maniFEM

USER MANUAL

Cristian Barbarosie and Anca-Maria Toader

version of this manual: 24.05
documents maniFEM release 24.05

This document describes maniFEM, a `C++` library for solving partial differential equations through the finite element method. The name comes from "finite elements on manifolds". ManiFEM was designed with the goal of coping with very general meshes, in particular meshes on Riemannian manifolds, even manifolds which cannot be embedded in $\mathbb{R}^3$, like the torus $\mathbb{R}^2/\mathbb{Z}^2$. Also, maniFEM was written with the goal of being conceptually clear and easy to read. We hope it will prove particularly useful for people who want fine control on the mesh, e.g. for implementing their own meshing or remeshing algorithms.

ManiFEM is just a collection of `C++` classes. It has no user-friendly interface nor graphic capabilities. The user should have some understanding of programming and of `C++`. However, maniFEM can be used at a basic level by people with no deep knowledge of `C++`. A gallery of examples is available at `https://maniFEM.rd.ciencias.ulisboa.pt/gallery/`. In paragraphs 1.8 – 1.10 of this manual, you can find three lists showing maniFEM's competitors, as well as strong and weak points.

In its current version, release 24.05, maniFEM works well for mesh generation, including meshes on quotient manifolds. Non-uniform meshes can be built (paragraphs 3.23 – 3.26), as well as anisotropic meshes (paragraphs 3.27 - 3.30). Lagrange finite elements of degree one and two are implemented for triangular and quadrangular cells; Lagrange finite elements of degree one are implemented for tetrahedral cells; many other types of finite elements are still to be implemented. In the future, variational formulations will be implemented as `C++` objects, thus allowing for compact and elegant code. A changelog is included at the end of this manual, before the index. To check which version of maniFEM is installed in your computer, see at the beginning of the file `maniFEM.h`.

A component of maniFEM, `MetricTree`, can be used independently. It is a generalization of quad-trees for metric spaces. See paragraph 12.14.

Some examples use the `Eigen` library for storing matrices and for solving systems of linear equations, see `https://eigen.tuxfamily.org/index.php?title=Main_Page`. Many drawings in this manual have been produced using `gmsh`, see `https://gmsh.info/`

ManiFEM is being developed by Cristian Barbarosie[1] and Anca-Maria Toader.

The home page of maniFEM is `https://maniFEM.rd.ciencias.ulisboa.pt` (where this manual can be found).

To use maniFEM, visit `https://codeberg.org/cristian.barbarosie/maniFEM`, choose a release and download all files to some directory in your computer. Current code may be unstable, releases are stable. You can then run the examples in this manual: just `make run-parag-1.1` for the example in paragraph 1.1, `make run-parag-2.7` for the example in paragraph 2.7, and so on. Paragraph 11.14 gives more details.

[1] `cristian.barbarosie@gmail.com`

Structure of this manual

This manual is divided in parts and sections describing maniFEM with increasing degree of technical detail.

1. General description A quick overview of maniFEM's capabilities.

Part I. Basic usage Sections 2 to 7 show how maniFEM can be used for building meshes and solving PDEs using high-level commands, in a style somewhat similar to `FreeFEM` or `fenics`. It can be used by people with no deep knowledge of `C++`.

2. Meshes and manifolds; patchwork Shows how to build meshes by joining simple shapes, like patches, some of them on manifolds.

3. Frontal mesh generation; knitting Shows how to build meshes starting from their boundary alone, some of them on manifolds.

4. Exchanging information with third-party software About writing and reading information to and from files.

5. Functions and variational formulations Some details about functions, a lot of work to do.

6. Finite elements and integrators Shows examples of finite element computations.

7. Quotient manifolds Describes meshes on quotient manifolds.

8. Miscellanea Gives informations which do not fit in other sections of the manual.

Part II. Advanced Usage Sections 9 and 10 show how maniFEM can be used for controlling the details of the mesh. We hope it will be particularly useful for people interested in implementeing their own meshing or remeshing algorithms.

9. Cells, meshes, iterators Gives details about neighbourhood relations between cells in a mesh, orientation and iterators.

10. Remeshing Some simple examples showing how to manipulate a mesh.

Part III. For programmers Sections 11 and 12 are aimed at people interested in maintaining and developing maniFEM.

11. Technical details Various implementation details, programming style, compilation options, frequent errors.

12. Internal details Mainly drawings intended to support the developer in understanding the source code.

13. Frequently asked questions

Aknowledgements

Development of maniFEM is supported by national funding from FCT – *Fundação para a Ciência e a Tecnologia* (Portugal), through *Faculdade de Ciências da Universidade de Lisboa* and *Centro de Matemática, Aplicações Fundamentais e Investigação Operacional*, project UIDB/04561/2020.

Anabela Silva contributed with drawings and source text coloring.

Armando Machado contributed with comments in paragraph 8.2.

Colophon

See file `misc/colophon.md` at `https://codeberg.org/cristian.barbarosie/maniFEM`

Contents

1. General description

This section is a quick tour through maniFEM's capabilities. There is also a gallery with several examples at `https://maniFEM.rd.ciencias.ulisboa.pt/gallery/`

1.1. An elementary example

In this paragraph, we show how to build a rectangular mesh immersed in $\mathbb{R}^3$ and then compute the integral of a given function. Paragraph 1.4 shows a purely two-dimensional example.

parag-1.1.cpp

```
#include "maniFEM.h"

using namespace maniFEM;
using namespace std;

int main ()

{  // we choose our (geometric) space dimension :
   Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );

   // xyz is a map defined on our future mesh with values in RR3 :
   Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );

   // we can extract components of xyz using the [] operator
   Function x = xyz[0], y = xyz[1], z = xyz[2];

   // Let's build a rectangular mesh. First, the four corners :
   Cell SW ( tag::vertex );  x(SW) = -1.;  y(SW) = 0.;  z(SW) = 0.;
   Cell SE ( tag::vertex );  x(SE) =  1.;  y(SE) = 0.;  z(SE) = 0.;
   Cell NE ( tag::vertex );  x(NE) =  1.;  y(NE) = 1.;  z(NE) = 0.;
   Cell NW ( tag::vertex );  x(NW) = -1.;  y(NW) = 1.;  z(NW) = 1.;
```

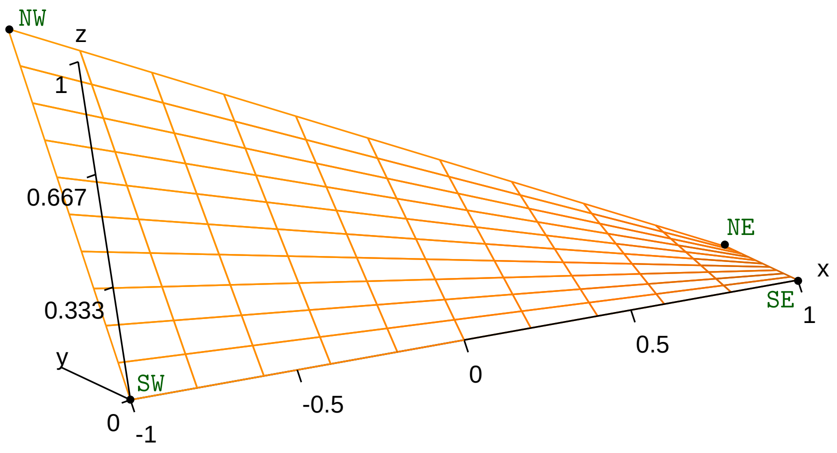


Figure 1.1.: a twisted rectangular mesh

```
   // alternative syntax for declaring vertices :
   // Cell SW ( tag::vertex, tag::of_coords, { -1., 0., 0. } )
   // Cell SE ( tag::vertex, tag::of_coords, {  1., 0., 0. } )
   // Cell NE ( tag::vertex, tag::of_coords, {  1., 1., 0. } )
   // Cell NW ( tag::vertex, tag::of_coords, { -1., 1., 1. } )

   // we access the coordinates of a point using the () operator :
   cout << "coordinates of NW : " << x(NW) << " " << y(NW) << " " << z(NW) << endl;

   // now build the four sides of the rectangle :
   Mesh south ( tag::segment, SW .reverse(), SE, tag::divided_in, 10 );
   Mesh east  ( tag::segment, SE .reverse(), NE, tag::divided_in, 10 );
   Mesh north ( tag::segment, NE .reverse(), NW, tag::divided_in, 10 );
   Mesh west  ( tag::segment, NW .reverse(), SW, tag::divided_in, 10 );

   // and now the rectangle :
   Mesh rect_mesh ( tag::rectangle, south, east, north, west );

   // we may want to visualize the resulting mesh
   // here is one way to export the mesh in the "msh" format :
   rect_mesh .export_to_file ( tag::gmsh, "rectangle.msh" );

   // let's define a symbolic function
   Function f = x*x + 1/(5+y);

   // and compute its integral on the rectangle,
   // using, in each rectangular cell, Gauss quadrature with 9 points :
   Integrator integr ( tag::Gauss, tag::quad_9 );

   cout << "integral of f " << integr ( f, tag::on, rect_mesh ) << endl;

}  // end of main
```

Paragraph 11.2 explains the coloring conventions observed in this manual for `C++` code.

