3-D SURFACE REMESHERS

CM2 SURFREMESH T3
&
CM2 SURFREMESH Q4

Series 4.6.x

TUTORIAL

AND

USER'S MANUAL

Computing Objects

25 rue du Maréchal Foch, F-78000 Versailles.
www.computing-objects.com

Revision of manual: June 2015

Figures

Tables

This manual describes the 3-D surface remeshers of the **CM2 MeshTools** library: **CM2 SurfRemesh T3** and **CM2 SurfRemesh Q4**.

CM2 SurfRemesh T3 and CM2 SurfRemesh Q4 address the problem of regenerating a mesh on a 3-D surface defined by a discrete representation. This initial discrete representation may have elements with very high aspect ratio such as in STL meshes and even topological errors (degenerated elements, holes, overlapping elements).

These remeshers can generate a mesh finer than the initial mesh (*refinement*) or coarser (*decimation*). They can also be used to regularize a 3-D surface mesh (node-smoothing only, local node inserting, local node removal…). Finally, they can be used to find out patches of connected triangles on a 3-D surface that can be unfolded within a limited strain tolerance.
Though these remeshers usually generate automatically good meshes at first, several remeshings can be chained and sometimes a few manual optimizations can be needed on the final mesh.

CM2 SurfRemesh Q4 can generate mixed *quad-dominant* meshes (the default) or *all-quad* meshes. The former mode is recommended since it usually gives better meshes.

If the surface to be remeshed is closed, the tetrahedron mesh generator CM2 TetraMesh can be used after CM2 SurfRemesh T3 to fill the domain with tetrahedrons. The reader can read more about CM2 TetraMesh in the "3-D Mesh Generators - Tutorial and User's Manual".

Like all the other meshers of the library, CM2 SurfRemesh T3 and CM2 SurfRemesh Q4 are thread-safe (several instances or the remeshers can be ran concurrently). In addition, CM2 SurfRemesh T3 and CM2 SurfRemesh Q4 are parallelized and can use several cores to speed up the remeshings.

The `math1` library exports the vector and matrix classes needed to communicate with the meshers.
The additional libraries `meshtools`, `meshtools1D`, `meshtools2D` and `meshtools3D` can be used to generate simple 1-D and 2-D meshes for the boundaries or to do some mesh transformations (translation, rotations, concatenation, merging...)

For maximum performance, these software components are developed using the standard C++ language with efficient object-oriented programming techniques.

The full sources are available and they have been ported to most major platforms.
With a binary license, these libraries are shipped as precompiled dynamic libraries - DLL Win32 or shared Linux i386 - with .lib and C++ headers files.

The source code of CM2 MeshTools (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.001.250030.00.S.P.1999.000.20700 (16/06/1999) is regularly deposited since then.

The source code specific to CM2 SurfRemesh T3, together with this manual, has been registered with the APP under Inter Deposit number IDDN.FR.001.450010.000.S.P.2007.000.20600 (9/11/2007) and is regularly deposited since then.

The source code specific to CM2 SurfRemesh Q4, together with this manual, has been registered with the APP under Inter Deposit number IDDN.FR.001.440018.000.R.P.2008.000.20700 (31/10/2008) and is regularly deposited since then.

I – OVERVIEW OF THE MATH1 LIBRARY

Before digging into the meshers, let us have a look into the `math1` library which exports the types of vector and matrix used by the meshers. We do not intend here to give a full description of this `math1` library, nor the associated template libraries `vecscal`, `vecvec`, `matscal`, `matvec`, and `matmat`. We will simply explain the basic traits of these math classes in order to use the meshers properly. A full description can be found in the math1 API[1].

The `math1` library exports 16 types of vector, 14 types of rectangular matrix and 8 types of symmetric matrix[2].
They can be divided into two main categories: the *variable-size* containers, such as `DoubleVec`, `DoubleMat`, or `DoubleSym`, and the *fixed-size* containers, such as `DoubleVec3`, `DoubleMat2x2` or `DoubleSym2`.

☞ Following the C usage, all these math containers are *zero based*: a vector with size N extends from index 0 to N-1.

Variable-size containers and fixed-size containers

Besides the fact that only the containers of the first category can be resized - automatically or manually - they differ by the way the copy constructors and the copy operators work. The variable-size containers are merely *views* to data arrays, whereas the fixed-size containers actually hold the data as a member. The first category only holds a reference, and a copy implies only a copy of that reference, not the data (*shallow copy*). A copy of a container of the second category actually copies the data (*deep copy*).

Example:

```
DoubleVec   V1;             // Empty vector.
DoubleVec   V2(10, +1.0);   // Vector of 10 values, all intitialized to 1.0

V1.push_back (3.0)          // V1.size() == 1
V1.push_back (2.0)          // V1.size() == 2

V1 = V2;                    // Shallow copy[3] (V1.size() == 10).
                            // Previous values of V1 are lost (3.0 and 2.0).

V1[0] = 0.0;                // V2[0] == 0.0 (because the data are shared).
V2.clear();                 // V2.size() == 0 but V1.size() == 10
                            // The data are not deleted because still viewed
                            // from V1.
```

[1] *math1.html or math1.chm.*
[2] *Some other types are defined but not exported (they cannot be exchanged between dynamic libraries, only between statically linked entities): `vector_fixed<T,N>`, `matrix_fixed<T,M,N>`, `dense1D<T>`, `dense2D<T>` and `symmetric_full<T>`.*
[3] *A deep copy can be obtained with the template function `vecvec::copy` :*

```
   V1.resize (V2.size());     // Resize V1 to V2.size().
   vecvec::copy (V2, V1);     // Copy all V2 values into V1.
```

or with the "copy" member:

```
   V1.copy (V2);              // Resize V1 to V2.size() and copies the data.
```

Vectors	Rectangular matrices	Symmetric matrices
DoubleVec	DoubleMat	DoubleSymMat
FloatVec	FloatMat	FloatSymMat
IntVec	IntMat	IntSym
UIntVec	UIntMat	UIntSym
DoubleVec2		
DoubleVec3	DoubleMat3x1	DoubleSym2
DoubleVec4	DoubleMat2x2	
DoubleVec6	DoubleMat3x2 DoubleMat2x3	DoubleSym3
DoubleVec8		
DoubleVec9	DoubleMat3x3	
UIntVec2		
UIntVec3	UIntMat3x1	UIntSym2
UIntVec4	UIntMat2x2	
UIntVec6	UIntMat3x2 UIntMat2x3	UIntSym3
UIntVec8		
UIntVec9	UIntMat3x3	

Table 1 – Vectors and matrices exported by the `math1` library.

Views of the variable-size containers

Several variable-sized containers can have a view on the same array of data, but the view can be different from each other. The beginning and the size in the array are specific to each container. For instance in an array of 30 items, a first vector views items from 0 to 9 and a second one views items from 5 to 20.

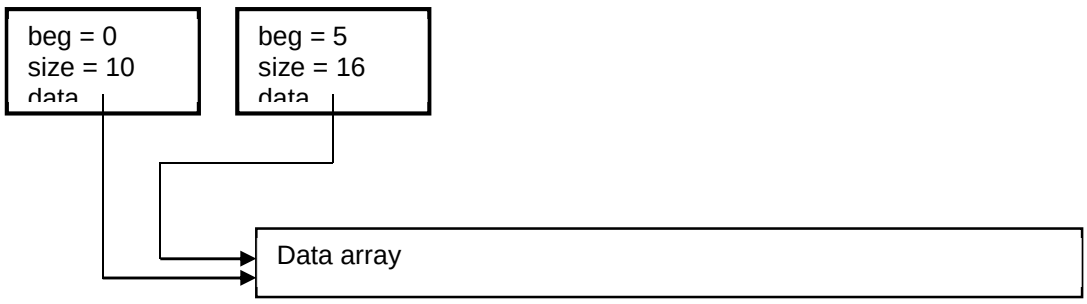


Figure 1 – Views and data in variable-sized vectors.

Elements from 5 to 9 are accessible from the two vectors.
When a destructor is called on a variable-size container, the data are destroyed only when no other container shares them anymore. A *smart pointer* mechanism is used to count the number of references on the data and the deallocation actually occurs when the count reaches null. The memory management is automatic (automatic garbage collection).

Example:

```
DoubleVec  *V1 = new DoubleVec(10, -1.0);
DoubleVec  *V2 = new DoubleVec(*V1);    // Shallow copy (share the data).

delete V1;   // The data are still referenced by V2.
delete V2;   // Now the data are destroyed too.
```

Fixed-size containers

The fixed-size math containers are deep-copy containers. The copy-constructor and the copy-operator do not make the data to be shared anymore but leads to actually different arrays in memory. They are simpler than the variable-size containers, and faster for short arrays, whereas the variable-size containers are more suited for big arrays.

Example:

```
DoubleVec2   V1;           // Vector of 2 unintitialized values (double).
DoubleVec2   V2(+1.0);     // Vector of 2 values intitialized to 1.0

V1[0] = 0.0;
V1[1] = -1.0;

V1 = V2;                   // Deep copy: V1 = {1, 1}, V2 = {1, 1}.
V2[0] = 0.0;               // V1 = {1, 1}, V2 = {0, 1}
V1[1] = 0.0;               // V1 = {1, 0}, V2 = {0, 1}
```

STL-like iterators and the generic math library

All these math containers are optimized for numerical computation.
The vector containers - variable-size and fixed-size - are equipped with STL-like iterators `begin()` and `end()`, to make them compatible with most of the STL algorithms. They also have access operators such as `operator[]`, and the usual functions for a vector class: `size()`, `empty()`, `front()`, `back()`...
The variable-size vectors are also equipped with members such as `reserve`, `resize`, `push_back`, and `pop_back`.
Aside from the STL algorithms, one can also use the math-specific template functions of the `vecscal`, `vecvec`, `matscal`, `matvec` and `matmat` libraries (cf. math1 API).

Example:

```
DoubleVec    V1(3), V2(3,-1.0);
DoubleMat    M(3, 10, 0.0);       // Matrix of doubles 3 by 10 set to 0.

vecvec::copy (V2, V1);            // Hard copy of V2 into V1 (sizes match).
vecvec::axpy (2.0, V1, V2);       // V2 += 2 * V1
vecscal::mult (-1.0, V1);         // V1 = -V1
```

The variable-size matrix types (`DoubleMat`, `IntMat` and `UIntMat`) have a behavior similar to the vectors with respect to the memory management and the copy operators (and the copy-constructors). In addition, the rows and the columns are equipped with iterators, just like the vectors, and the same template functions can be used on them.

Example:

```
const unsigned N  = 10;
const double   PI = 3.14159;
DoubleMat      pos(2, N);         // Uninitialized 2xN matrix.
DoubleVec2     V;                 // Uninitialized vector of size 2.

// Points on a circle.
for (unsigned j = 0; j < N; ++j)
```



```
{
   pos(0,j) = ::cos(j*2*PI/N);
   pos(1,j) = ::sin(j*2*PI/N);
}

// Set radius to 3.0
matscal::mult (3.0, pos);

// Copy column #2 to V (ok, dimensions match).
vecvec::copy (pos.col(2), V);

// Copy V to column #9 (the last one).
vecvec::copy (V, pos.col(9));

// Copy column #9 to column #0
vecvec::copy (pos.col(9), pos.col(0));

// Append V in a new column of pos (dimensions match).
pos.push_back (V);              // pos.cols() == 11 after this line.
```

Bound checking

In debug mode (with macro `_DEBUG` defined), bound violations abort the program. In release mode however, for best performance no check is performed and the user must take care not to out value the limits of the vectors and matrices.

Interoperability with other math containers

The API of the MeshTools library use exclusively vectors and matrices of the math1 library (such as `DoubleMat`, `UIntMat`, `FloatVec`...). To use the meshers with other types of vectors and matrices, the variable-size containers are equipped with constructors with raw pointers as arguments. Hence, they can view the data in any other math containers as long as the latter provide a way to get a raw pointer to their data, and that these data are *contiguous* in memory.
Remember that the variable-size containers implement *shallow copies*. This means that the arrays are shared, not copied. Therefore the memory management becomes a point to take care of and the user must keep in mind which library is responsible for deleting the memory upon exit. The default is that the allocator of the array remains responsible for its deletion.

Example:

```
double               buff1[100];                // Static raw C array.
double*              buff2 = new double[100];   // Dynamic raw C array.
std::vector<double>  buff3(100);                // STL vector.

DoubleVec   V1(50, buff1);         // Views the first 50 elements in buff1.
DoubleVec   V2(50, buff2);         // Views the first 50 elements in buff2.
DoubleVec   V3(50, buff3.begin()); // Views the first 50 elements in buff3.

V2.clear();          // buff2 is not deallocated.
V2 = V1;             // buff2 is not deallocated.
V3.clear();          // buff3 is not deallocated.
delete[] buff2;      // Dangerous but correct because buff2 is no longer
                     // referenced by any DoubleVec.
buff3.clear();       // Dangerous: data in buff3 may have been deallocated.
                     // Don't use V3 anymore.
```

Note that this can be done with the fixed-size containers of math1:

```
DoubleVec3  V3(1.0, 0.0, -1.0);
DoubleVec   V(3, V1.begin());     // Views the elements in V3.
```

The matrices of math1 can view also the data in an external container. In addition to the raw pointer to the data, the user must provide the number of rows, the number of columns and the stride between two columns (so-called *leading* dimension):

```
unsigned*   buff = new unsigned[30];
UIntMat     M(3, 10, /*ld=>*/ 3, buff);
```

As before, the matrix is not responsible for the deletion of the underlying buffer[4].

In the case where a container constructed this way is subsequently resized, it may "point" to another array of memory but the initial buffer remains valid:

```
double*     buff = new double[6];
DoubleVec   V(5, buff);

V.push_back (2.0);   // Reallocation and copy performed.
                     // buff is still alive, but V does not "point" to
                     // it anymore.
V.clear_hard();      // The new array of V is deallocated, not buff.
```

☞ As a rule of thumb, the lifetime of the external buffer must span the lifetime of the math1 container.

```
double*     buff = new double[6];

{
   DoubleVec    V(5, buff);
   ...    // Use V here.
}  // V is killed here but the buffer is spared.

delete[] buff;  // So long with buff.
```

We have seen how to construct math1 variable-size container upon other containers or buffers. To do the other way, we use the `data()` or `begin()` members to access the underlying data:

Example:

```
DoubleVec     V(50, 0.0);
double*       buff = V.data();
unsigned      N    = V.size();     // Equals to 50

for (unsigned i = 0; i < N; ++i, ++buff)
   *buff = double(i);

assert (V[10] == 10.);            // Changes in buff have been seen in V.
```

```
DoubleMat     P(3, 40);
double*       buff = P.data();
unsigned      M    = P.rows();     // Equals to 3
unsigned      N    = P.cols();     // Equals to 40
unsigned      LD   = P.ld();       // Equals to 3 (here stride == rows).
```

```
for (unsigned j = 0; j < N; ++j)
   for (unsigned i = 0; i < M; ++i)
      buff[i + j*LD] = double(i + j*LD);
```

[4] *A default parameter "protect" in the constructors can be used to change this behaviour.*


```
assert (P(0,10) == 30.);           // Changes in buff have been seen in P.
```

Here, the math1 vectors and matrices are responsible for the deletion of their data:

```
DoubleMat     P(3, 40);
double*       buff = P.data();

delete[] buff;       // Don't do that !
P.resize (3, 80);    // Crash now, or maybe later...
```

II – TUTORIAL

This chapter shows examples of surface remeshings illustrating along the way some of the major options of the remeshers.

Each example starts with including the file `stdafx.h` (can be a precompiled header), giving access to the classes and the routines of the libraries (API).

The main namespace `cm2` has nested namespaces such as `vecscal`, `vecvec`, `meshtools` or `surf_remesh_t3`. The user can add "using namespace" directives in this `stdafx.h`. Keeping namespaces in the user's source code can however be useful to improve the legibility and to avoid name conflicts.

File "stdafx.h":

```
// MESHTOOLS
#include "meshtools.h"      // General purpose mesh routines.

// REMESHERS
#include "surf_remesh_t3.h" // 3-D surface triangle remesher.
#include "surf_remesh_q4.h" // 3-D surface quadrangle remesher.
```

Required libraries[5]:

```
cm2math1
cm2misc
cm2meshtools
cm2meshtools1d
cm2meshtools2d
cm2surf_remesh_t3
cm2surf_remesh_q4
```

[5] *On Windows, the lib names end with `_Win32_45` or `_x64_45`. For instance `cm2math1_x64_45.dll`.*
On Windows, file extensions for the librariess are .lib and .dll. On Linux/Unix/Mac OS platforms, file extensions are usually .a (static archive), .so or .dylib (dynamic lib).

II-1 GETTING STARTED – DEFAULT MODE REMESHING

This first example is a regular cylinder with circular base. The diameter and the height of this cylinder are 100. The input mesh is a STL-mesh provided by a CAD modeler (Figure 2).

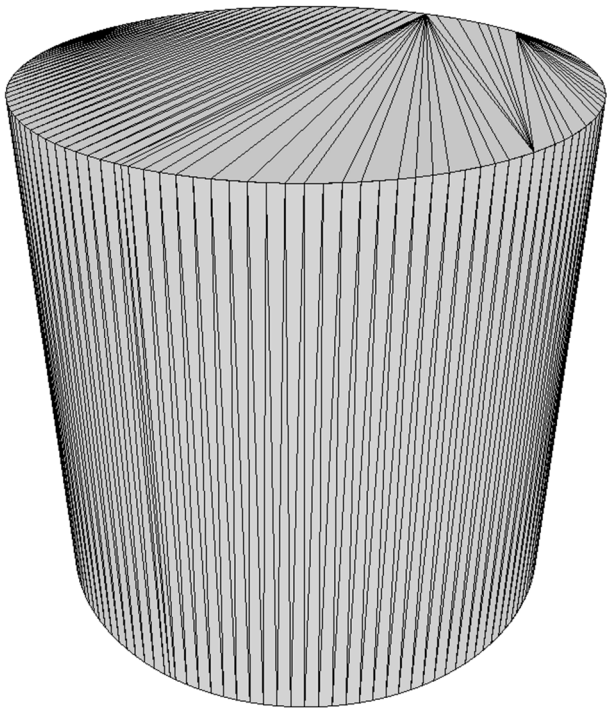


Figure 2 – Initial mesh of the cylinder (H = 100, D = 100).

Many triangles have high aspect ratio making this initial mesh not suited for FEM computations for instance. The program to generate a new better mesh on this tessellated surface is shown below. Figure 3 shows the resulting mesh.

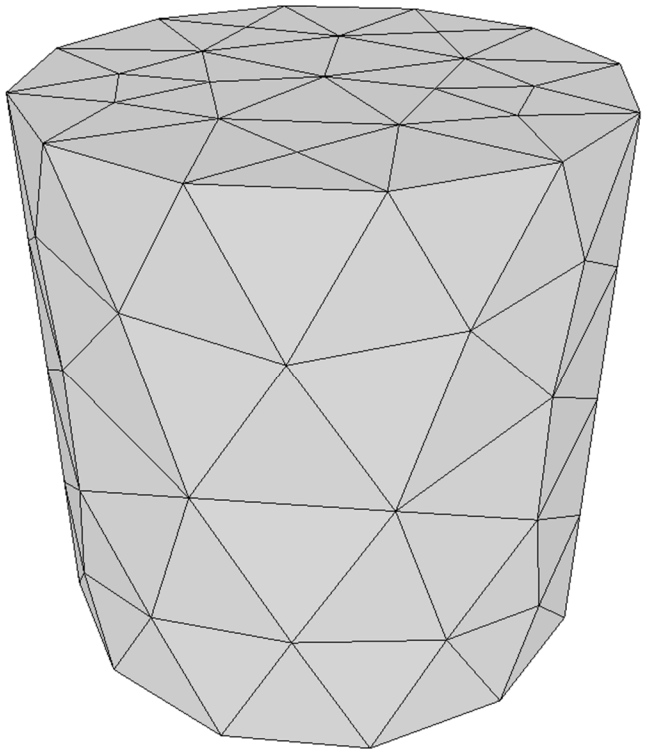


Figure 3 – The cylinder remeshed with default mode.

```
#include "stdafx.h"

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


int main()
{
   surf_remesh_t3::mesher               the_remesher;
   surf_remesh_t3::mesher::data_type    data;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH (STL FORMAT).
   meshtools::STL_input ("cylinder.stl", data.pos, data.connectM);

   // REMESH THE SURFACE.
   the_remesher.run (data);            // Remesh the surface.

   // 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
```

Let us explain this program line by line.

Some declarations

`the_remesher` is an instance of remesher. `data` is a key structure for the remesher. It stores, among others, the matrix `pos` which contains all the points' coordinates and the matrix `connectM` which is the connectivity matrix of the mesh.

