/Q4 - tutorials** and **reference manual**.
- A similar template function (`meshtools1d::mesh_curve_param`) is available for parametric curve meshing.

COMPUTING
OBJECTS