To run this example, you will need a recent `C++` compiler and the `make` utility. Visit `https://codeberg.org/cristian.barbarosie/maniFEM`, choose a release and download all files to some directory in your computer. Paragraph 11.14 gives more details. Launch the program through the command `make run-parag-1.1`; a file `rectangle.msh` should appear in the working directory. You may view the mesh using the software `gmsh`.

Expressions like `tag::of_dim` and `tag::vertex` are objects belonging to the `namespace tag`; we use them as arguments to many functions. See paragraph 11.3 for some details.

Paragraph 5.1 explains why we build `Function xyz` with arguments `tag::Lagrange, tag::of_degree, 1`.

When declaring a segment `Mesh`, we must `reverse` the first vertex (paragraph 1.2 discusses the `reverse` method). Paragraph 1.4 explains why we build the rectangle based on its four sides rather than on its four vertices.

Note that in this example we do not have exact control on the shape of the surface being meshed. The coordinates of the inner vertices are defined rather vaguely by interpolating the coordinates of the four corners. See sections 2 and 3 for ways to precisely define a submanifold in $\mathbb{R}^3$ and mesh (a bounded region of) it.

1.2. Cells and meshes

In maniFEM, all basic constituents of meshes are called "cells". Points are zero-dimensional cells, segments are one-dimensional cells, triangles are two-dimensional cells, and so on.

Roughly speaking, a mesh is a collection of cells of the same dimension. For efficiency reasons, meshes keep lists of cells of lower dimension, too. For instance, the mesh built in paragraph 1.1 is roughly a list of two-dimensional cells (quadrilaterals), but lists of segments and points are also kept. This represents quite some amount of redundant information, but it is useful e.g. for quickly sweeping over all vertices of a mesh.[1]

A cell of dimension $d > 0$ is defined by its boundary, which in turn is a mesh of dimension $d - 1$. The boundary of a segment is a (zero-dimensional) mesh made of two points. The boundary of a triangle is a one-dimensional mesh made of three segments. Thus, a segment is essentially a pair of points, a triangle is essentially a triplet of segments, and so on.

Cells and meshes are oriented. An orientation of a mesh is just an orientation for each of its component cells; these orientations must be mutually compatible. Although this is not how the orientation is implemented internally (see paragraph 11.4), an oriented point can be conceived simply as a point with a sign attached (1 or -1). The orientation of a cell of dimension higher than zero is given by an orientation of its boundary, which is a lower-dimensional mesh.

Thus, an oriented segment is essentially a pair of points, one of which has a -1 attached, the other having a 1. We call the former "base" and the latter "tip". These signs are related to integration of functions along that segment. The integral of a function of one variable is equal to the value of the primitive function at one end of the segment minus the value of the primitive at the other end.

An oriented triangle is essentially a triplet of segments, each one with its own orientation. The orientations must be compatible to each other in the sense that each vertex must be seen as positive by one of the segments and as negative by another one. An oriented tetrahedron can be identified with four triangles, each one with its own orientation. In such a tetrahedron, each segment must be seen as positive by one of the triangles and as negative by another one.

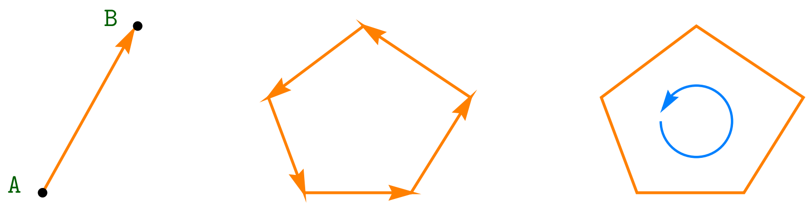


Figure 1.2.: oriented segment, oriented pentagon

We can think of an oriented segment as an arrow pointing from its negative extremity (base) towards its positive extremity (tip). We can think of an oriented polygon as having an arrow attached to each of it sides, or we can imagine a small oriented circle inside the polygon.

A one-dimensional oriented mesh can be thought of as a chain of arrows, each one pointing to the next segment's base (like in figure 1.3 left). A two-dimensional oriented triangular mesh can be thought of as a web of triangles, each triangle having a small oriented circle inside (like in figure 1.3 right). The orientations of neighbour cells must be compatible: each segment must be seen in opposite orientations from the point of view of its two neighbour triangles.

[1] Actually, the implementation details are more complicated. There are different kinds of meshes, some of them keep more information (lists of cells) and are faster; others are slower but lighter in terms of memory occupied. See e.g. paragraph 11.6.

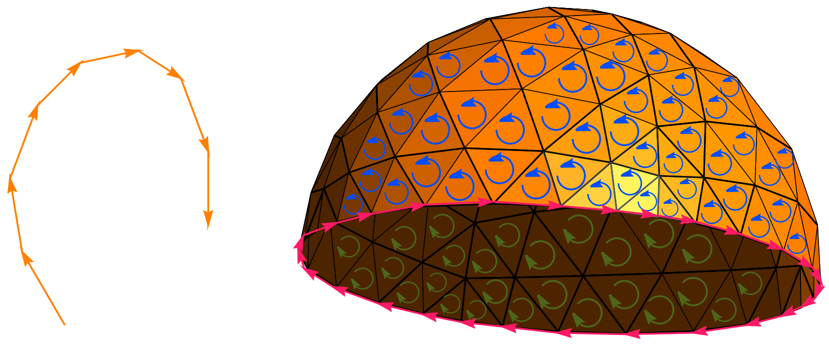


Figure 1.3.: oriented meshes of dimension one and two

The above description of the orientation of a two-dimensional mesh does not depend on the surrounding space. A mesh can be immersed in some Euclidian space $\mathbb{R}^d$ or not. As an extreme example, vertices may even have no coordinates at all. However, in the case of a two-dimensional mesh immersed in $\mathbb{R}^3$, the right hand rule establishes a correspondence between the oriented circle described above and an arrow normal to the surface being meshed. Note that the right hand rule is a convention based on the assuption that the surrounding space $\mathbb{R}^3$ has a certain orientation.

Note also that an orientation of a mesh defines an orientation of its boundary. In figure 1.3 (right) we can see the orientation of the boundary of the mesh, shown with pink arrows. This convention is used by Stokes' theorem.

Figure 1.4 shows two opposite orientations of a mesh, together with the corresponding orientation of its boundary.

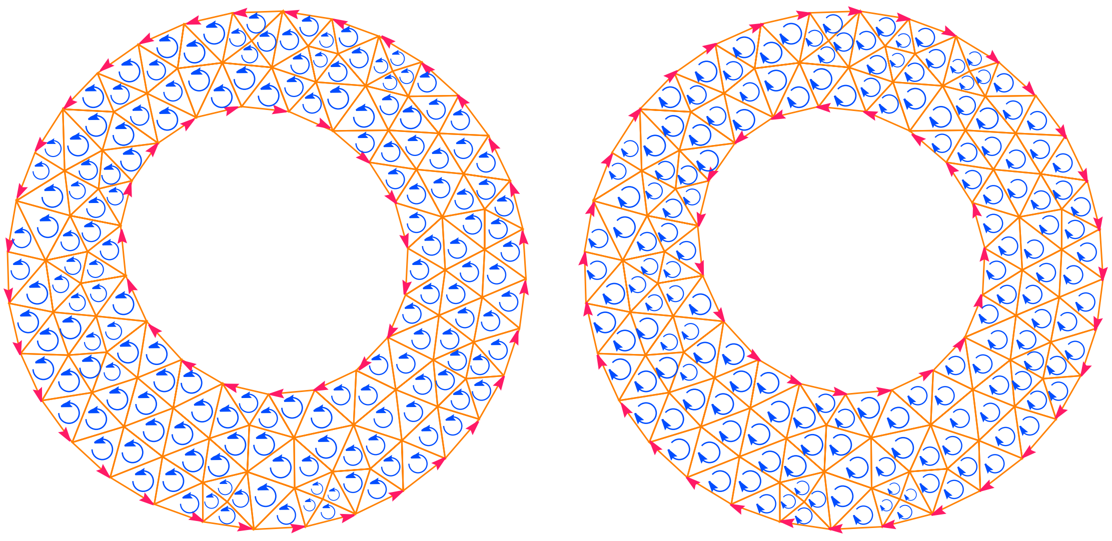


Figure 1.4.: mesh with two opposite orientations

Often, when building a `Mesh`, maniFEM chooses the orientation based on the orientation of the boundary, as provided by the user. For instance, in a statement like

```
Mesh rect_mesh ( tag::rectangle, south, east, north, west );
```

(used in paragraphs 1.1, 1.4 and many others) the orientation of `rect_mesh` is defined by the orientation of its four sides (which must be mutually compatible). If we switch all four orientations, we will obtain the same mesh with the opposite orientation. This can be done either by changing the declaration of each side (switching the first vertex with the last) or by replacing (in the above declaration of `rect_mesh`) `south` by `south .reverse()`, `east` by `east.` `.reverse()` and so on. The `reverse` method is explained below. If we switch the orientation of one side only, maniFEM will complain that the orientations are not compatible with each other.