`data.pos` stores all the nodes involved in the meshing process: the nodes of the initial mesh and the nodes of the final mesh (some of them may be in both). It is stored as a `DoubleMat` matrix (variable-sized matrix of doubles).

`data.connectM` stores the connectivity matrix of the meshes (first of the initial mesh then of the final mesh after remeshing) as `UIntMat` matrices (natural integers). `connectM(i,j)` is the i^{th} local node of the j^{th} initial element. This integer is the column number in matrix `pos` where the coordinates of this node can be found[6].

Authorization of the library

The library needs to be unlocked through a call to `surf_remesh_t3::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 - contact license@computing-objects.com for any licensing enquiry. Note that both strings are case sensitive and the registration call must be made each time the library is loaded into memory and *before* any run of the remesher.

```
surf_remesh_t3::registration ("Licensed to SMART Inc.", "F5BEA10ABCWX");
```

The initial mesh

In this example, the initial mesh is read from a STL file (ASCII). The `meshtools::STL_input` function reads the points' coordinates, the node's indices of the triangles and merge together any coincident nodes.

```
meshtools::STL_input ("cylinder.stl", data.pos, data.connectM);
```

The new mesh

```
the_remesher.run (data);
```

Upon exit, the matrix `data.pos` is bigger and contains the new nodes generated on the surface by the remesher. These new points are appended to the original matrix. The initial points of the input mesh are unchanged. The connectivity of the final mesh is stored in the matrix `data.connectM`, each column storing the indices of the nodes for an element.

Output information

Printed information about the generated mesh and a MEDIT[7] output file are obtained with:

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

Figure 4 shows the output of `data.print_info (&display_hdl)`:

The new mesh has 98 nodes and 192 triangles and its quality is much better. In the initial mesh, the average quality was 0.042. In the final mesh, it is 0.88.

The `max_error_distance` is a measure of the geometric error between the two meshes. The nodes of the new mesh are located exactly onto the initial surface. It is not always the case however for the centroïd of the new elements or for the initial nodes. If the surface is not flat, the new mesh may indeed be some distance away from it. The `max_error_distance` is a measure of this maximum distance (see § III-1).

---

[6] We recall that all matrix and vector indices are zero based (from 0 to N-1).[7] *MEDIT is a free vizualisation program (http://www.ann.jussieu.fr/~frey/logiciels/medit.html). Other output formats are: VTK, Ensight, STL, FEMAP neutral, Nastran and Wavefront OBJ.*

[7] MEDIT is a free vizualisation program (http://www.ann.jussieu.fr/~frey/logiciels/medit.html). Other output formats are: VTK, Ensight, STL, FEMAP neutral, Nastran and Wavefront OBJ.

The times spent in the several steps of the meshing process are given in seconds. The first step is for cleaning the initial mesh (node merging and edges swapping).
In the second step, the mesh is analyzed to find out groups of connected triangles (called *patches*) that can be remeshed individually. The boundaries of the patches are located where the angle between two adjacent triangles is greater than a prescribed value[8] (parameter `patch_angle_tolerance` – see § III-3). The boundaries of these patches are called the *skeleton lines*. The singular nodes among the skeleton lines (nodes connected to one or three or more skeleton lines) are called *skeleton nodes*. The skeleton lines and the patches are remeshed during the third step. The skeleton nodes remain present in the new mesh.
Finally, geometrical and topological optimizations are performed to improve the quality of the elements.

[8] For the sake of clarity, we can consider that the angle between adjacent triangles is the only parameter used by the algorithm (it is indeed not the only one, but the most important).

```
**************************************
********** CM2 SURFREMESH T3 **********
**************************************
Version         : 4.6.0.0
Initial Mesh
  Nodes           : 256
  Triangles       : 508
  Surface         : 4.711434E+004
  Qavg            : 4.163685E-002
  Qmin            : 2.128816E-002
New Mesh
  Nodes           : 98
  Triangles       : 192
  Surface         : 4.613917E+004
  Qavg            : 8.838215E-001
  Qmin            : 7.037609E-001
Max error distance : 1.758797E+000
Cleaning time      : 0.00 s
Analysis time      : 0.04 s
Remesh time        : 0.01 s
Optim time         : 0.00 s
Total time         : 0.05 s (3997.00 t/s)

************ HISTOGRAM QS ************
Total number of bins     :           11
Total number of counts   :          192
Number of larger values  :            0
Number of smaller values :            0
V max                    : 9.987371E-001
V mean                   : 8.838215E-001
V min                    : 7.037609E-001

Bin number        -- Bin boundaries --         Hits

   10            0.90             1.00          112
    9            0.80             0.90           32
    8            0.70             0.80           48
    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 4 – Information output for the cylinder example[9].

[9] *All runs are done on Windows 8 x64 with Intel Xeon E3 V2 1270 3.5 GHz, 1 thread, turbo boost disabled.*

Note that the initial connectivity matrix is overwritten by the new one in matrix `data.connectM` and some nodes in `data.pos` are no longer referenced[10]. To keep the initial mesh, the following code should be used instead:

```
#include "stdafx.h"

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


int main()
{
   surf_remesh_t3::mesher               the_remesher;
   surf_remesh_t3::mesher::data_type    data;
   UIntMat                              connectMi;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH (STL FORMAT).
   meshtools::STL_input ("cylinder.stl", data.pos, data.connectM);
   connectMi.copy (data.connectM);     // Hard copy.

   // REMESH THE SURFACE.
   the_remesher.run (data);            // Remesh the surface.

   // 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 initial mesh is in `connectMi` and the new mesh in `data.connectM`. Indices in both matrices refer to columns in `data.pos` (containing nodes of the old and the new mesh).

We can also shallow-copy the new mesh:

```
...// REMESH THE SURFACE.
   the_remesher.run (data);               // Remesh the surface.
   UIntMat    connectMr (data.connectM);  // Shallow copy.

...
```

Another way to do this could be:

```
#include "stdafx.h"

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


int main()
{
```

[10]. *You can get ride of the nodes unused anymore with `meshtools::simplify`. Please refer to the Reference Manual of CM2 MeshTools (HTML doc).*

```
   surf_remesh_t3::mesher               the_remesher;
   surf_remesh_t3::mesher::data_type    data;
   DoubleMat                            pos;
   UIntMat                              connectMi, connectMr;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH (STL FORMAT).
   meshtools::STL_input ("cylinder.stl", pos, connectMi);

   // REMESH THE SURFACE.
   data.pos = pos;                      // Shallow copy.
   data.connectM.copy (connectMi);      // Hard copy.
   the_remesher.run (data);             // Remesh the surface.
   pos = data.pos;                      // Shallow copy.
   connectMr = data.connectM;           // Shallow copy.

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

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

   return 0;
}   // main
```

In the following example (called "part #1"), the skeleton lines found by the algorithm are the natural ridges of the solid (and some lines inside the circular holes). Hence, most of the skeleton nodes are the corners of the solid.
Here also, we do not specify any size value for the new mesh. We let the remesher compute default values based on the distance between each skeleton node.

Concerning the program, the main difference with the previous one concerns the reading of the initial mesh. The coordinates and the connectivity matrices are no longer in a STL format file, but are read with `matio::transpose_read` from a "matrix-formatted" file.
Here the expected format for the matrices is:

n X *m* [
$d_{0,0}$ $d_{0,1}$ $d_{0,2}$ ... $d_{0,m-1}$
$d_{1,0}$ $d_{1,1}$ $d_{1,2}$ ... $d_{1,m-1}$
...
$d_{n-1,0}$ $d_{n-1,1}$ $d_{n-1,2}$... $d_{n-1,m-1}$]

The format for each component of the matrix is free.
For instance for the coordinate matrix (4 nodes, 3 coordinates per node):

```
4 X 3 [
  0 0.5 2.0
  0 1 1
  1E-3 –7 1.0
  3.4 –2.1 –1.0 ]
```

```
#include "stdafx.h"

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


int main()
{
   surf_remesh_t3::mesher               the_remesher;
   surf_remesh_t3::mesher::data_type    data;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH.
   ifstream       istrm (“part1.dat”);
   matio::transpose_read (istrm, data.pos);
   matio::transpose_read (istrm, data.connectM);

   // REMESHING THE SURFACE.
   the_remesher.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
```

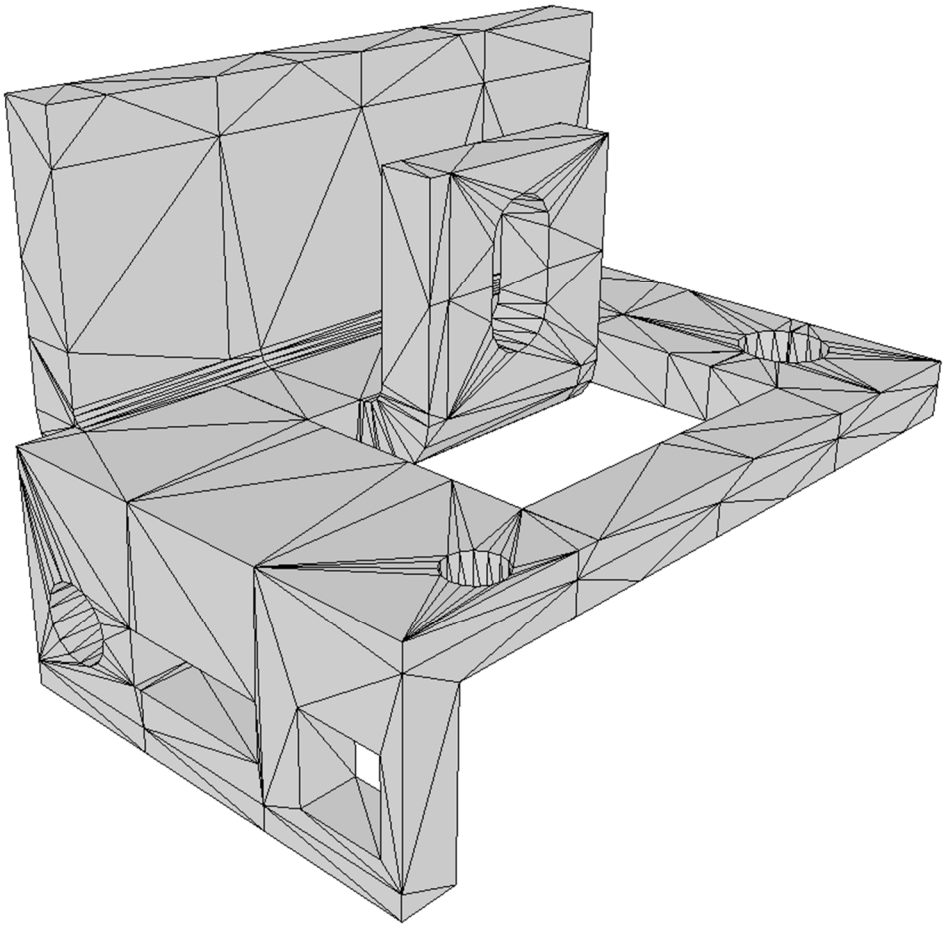



Figure 5 – Part #1 - Initial mesh
(bounding box X: 0.10, Y: 0.10, Z: 0.15).

```
Initial Mesh
  Nodes             : 379
  Triangles         : 782
  Surface           : 4.428174E-002
  Qavg              : 3.987199E-001
  Qmin              : 9.139058E-004
```

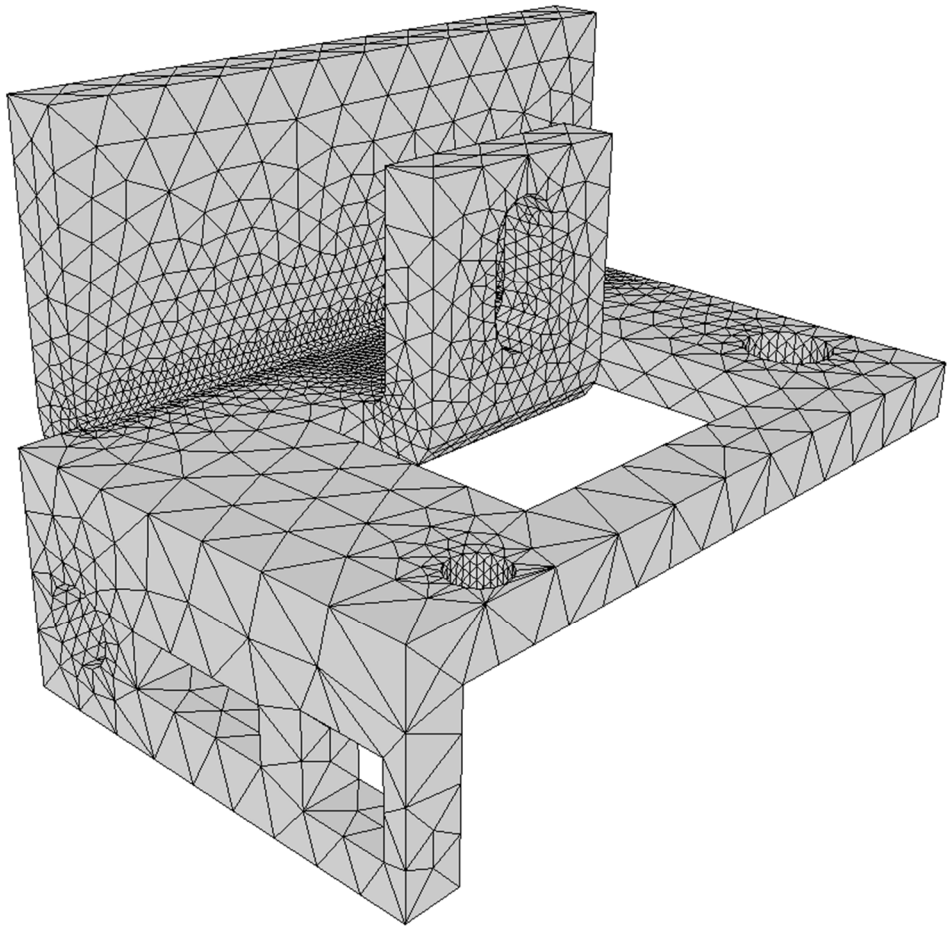


Figure 6 – Part #1 remeshed with default mode

```
New Mesh
  Nodes             : 3051
  Triangles         : 6126
  Surface           : 4.427879E-002
  Qavg              : 8.308896E-001
  Qmin              : 1.598984E-001

Max error distance : 1.947421E-004
```

In this example where most of the patches are flat, there is a very small geometric error between the two meshes.

Note also the variations of the elements' size in the new mesh. It is a typical result in the default mode of the surface remesher. The next section will address this point.

II-2 LIMITING THE RANGE OF THE ELEMENTS' SIZE

Usually, the default elements' size does not give satisfactory meshes. They are either too fine or too coarse. For this matter, the user can limit the range of the elements' size field with two parameters of the operating mode: `min_h` and `max_h`. In the default mode `min_h` = 0 and `max_h` = `DBL_MAX`.

Let us use again the cylinder example. We now set the two bounds to the same value: 10.
This will give a uniform mesh in which all elements' edges have a length of 10 on average.

```
#include "stdafx.h"

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


int main()
{
   surf_remesh_t3::mesher               the_remesher;
   surf_remesh_t3::mesher::data_type    data;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH.
   ifstream       istrm (“part1.dat”);
   matio::transpose_read (istrm, data.pos);
   matio::transpose_read (istrm, data.connectM);

   // REMESHING THE SURFACE.
   the_remesher.mode.min_h = 10.0;
   the_remesher.mode.max_h = 10.0;
   the_remesher.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
```

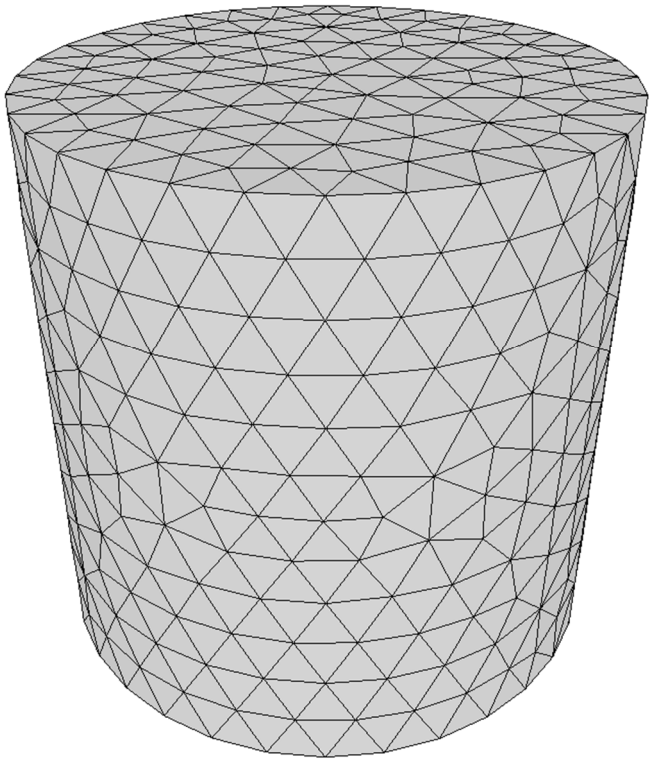


Figure 7 – Cylinder remeshed with min_h = 10 and max_h = 10

```
Initial Mesh
  Nodes         : 256
  Triangles     : 508
  Area          : 4.711434E+004
  Qavg          : 4.163685E-002
  Qmin          : 2.128816E-002
New Mesh
  Nodes         : 554
  Triangles     : 1104
  Area          : 4.697452E+004
  Qavg          : 9.381782E-001
  Qmin          : 7.172157E-001
Max error dist. : 2.986012E-001
```

The values min_h = max_h = 5 give the mesh shown in Figure 8.

Note that the geometric error between the initial STL mesh and the new mesh decreases as it gets finer.

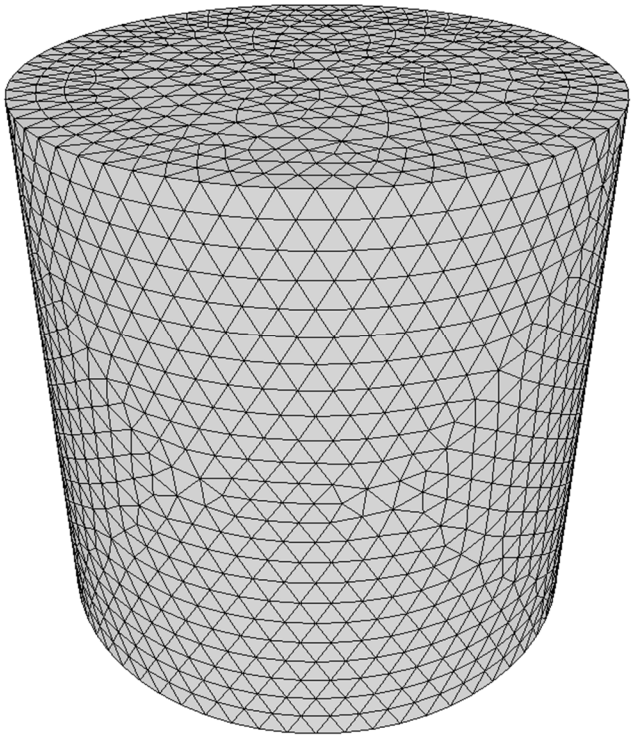


Figure 8 – Cylinder remeshed with min_h = 5 and max_h = 5

```
Initial Mesh
  Nodes          : 256
  Triangles      : 508
  Area           : 4.711434E+004
  Qavg           : 4.163685E-002
  Qmin           : 2.128816E-002
New Mesh
  Nodes          : 2172
  Triangles      : 4340
  Area           : 4.708151E+004
  Qavg           : 9.547583E-001
  Qmin           : 7.996002E-001
Max error dist. : 6.471375E-002
```

Below are two resulting meshes for the "Part #1" example with different values for `min_h` and `max_h`:

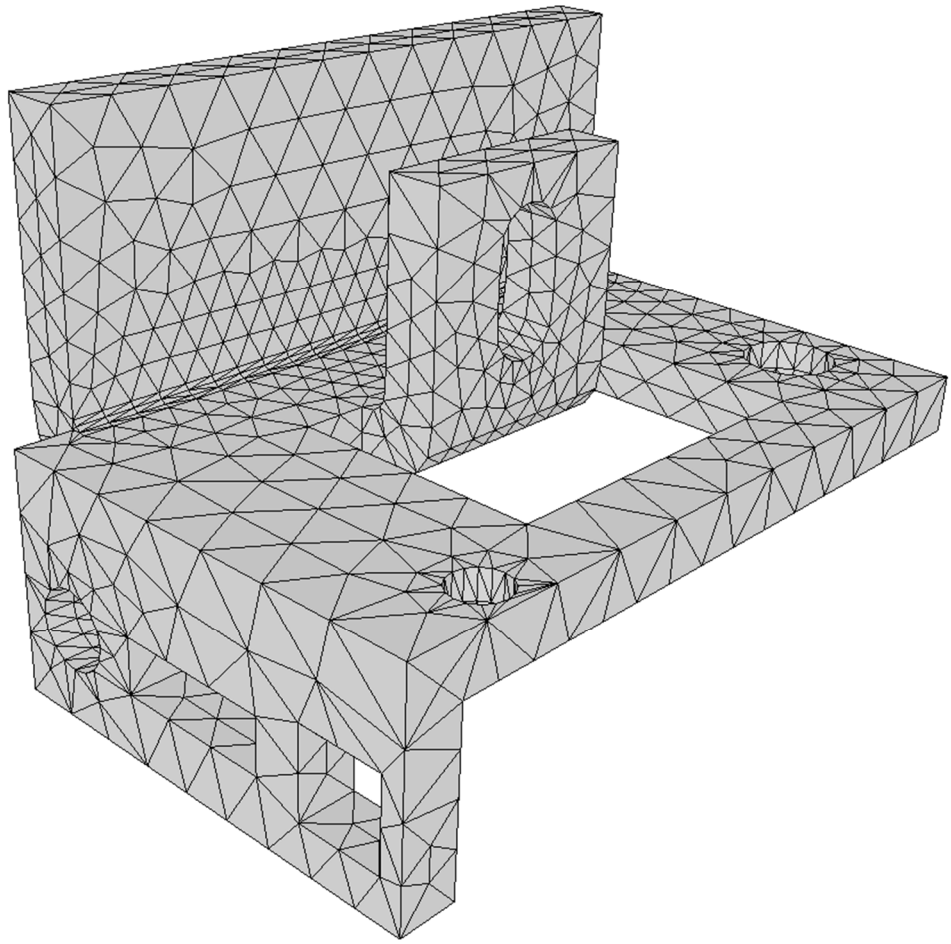


Figure 9 – Part #1 remeshed
with min_h = 0.005 and max_h = DBL_MAX

```
Initial Mesh
  Nodes          : 379
  Triangles      : 782
  Area           : 4.428174E-002
  Qavg           : 3.987199E-001
  Qmin           : 9.139058E-004
New Mesh
  Nodes          : 1290
  Triangles      : 2604
  Area           : 4.427485E-002
  Qavg           : 7.823978E-001
  Qmin           : 1.598984E-001
Max error dist. : 4.184950E-004
```

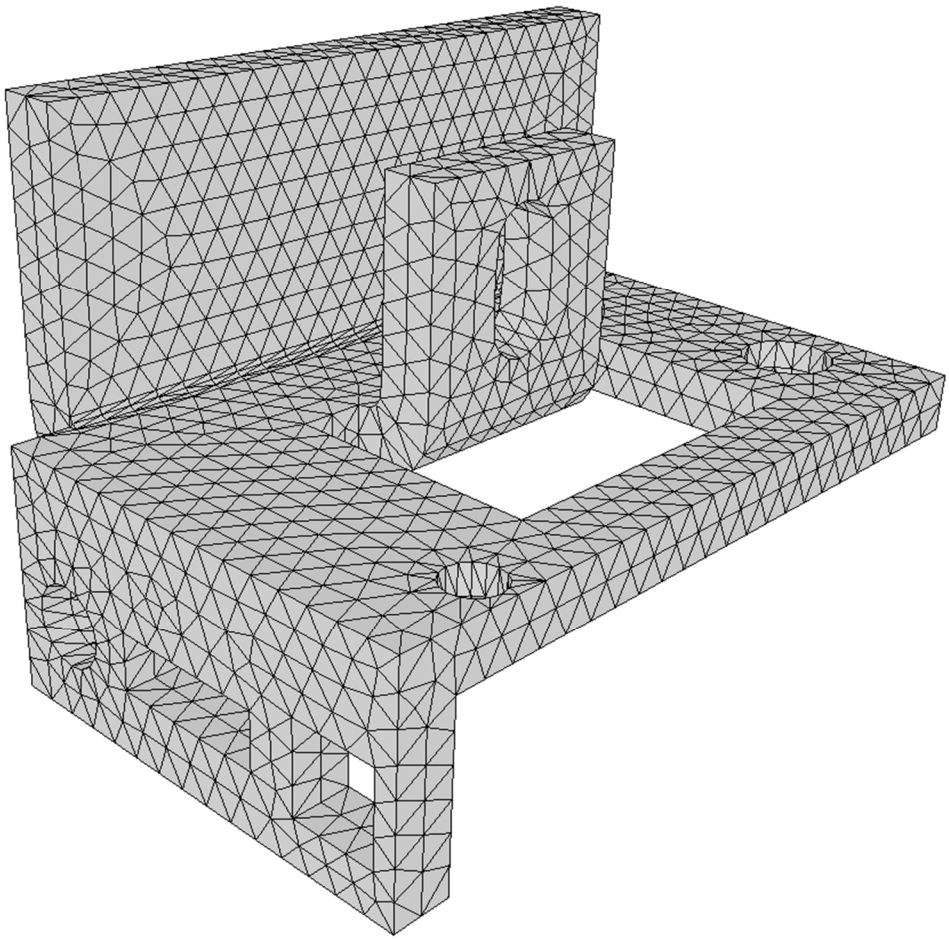



Figure 10 – Part #1 remeshed
with min_h = 0.005 and max_h = 0.005

```
Initial Mesh
  Nodes         : 379
  Triangles     : 782
  Area          : 4.428174E-002
  Qavg          : 3.987199E-001
  Qmin          : 9.139058E-004
New Mesh
  Nodes         : 2136
  Triangles     : 4296
  Area          : 4.428039E-002
  Qavg          : 8.711093E-001
  Qmin          : 2.734789E-001
Max error dist. : 2.760341E-004
```

Note that in this last example where `min_h` = `max_h`, most of the domain is meshed uniformly. But due to the constraints on patches it is not the case everywhere. Some areas – such as the high-curvature zones - remain finer.

II-3 USER'S SPECIFIED PUNCTUAL SIZES

Until now, we have seen two fields in the structure used to exchange data with the mesher:

- The `pos` matrix for the points' coordinates.
- The `connectM` matrix for the connectivity of the 3-D meshes (same matrix used for the initial mesh and for the final mesh).

We have seen also the effects of two global parameters of the operating mode: `min_h` and `max_h`. In a more specific way, the user can set elements' size values at some nodes of the initial mesh. This is done using the `metrics` field of the data structure.
To illustrate this feature, let us use again the "part #1" example with min_h = 0.05 and max_h =0.05. We now specify a size value of 0.001 at nodes #68 and #79[11]. At these nodes, these values supersede the operating mode bounds `min_h` and `max_h` and the new mesh follows locally these sizes.

```
#include "stdafx.h"

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


int main()
{
   surf_remesh_t3::mesher               the_remesher;
   surf_remesh_t3::mesher::data_type    data;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH.
   ifstream       istrm (“part1.dat”);
   matio::transpose_read (istrm, data.pos);
   matio::transpose_read (istrm, data.connectM);

   // REMESHING THE SURFACE.
   the_remesher.mode.min_h = 0.005;
   the_remesher.mode.max_h = 0.005;
   data.metrics.resize (pos.cols(), 0.0);
   data.metrics[68] = 0.001;
   data.metrics[79] = 0.001;
   the_remesher.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 result is a new mesh with uniform size of 0.005 everywhere on the surface except near these four nodes where the elements' size drops to 0.001.

[11] *The node IDs can also be found with the help of meshtools::node_detector. This class can find the closest node to a point, or all nodes inside a box.*

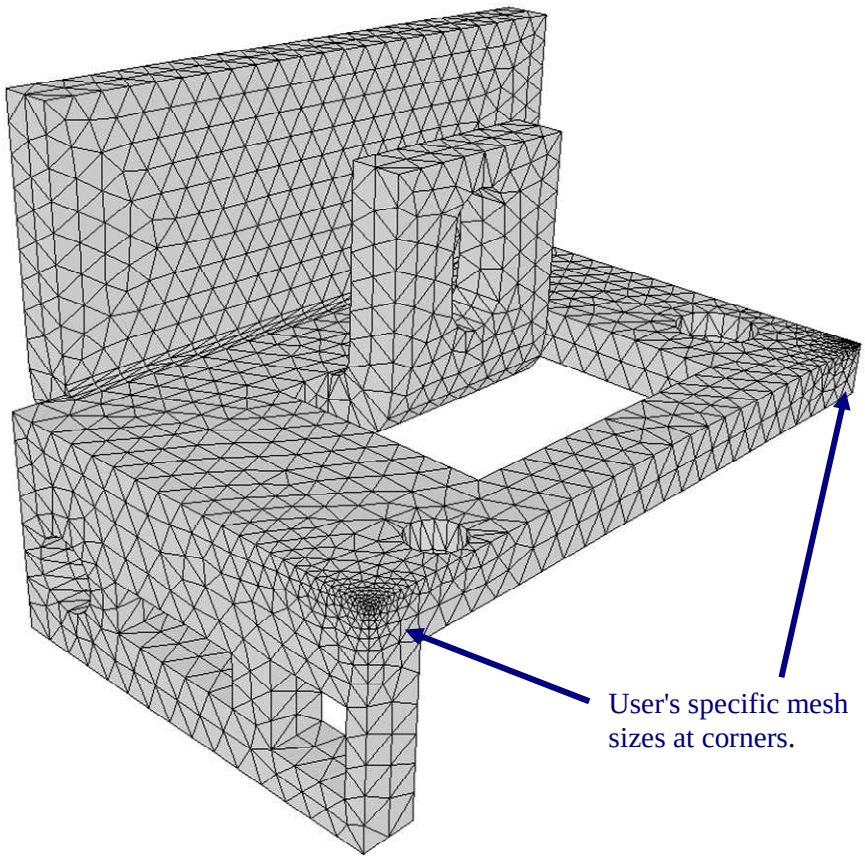


Figure 11 – Part #1 remeshed with h = 0.001 at four nodes (min_h = 0.005, max_h = 0.005)

Specific mesh sizes can be set on any node of the initial mesh. In the following example, we set a sinus variation in `data.metrics` for all initial nodes (we used mesh at Figure 10 as initial mesh):

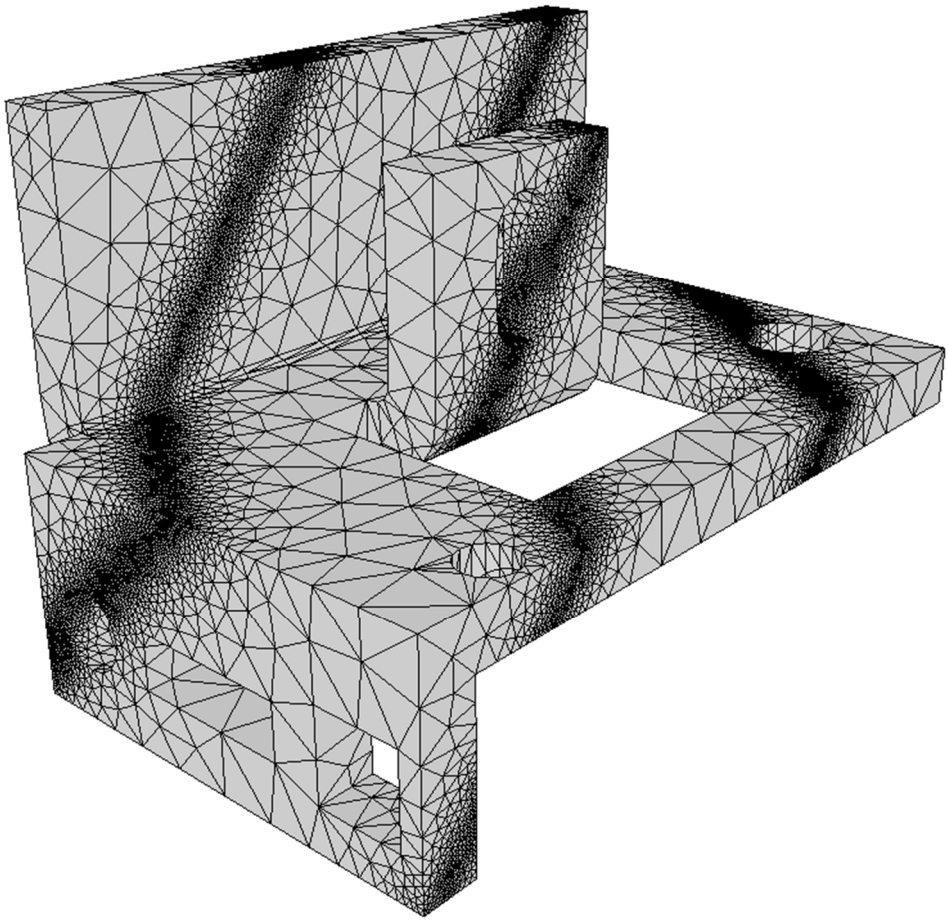


Figure 12 – Part #1 remeshed with user metrics on all initial nodes.

Hence, surface meshes can be coarsened in some areas and refined in some others.

II-4 PATCHES

The algorithm of the remeshers has three major steps. First the initial mesh is split into patches/subdomains (groups of connected triangles), second the patch boundaries (called "skeleton lines" or "skeleton edges") are remeshed, and finally the patches are remeshed one by one. This third step is divided into three sub steps: each patch of the initial mesh (called an *initial* patch) is unfolded to the Z = 0 plane, remeshed into a new patch and then mapped back to the initial surface (called then a *final* patch). Similarly, we define initial skeleton lines and final skeleton lines.

The algorithm to find out the patches is based mostly on the angle between adjacent triangles. Starting from some element seeds, the patches are grown by neighboring traversal until the angle between two adjacent triangles is greater than a fixed parameter (`patch_angle_tolerance` in the remeshers' operating mode)[12] or when a user-specified edge is encountered.
The value of `patch_angle_tolerance` is relevant when the 3-D surface has soft angles. When this value is low, the patches tend to be small and the final mesh is close to the initial mesh (small geometric error). The remesher can generate better meshes when the patches are big. This is the case when the angle tolerance is high. The following example illustrates this point.

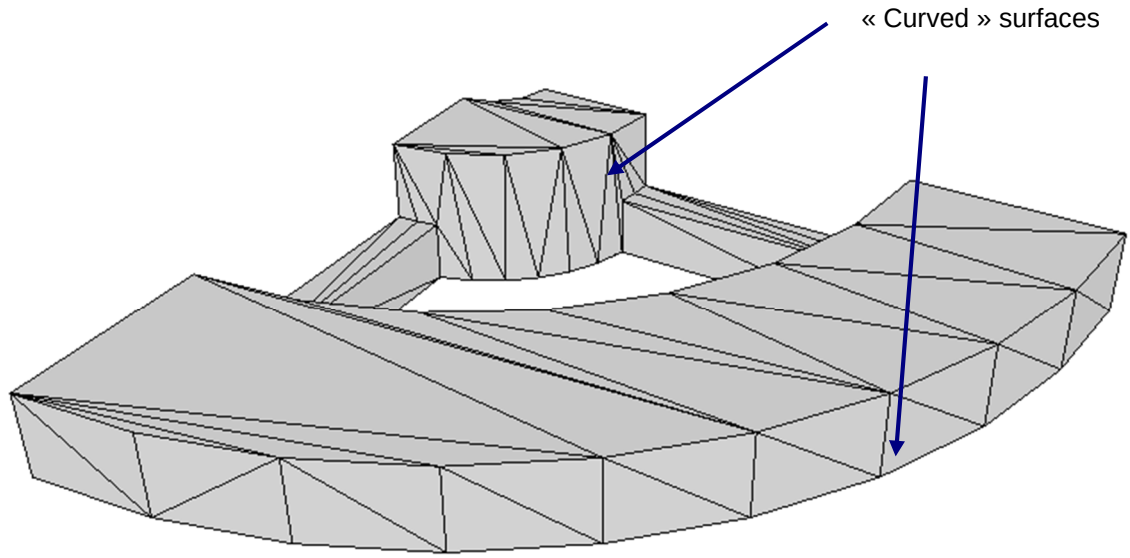


Figure 13 – Part #2 - Initial mesh

```
Initial Mesh
  Nodes          : 86
  Triangles      : 172
  Area           : 5.352784E+004
  Qavg           : 4.778100E-001
  Qmin           : 1.452030E-002
```

We first set the value of `patch_angle_tolerance` to 5 degrees:

[12] *The strain induced by the unfolding of each patch must also be lesser than `mode.strain_tolerance` (see § III.3). Other special conditions can also stop the patch growth that are beyond the scope of this manual. We can consider that only `patch_angle_tolerance` drives the algorithm.*

```
#include "stdafx.h"

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


int main()
{
   surf_remesh_t3::mesher              the_remesher;
   surf_remesh_t3::mesher::data_type   data;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH.
   ifstream       istrm (“part1.dat”);
   matio::transpose_read (istrm, data.pos);
   matio::transpose_read (istrm, data.connectM);

   // REMESHING THE SURFACE.
   the_remesher.mode.min_h = 10.0;
   the_remesher.mode.max_h = 10.0;
   the_remesher.mode.patch_angle_tolerance = 5.0;
   the_remesher.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 new mesh is close to the initial mesh. Many angles of the initial mesh remain in the new mesh (Figure 15).

With `patch_angle_tolerance` = 20, the patches are bigger. This gives a better and smoother new mesh at the cost of a higher geometric error between the two meshes, i.e. chordal error (Figure 17).

With `patch_angle_tolerance` = 0, the initial mesh is divided into pure flat patches (can be limited to only one element). In this case, there is no geometric error between the initial mesh and the new mesh.

The default value `patch_angle_tolerance` = 20 degrees is a good compromise in most cases. A value of 45 degrees should be considered as a maximum.