In paragraphs 1.4 and 3.18, special attention is given to the orientation of the common boundary of meshes to be subsequently joined. In frontal mesh generation (described in section 3) we often provide the boundary of the future mesh, whose orientation determines the output of the algorithm. See also paragraph 10.5.

ManiFEM distinguishes between cells (which are geometric entities) and finite elements. We may have different finite elements whose geometry is the same. For instance, there are triangular Lagrange finite elemens of different degrees; they will all dock on the same triangular cell. Paragraph 6.1 gives more details.

Cells have a `reverse` method returning the reversed cell. Segment `Cell`s have methods `base` and `tip` returning their extremities. For instance :

```
Cell A ( tag::vertex );  Cell B ( tag::vertex );
assert ( A .is_positive() );
assert ( not A .reverse() .is_positive() );
assert ( A .reverse() .reverse() == A );
Cell AB ( tag::segment, A .reverse(), B );  // here, AB is a segment Cell, not a Mesh
assert ( AB .base() == A .reverse() );
assert ( AB .tip() == B );

Cell BA = AB .reverse();
assert ( BA .base() == B .reverse() );
assert ( BA .tip() == A );
assert ( BA .reverse() == AB );
```

Paragraph 9.5 gives more details about the orientation of cells. See also paragraph 11.4.

Cells are topological entities; they carry no geometric information. In particular, points do not have coordinates. Coordinates are stored (semi-)externally, see paragraphs 5.1 and 13.6.

`Mesh`es have a `reverse` method, too. It is used mainly when we want to `join` meshes having a common piece of boundary; see paragraphs 1.4, 2.1, 2.7, 2.8, 2.15, 3.18, 3.19, 3.24, 7.9.

Below we show a low-level way of building a tiny mesh, made of two triangles only. We stress that there are shorter and more elegant mesh constructors, see paragraphs 1.1, 1.4 – 1.6, sections 2 and 3.

```
Cell C ( tag::vertex );
Cell BC ( tag::segment, B .reverse(), C );
Cell CA ( tag::segment, C .reverse(), A );
Cell ABC ( tag::triangle, AB, BC, CA );
Cell D ( tag::vertex );
Cell AD ( tag::segment, A .reverse(), D );
Cell DB ( tag::segment, D .reverse(), B );
Cell BAD ( tag::triangle, AB .reverse(), AD, DB );
Mesh msh ( tag::fuzzy, tag::of_dim, 2 );
ABC .add_to_mesh ( msh );
BAD .add_to_mesh ( msh );
```

1.3. Functions

Minimalist programs like the pieces of code presented in paragraph 1.2 (consisting mainly of manipulation of `Cells`) work without declaring a `Manifold` and without `Functions`. However, most `Mesh` constructors will try to interpolate geometric coordinates (which are `Functions`) based on the shape of the current `Manifold`.

`Functions` in maniFEM can be divided roughly in two categories. Some keep numeric values, like `x`, `y` and `z` encountered in many places in this manual. After building a function like

```
Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
```

the process of creating a new vertex will change. Space in the computer's memory will be reserved for `d` values of type `double` (`d` being the dimension of the `Manifold`, here `RR3`) for each new vertex. The components of `xyz` can be extracted like in

```
Function x = xyz[0], y = xyz[1], z = xyz[2];
```

They are mere handlers through which we access the above referred `double` values, like in `x(A) = 1.` or `double sum = x(A) + y(B) + z(C)`. The interaction between the `Function` object and the values associated to each vertex involves other layers, one of which is a `Field` object.

Other `Functions` are not related with space in the computer's memory (they have no subjacent `Field`). For instance, there are arithmetic expressions like

```
Function norm = power ( x*x + y*y + z*z, 0.5 );
```

We can still access the value of such a `Function` at a cell like in `double n = norm(A)`; the computer will return the value of the corresponding arithmetic expression, replacing the symbols `x`, `y` and `z` by the corresponding values at cell `A`. But it makes no sense to change the value of `norm` at cell `A`. Thus, statements like `x(A) = 1.` work fine, while `norm(A) = 1.` produces a run-time error.

The `Function::deriv` method performs symbolic differentiation :

```
Function d_norm_d_x = norm .deriv (x);
Function d_norm_d_y = norm .deriv (y);
```

Integrals of `Functions` are computed by means of `Integrators`, either alone or coupled with a `FiniteElement`. Section 6 shows several examples.

Paragraph 5.1 gives more details on `Functions`.

1.4. Joining meshes

The example in paragraph 1.1 could have been shortened had we used the overloaded version of the `Mesh` constructor with `tag::rectangle` which accepts the four corners as arguments. This overloaded version exists in maniFEM, but we prefer to build meshes in a structured way, first corners, then sides and then the plane region. This has the advantage that one can build more complex meshes from simple components. For instance, one can build an L-shaped mesh by joining three rectangular meshes :

parag-1.4.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
// in this example, we do not need to give names to the two components of xy
```

```
  Cell A ( tag::vertex, tag::of_coords, { -1., 0.  } );
  Cell B ( tag::vertex, tag::of_coords, {  0., 0.  } );
  Cell C ( tag::vertex, tag::of_coords, {  0., 0.5 } );
  Cell D ( tag::vertex, tag::of_coords, { -1., 0.5 } );
  Cell E ( tag::vertex, tag::of_coords, {  0., 1.  } );
  Cell F ( tag::vertex, tag::of_coords, { -1., 1.  } );
  Cell G ( tag::vertex, tag::of_coords, {  1., 0.  } );
  Cell H ( tag::vertex, tag::of_coords, {  1., 0.5 } );

  Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 10 );
  Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in,  8 );
  Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, 10 );
  Mesh DA ( tag::segment, D .reverse(), A, tag::divided_in,  8 );
  Mesh CE ( tag::segment, C .reverse(), E, tag::divided_in,  7 );
  Mesh EF ( tag::segment, E .reverse(), F, tag::divided_in, 10 );
  Mesh FD ( tag::segment, F .reverse(), D, tag::divided_in,  7 );
  Mesh BG ( tag::segment, B .reverse(), G, tag::divided_in, 12 );
  Mesh GH ( tag::segment, G .reverse(), H, tag::divided_in,  8 );
  Mesh HC ( tag::segment, H .reverse(), C, tag::divided_in, 12 );

  Mesh ABCD ( tag::rectangle, AB, BC, CD, DA );
  Mesh CEFD ( tag::rectangle, CE, EF, FD, CD .reverse() );
  Mesh BGHC ( tag::rectangle, BG, GH, HC, BC .reverse() );

  Mesh L_shaped ( tag::join, ABCD, CEFD, BGHC );
```

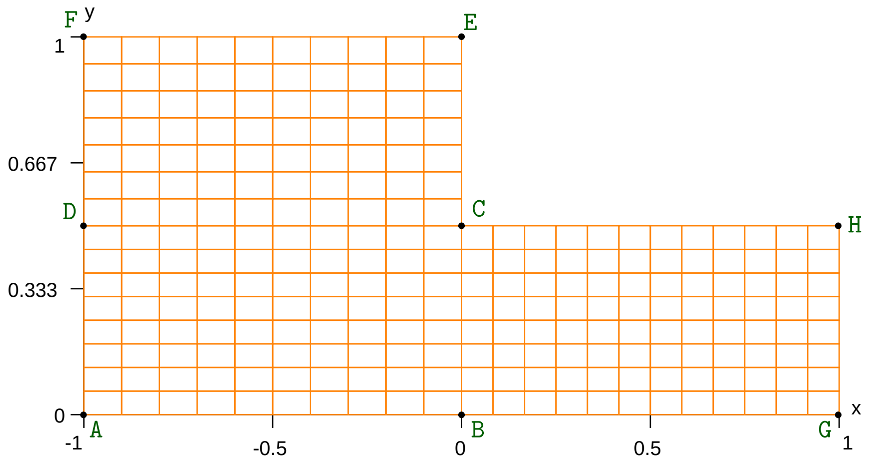


Figure 1.5.: an L-shaped mesh

Meshes in $\mathbb{R}^2$ like the one above may be exported in the `msh` format or directly drawn in `PostScript`, by one of the two statements below