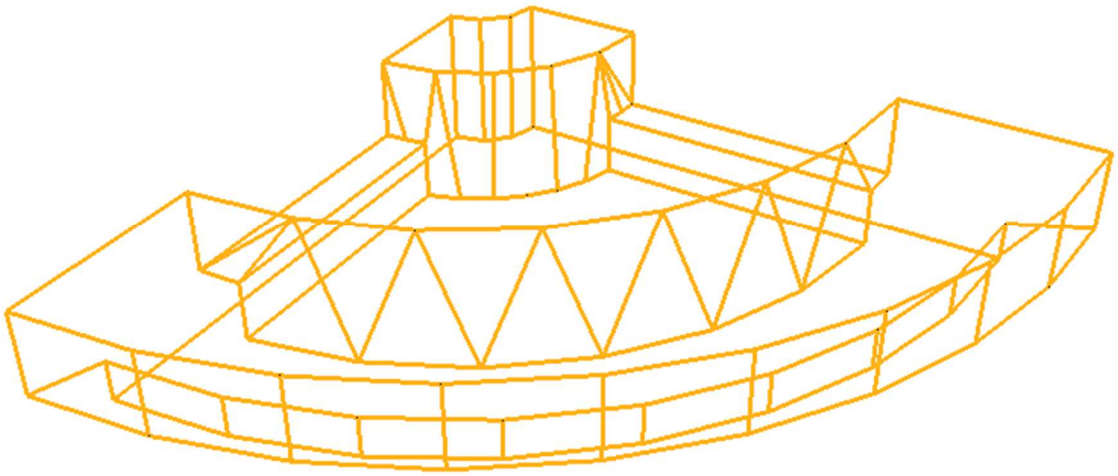


Figure 14 – Part #2 - Skeleton lines in the initial mesh (patch_angle_tolerance = 5)

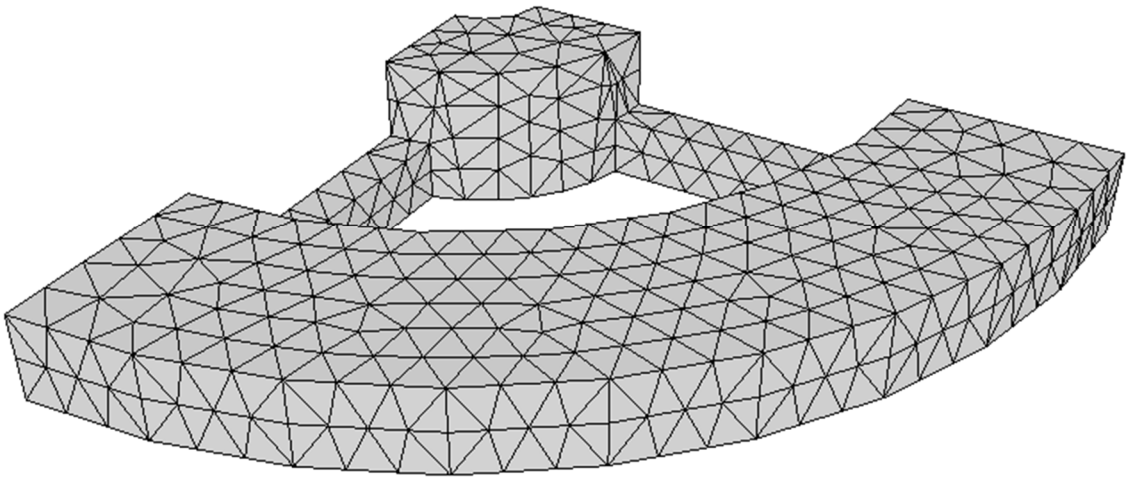


Figure 15 – Part #2 remeshed with min_h = 10, max_h = 10 and patch_angle_tolerance = 5

```
New Mesh
  Nodes          : 659
  Triangles      : 1318
  Area           : 5.352579E+004
  Qavg           : 8.621966E-001
  Qmin           : 2.777585E-001
Max error dist.  : 1.912226E-001
```

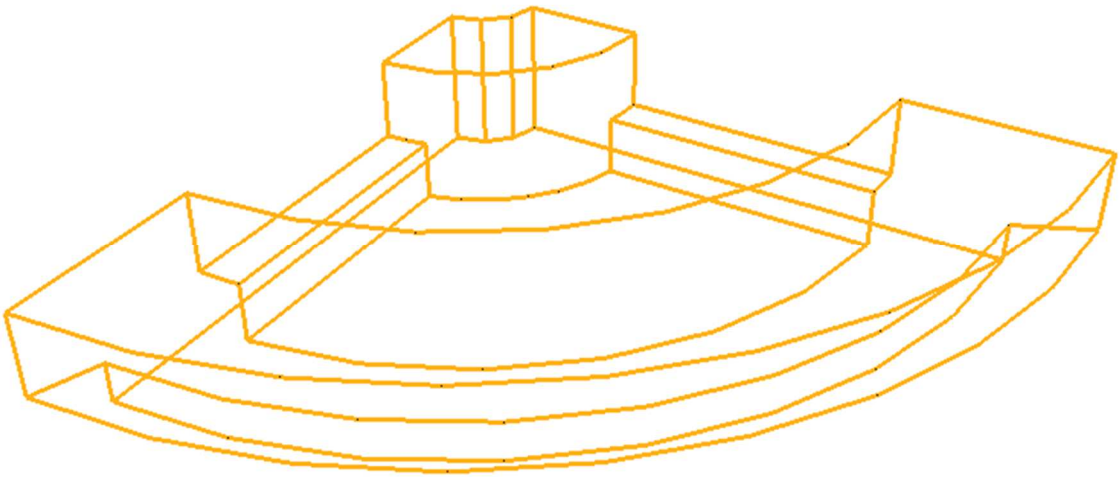


Figure 16 – Part #2 - Skeleton lines in the initial mesh (patch_angle_tolerance = 20)

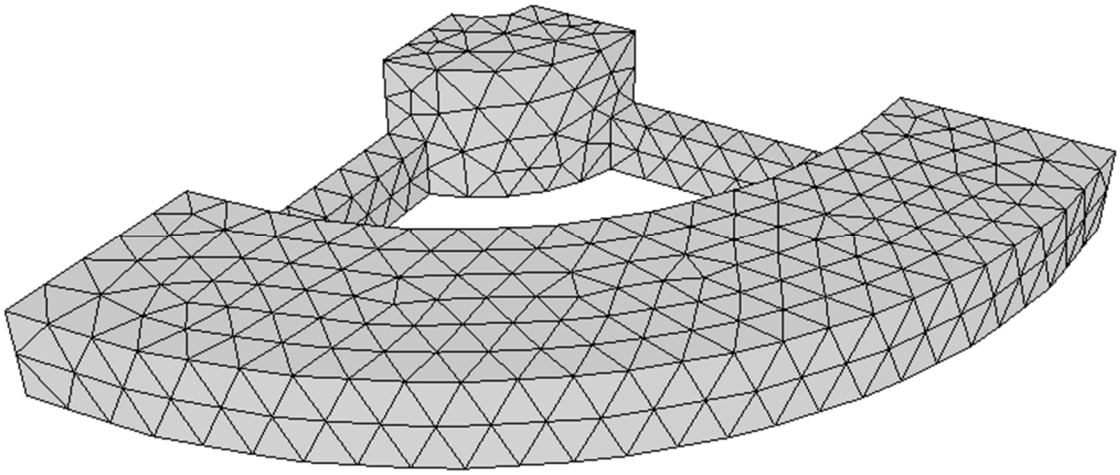


Figure 17 – Part #2 remeshed with min_h = 10, max_h = 10 and patch_angle_tolerance = 20

```
New Mesh
  Nodes          : 653
  Triangles      : 1306
  Area           : 5.356828E+004
  Qavg           : 8.813238E-001
  Qmin           : 6.071915E-001
Max error dist.  : 3.821892E-001
```

☞ Usually, the higher the angle tolerance, the better the mesh, the smaller the number of elements but the higher the geometric error.

II-5 REMESHING WITH QUADRANGLES

CM2 SurfRemesh Q4 can be used in place of CM2 SurfRemesh T3 to remesh with quadrangles instead of triangles. From the previous example, simply changing the namespace changes the type of the remesher:

```
#include "stdafx.h"

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


int main()
{
   surf_remesh_q4::mesher               the_remesher;
   surf_remesh_q4::mesher::data_type    data;

   // UNLOCK THE DLL.
   surf_remesh_q4::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH.
   ifstream       istrm (“part1.dat”);
   matio::transpose_read (istrm, data.pos);
   matio::transpose_read (istrm, data.connectM);

   // REMESHING THE SURFACE.
   the_remesher.mode.min_h = 10.0;
   the_remesher.mode.max_h = 10.0;
   the_remesher.run (data);

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

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

   return 0;
}   // main
```

The default mode of this remesher is the *quad-dominant* mode: it generates mixed quadrangle-triangle meshes. This is the reason why the output file is done here with the type `CM2_FACE_MIX`.

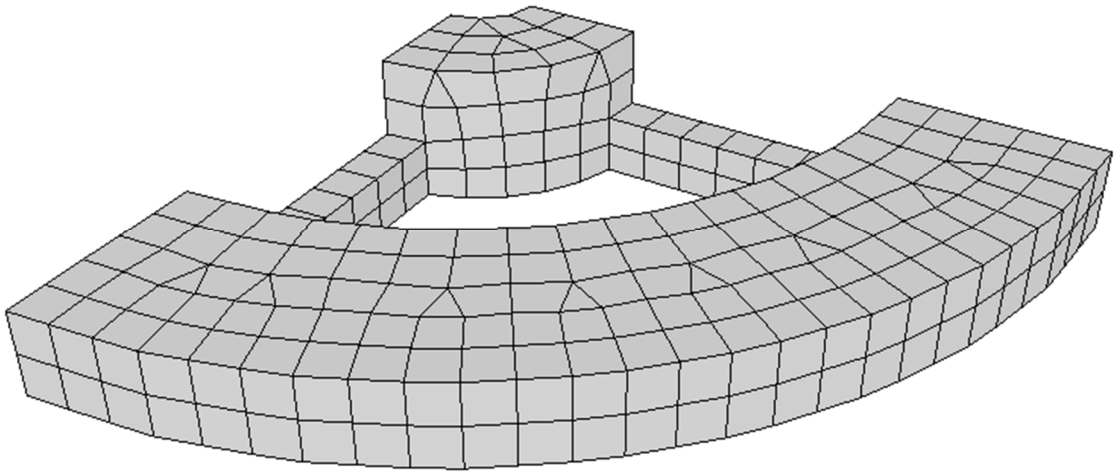


Figure 18 – Part #2 remeshed in quad-dominant mode (min_h = 10, max_h = 10, patch_angle_tolerance = 20)

```
New Mesh
  Nodes          : 550
  Elements       : 564
    Quadrangles  : 536 (95.04 %, 97.92 %)
    Triangles    : 28 (4.96 %, 2.08 %)
  Area           : 5.356221E+004
  Qavg           : 9.109910E-001
  Qmin           : 5.229764E-001
Max error dist.  : 3.824302E-001
```

An all-quad mesh can be generated by changing a setting in the remesher's mode:


```
int main()
{
   surf_remesh_q4::mesher               the_remesher;
   surf_remesh_q4::mesher::data_type    data;

   // UNLOCK THE DLL.
   surf_remesh_q4::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // READ THE INITIAL MESH.
   ifstream       istrm (“part1.dat”);
   matio::transpose_read (istrm, data.pos);
   matio::transpose_read (istrm, data.connectM);

   // REMESHING THE SURFACE.
   the_remesher.mode.min_h = 10.0;
   the_remesher.mode.max_h = 10.0;
   the_remesher.mode.all_quad_flag = true;
   the_remesher.run (data);

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

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

   return 0;
}   // main
```

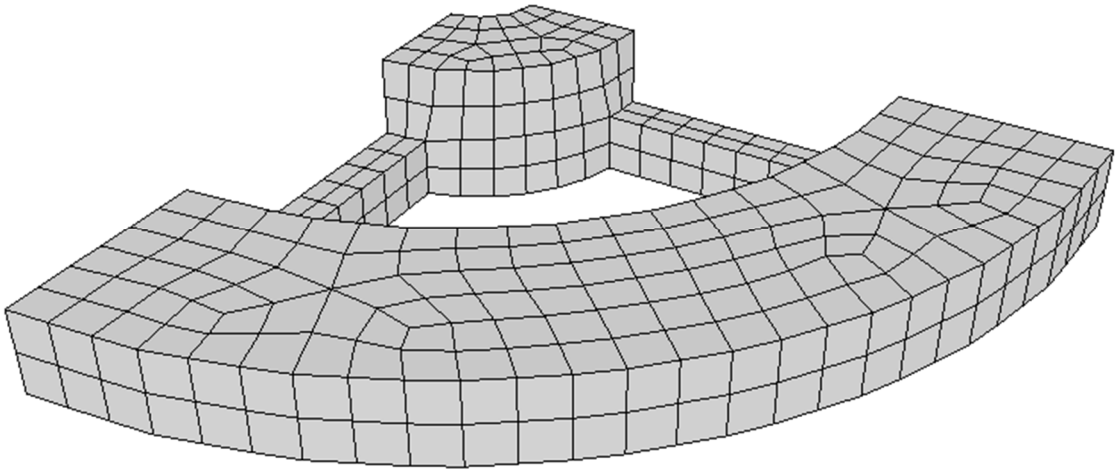


Figure 19 – Part #2 remeshed in all-quad mode
(min_h = 10, max_h = 10, patch_angle_tolerance = 20)

```
New Mesh
  Nodes          : 706
  Elements       : 706
    Quadrangles  : 706 (100.00 %, 100.00 %)
    Triangles    : 0 (0.00 %, 0.00 %)
  Area           : 5.357723E+004
  Qavg           : 8.113556E-001
  Qmin           : 4.757265E-001
Max error dist.  : 3.326376E-001
```

If triangles are tolerated, the quad-dominant mode (the default mode) should be preferred against the all-quad mode. It gives usually better meshes, with fewer elements. The all-quad mode should be restricted to very simple surfaces such as this one.

II-6 KEEPING THE SKELETON LINES

Let us now consider the remeshing of a sphere (Figure 20):

```
#include "stdafx.h"

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


int main()
{
   const double             R = 10.0;
   const unsigned           N = 64;
   UIntVec                  indices;
   UIntMat                  connectE;
   DoubleMat                pos;
   UIntMat                  connectMi, connectMr;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // THE INITIAL MESH.
   matvec::push_back (DoubleVec3(0.,0.,R), pos);
   meshtools1d::extrude_rotate (pos, 0u, DoubleVec3(0.,0.,0.),
                                DoubleVec3(0.,M_PI,0.), N/2, indices);
   meshtools1d::indices_to_connectE2 (indices, connectE);
   meshtools2d::extrude_rotate_T3 (pos, connectE, DoubleVec3(0.,0.,0.),
                                   DoubleVec3(0.,0.,2.*M_PI), N, 1,
                                   connectMi);

   // REMESHING THE SURFACE.
   surf_remesh_t3::mesher                  the_remesher;
   surf_remesh_t3::mesher::data_type       data;
   data.pos = pos;
   data.connectM.copy (connectMi);       // Hard copy of connectMi
   the_remesher.mode.min_h = 1.0;
   the_remesher.mode.max_h = 1.0;
   the_remesher.run (data);
   pos = data.pos;
   connectMr = data.connectM;

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

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

   return 0;
}   // main
```

To generate the initial mesh we start with meshing a "Greenwich" meridian[13]:

```
matvec::push_back (DoubleVec3(0.,0.,R), pos);
meshtools1d::extrude_rotate (pos, 0u, DoubleVec3(0.,0.,0.),
                             DoubleVec3(0.,M_PI,0.), N/2, indices);
meshtools1d::indices_to_connectE2 (indices, connectE);
```

This arc is then extruded into a triangle surface with a 2 π rotation:

```
meshtools2d::extrude_rotate_T3 (pos, connectE, DoubleVec3(0.,0.,0.),
                                DoubleVec3(0.,0.,2.*M_PI), N, 1,
                                connectMi);
```

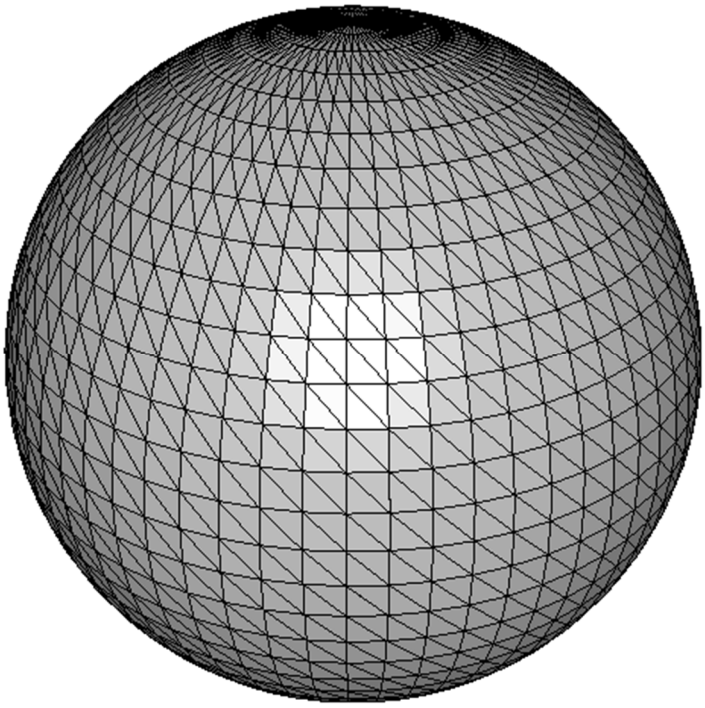


Figure 20 – Sphere. Initial mesh.

Note that this way of generating the mesh leads to multiple coincident nodes (along the initial meridian) and degenerated triangles at the poles. Fortunately, the remeshers are able to fix these issues and the new meshes are perfectly closed (watertight) with no degenerated element.

[13] *For explanation on `meshtools1D::extrude_rotate`, `meshtools1D::indices_to_connectE2` and `meshtools2D::extrude_rotate_T3` please refer to the Reference Manual of CM2 MeshTools (HTML doc).*

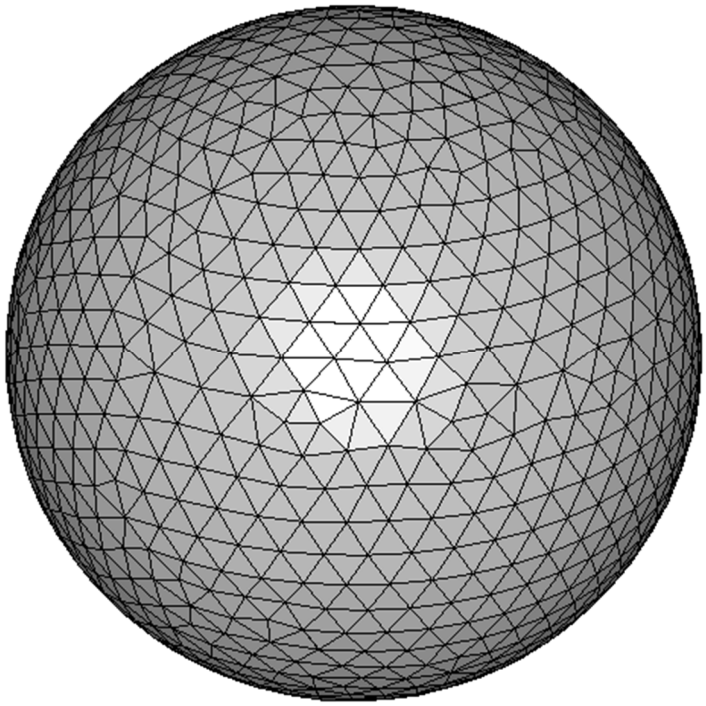


Figure 21 – Sphere remeshed with triangles.

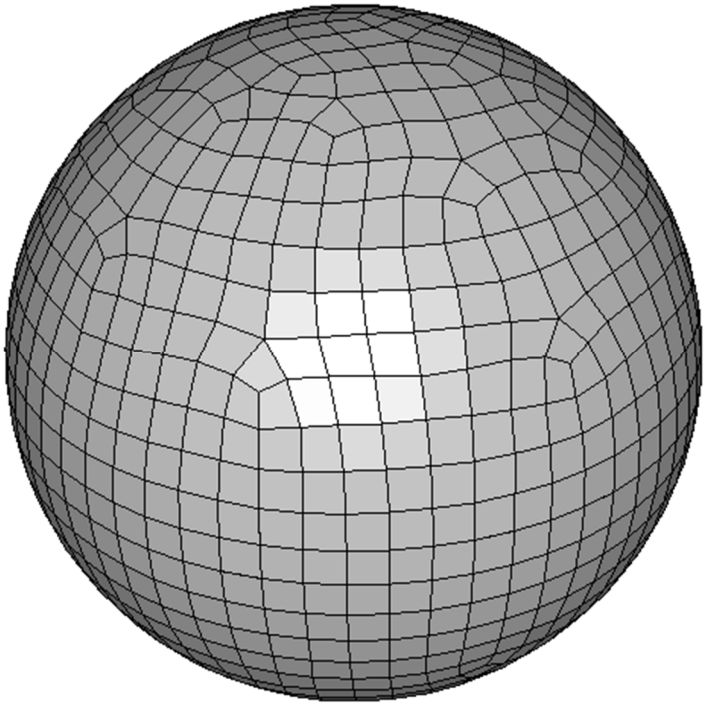


Figure 22 – Sphere remeshed with quadrangles only (all-quad mode).

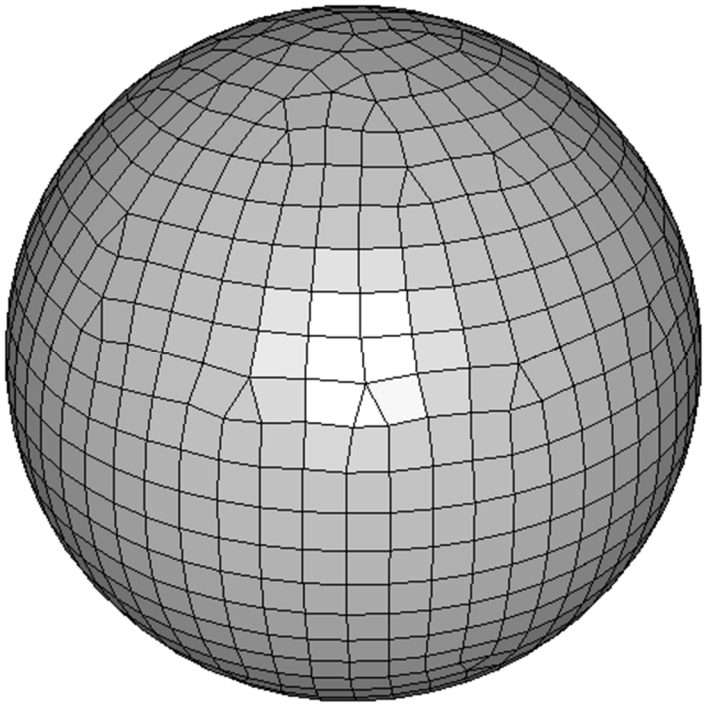


Figure 23 – Sphere remeshed with quads and triangles (mixed mode).