```
L_shaped .export_to_file ( tag::gmsh, "L-shaped.msh" );
L_shaped .export_to_file ( tag::PostScript, "L-shaped.eps" );
// tag::PostScript and tag::eps are interchangeable :
// L_shaped .export_to_file ( tag::eps, "L-shaped.eps" )
```

Note that, in *maniFEM*, cells and meshes are oriented (see paragraph 1.2). To build `CEFD` one must use not `CD` but its reverse; to build `BGHC` one must use not `BC` but its reverse. In figure 1.6 we see a zoom around `C`; the three rectangular meshes have been separated just for visualization purposes. The drawing illustrates that the common boundary has a certain orientation when seen from one mesh and has the opposite orientation when seen from the neighbour mesh. If we do not respect these orientations, *maniFEM* will be unable to `join` these meshes.

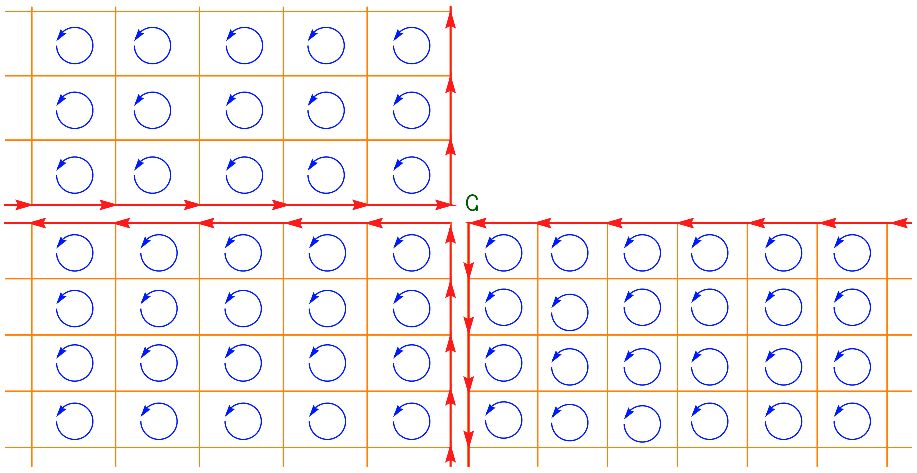


Figure 1.6.: a zoom around `C`

We can imagine a magnet attached to each face of a cell. For two neighbour cells, the common face will have two magnets, one from each cell. If their polarities are not opposite, the magnets will not cling. Likewise, the common boundary of two meshes that we intend to join can be thought of as a pair of magnets. If their polarities are not opposite, the two meshes will not cling.

Note also that if we define the rectangles based on their vertices instead of their sides, the `Mesh` constructor with `tag::join` will not work properly. For instance, the two rectangles defined by

```
Mesh ABCD ( tag::rectangle, A, B, C, D, tag::divided_in, 10, 8 );
Mesh CEFD ( tag::rectangle, C, E, F, D, tag::divided_in, 7, 10 );
```

cannot be joined[2] because the side `CD` of `ABCD` has nothing to do with the side `DC` of `CEFD`. These two sides are one-dimensional meshes both made of 10 segments but with different interior points (only `C` and `D` are shared) and different segments. In contrast, `CD` and `CD.reverse()` share the same 11 points and the same 10 segments (reversed).

Paragraph 9.2 shows a more complex use of the `Mesh` constructor with `tag::join`.

Incidentally, note that the `Mesh` constructor with `tag::rectangle` accepts any position for the vertices. Thus, you can use it to build any quadrilateral; the inner vertices' coordinates are simply interpolated from the coordinates of vertices on the boundary, as shown in paragraphs 1.6 and 2.1. This can be done even in more than two (geometric) dimensions, like in paragraphs 1.1, 2.8, 2.10, 2.12 and 2.13. Here, tags `rectangle`, `quadrilateral` and `quadrangle` can be used interchangeably.

An almost identical mesh is built in paragraph 2.1.

This is a good place to mention that *maniFEM* can also import a mesh from a file in `msh` format:

```
Mesh msh ( tag::import, tag::gmsh, "filename.msh" );
```

[2] Actually, they can be joined but the resulting mesh will have a crack along `CD` – probably not what the user wants.

For geometric dimension higher than 3, the msh file exported with method Mesh::export_to_file will not be readable by gmsh. However, it can still be used by maniFEM to read back (import) the Mesh object. See also paragraphs 4.1 and 4.2.

1.5. Triangular meshes

We can also build meshes on triangular domains and join them as we wish :

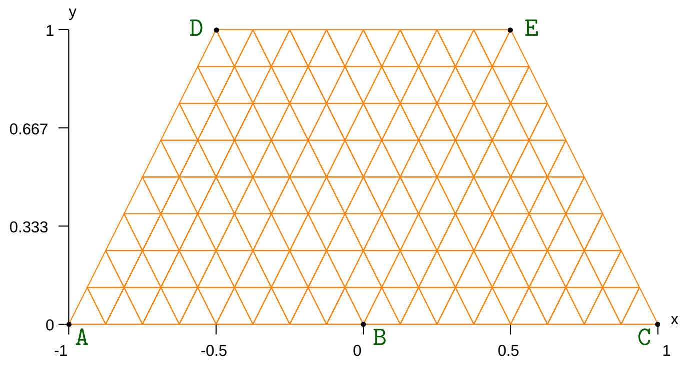


Figure 1.7.: mesh obtained by joining three triangles

parag-1.5.cpp
```
Cell A ( tag::vertex );  x(A) = -1. ;  y(A) = 0.;
Cell B ( tag::vertex );  x(B) =  0. ;  y(B) = 0.;
Cell C ( tag::vertex );  x(C) =  1. ;  y(C) = 0.;
Cell D ( tag::vertex );  x(D) = -0.5;  y(D) = 1.;
Cell E ( tag::vertex );  x(E) =  0.5;  y(E) = 1.;

Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 8 );
Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in, 8 );
Mesh AD ( tag::segment, A .reverse(), D, tag::divided_in, 8 );
Mesh BD ( tag::segment, B .reverse(), D, tag::divided_in, 8 );
Mesh BE ( tag::segment, B .reverse(), E, tag::divided_in, 8 );
Mesh CE ( tag::segment, C .reverse(), E, tag::divided_in, 8 );
Mesh ED ( tag::segment, E .reverse(), D, tag::divided_in, 8 );

Mesh ABD ( tag::triangle, AB, BD, AD .reverse() );
Mesh BCE ( tag::triangle, BC, CE, BE .reverse() );
Mesh BED ( tag::triangle, BE, ED, BD .reverse() );

Mesh three_tri ( tag::join, ABD, BCE, BED );
```

1.6. Mixing triangles and rectangles

It is possible to have triangles and quadrilaterals mixed in the same mesh :

parag-1.6.cpp
```
Mesh ABD ( tag::triangle, AB, BD, AD .reverse() );
Mesh BCE ( tag::triangle, BC, CE, BE .reverse() );
Mesh BEFD ( tag::quadrangle, BE, EF, FD, BD .reverse() );
Mesh two_tri_one_rect ( tag::join, ABD, BEFD, BCE );
```

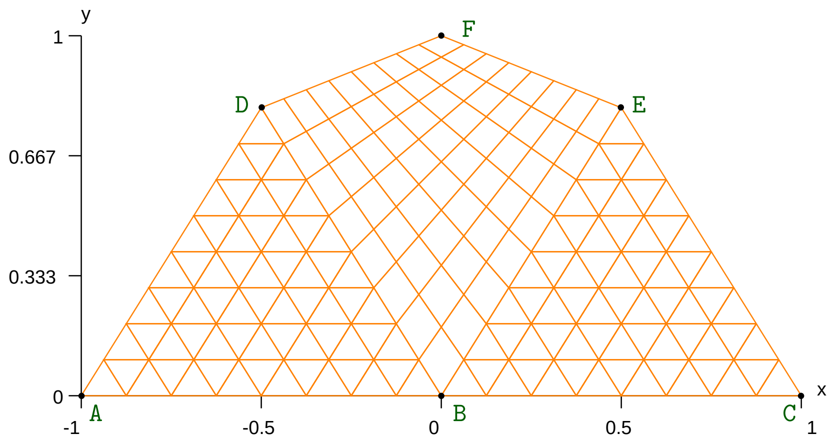


Figure 1.8.: mixed mesh

The resulting mesh is shown in figure 1.8.
Paragraphs 2.2 and 2.4 show other examples of mixed meshes.

1.7. Meshing a cube

To build a mesh of cubic cells, we first build eight vertices, then twelve segments, then six squares, then finally invoke the `Mesh` constructor with `tag::cube` (or, equivalently, `tag::hexahedron`)

parag-1.7.cpp
```
  Mesh cube ( tag::cube, ABCD, HGFE, BFGC, DHEA, CGHD, AEFB );
```

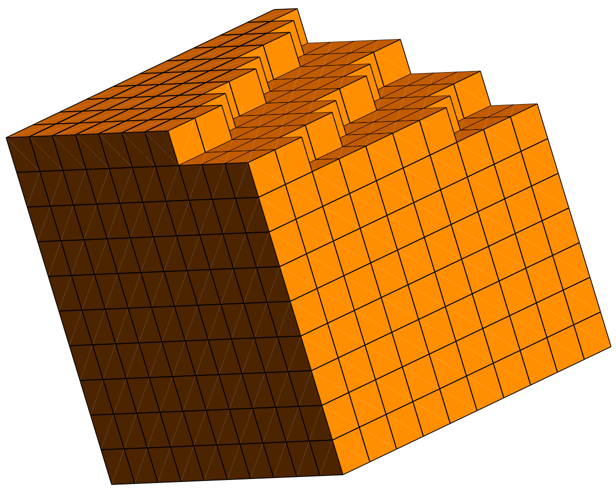


Figure 1.9.: mesh of cubes

Just like for two-dimensional rectangular meshes, the orientation of the resulting mesh is determined by the orientation of the six faces, as provided by the user. These orientations must be mutually compatible.

The order of the arguments of the above `Mesh` constructor is not rigid; the only constraint is that the first two meshes provided should be opposite faces of the cube, just like the third and the fourth and the last two.

In figure 1.9, some cells have been removed so that the reader can see the inside of the cube. This was done using the following options of `gmsh`: `Tools` → `Clipping` → `Mesh` → `Planes` → `Keep whole elements`, $A = 1, B = -0.2, C = -0.3, D = 0.1$.

Just like for two-dimensional rectangular meshes, vertices may be in any position in the space, the space may have more than three dimensions, edges and faces may be curved. Paragraphs 2.14 and 2.17 show a fancier use of the `Mesh` constructor with `tag::cube`.

1.8. Similar products (competitors)

deal.II : `https://dealii.org/`
FEniCS/Dolphin : `https://fenicsproject.org/`
FreeFem : `http://www3.freefem.org/`
GetFEM : `https://getfem.org/`
gmsh : `https://gmsh.info`
libMesh : `https://libmesh.github.io/`
MFEM : `https://opensourcelibs.com/lib/mfem`
OFELI : `https://ofeli.org/`
PETSc-FEM : `https://cimec.org.ar/foswiki/Main/Cimec/PETScFEM`
Pablo : `https://optimad.github.io/PABLO/` (dedicated to octree meshing)
PolyFEM : `https://github.com/polyfem/polyfem`
Rodin : `https://github.com/cbritopacheco/rodin`

1.9. Strong points

ManiFEM offers a flexible and general approach for defining meshes on manifolds (sections 2 and 3). Meshes on quotient manifolds (section 7) are handy for solving problems with periodic boundary conditions.

One of the goals of maniFEM is to make remeshing easy for end users. Thus, meshes can be manipulated (cells can be added and removed) using intuitive commands (section 10).

In maniFEM, the user defines finite elements using a clear and intuitive syntax (section 6). Some types of finite elements are rather slow; they are interesting for dydactic purposes. Fast versions are available; they use a slightly less elegant syntax.

The manual is clear and presents many examples.

1.10. Weak points

ManiFEM is not intended for parallel processing (see paragraph 13.2).

Some finite elements are slow (but fast versions are available).

Since maniFEM is under heavy development, some features are still missing : many types of finite elements, compact syntax for abstract variational formulations. See the changelog at the end of this manual (just before the index), especially the “To do” paragraph.

ManiFEM relies on oriented cells and meshes. Some people may find it confusing (the distinction between positive cells and negative cells).

Part I.

Basic usage

Sections 2 to 7 show how maniFEM can be used for building meshes and solving PDEs using high-level commands, in a style somewhat similar to `FreeFEM` or `fenics`. It can be used by people with no deep knowledge of `C++`.

2. Meshes and manifolds; patchwork

This section describes several examples of meshes, some of them built on specific manifolds. Note that in this manual we use the term “manifold” to mean a manifold without boundary.

This section focuses on building meshes by joining together several regular meshes (triangles or quadrangles), like patches. In contrast, section 3 shows how to mesh regions of a manifold by propagating a front and progressively adding cells, one by one.

Paragraphs 2.5 and 2.6 deal with one-dimensional meshes (curves) in $\mathbb{R}^2$, paragraphs 2.16, 2.19 and 2.20 show curves in $\mathbb{R}^3$, paragraphs 2.1, 2.2, 2.4, 2.10, 2.11 and 2.12 show plane domains (two-dimensional meshes in $\mathbb{R}^2$), while paragraphs 2.7, 2.8, 2.9, 2.15, 2.18, 2.21 and 2.23 focus on two-dimensional meshes in $\mathbb{R}^3$ (surfaces). Paragraphs 2.14, 2.17 and 2.22 show volumic meshes, made of cubic cells.

Paragraphs 2.5 – 2.18 are about manifolds defined implicitly as level sets; paragraphs 2.19 – 2.23 describe parametric manifolds.

2.1. Joining segments

Here is another way of meshing the same L-shaped domain as in paragraph 1.4:

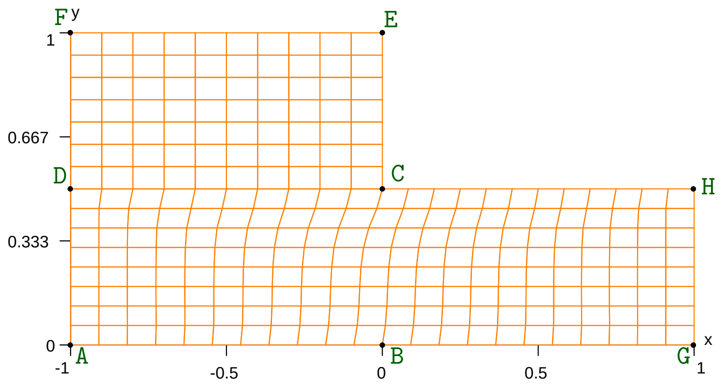


Figure 2.1.: an L-shaped mesh again

```
                                parag-2.1.cpp
    Mesh AG ( tag::segment, A .reverse(), G, tag::divided_in, 22 );
    Mesh GH ( tag::segment, G .reverse(), H, tag::divided_in,  8 );
    Mesh HC ( tag::segment, H .reverse(), C, tag::divided_in, 12 );
    Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, 10 );
    Mesh HD ( tag::join, HC, CD );
    Mesh DA ( tag::segment, D .reverse(), A, tag::divided_in,  8 );
    Mesh CE ( tag::segment, C .reverse(), E, tag::divided_in,  7 );
    Mesh EF ( tag::segment, E .reverse(), F, tag::divided_in, 10 );
    Mesh FD ( tag::segment, F .reverse(), D, tag::divided_in,  7 );
    Mesh AGHD ( tag::rectangle, AG, GH, HD, DA );
    Mesh CEFD ( tag::rectangle, CE, EF, FD, CD .reverse() );
    Mesh L_shaped ( tag::join, AGHD, CEFD );
```

The only difference between this mesh and the one presented in paragraph 1.4 is a slight distortion in the lower half of the domain, due to the non-uniform distribution of the vertices along `HD`.

Paragraph 11.2 explains the coloring conventions used in this manual for `C++` code. Paragraph 11.3 gives details about tags.

2.2. Exercise

Build the mesh shown in figure 2.2. We ask for one connected mesh, that is, the triangle and the rectangle must share some vertices (and segments, reversed).

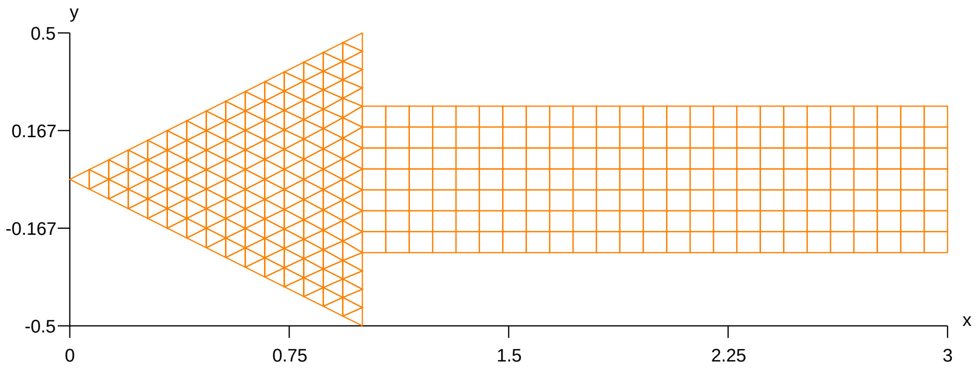


Figure 2.2.: an arrow

2.3. Joining boxes

In this paragraph we build a parallelipiped made of nine parts. Paragraph 1.7 describes the constructor of individual parallelipipeds. Here the upper part touches four other parts, so we need to join some segments and also some rectangles.

For the final mesh made of cubic cells, if we merely join these nine parallelipipeds, we get a mesh which is indistinguishable from one large parallelipiped. In order to keep track of the nine parts composing the large box and obtain the colorful drawing in figure 2.3, we use a more advanced feature : a `Mesh::Composite` object, described in section 4.

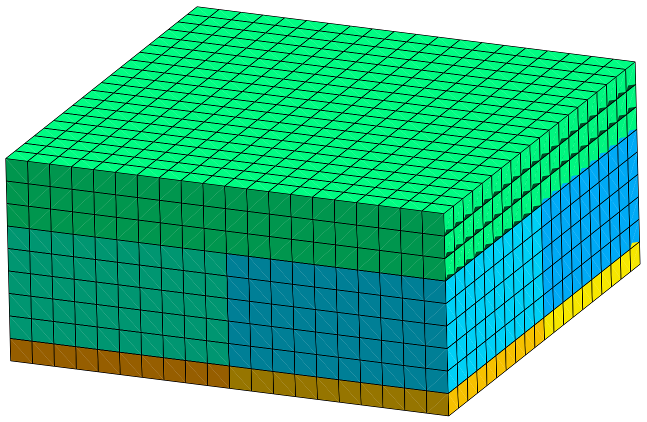


Figure 2.3.: nine boxes

2.4. Triangular meshes on rectangles

On a rectangular domain, we can build a mesh of triangles by using the `Mesh` constructor with `tag::rectangle`, providing as last argument the `tag::with_triangles`. For instance, in example 1.4, if we re-write the definition of `BGHC` as