It appears that the remeshers divide the initial sphere into two patches (Figure 24) and that the initial G meridian and the equator lines are lost in the remeshing process. Should we be interested in keeping them, we have to put the edges of these lines into the edge connectivity matrix "skeleton_edges_in" as shown in the following example[14].

[14] *The meridian edges are readily available. However the equator edges need to be searched for (the code to do that is not shown here).*

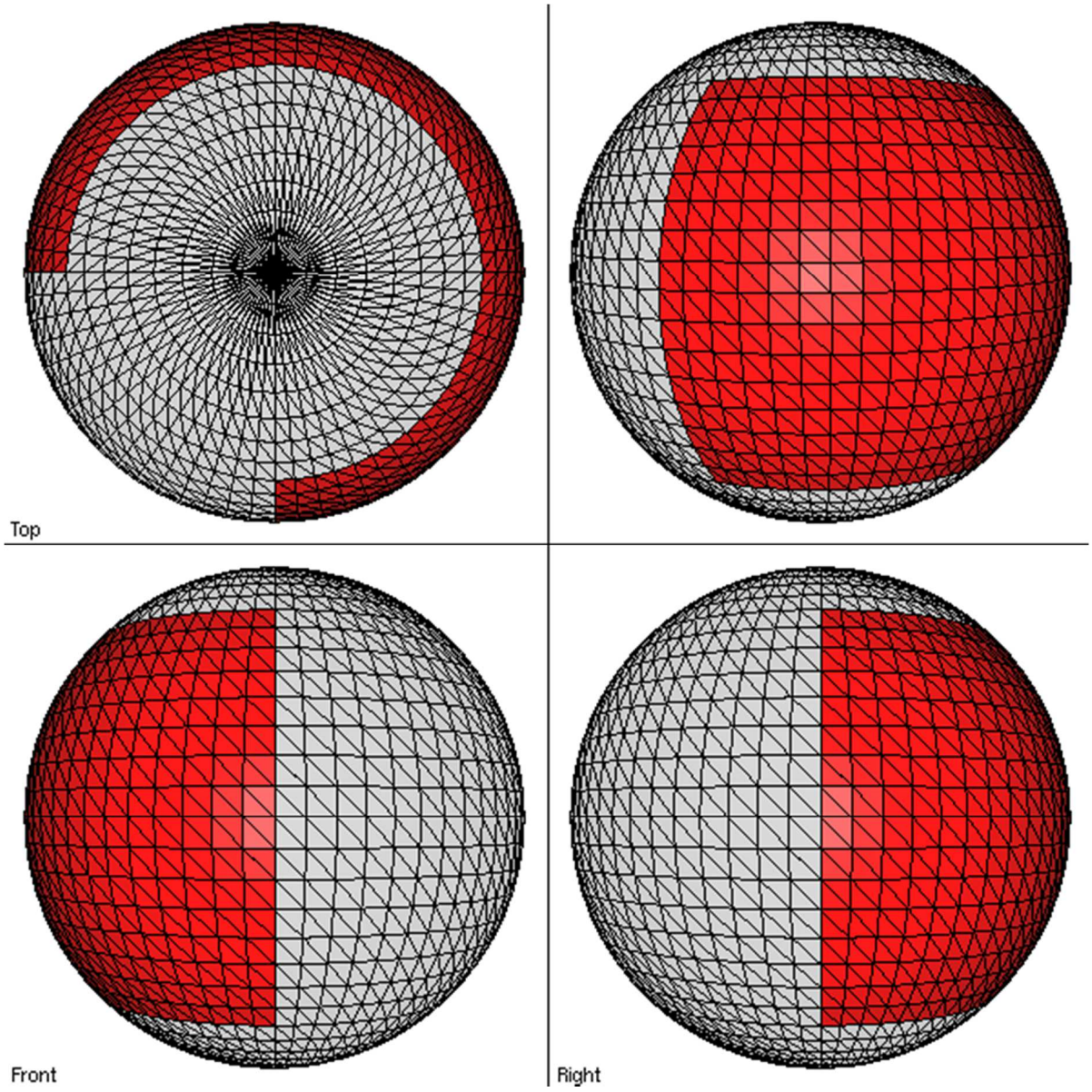


Figure 24 – Patches on the initial sphere (two patches).

```
#include "stdafx.h"

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

int main()
{
   const double             R = 10.0;
   const unsigned           N = 64;
   UIntVec                  indices;
   UIntMat                  connectE;
   DoubleMat                pos;
   UIntMat                  connectMi, connectMr;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // THE INITIAL MESH.
```

```
   matvec::push_back (DoubleVec3(0.,0.,R), pos);
   meshtools1d::extrude_rotate (pos, 0u, DoubleVec3(0.,0.,0.),
                                DoubleVec3(0.,M_PI,0.), N/2, indices);
   meshtools1d::indices_to_connectE2 (indices, connectE);
   meshtools2d::extrude_rotate_T3 (pos, connectE, DoubleVec3(0.,0.,0.),
                                   DoubleVec3(0.,0.,2.*M_PI), N, 1,
                                   connectMi);

   // REMESHING THE SURFACE.
   surf_remesh_t3::mesher                  the_remesher;
   surf_remesh_t3::mesher::data_type       data;
   data.pos = pos;
   data.connectM.copy (connectMi);       // Hard copy of connectMi
   data.skeleton_edges_in = connectE;    // Keep the meridian line.
   the_remesher.mode.min_h = 1.0;
   the_remesher.mode.max_h = 1.0;
   the_remesher.run (data);
   pos = data.pos;
   connectMr = data.connectM;

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

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

   return 0;
}   // main
```

Now the remesher divides the initial sphere into four patches in order to keep the constrained line.

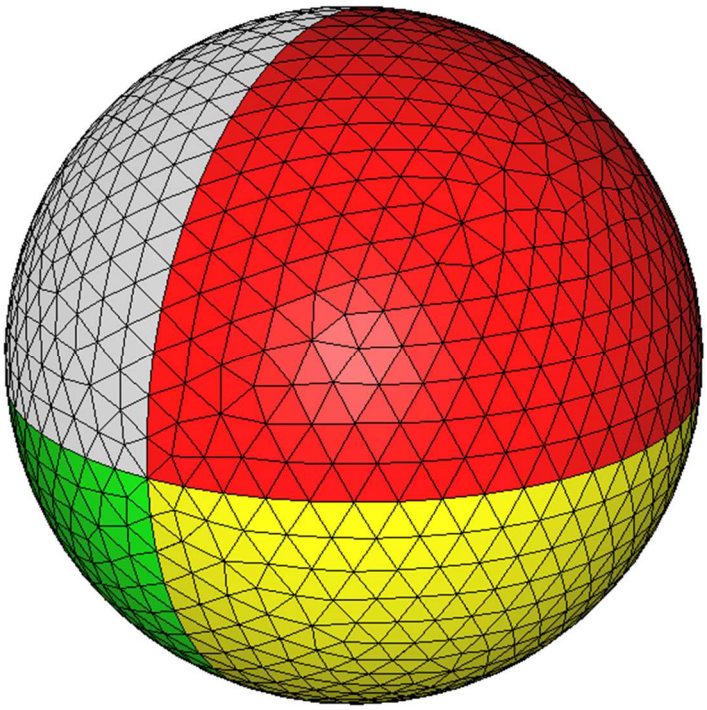


Figure 25 – Patches on the remeshed sphere when G meridian is kept (four patches).

Here the G meridian line (and by chance the equator line also) is kept in the final mesh. However it is remeshed: the nodes and edges along it are different now. The geometry of the line is preserved but not its mesh.
To keep the initial edges, we shall use the "hard_edges" field instead of "skeleton_edges_in":

```
#include "stdafx.h"

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

int main()
{
   const double             R = 10.0;
   const unsigned           N = 64;
   UIntVec                  indices;
   UIntMat                  connectE;
   DoubleMat                pos;
   UIntMat                  connectMi, connectMr;

   // UNLOCK THE DLL.
   surf_remesh_t3::registration ("Licensed to SMART Inc.", "0BEE453363E8");

   // THE INITIAL MESH.
   matvec::push_back (DoubleVec3(0.,0.,R), pos);
   meshtools1d::extrude_rotate (pos, 0u, DoubleVec3(0.,0.,0.),
                                DoubleVec3(0.,M_PI,0.), N/2, indices);
   meshtools1d::indices_to_connectE2 (indices, connectE);
   meshtools2d::extrude_rotate_T3 (pos, connectE, DoubleVec3(0.,0.,0.),
                                   DoubleVec3(0.,0.,2.*M_PI), N, 1,
                                   connectMi);

   // REMESHING THE SURFACE.
   surf_remesh_t3::mesher                  the_remesher;
   surf_remesh_t3::mesher::data_type       data;
   data.pos = pos;
   data.connectM.copy (connectMi);    // Hard copy of connectMi
   data.hard_edges = connectE;        // Keep the edges along the meridian.
   the_remesher.mode.min_h = 1.0;
   the_remesher.mode.max_h = 1.0;
   the_remesher.run (data);
   pos = data.pos;
   connectMr = data.connectM;

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

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

   return 0;
}   // main
```


III – USER'S MANUAL

III-1 REMESHERS' DATA

All data for a run of the remesher are gathered into a single structure of type `data_type`:

```
void surf_remesh_t3::mesher::run (surf_remesh_t3::mesher::data_type& data);
void surf_remesh_q4::mesher::run (surf_remesh_q4::mesher::data_type& data);
```

Some of the most important fields of these structures have already been seen in the previous chapter. This one details all the fields.

Points' coordinates

Matrix `pos` of size 3xN (IN and OUT).

It stores the coordinates of *all* points. The coordinates are stored column-wise. The column index is the index of the node (zero-based, i.e. from 0 to N-1). The *X*-coordinates are in the first row, the *Y*-coordinates in the second row, and the *Z*-coordinates in the third row.
Upon exit, the coordinates of the newly generated nodes are appended to the back of the matrix as new columns. The initial columns are left unchanged[15].

Elements

Matrix `connectM` (IN and OUT).

For the triangle remesher, the dimension of this matrix is always 3xNEFS (input and output). For the quadrangle remesher, the dimension is 3xNEFS upon entry (only triangle mesh upon entry[16]) and 4xNEFS upon exit. In this case, the leading part of the matrix (from columns 0 to `nefs_Q4-1`) is the connectivity of the quadrangle elements. The tailing part (from columns `nefs_Q4` to `nefs-1`) is the connectivity of the triangle elements, the fourth node ID in this part being `CM2_NONE` (`unsigned(-1)`).
The ordering of the elements in this matrix is irrelevant.
The initial connectivity is lost and overwritten by the new mesh connectivity, except when the remesher fails.

Metric field

Vector `metrics` (IN).

The user can specify a target mesh size at each initial node. If the value for a node is zero - or negative or not present – a default value will be used instead[17]. The new mesh is generated to fit best locally the metric values. In the current release (4.0), there is no way to control more precisely the elements' size far from the hard nodes. There is no "background mesh" option for instance.

Only the values for the hard nodes will be considered.

Neighbors

Matrix `neighbors` of dimension 3xM (OUT).
This matrix gives, for each element in the final mesh, the indices of the four neighboring elements (`CM2_NONE` if none or several).

---

[15] To keep only the nodes of the final mesh, use `function cm2::meshtools::simplify` after the remeshing:

```
cm2::meshtools::simplify (data.pos, data.connectM);
```

[16] To remesh a all-quad or quad-dominant mesh, use `meshtools2D::split_Q4_into_T3` to transform into a all-triangle mesh prior to remeshing.

[17] The default mesh size value at a hard node is the minimum distance to any other hard node caped by `min_h` and `max_h`.

`neighbors(i,j)` is the neighbor of the j^{th} element sharing its i^{th} edge. See Figure 26 for the local numbering of the faces.

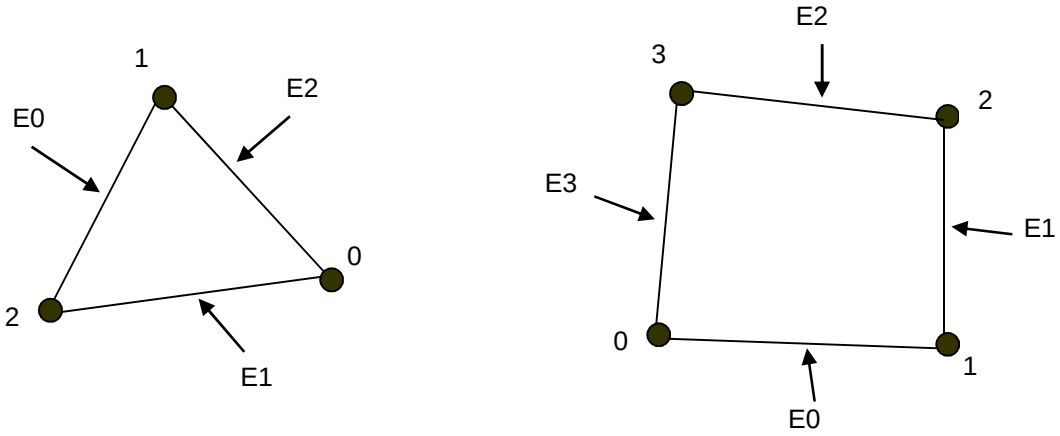


Figure 26 – Nodes and edges local numbering in triangles and quads.

Ancestors

`ancestors` is a vector of size N, i.e. the number of columns in the `pos` matrix upon exit (OUT).

This vector gives, for each node, the index of one of the elements in which it belongs (`CM2_NONE` if the node is not in the final connectivity matrix).
Together with the `neighbors` matrix, this can make easy the design of search and traversal algorithms (such as looking for all elements connected to a node).

Shape qualities

Vector `shape_qualities` of size equal to the number of columns in `connectM` upon exit (OUT).

This vector gives the shape quality of each element.

The formula for the shape quality of a triangle writes:

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

with:

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

The formula for the shape quality of a 2-D quadrangle writes:

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

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 quadrangle.

The formula for the shape quality of a 3-D quadrangle writes:

$$Q_s^{3D} = Q_s^{2D}\ Q_w$$

with:

Q_w Warp quality of the quadrangle:

$$Q_w = 1 - \frac{a\cos\left(\max\left(\langle N_0, N_2\rangle, \langle N_1, N_3\rangle\right)\right)}{\pi}$$

N_i Normal to the quad at node i.

Histograms

Histograms `histo_Qs_in`, `histo_Qs` and `histo_Qh` (OUT).

`histo_Qs_in` is the histogram of the shape qualities in the initial mesh.
`histo_Qs` is the histogram of the vector `shape_qualities` in the final mesh.
`histo_Qh` is the histogram of the normalized edge lengths in the final mesh (computed only when the option flag `compute_Qh_flag` is up, see § III-3).
Each histogram stores the minimum, the maximum and the average value as data members.

Hard faces

Vector `hard_faces` (IN).

Put in this vector the IDs of the initial triangles (column IDs in input matrix `connectM`) that must be kept in the final mesh. Their edges and nodes will be considered as hard edges and hard nodes.

Skeleton edges of the initial mesh

Matrix `skeleton_edges_in` of dimension 2xNE_{ski} (IN-OUT).

These skeleton edges are the boundaries of the patches in the initial mesh. They are the "stitches" in the initial mesh found by the algorithm. Upon entry, the user can specify some skeleton edges. This constrains the algorithm to "cut" the 3D mesh along these edges. The skeleton lines (successively connected skeleton edges) are remeshed in the final mesh.

Skeleton edges of the final mesh

Matrix `skeleton_edges` of dimension 2xNE_{skf} (OUT).
These skeleton edges are the remeshed skeleton lines. They are the boundaries of the patches in the final mesh.

Edges to exclude from skeleton edges

Matrix `exclude_skeleton_edges` of dimension 2xNE_{xsk} (IN).

This matrix contains the edges that should not be considered as skeleton edges, if possible. Only the edges that violate the `patch_angle_tolerance` criterion can be excluded this way. The algorithm will keep any skeleton edge if it violates any other criterion (`strain_tolerance` or if it is impossible to remesh without this edge). The user can use this matrix to make the algorithm select bigger patches than what it normally would.

Hard edges

Matrix `hard_edges` of dimension 2xNE_{he} (IN).

Hard edges are skeleton edges that should not be remeshed. A "hard edge" is always considered as a skeleton edge: it is a stitch in the initial mesh, and its vertices are considered as skeleton nodes (= hard nodes).

Hard nodes

Vector `hard_nodes` (IN-OUT).

Upon entry, the user can put in this vector some node IDs to constrain them to remain in the final mesh. Upon exit, this vector will contain in addition all the "skeleton" nodes, e.g. the topologically singular nodes (nodes connected to one, three or more skeleton lines) and the nodes of the skeleton lines for which the angle between coincident skeleton lines is greater than `mode.patch_angle_tolerance`. These skeleton nodes are also kept from the initial mesh.

Nodes to exclude from the hard nodes

Vector `exclude_hard_nodes` (IN).

This vector contains the nodes that should not be selected as skeleton nodes, if possible.
Only the nodes that violate the `patch_angle_tolerance` criterion of the skeleton lines can be excluded. The algorithm will keep any other skeleton nodes, as well as the user-defined hard nodes.

Field to interpolate on the new mesh

Matrix `interpolate_field` (IN-OUT).

This matrix contains nodal data to be interpolated to the new mesh (such as FEM data, textures...)
Data for node #i is at column #i (can be a single scalar or a vector).
As new nodes are appended to the coordinate matrix, new values are linearly interpolated and appended to this matrix[18].

Elements' color

Vectors `colors_in` and `colors` (OUT).

The `colors_in` (resp. `colors`) vector gives the patch ID[19] of the elements of the initial (resp. final) mesh. This helps transferring information data (such as references to material properties) from the initial mesh to the final mesh.

Skeleton edges' color

Vectors `skeleton_colors_in` and `skeleton_colors` (OUT).
The `skeleton_colors_in` (resp. `skeleton_colors`) vector gives the skeleton line ID[20] of the skeleton edges of the initial (resp. final) mesh. This helps transferring information data (such as references to beam material properties) from the initial mesh to the final mesh.

---

[18] To eliminate the old nodes together with their data, use the following code snippet after the remeshing:

```
UIntMat     connectM;
connectM.copy (data.connectM);                    // Copy of initial connectivity.
cm2::meshtools::simplify (data.pos, connectM);    // Eliminate unused nodes.
cm2::meshtools::simplify (data.interpolate_field,
                          data.connectM);         // Eliminate unused data.
```

[19] Patches are numbered consecutively starting from 0. Elements eliminated during the initial clean-up and final optimization steps have no patch ID (equals to `CM2_NONE`).

[20] Skeleton lines are numbered consecutively starting from 0. Elements eliminated during the initial clean-up and final optimization steps have no ID (i.e. equals to `CM2_NONE`).

Maximum error distance

Scalar (`double`) `max_error_distance` (OUT).

This is an approximate maximum distance between the initial mesh and the final mesh. This is a measure of the geometric error between the two meshes.

Errors and warnings

Two enums, `error_code` and `warning_code`, give upon exit the error and warning codes. The string `msg1` holds explanations messages about the error/warning raised if any. In case of error, the meshing process is aborted and the output mesh (`connectM`) is left unchanged (equal to the input mesh).
With a mere warning, the process goes to its end, though the final mesh may be of poor quality.
A correct run gives the `CM2_NO_ERROR` and `CM2_NO_WARNING` codes (zero values).
See § III-2 for more detailed explanations of the error and warning codes.