```
Mesh BGHC ( tag::rectangle, BG, GH, HC, BC .reverse(), tag::with_triangles );
```

we get the mesh shown in figure 2.4.

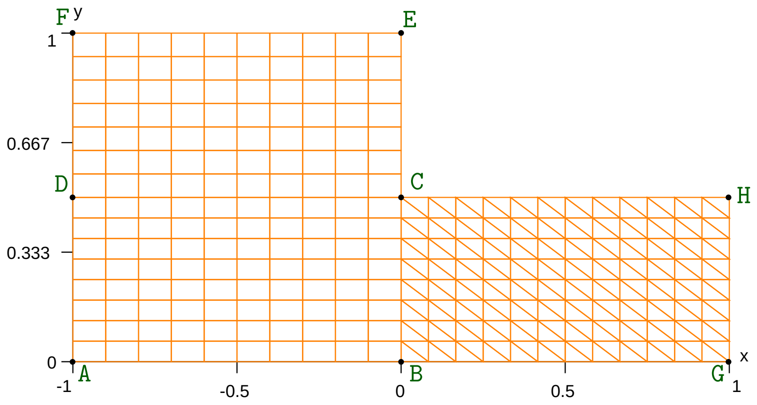


Figure 2.4.: an L-shaped mesh with triangular cells

If we give the sides of the rectangle in a different order, like in

```
Mesh BGHC ( tag::rectangle, GH, HC, BC .reverse(), BG, tag::with_triangles );
```

the rectangles will be cut along the other diagonal (check it yourself).

The mesh in paragraph 1.5 could have been built like this :

```
Mesh ABD ( tag::triangle, AB, BD, AD .reverse() );
Mesh BCED ( tag::quadrangle, CE, ED, BD .reverse(), BC, tag::with_triangles );
Mesh one_tri_one_rect ( tag::join, ABD, BCED );
```

2.5. A manifold defined as a level set in $\mathbb{R}^2$

maniFEM allows one to define manifolds and submanifolds, and this feature may be used to build domains of the desired shape.

Until now, we have only met the trivial Euclidian manifold, defined as `Manifold ( tag::Euclid, tag::of_dim, d )`. One can define a submanifold in terms of an implicit equation, that is, as a level set, using the method `implicit` of the Euclidian manifold. The code below introduces a one-dimensional submanifold of $\mathbb{R}^2$ (a hiperbola).

parag-2.5.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

Manifold hiperbola = RR2 .implicit ( x*y == 1. );

Cell A ( tag::vertex );  x(A) = 0.5;   y(A) = 2.;
Cell B ( tag::vertex );  x(B) = 3.;    y(B) = 0.333333333333;
```

```
    Mesh arc_of_hiperbola ( tag::segment, A .reverse(), B, tag::divided_in, 7 );

    arc_of_hiperbola .export_to_file ( tag::eps, "hiperbola.eps" );
    arc_of_hiperbola .export_to_file ( tag::gmsh, "hiperbola.msh" );
```

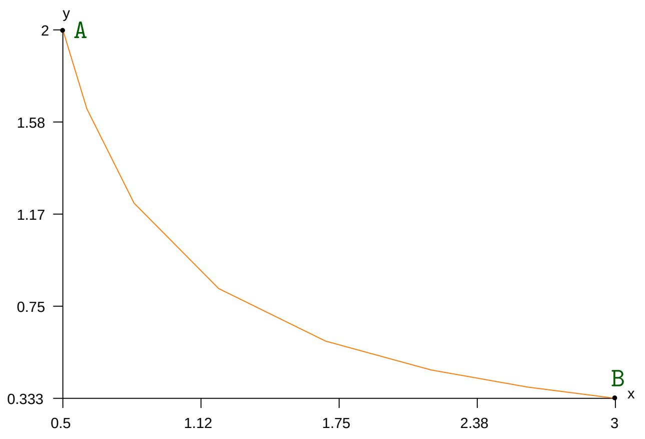


Figure 2.5.: an arc of hiperbola

In `gmsh`, you must select `Tools` → `Options` → `Mesh` → `1D elements` in order to see this mesh.

In maniFEM there is a global variable `Manifold::working`. Any `Manifold` constructor sets this variable to the manifold being defined. Subsequently, most `Mesh` constructors, when building inner vertices, define their geometric coordinates by interpolating coordinates of vertices on the boundary of the mesh and then projecting them on the current working manifold. The projection operation is a method of the `Manifold` itself. It does nothig for a Euclidian manifold. For an implicit manifold, it applies a few steps of a Newton-type algorithm; details are presented in paragraph 8.1.

Note that the vertices are not perfectly uniformly distributed along the curve because they are obtained as projections of points uniformly distributed along the straight segment `AB` onto the `hiperbola` manifold. Paragraph 3.5 shows another way of meshing a curve, producing equidistant vertices.

Note also that when defining individual points `A` and `B` we must be careful to set coordinates `x` and `y` within the `hiperbola` manifold. As an alternative, we might explicitly project them onto the `hiperbola` like this :

```
Cell P ( tag::vertex );  x(P) = 0.6;  y(P) = 2.1;
hiperbola .project (P);
```

There is also a `Cell` constructor with `tag::of_coords` and `tag::project`. This constructor first sets the values of `x` and `y` and then performs, under the curtains, a projection on the current working `Manifold` :

```
Cell P ( tag::vertex, tag::of_coords, { 0.6, 2.1 }, tag::project );
```

Also, the `Mesh` constructor with `tag::segment` builds points in the surrounding space `RR2` and then projects them onto the `hiperbola` without the user's assistance.

2.6. A circle defined by four curved segments

We can define several arcs of curve and `join` them, thus obtaining a closed curve :

parag-2.6.cpp
```
Manifold circle_manif = RR2 .implicit ( x*x + y*y == 1. );

Cell N ( tag::vertex );  x(N) =  0.;  y(N) =  1.;
Cell W ( tag::vertex );  x(W) = -1.;  y(W) =  0.;
Cell S ( tag::vertex );  x(S) =  0.;  y(S) = -1.;
Cell E ( tag::vertex );  x(E) =  1.;  y(E) =  0.;

Mesh NW ( tag::segment, N .reverse(), W, tag::divided_in, 5 );
Mesh WS ( tag::segment, W .reverse(), S, tag::divided_in, 5 );
Mesh SE ( tag::segment, S .reverse(), E, tag::divided_in, 5 );
Mesh EN ( tag::segment, E .reverse(), N, tag::divided_in, 5 );

Mesh circle ( tag::join, NW, WS, SE, EN );
```

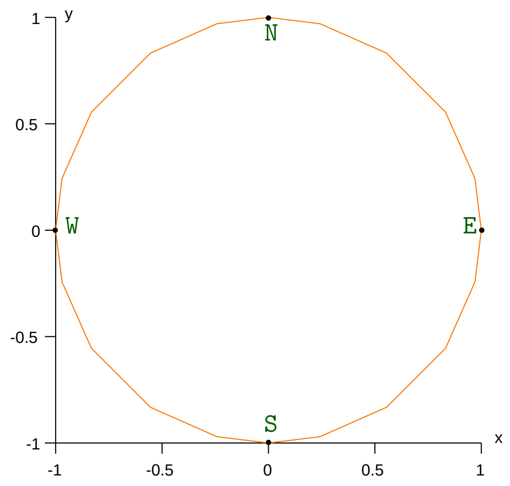


Figure 2.6.: a circle obtained by `join`ing four segments

Again, the distribution of vertices along the circle is not perfect because vertices are obtained as projections (on the circle) of points along straight segments `NW`, `WS` and so forth. Paragraph 3.2 shows another way of meshing a closed curve, producing equidistant vertices.

Note that applying the `Mesh` constructor with `tag::join` to four segments is very different from applying the `Mesh` constructor with `tag::quadrangle` to the same four segments; see paragraphs 2.10 and 2.12.

2.7. A hemisphere defined by four curved triangles

In this paragraph, we mesh a surface in $\mathbb{R}^3$:

parag-2.7.cpp
```
Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xyz[0], y = xyz[1], z = xyz[2];

Manifold sphere = RR3 .implicit ( x*x + y*y + z*z == 1. );
```

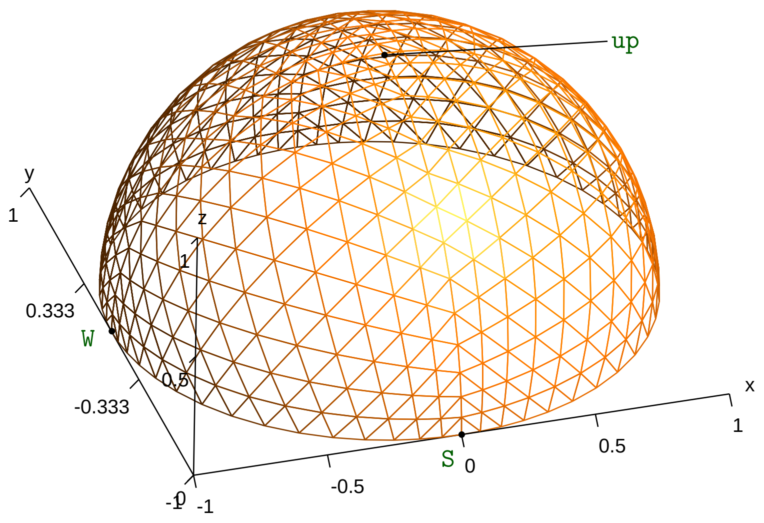


Figure 2.7.: a hemisphere obtained by `joining` four triangles

```
// let's mesh half of a sphere
Cell E  ( tag::vertex );  x(E) =  1.;  y(E) =  0.;  z(E) = 0.;
Cell N  ( tag::vertex );  x(N) =  0.;  y(N) =  1.;  z(N) = 0.;
Cell W  ( tag::vertex );  x(W) = -1.;  y(W) =  0.;  z(W) = 0.;
Cell S  ( tag::vertex );  x(W) =  0.;  y(W) = -1.;  z(W) = 0.;
Cell up ( tag::vertex );  x(up)=  0.;  y(up)=  0.;  z(up)= 1.;

int n = 15;
Mesh EN ( tag::segment, E .reverse(), N, tag::divided_in, n );
Mesh NW ( tag::segment, N .reverse(), W, tag::divided_in, n );
Mesh WS ( tag::segment, W .reverse(), S, tag::divided_in, n );
Mesh SE ( tag::segment, S .reverse(), E, tag::divided_in, n );
Mesh upE ( tag::segment, up .reverse(), E, tag::divided_in, n );
Mesh upN ( tag::segment, up .reverse(), N, tag::divided_in, n );
Mesh upW ( tag::segment, up .reverse(), W, tag::divided_in, n );
Mesh upS ( tag::segment, up .reverse(), S, tag::divided_in, n );

// build four triangles
Mesh ENup ( tag::triangle, EN, upN .reverse(), upE );
Mesh NWup ( tag::triangle, NW, upW .reverse(), upN );
Mesh WSup ( tag::triangle, WS, upS .reverse(), upW );
Mesh SEup ( tag::triangle, SE, upE .reverse(), upS );

// and finally join the triangles :
Mesh hemisphere ( tag::join, ENup, NWup, WSup, SEup );
```

Again, when we define individual points `E`, `N`, `W`, `S` and `up` we must be careful to provide coordinates on the `sphere` (or `project` them explicitly as shown in paragraph 2.8). In contrast, the `Mesh` constructors with `tag::segment`, `tag::quadrangle` and `tag::triangle` build inner points in the surrounding space ($\mathbb{R}^2$ or $\mathbb{R}^3$) and then project them onto the current manifold without the user's assistance (paragraph 8.1 describes this projection operation). As a side effect, within each triangle (`ENup`, `NWup` and so forth), the distribution of the vertices is not perfectly uniform.

Note that, when we build the segments `WS`, `upS` and so on, we know that those segments will be (polygonal approximations of) arcs of circle on the sphere. This is so due to the particular geometry of our manifold (we know that the projection of a straight line segment on the sphere is an arc of circle of radius equal to the radius of the sphere); the shape of such segments is less clear in other examples (like the one in paragraph 2.8).

Section 3 shows other ways of meshing a surface.

2.8. A more complex surface

If the surface is more "bumpy", we must use smaller patches in order to get a mesh of good quality. Below we use twelve rectangles to get a bumpy hemisphere, shown in figure 2.8.

parag-2.8.cpp

```
Manifold nut = RR3 .implicit ( x*x + y*y + z*z + 1.5*x*y*z == 1. );

// let's mesh a hemisphere (much deformed)
Cell S ( tag::vertex );   x(S)  =  0.;  y(S)  = -1.;  z(S)  =  0.;
Cell E ( tag::vertex );   x(E)  =  1.;  y(E)  =  0.;  z(E)  =  0.;
Cell N ( tag::vertex );   x(N)  =  0.;  y(N)  =  1.;  z(N)  =  0.;
Cell W ( tag::vertex );   x(W)  = -1.;  y(W)  =  0.;  z(W)  =  0.;
Cell up ( tag::vertex );  x(up) =  0.;  y(up) =  0.;  z(up) =  1.;
// no need to project these

Cell mSW ( tag::vertex );  x (mSW) = -1.;   y (mSW) = -1.;   z (mSW) = 0.;
nut .project ( mSW );  // midway between S and W
Cell mSup  ( tag::vertex );  x (mSup) =  0.;   y (mSup) = -1.;   z (mSup) = 1.;
nut .project ( mSup );  // midway between S and up
Cell mSWup ( tag::vertex );  x (mSWup) = -1.;  y (mSWup) = -1.;  z (mSWup) = 1.;
nut .project ( mSWup );  // somewhere between S, W and up
// ... and so forth ...
// now build segments :
int n = 10;
Mesh W_mSW  ( tag::segment, W .reverse(), mSW,  tag::divided_in, n );
Mesh W_mNW  ( tag::segment, W .reverse(), mNW,  tag::divided_in, n );
Mesh W_mWup ( tag::segment, W .reverse(), mWup, tag::divided_in, n );
Mesh S_mSW  ( tag::segment, S .reverse(), mSW,  tag::divided_in, n );
// ... and so forth ...

// now the twelve rectangles :
Mesh rect_W_SW  ( tag::quadrangle,
   mSW_mSWup, mWup_mSWup .reverse(), W_mWup .reverse(), W_mSW );
Mesh rect_S_SW  ( tag::quadrangle,
   mSup_mSWup, mSW_mSWup .reverse(), S_mSW .reverse(), S_mSup );
Mesh rect_up_SW ( tag::quadrangle,
   mWup_mSWup, mSup_mSWup .reverse(), up_mSup .reverse(), up_mWup );
// ... and so forth ...

// and finally join the rectangles :
Mesh hemisphere ( tag::join,
   { rect_E_NE, rect_E_SE, rect_S_SE, rect_S_SW, rect_W_SW, rect_W_NW,
     rect_N_NE, rect_N_NW, rect_up_SE, rect_up_SW, rect_up_NE, rect_up_NW } );
```

Note how we use a version of the `Mesh` constructor with `tag::join` taking as argument a vector of `Mesh`es; the same constructor is used in paragraph 9.2.

Unlike in paragraph 2.7, here we do not control the exact shape of the segments `S_mSW`, `S_mSup` and so on. They are projections of straight line segments onto our surface but since the equation of the surface is rather complicated we do not know the exact shape of these projections. Since points like `mSW` and `mSE` have been placed initially in `RR3` not belonging to the

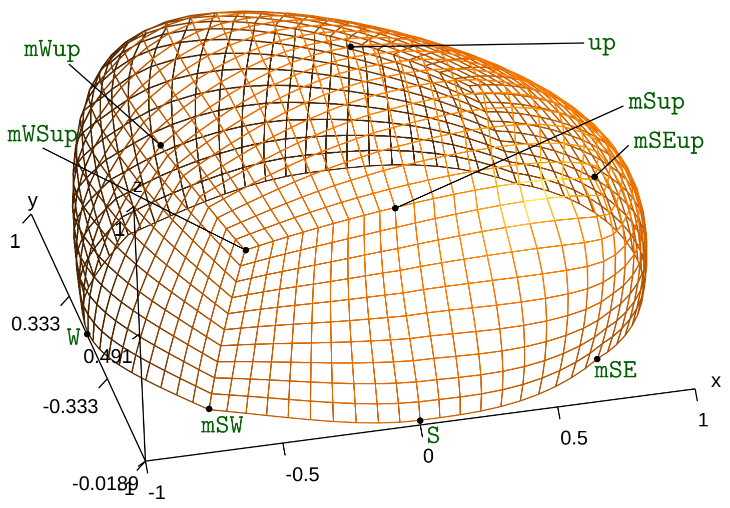


Figure 2.8.: here we have `joined` twelve rectangles

`bupmy` manifold and then explicitly `projected`, there is no guarantee that they lie in the plane $z = 0$ (they probably don't). We notice an angle between `W_mSW` and `S_mSW` at `mSW`.

Paragraph 2.18 shows a way to control the shape of the segments `S_mSW`, `S_mSE` and so on.

Section 3 shows other ways of meshing a surface.

2.9. Exercise

Slightly change the code in paragraph 2.8 in order to obtain the mesh shown in figure 2.9. (Hint: have a look at paragraph 2.4.)