Complementary information

In this section are gathered the remaining fields of the data structure. They are all output values about the initial and the final mesh:

- The number of elements in the initial and in the final mesh (also equals to the number of columns in `connectM` upon entry and upon exit).
- The number of quads (always null with SurfRemesh T3) and the number of triangles (always null with SurfRemesh Q4 in all-quad mode).
- The number of nodes in the initial and final meshes (less or equal to the number of columns in `pos` upon entry and upon exit).
- The total number of points (i.e. columns) in matrix `pos` upon exit.
- The area of the initial and final mesh.
- The area of the quads (always null with SurfRemesh T3) and the area of the triangles (always null with SurfRemesh Q4 in all-quad mode). The sum equals to the area of the mesh above.
- The worst shape quality of the elements in the initial and in the final mesh.
- The error and warning codes (see III-2).
- The error message.
- The times spent in the successive steps of the remesher.
- The total time spent in the remesher.
- The global speed of the remesher (number of generated elements per second).

```
struct data_type
{
   DoubleMat               pos;
   UIntMat                 connectM;
   DoubleVec               metrics;

   UIntMat                 neighbors;
   UIntVec                 ancestors;
   DoubleVec               shape_qualities;
   cm2::misc::histogram    histo_Qs_in;
   cm2::misc::histogram    histo_Qs;
   cm2::misc::histogram    histo_Qh;

   UIntVec                 hard_faces;
   UIntMat                 skeleton_edges_in;
   UIntMat                 skeleton_edges;
   UIntMat                 exclude_skeleton_edges;
   UIntMat                 hard_edges;
   UIntVec                 hard_nodes;
   UIntVec                 exclude_hard_nodes;

   DoubleMat               interpolate_field;

   UIntVec                 colors_in;
   UIntVec                 colors;
   UIntVec                 skeleton_colors_in;
   UIntVec                 skeleton_colors;

   double                  max_error_distance;

   unsigned                nefs_in;
   unsigned                nods_in;
   double                  surface_in;
   double                  Qmin_in;

   unsigned                nefs;
   unsigned                nefs_Q4;
   unsigned                nefs_T3;
   unsigned                nods;
   double                  area;
   double                  area_Q4;
   double                  area_T3;
   double                  Qmin;

   unsigned                total_nods;

   double                  cleaning_time;
   double                  analysis_time;
   double                  remesh_time;
   double                  optim_time;
   double                  total_time;
   double                  speed;

   error_type              error_code;
   warning_type            warning_code;
   char                    msg1[256];
};
```

Table 2 – The `surf_remesh_t3::mesher::data_type` and `surf_remesh_q4::mesher::data_type` structures (only the data members are shown).

III-2 ERROR AND WARNING CODES

Error codes

The error code is located in the structure `data_type`.

```
enum error_type
{
   CM2_NO_ERROR,                    // 0
   CM2_LICENSE_ERROR,               // -100
   CM2_MODE_ERROR,                  // -101
   CM2_DATA_ERROR,                  // -102
   CM2_NODES_LIMIT_ERROR,           // -103
   CM2_DEGENERATED_ELEMENT,         // -107
   CM2_REMESHING_ERROR,             // -108
   CM2_SYSTEM_MEMORY_ERROR,         // -199
   CM2_INTERNAL_ERROR               // -200
};
```

Table 3 – Error codes for CM2 SurfRemesh T3 & Q4.

`CM2_NO_ERROR`	OK, no problem.
`CM2_LICENSE_ERROR`	The registration must occur before the instantiation of any meshers. Check also your license. You may renew it. Please, contact license@computing-objects.com
`CM2_MODE_ERROR`	The operating mode is not valid (see § III-3). Check the positivity/range of scalar values such as `shape_quality_weight`, `patch_angl_tolerance`...
`CM2_DATA_ERROR`	The input data are not valid. Check the sizes of matrices, of vectors, node indices in the connectivity matrices, look for insane values…
`CM2_NODES_LIMIT_ERROR`	The limit on the number of nodes is too low.
`CM2_DEGENERATED_ELEMENT`	At least one of the elements is invalid (null surface). Check the input data.
`CM2_REMESHING_ERROR`	Can't generate a better mesh. Reduce parameter `min_h`, `max_h`, `patch_angle_tolerance`, `strain_tolerance`..
`CM2_SYSTEM_MEMORY_ERROR`	Insufficient memory available. Mesh is too big to be generated (over several tens millions elements).
`CM2_INTERNAL_ERROR`	Unknown cause of error.

Table 4 – Error codes.

In case of error, the output mesh in field `data.pos` and `data.connectM` is identical to the input mesh.

For error codes `CM2_DEGENERATED_ELEMENT`, `CM2_REMESHING_ERROR` and `CM2_INTERNAL_ERROR`, save the data by calling `data_type::save` and send the zipped file to support@computing-objects.com.

Example:

```
if (data.error_code != surf_remesh_t3::mesher::data_type::CM2_NO_ERROR)
{
  // Error, do something.
}
```

Warning codes

The warning code is located in the structure `data_type`.

```
enum warning_type
{
   CM2_NO_WARNING,                    //   0
   CM2_INTERRUPTION,                  // -10
   CM2_NODES_LIMIT_WARNING,           // -11
   CM2_SHAPE_QUALITY_WARNING,         // -12
   CM2_NON_MANIFOLD_SOLID_WARNING     // -13
};
```

Table 5 – Warning codes for CM2 SurfRemesh T3 & Q4.

`CM2_NO_WARNING`	OK, no problem.
`CM2_NODES_LIMIT_ WARNING`	The node limit has been reached and the mesh may be far from optimal.
`CM2_INTERRUPTION`	The user has aborted the run (through the interrupt handler). The final mesh may be empty or valid but of poor quality.
`CM2_SHAPE_QUALITY_ WARNING`	The final mesh is valid but at least one of the elements is very bad (shape quality < 0.01). Reduce parameter `min_h`, `max_h`. Increase parameters `patch_angle_tolerance`, `strain_tolerance`, `optim_tolerance`..
`CM2_NON_MANIFOLD_SOLID_ WARNING`	The 3-D surface mesh could not be made watertight. This warning can be raised only in the solid mode (`solid_flag` = `true`).

Table 6 – Warning codes.

Example:

```
if (data.warning_code ==
    surf_remesh_t3::mesher::data_type::CM2_SHAPE_QUALITY_WARNING)
{
  // Warning, do something.
}
```

III-3 OPTIONS OF THE REMESHERS

CM2 SurfRemesh T3 & Q4 have several operating options that can change drastically its outputs. They are gathered into a structure of type `operating_mode_type` as a public field of the remeshers:

```
cm2::surf_remesh_t3::mesher::operating_mode_type   mode;
cm2::surf_remesh_q4::mesher::operating_mode_type   mode;
```

Minimum size of the elements

`min_h`. Default = 0.0

Lower bound for the metric field used to generate the new mesh.
This value controls the size of the smallest elements[21].
The user's specified values in `metrics` are not limited by this value.

Maximum size of the elements

`max_h`. Default = `DBL_MAX`

Upper bound for the metric field used to generate the new mesh.
This value controls the size of the largest elements.
The user's specified values in `metrics` are not limited by this value.

Max chordal error

`max_chordal_error`. Default = -0.05 (i.e. 5% of local radius)

Maximum chordal error allowed.
The mesh size is reduced locally to limit the chordal error between the mesh and the surface:
If negative, this value is *relative* to the local radii (for instance -0.01 => max chordal error < 1% of local radii).
If positive, this value is *absolute* (for instance 0.1 => max chordal error < 0.1).

Max gradation

`max_gradation`. Positive value. Default = 0.5

This parameter controls the gradation of the elements size along the skeleton lines and inside the patches towards the `max_h` mesh size.
A value close to 0 leads to a more progressive variation of mesh size (smoother).

Patch angle tolerance

`patch_angle_tolerance`. Default = 20.0

Parameter used to delimit the patches (groups of connected triangles in the initial mesh).
An angle between two adjacent triangles greater than this value will draw a limit between two patches (we call such limit a "skeleton edge"). A small value tends to give numerous small patches and the final mesh will be close to the initial mesh but may be of poor quality. A big value tends to give fewer bigger patches. The remesher can do a better job on big patches but the final mesh may be more distant from the initial mesh (increased geometric error).
A value of 45° should be considered as the maximum for this parameter.

[21] *However some elements may be smaller to satisfy geometric constraints.*

Initial fixing tolerance

`fix_tolerance`. Default = -0.0025 (i.e. 0.25% of min_h)

Controls the merging of nodes and the swapping of edges during the initial clean-up step.
This parameter is the maximum allowable distance (absolute or relative to `min_h`) that a node merging or an edge swapping can induce on the initial surface.
If `fix_tolerance` > 0, `fix_tolerance` is taken as an *absolute* tolerance.
If `fix_tolerance` < 0, this is a *relative* tolerance. The distance of merging is then computed as the product of `-fix_tolerance` and `min_h`, or if `min_h` is null, as the product of `-fix_tolerance` and a mean of the lengths of the edges in the initial mesh.

This parameter should be used only to treat pathologically close nodes that are present in the initial mesh. A value of 1% (i.e. `-0.01`) should be considered as the maximum for this parameter.

Final optimization tolerance

`optim_tolerance`. Default = -0.05 (i.e. 5% of min_h)

Controls the merging of nodes and the swapping of edges during the final optimization step.
This parameter is the maximum allowable distance (absolute or relative to `min_h`) that a node merging or an edge swapping can induce on the final surface.
If `optim_tolerance` > 0, `optim_tolerance` is taken as an absolute tolerance.
If `optim_tolerance` < 0, this is a relative tolerance. The distance of merging is then computed as the product of `-optim_tolerance` and `min_h`, or if `min_h` is null, as the product of `-optim_tolerance` and a mean of the lengths of the edges in the initial mesh.

A value of 20% (i.e. `-0.20`) should be considered as the maximum for this parameter.

Strain tolerance

`strain_tolerance`. Default = 0.30

This is the maximum allowable strain for the elements in a patch when unfolded.
Set `strain_tolerance` = 0 if no strain is allowed. Only perfectly unfoldable patches will be selected in this case (i.e. flat or simply curved patches). A small value tends to reduce the size of the patches. A large value tends to give fewer bigger patches.
A value of 0.60 should be considered as the maximum for this parameter.

Initial clean-up

`initial_cleanup_flag`. Default = true

Flag to allow a clean-up of the initial mesh (node merging, gaps filling and topological fixing). The patch ID of the elements (see `data_type::patch_IDs`) is computed after the initial clean-up.

Flag for solids

`solid_flag`. Default = false

Flag to tell if the 3-D surface should be considered as closed and simple (no internal faces) or not.
If true, a specific algorithm is used to correct the initial mesh in the cases when some elements are under-connected or over-connected (edges not shared exactly by two elements). The remesher emits a warning (`CM2_NOT_SIMPLE_SOLID_WARNING`) when it cannot enforce all edges in the initial mesh to be shared by exactly two elements.

All-quad or quad-dominant mode (CM2 SurfRemesh Q4)

`all_quad_flag`. Default = false

Flag to force the generation of an all-quad mesh (no triangle):

- `false`: generates a mixed triangle-quad mesh (the default).
- `true`: generate a all-quad mesh.

A good all-quad mesh can only be generated in simple cases. If possible, rather use the mixed quad-dominant generation mode.
When this flag is set to `false`, parameter `quadrangle_weight` can be used to control the trade-off between meshes with more numerous quads and meshes with more numerous triangles but of better shape quality.

Remeshing

`remesh_flag`. Default = true.

This flag enables the remeshing of the patches. Set this flag to `false` if you don't want to remesh but only to optimize it (or simply to compute the patches in the initial mesh, the skeleton edges and the skeleton nodes).
☞ Note that the initial clean-up and final optimization, if not disabled, may still modify the mesh.

Flag to force parity along the skeleton lines

`force_even_flag`. Default = false.

Flag to force the number of edges in the final skeleton lines to be even. This is always the case with CM2 SurfRemesh Q4 when `all_quad_flag` is set to `true`.

Node smoothing

`node_smoothing_flag`. Default = `true`.

This flag controls the node-smoothing scheme in the optimization step.
When `remeshing_flag = false`, node smoothing doesn't change the mesh connectivity, only the coordinates of nodes.
This flag has no effect when the optimization step is skipped (`optim_level` = 0)

Node inserting

`node_inserting_flag`. Default = `true`.

This flag controls the node-inserting scheme in the optimization step.
When `remeshing_flag = false`, node inserting increases the number of nodes, changes the mesh connectivity, but doesn't change the other nodes' coordinates.
This flag has no effect when the optimization step is skipped (`optim_level` = 0)

Node removing

`node_removing_flag`. Default = `true`.

This flag controls the node-removing scheme in the optimization step.
When `remeshing_flag = false`, node removing decreases the number of nodes, changes the mesh connectivity, but doesn't change the other nodes' coordinates.
This flag has no effect when the optimization step is skipped (`optim_level` = 0)

Shell remeshing

`shell_remeshing_flag`. Default = `true`.

This flag controls the local remeshing scheme in the optimization step.
When `remeshing_flag = false`, shell remeshing changes the mesh connectivity, but doesn't change the number of nodes nor their coordinates.
This flag has no effect when the optimization step is skipped (`optim_level` = 0)

Final optimization

`final_optimization_flag`. Default = true.

Flag to allow the final inter-patch optimization scheme.
If true, nodes can be merged together if they are within the merging distance (`optim_tolerance`) and edges can be swapped to improve the shape quality of the elements if the induced geometric error is lower than `optim_tolerance`.

Only smooth nodes

`node_smoothing_only_flag`. Default = `false`.

This flag tells the remesher to keep the initial mesh connectivity and do only node smoothing on the surface. When true, the other flags `initial_cleanup_flag` , `solid_flag`, `remesh_flag`, `force_even_flag` and `final_optimization_flag` are irrelevant (considered as `false`).

Computation of the size-qualities histogram

`compute_Qh_flag`. Default = `false`.

Before exiting the process, this flag tells the mesher to compute the histogram of the size quality of all the edges in the new mesh.

Pattern for structured meshes (CM2 SurfRemesh T3)

`structured_pattern`. Default = -1.

This option controls the way the generators does the structured meshes when possible (on rectangular-like patches).
It can take four values:

- -1: This is the default mode. The triangular meshes are always done with the frontal-Delaunay algorithm. This usually gives “optimal” meshes.
- 0: When possible, generates structured left-oriented meshes (simply oriented pattern).
- +1: When possible, generates structured right-oriented meshes (simply oriented pattern).
- +2: When possible, generates structured UJ meshes (“Union Jack” pattern).

Pattern for structured meshes (CM2 SurfRemesh Q4)

`structured_flag`. Default = `true`.

This option controls the way the generators do the structured meshes when possible (on rectangular-like patches):

- `true`: This is the default mode. When possible, generates structured (grid-like) meshes.

- `false`: The quad meshes are always done with the frontal-Delaunay algorithm[22].

Limit on the number of nodes

`nodes_limit`. Default = `UINT_MAX`.

When the mesh generator reaches this limit, the `CM2_NODES_LIMIT_WARNING` is issued and new nodes are no longer created. In this case, the quality of some elements can be far from optimal. When the limit is so low that the remesher cannot even insert all the skeleton nodes the `CM2_NODES_LIMIT_ERROR` is raised and the mesh is aborted.

Optimization level

`optim_level`. Integer between 0 and 10. Default = 3

A null value makes the mesher to skip the optimization step. The speed is maximal but the quality may be poor. From value 1 on, the optimizer algorithm uses several techniques to improve both the shape quality and the size quality of the elements, such as node smoothing, edge swapping, node insertion and node removal. Level 3 is usually a good trade-off between quality and speed.

Weight on shape quality

`shape_quality_weight`. Value between 0 and 1. Default = 0.6

This parameter controls the trade-off between shape optimization and size optimization. It is the weight of the shape quality in the measure of the global quality of an element. The default value (0.6) gives a slight preference to the shape quality over the size quality.

Weight on quadrangles (CM2 SurfRemesh Q4)

`quadrangle_weight`. Value between 0 and 1. Default = 0.70

Preference for quadrangles vs. triangles. Weight between 0 and 1 indicating the preference for quadrangles over triangles, when a all-quad mesh cannot be generated (i.e. when there is no parity on the boundaries, or `all_quad_flag = false`).
With `quadrangle_weight` = 0, quadrangles are never used.
With `quadrangle_weight` = 0.5, quadrangles are used only when they improve the quality of the mesh.
For values between 0.5 and 1, quadrangles are more and more used even if this lead to a lesser quality of the mesh.
With `quadrangle_weight` = 1, the minimum number of triangles are used (but may not be null).
This parameter is used only when `all_quad_flag` = `false`.
This parameter is not the ratio between quads and triangles. Furthermore, there is no linearity between this weight and the ratio between quads and triangles.

Minimum quadrangle quality (CM2 SurfRemesh Q4)

`min_Q4_angle_quality`. Double value between 0 and 1. Default = 0 (no minimum).

Minimum acceptable angle quality for the quadrangles.
This parameter is taken into account in mixed mode only (`all_quad_flag = false`).

[22] This may give also structured meshes. The true option enforces this and is much faster whenever a structured mesh can be generated. Since release 3.4 this flag is less useful because QuadMesh is able to generate naturally a perfect mesh on rectangle domains, even with this flag off.

This quality threshold is based on the *angle* quality of the quads (not the *geometric* quality which takes the length ratios also into account). The angle quality is computed as the minimum of the four angles at sommits[23]. Set min_Q4_angle_quality = 1 to allow rectangles only (quads with right angles only).
In this case, be aware that when boundaries are not straight very few rectangles may be generated (mostly triangles).

Upper bound on edges length

`length_upper_bound`. Value greater than 0. Default = 1.414

This parameter is used to limit the length of the edges in the generated mesh (normalized length). This is *not* a strict enforcement however. Most of the edges will be shorter than this limit, but some may remain somewhat longer. The default value (1.414) gives the optimal meshes with respect to the size qualities. With this default value, the average edge length tends to be 1 (optimal edge quality on average).
Sometimes, it can be useful to limit the length of the edges to a shorter value (usually between 1 and 1.414), and to accept an average value smaller than 1 (sub-optimal edge qualities on average).

Minimum number of edges along loops

`min_NE_for_loops`. Integer greater or equal to 3. Default = 6

This parameter can be useful when holes or loops need to be remeshed with a minimum number of edges. Works only for surface domains, not solid domains.

Maximum number of edges along loops

`max_NE_for_loops`. Integer greater or equal to 3. Default = `UINT_MAX`

This parameter can be useful when holes or loops need to be remeshed with a limited number of edges. Works only for surface domains, not solid domains.

Display handler

This user-supplied function is used to handle the messages issued by the mesher.

```
typedef void (*display_handler_type) (void* pass_thu,
                                      unsigned level, const char* msg);
```

The `pass_thru` parameter is the pointer set by the user in the operating mode structure.
The level parameter gives the importance of the message:

- +0 ➔ important (for instance entering a major step of the process)
- +1 ➔ somewhat important (minor step of the process)
- ≥ 2 ➔ not serious (debug messages that should not be printed for end-users).

The `msg` parameter is the string message (length <= 255 characters).

Note:
This handler is not called in case of error or warning. At the end of the run, the user must check for an error or a warning in the fields `data_type::error_code` and `data_type::warning_code` and then (in case of error or warning) process the string `data_type::msg1`.

Example:

```
void my_display_hdl (void* pass_thru, unsigned level, const char* msg)
{
```

[23] The angle quality of a rectangle equals to 1 (perfect) whereas its geometric quality is only equal to 1 when the rectangle is a square.


```
   window_type*   my_window = static_cast<window_type*> (pass_thru);
   my_window->show (msg);
}

cm2::surf_remesh_t3::mesher             my_mesher;
cm2:: surf_remesh_t3::mesher::data_type  my_data (pos, connectB);
window_type                        my_window;   // A “window” instance.

my_mesher.mode.display_handler = &my_display_handler;
my_mesher.mode.pass_thru = static_cast<void*> (&my_window);
my_mesher.run (my_data);      // Will call my_display_hdl with “my_window”
                              // in pass_thru parameter.
```

Interrupt handler

`interrupt_hdl`. Default = `NULL`. Used in all modes.

Can be useful for big meshes (over hundreds of thousands of elements).

```
typedef bool (*interrupt_handler_type)( void* pass_thu, double progress);
```

This handler, if any, is called periodically by the remesher to check for a stop signal. When the handler returns `true`, the remesher aborts its current step. If the interruption occurs early in the meshing stage - for instance in the front mesh step - the mesh is invalid, so it is cleared. From the refine step on, however, the user can get a valid mesh upon exit, though probably of poor quality.
An interruption also raises the `CM2_INTERRUPTION` warning.

The `pass_thru` parameter is a pointer set by the user in the operating mode structure (the same parameter is also passed to the display handler).
The parameter `progress` (between 0 and 1) gives a hint about the progress of the remeshing.

Example:

```
bool my_interrupt_handler (void* pass_thu, double progress)
{
   clock_t*   t_limit = static_cast<clock_t*> (pass_thru);
   return clock() > (*t_limit);
}

cm2::surf_remesh_t3::mesher            my_mesher;
cm2::surf_remesh_t3::mesher::data_type my_data (pos, connectB);
clock_t                          my_limit (clock() + 1E3*CLOCKS_PER_SEC);

my_mesher.mode.interrupt_handler = &my_interrupt_handler;
my_mesher.mode.pass_thru = static_cast<clock_t*> (&my_limit);
my_mesher.run (my_data);         // Will stop if duration > 1000 s.
```

```
struct operating_mode_type
{
   double                  min_h;
   double                  max_h;
   double                  max_gradation;
   double                  patch_angle_tolerance;
   double                  fix_tolerance;
   double                  optim_tolerance;
   double                  strain_tolerance;
   bool                    initial_cleanup_flag;
   bool                    solid_flag;
   bool                    remesh_flag;
   bool                    force_even_flag;
   bool                    node_smoothing_flag;
   bool                    node_inserting_flag;
   bool                    node_removing_flag;
   bool                    shell_remeshing_flag;
   bool                    final_optimization_flag;
   bool                    node_smoothing_only_flag;
   bool                    compute_Qh_flag;
   int                     structured_pattern;
   unsigned                nodes_limit;
   unsigned                optim_level;
   double                  shape_quality_weight;
   double                  length_upper_bound;
   unsigned                min_NE_for_loops;
   unsigned                max_NE_for_loops;
   display_handler_type    display_hdl;
   interrupt_handler_type  interrupt_hdl;
   void*                   pass_thru;
};
```

Table 7 – The `cm2::surf_remesh_t3::operating_mode_type` structure (only the data members are shown).

```
struct operating_mode_type
{
   double                  min_h;
   double                  max_h;
   double                  max_gradation;
   double                  patch_angle_tolerance;
   double                  fix_tolerance;
   double                  optim_tolerance;
   double                  strain_tolerance;
   bool                    initial_cleanup_flag;
   bool                    solid_flag;
   bool                    remesh_flag;
   bool                    all_quad_flag;
   bool                    force_even_flag;
   bool                    node_smoothing_flag;
   bool                    node_inserting_flag;
   bool                    node_removing_flag;
   bool                    shell_remeshing_flag;
   bool                    final_optimization_flag;
   bool                    node_smoothing_only_flag;
   bool                    compute_Qh_flag;
   bool                    structured_flag;
   unsigned                nodes_limit;
   unsigned                optim_level;
   double                  shape_quality_weight;
   double                  quadrangle_weight;
   double                  min_Q4_angle_quality;
   double                  length_upper_bound;
   unsigned                min_NE_for_loops;
   unsigned                max_NE_for_loops;
   display_handler_type    display_hdl;
   interrupt_handler_type  interrupt_hdl;
   void*                   pass_thru;
};
```

Table 8 – The `cm2::surf_remesh_q4::operating_mode_type` structure (only the data members are shown).

Most useful fields are “min_h”, “max_h”, “optim_level”, “all_quad_flag” (for CM2 SurfRemesh Q4) and possibly the handlers. Users can also adapt “patch_angle tolerance” to the specificity of his/her surfaces. Other parameters are rarely useful and should be left to expert users only.

III-4 GENERAL SCHEME OF THE REMESHERS

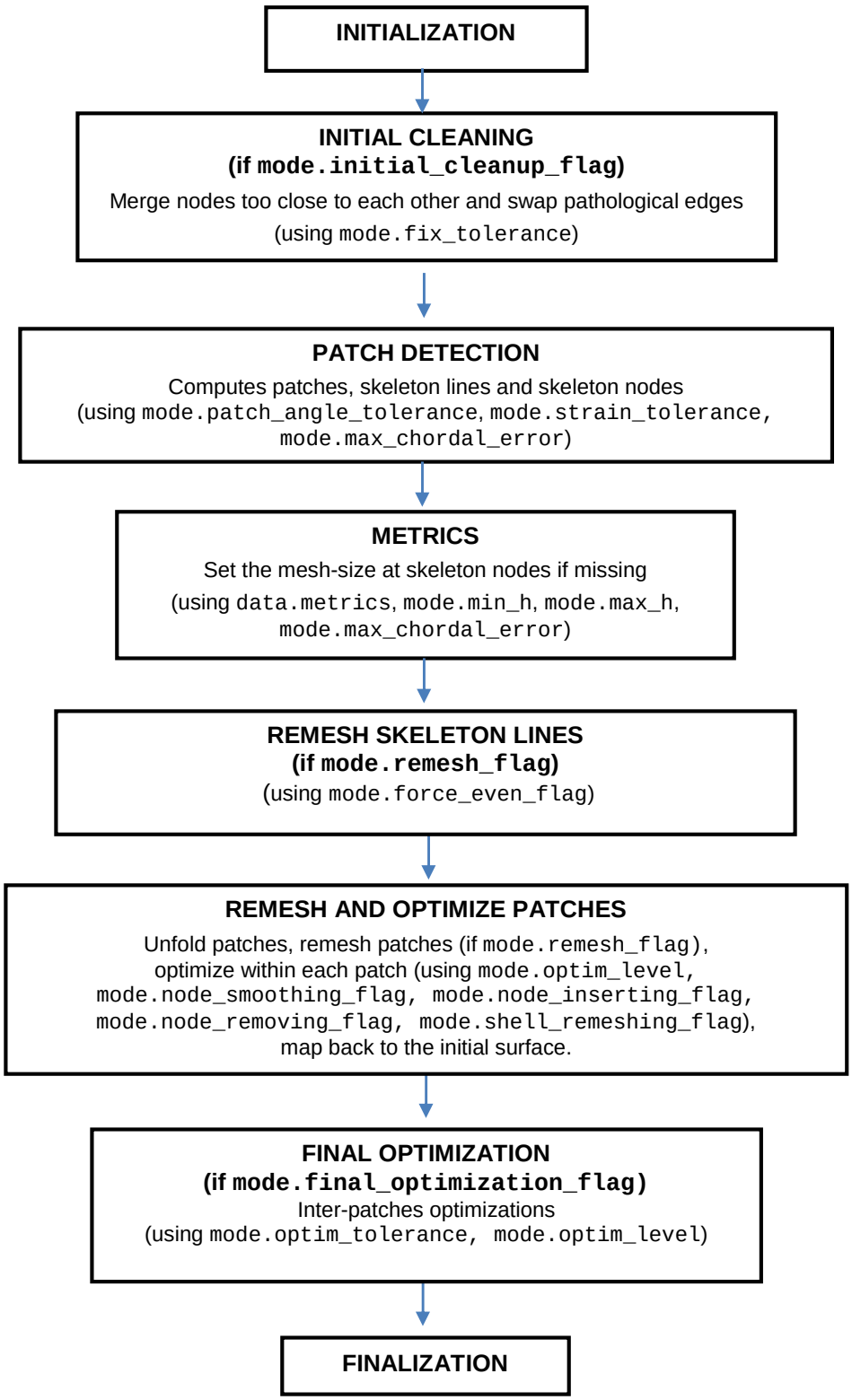
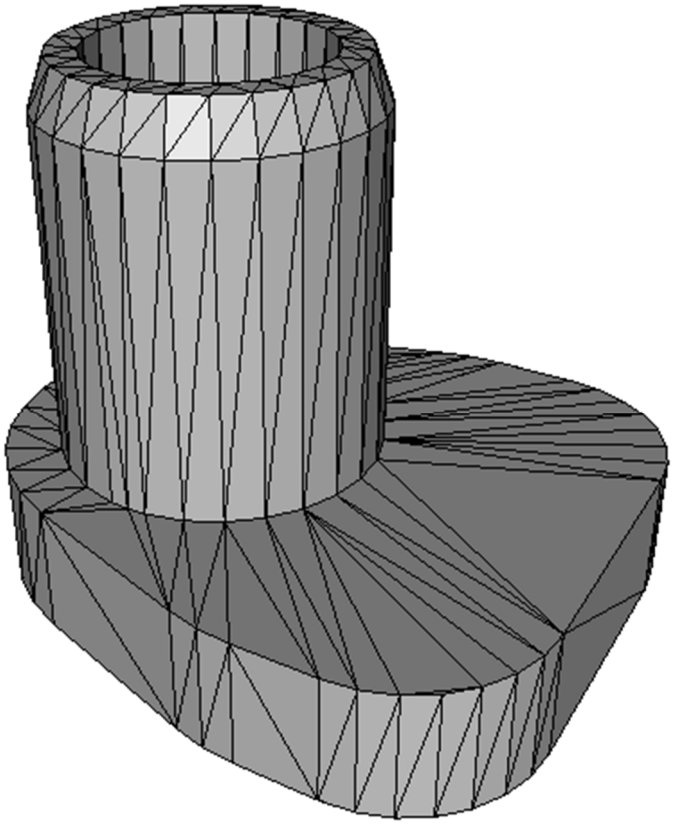


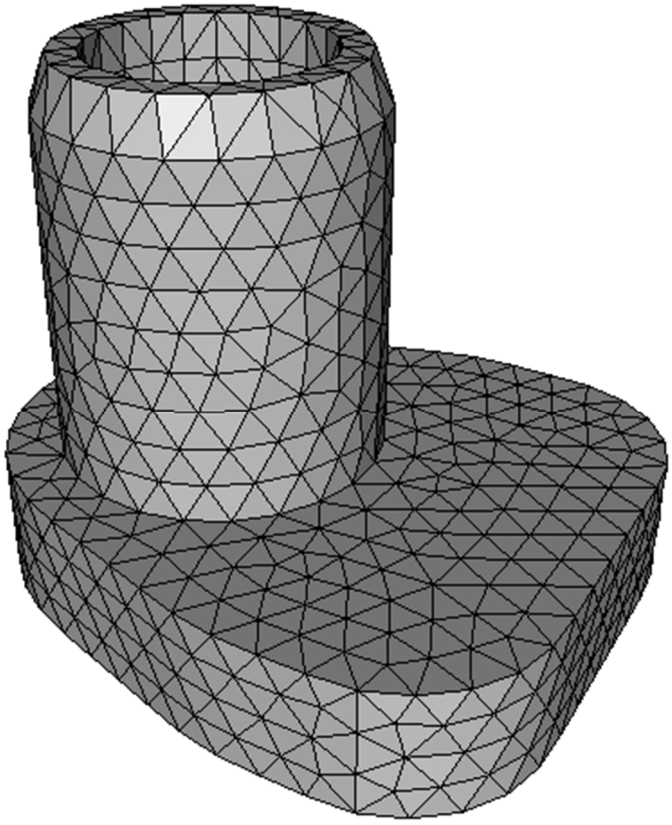
Figure 27 – General scheme of the remeshers.

IV – MESH GALLERY



Initial mesh

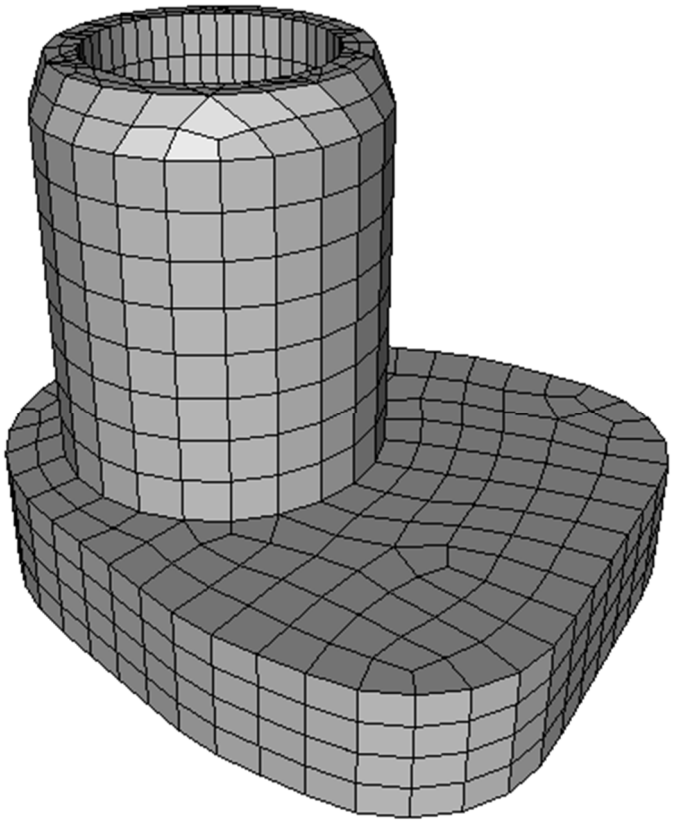
```
Nodes            : 202
Triangles        : 400
Area             : 1.7722E+05
Qavg             : 4.0583E-01
Qmin             : 6.8572E-02
```



New mesh (CM2 SurfRemesh T3)

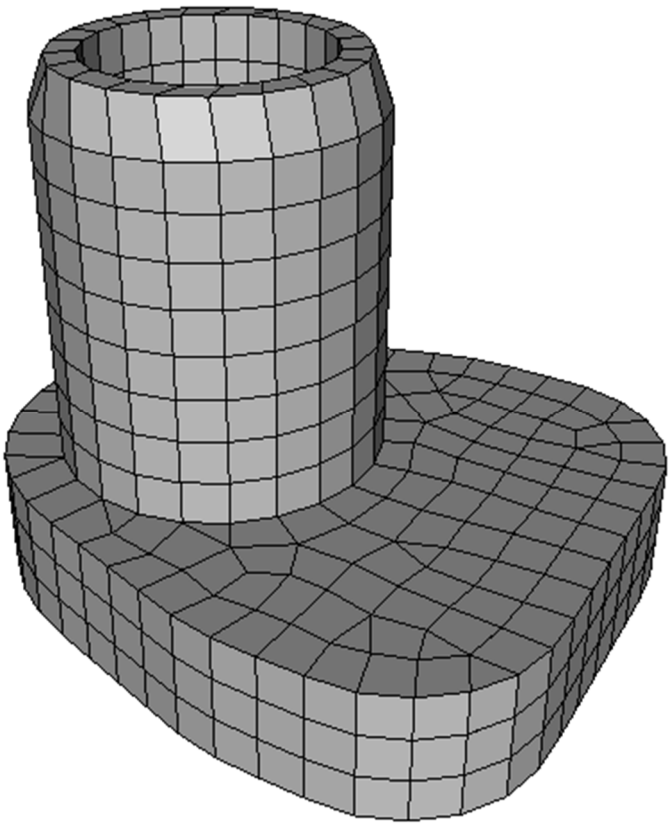
```
Nodes            : 940
Triangles        : 1876
Area             : 1.7670E+05
Qavg             : 9.176E-01
Qmin             : 1.706E+00

Cleaning time    : 0.00 s
Analysis time    : 0.03 s
Remesh time      : 0.02 s
Optim time       : 0.00 s
Total time       : 0.05 s (39987 t/s)
```

The output information given here are only indicative. All runs were done with x64 CM2 libs (Visual Studio 2010 MD build) on Windows 8 x64 with Intel Xeon E3-1270 V2 3.5 GHz (1 thread, turbo boost disabled).



New mesh (CM2 SurfRemesh Q4 in all-quad mode)

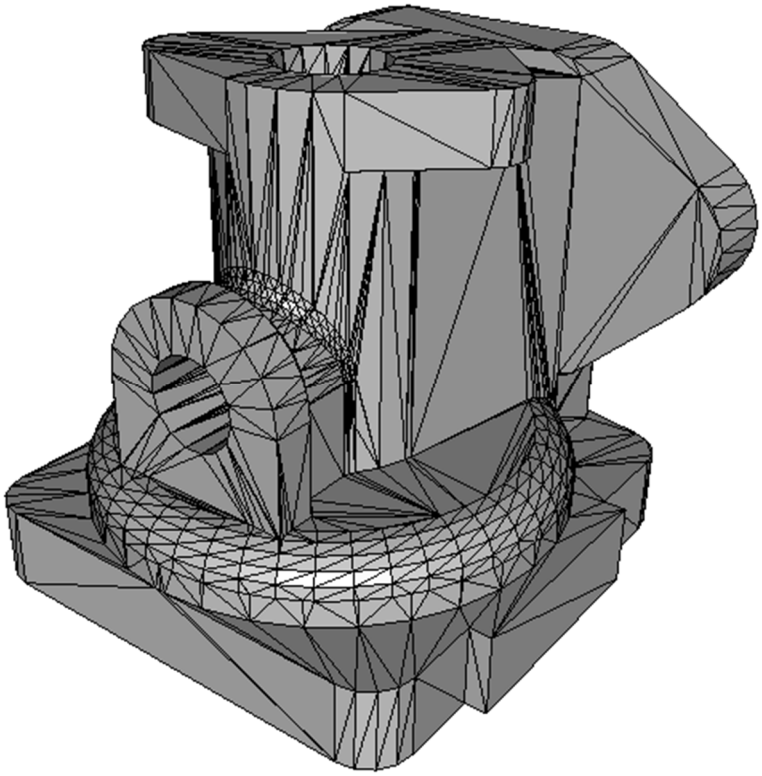
```
Nodes            : 948
Elements         : 946
  Quadrangles    : 946
  (100.00 %, 100.00 %)
  Triangles      : 0
  (0.00 %, 0.00 %)
Area             : 1.7662E+05
Qavg             : 8.437E-01
Qmin             : 2.539E-01

Cleaning time    : 0.00 s
Analysis time    : 0.03 s
Remesh time      : 0.05 s
Optim time       : 0.00 s
Total time       : 0.08 s (12100 e/s)
```



New mesh (CM2 SurfRemesh Q4 in quad-dominant mode)

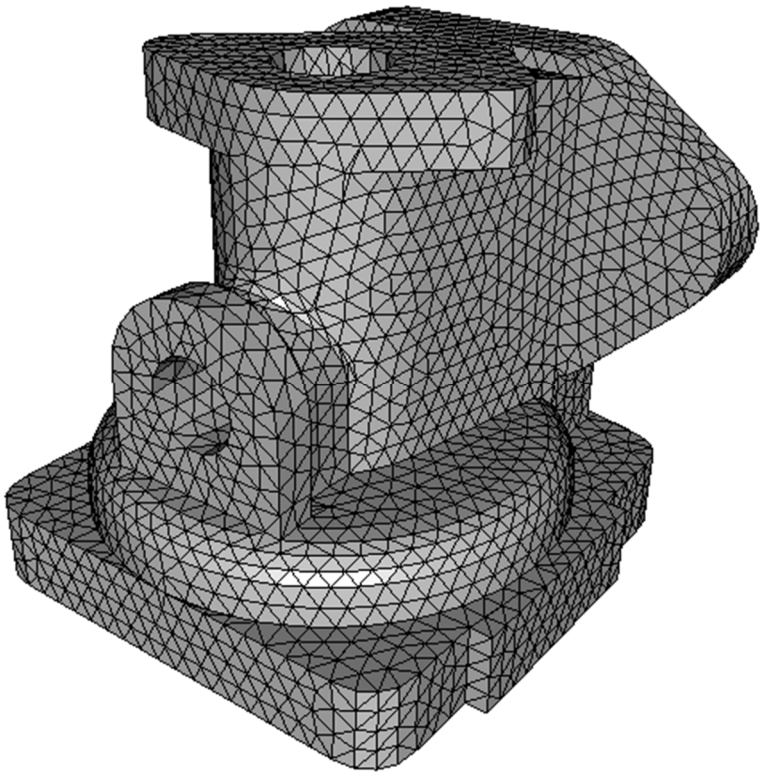
```
Nodes            : 811
Elements         : 825
  Quadrangles    : 793
  (96.12 %, 98.34 %)
  Triangles      : 32
  (3.88 %, 1.66 %)
Area             : 1.7659E+05
Qavg             : 9.128E-01
Qmin             : 5.049E-01

Cleaning time    : 0.00 s
Analysis time    : 0.03 s
Remesh time      : 0.04 s
Optim time       : 0.00 s
Total time       : 0.06 s (13306 e/s)
```

The percent figures indicate the ratio between the number of quads and triangles, and the ratio between the surfaces of quads and triangles.



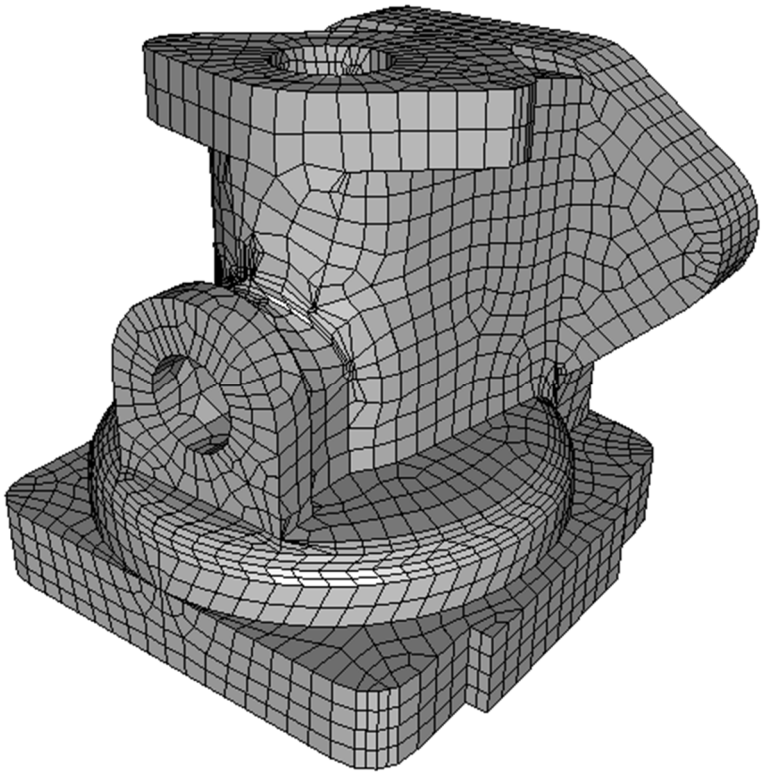
Initial mesh

Nodes	: 1152
Triangles	: 2316
Area	: 8.7787E+04
Qavg	: 3.76E-01
Qmin	: 3.23E-04



New mesh (CM2 SurfRemesh T3)

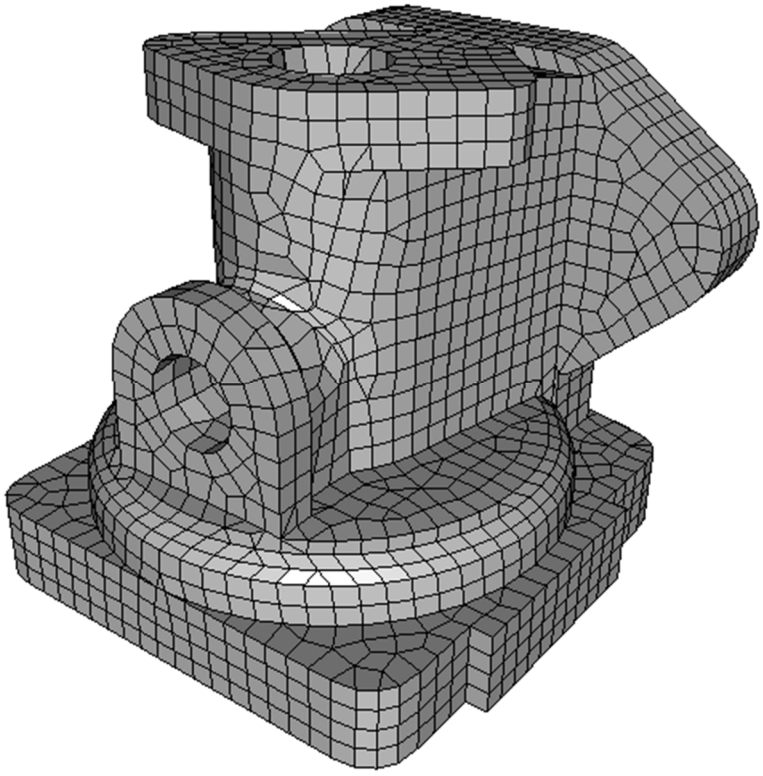
Nodes	: 4234
Triangles	: 8480
Area	: 8.7642E+04
Qavg	: 8.72E-01
Qmin	: 2.46E-02
Cleaning time	: 0.01 s
Analysis time	: 0.24 s
Remesh time	: 0.13 s
Optim time	: 0.05 s
Total time	: 0.44 s
(19405 t/s)	



New mesh (CM2 SurfRemesh Q4 in all-quad mode)

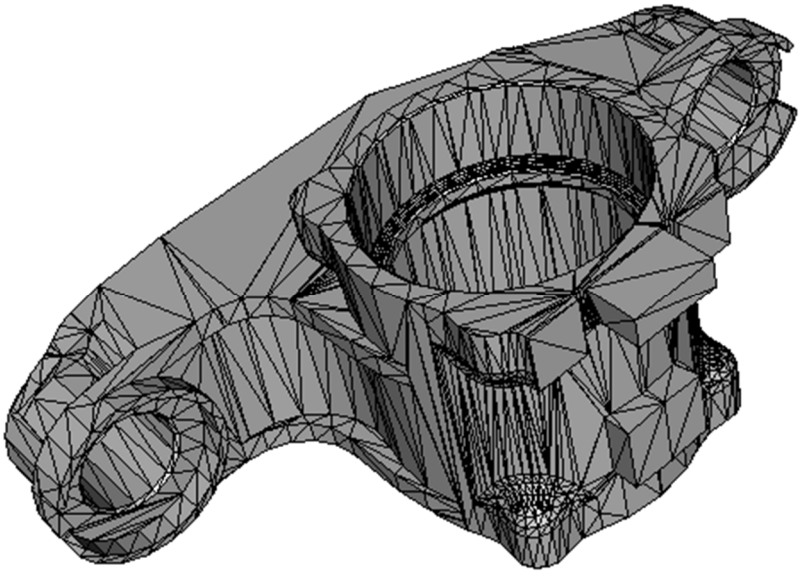
```
Nodes             : 4852
Elements          : 4858
  Quadrangles     : 4858
  (100.00 %, 100.00 %)
  Triangles       : 0
  (0.00 %, 0.00 %)
Area              : 8.7656E+04
Qavg              : 6.98E-01
Qmin              : 1.07E-03

Cleaning time     : 0.02 s
Analysis time     : 0.25 s
Remesh time       : 0.41 s
Optim time        : 0.03 s
Total time        : 0.70 s
(6910 e/s)
```

=> Some quads have a very bad shape quality: rather use the quad-dominant mode.



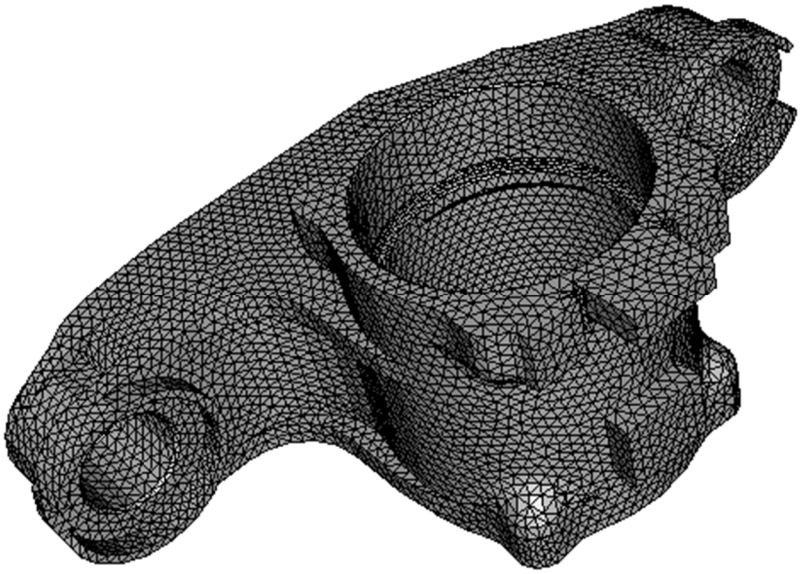
New mesh (CM2 SurfRemesh Q4 in quad-dominant mode)

```
Nodes             : 3825
Elements          : 3937
  Quadrangles     : 3725
  (94.62 %, 97.95 %)
  Triangles       : 212
  (5.38 %, 2.05 %)
Area              : 8.7643E+04
Qavg              : 8.21E-01
Qmin              : 3.14E-02

Cleaning time     : 0.02 s
Analysis time     : 0.25 s
Remesh time       : 0.23 s
Optim time        : 0.03 s
Total time        : 0.53 s
(7414 e/s)
```



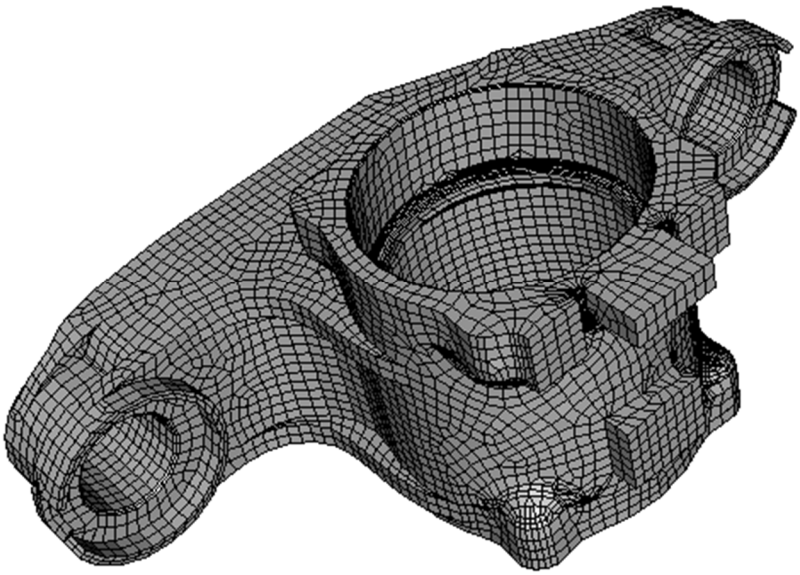
Initial mesh

```
Nodes          : 2674
Triangles      : 5364
Area           : 8.9217E+05
Qavg           : 3.6672E-01
Qmin           : 2.0704E-04
```



New mesh (CM2 SurfRemesh T3)

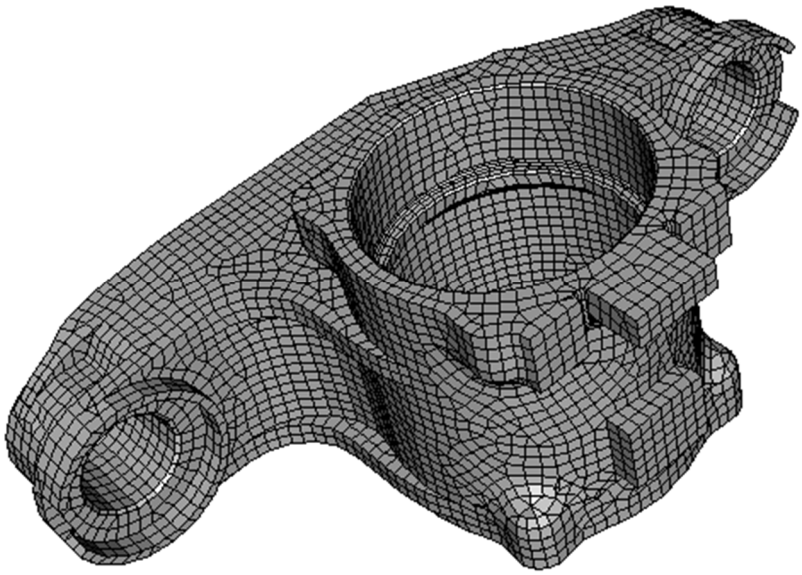
```
New Mesh
Nodes          : 11304
Triangles      : 22624
Area           : 8.9197E+05
Qavg           : 8.60E-01
Qmin           : 3.34E-02

Cleaning time  : 0.03 s
Analysis time  : 0.61 s
Remesh time    : 0.61 s
Optim time     : 0.17 s
Total time     : 1.44 s
(15732 t/s)
```



New mesh (CM2 SurfRemesh Q4 in all-quad mode)

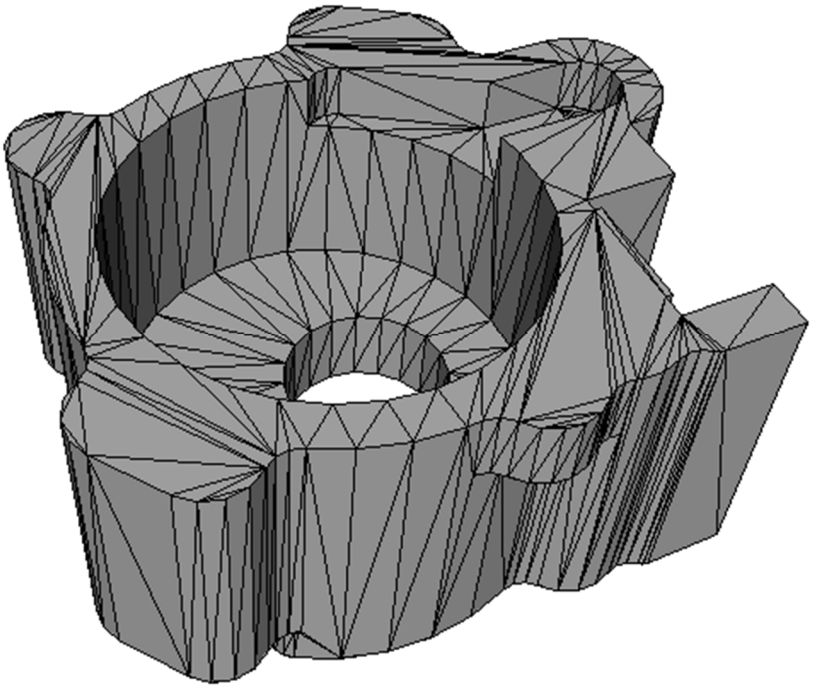
```
Nodes           : 13317
Elements        : 13325
  Quadrangles   : 13325
  (100.00 %, 100.00 %)
  Triangles     : 0
  (0.00 %, 0.00 %)
Area            : 8.9269E+05
Qavg            : 6.83E-01
Qmin            : 2.01E-03

Cleaning time   : 0.03 s.
Analysis time   : 0.61 s.
Remesh time     : 4.69 s.
Optim time      : 0.13 s.
Total time      : 5.47 s.
(2437 e/s)
```

Some quads have a very bad shape quality: => rather use the quad-dominant mode.



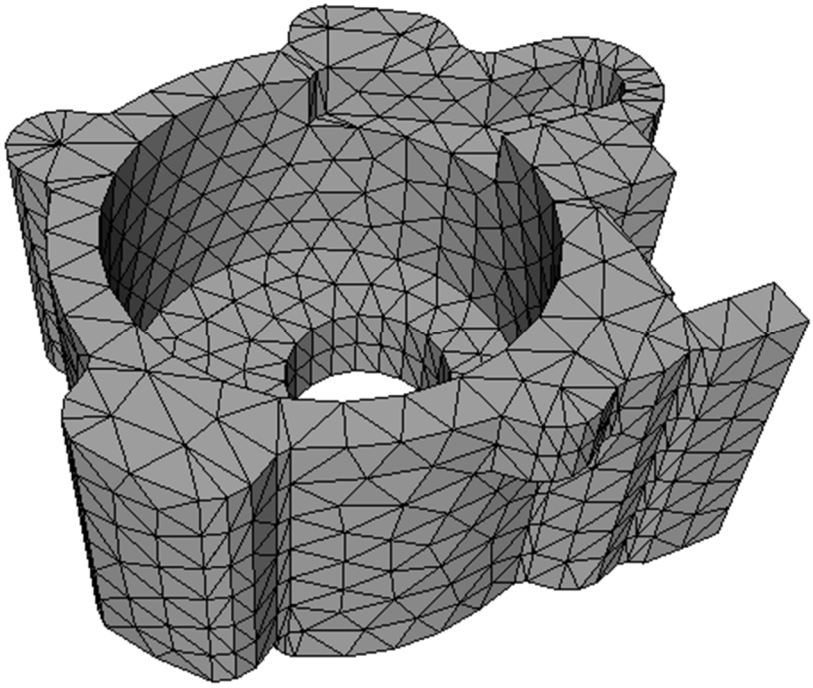
New mesh (CM2 SurfRemesh Q4 in quad-dominant mode)

```
Nodes           : 10160
Elements        : 10413
  Quadrangles   : 9923
  (95.29 %, 98.63 %)
  Triangles     : 490
  (4.71 %, 1.37 %)
Area            : 8.9200E+05
Qavg            : 8.37E-01
Qmin            : 3.345E-02

Cleaning time   : 0.01 s
Analysis time   : 0.61 s
Remesh time     : 1.03 s
Optim time      : 0.13 s
Total time      : 1.80 s
(5795 e/s)
```



Initial mesh

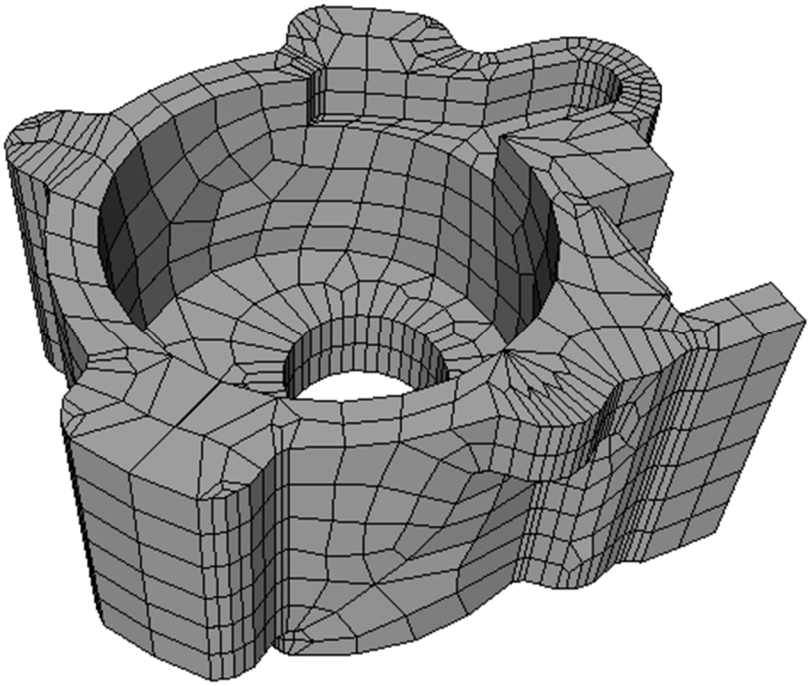
Nodes	: 406
Triangles	: 812
Area	: 6.8760E+04
Qavg	: 3.0524E-01
Qmin	: 2.5909E-03



New mesh
(CM2 SurfRemesh T3)

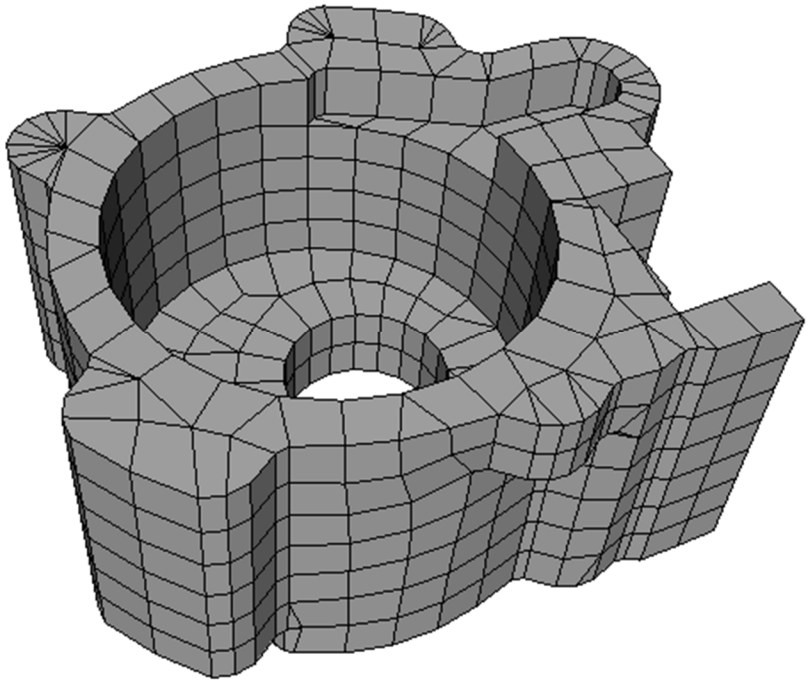
Nodes	: 993
Triangles	: 1986
Area	: 6.8620E+04
Qavg	: 7.79E-01
Qmin	: 1.50E-01

Cleaning time	: 0.00 s
Analysis time	: 0.05 s
Remesh time	: 0.03 s
Optim time	: 0.01 s
Total time	: 0.09 s
(21355 t/s)	



New mesh (CM2 SurfRemesh Q4 in all-quad mode)

```
Nodes            : 1487
Elements         : 1487
  Quadrangles    : 1487
  (100.00 %, 100.00 %)
  Triangles      : 0
  (0.00 %, 0.00 %)
Area           : 6.8609E+04
Qavg           : 5.72E-01
Qmin           : 7.04E-02

Cleaning time    : 0.00 s
Analysis time    : 0.05 s
Remesh time      : 0.14 s
Optim time       : 0.00 s
Total time       : 0.19 s
(7952 e/s)
```



New mesh (CM2 SurfRemesh Q4 in quad-dominant mode)

```
Nodes            : 909
Elements         : 938
  Quadrangles    : 880
  (93.82 %, 98.04 %)
  Triangles      : 58
  (6.18 %, 1.96 %)
Area           : 6.8593E+04
Qavg           : 7.74E-01
Qmin           : 1.50E-01

Cleaning time    : 0.02 s
Analysis time    : 0.05 s
Remesh time      : 0.06 s
Optim time       : 0.00 s
Total time       : 0.13 s
(7504 e/s)
```