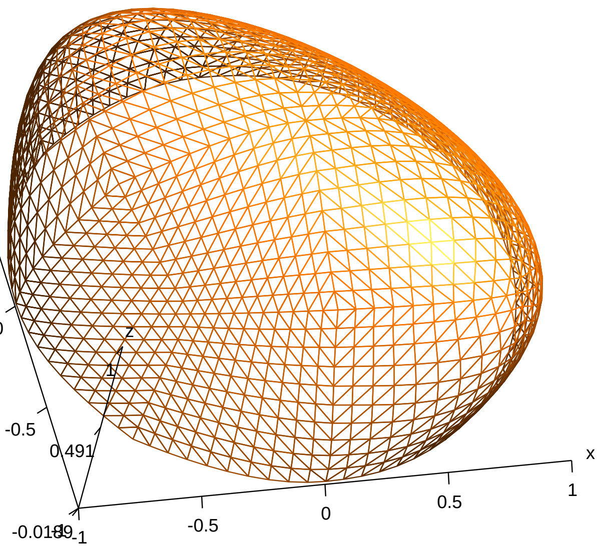


Figure 2.9.: using triangular cells

2.10. Alternating between manifolds

Let's go back to the example in paragraph 2.6. Suppose we want to mesh the whole disk, not just its boundary. We can build the boundary of the disk just like in paragraph 2.6, by placing ourselves in the manifold `circle`. But if we want to subsequently mesh the interior of the disk, we must leave `circle` and switch back to the original `RR2` manifold. Method `set_as_working_manifold` allows us to do that.

parag-2.10.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

Manifold circle = RR2 .implicit ( x*x + y*y == 1. );

Cell N ( tag::vertex );  x(N) =  0.;  y(N) =  1.;
Cell W ( tag::vertex );  x(W) = -1.;  y(W) =  0.;
Cell S ( tag::vertex );  x(S) =  0.;  y(S) = -1.;
Cell E ( tag::vertex );  x(E) =  1.;  y(E) =  0.;

Mesh NW ( tag::segment, N .reverse(), W, tag::divided_in, 10 );
Mesh WS ( tag::segment, W .reverse(), S, tag::divided_in, 10 );
Mesh SE ( tag::segment, S .reverse(), E, tag::divided_in, 10 );
Mesh EN ( tag::segment, E .reverse(), N, tag::divided_in, 10 );

RR2 .set_as_working_manifold();
Mesh disk ( tag::quadrangle, NW, WS, SE, EN );
```

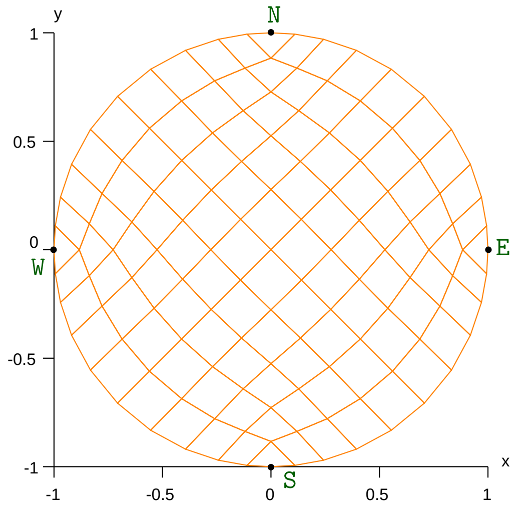


Figure 2.10.: a disk treated as a rectangle

The mesh (shown in figure 2.10) is of poor quality; we obtain quadrilaterals having a wide angle near `S`, `N`, `E` and `W`. Paragraph 2.11 shows a better way of meshing a disk. Paragraphs 3.1 and 3.2 show how to mesh a disk through a frontal mesh generation algorithm, starting from its boundary.

Each time a `Manifold` object is created, its constructor sets it as working manifold; this is why in many cases we don't need to know about method `set_as_working_manifold`. We need it, however, in cases like the one presented here.

2.11. Exercise

Mesh half of a disk by joining three triangular meshes, like in figure 2.11. Mesh the entire disk by joining six triangular meshes.

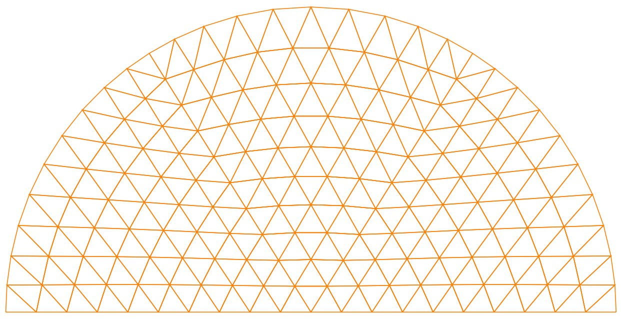


Figure 2.11.: half disk, made of three triangles

2.12. Alternating between manifolds, again

Here is an example similar to the one in paragraph 2.10, this time with four arcs of hiperbola.

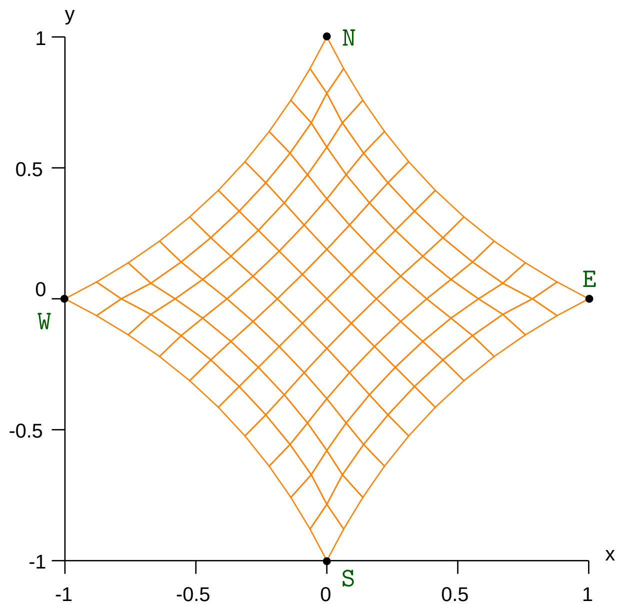


Figure 2.12.: a diamond shape

parag-2.12.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

Cell N ( tag::vertex );  x(N) =  0.;  y(N) =  1.;
Cell W ( tag::vertex );  x(W) = -1.;  y(W) =  0.;
Cell S ( tag::vertex );  x(S) =  0.;  y(S) = -1.;
Cell E ( tag::vertex );  x(E) =  1.;  y(E) =  0.;
```

```
Manifold first_arc  = RR2 .implicit ( x*y + x - y == -1. );
Mesh NW ( tag::segment, N .reverse(), W, tag::divided_in, 10 );
Manifold second_arc = RR2 .implicit ( x*y - x - y ==  1. );
Mesh WS ( tag::segment, W .reverse(), S, tag::divided_in, 10 );
Manifold third_arc  = RR2 .implicit ( x*y - x + y == -1. );
Mesh SE ( tag::segment, S .reverse(), E, tag::divided_in, 10 );
Manifold fourth_arc = RR2 .implicit ( x*y + x + y ==  1. );
Mesh EN ( tag::segment, E .reverse(), N, tag::divided_in, 10 );

RR2 .set_as_working_manifold();
Mesh diamond ( tag::quadrangle, NW, WS, SE, EN );
```

Paragraph 3.16 shows another way of meshing the same domain.

2.13. Zero-dimensional manifolds

This may sound strange, but we can define zero-dimensional manifolds and such manifolds may even be useful.

In this paragraph, we define two parabolas, one having vertical axis and the other one with horizontal axis.

parag-2.13.cpp
```
Manifold parab_vert  = RR2 .implicit ( y == x * x - 2. );
Manifold parab_horiz = RR2 .implicit ( x == y * y - 2. );
```

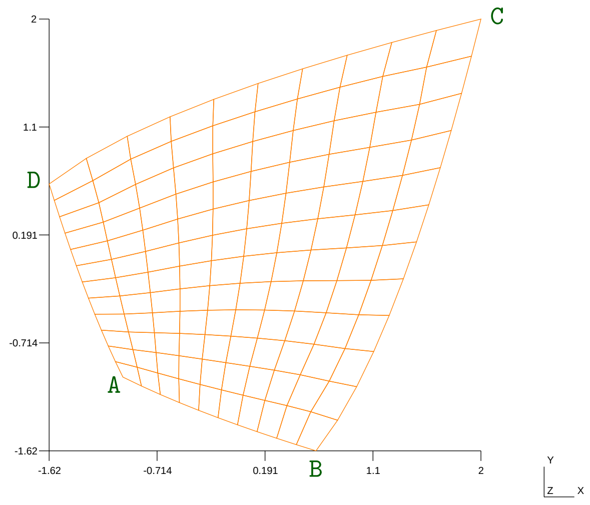


Figure 2.13.: a curved quadrilateral

We want to mesh the region contained between these two parabolas (a curved quadrilateral). Just like in paragraphs 2.10 and 2.12, we must start with the four corners. We could certainly compute their coordinates by hand, but we can also ask maniFEM to do that. We define a zero-dimensional manifold which in this case consists in four points (the intersection between the two parabolas) :

parag-2.13.cpp
```
Manifold four_points ( tag::intersect, parab_vert, parab_horiz );
```

Then we build points and project them explicitly onto this manifold. The projection will (usually) find the closest point to the given one, so we only need to define initial coordinates in the desired zone of the plane. Of course, it is better to give initial positions close to the desired point; this way the projection algorithm will find more easily its target.

```
Cell A ( tag::vertex );  x(A) = -1.;  y(A) = -1.;
four_points .project (A);
```

Equivalently, we can use a constructor with `tag::of_coords` and `tag::project`. This constructor first sets the values of `x` and `y` and then performs, under the curtains, a projection on the current working `Manifold`:

parag-2.13.cpp

```
Cell B ( tag::vertex, tag::of_coords, { 1., -1. }, tag::project );
```

2.14. A three-dimensional star-like shape

Here is a three-dimensional version of the diamond in paragraph 2.12.

In figure 2.14, some layers have been removed so that the reader can see inner cells. This was done using the following options of `gmsh`: `Tools` → `Clipping` → `Mesh` → `Planes` → `Keep whole elements`, $A = 1, B = 0.2, C = 0.3, D = 0.6$.

Comments in paragraph 1.7 apply.

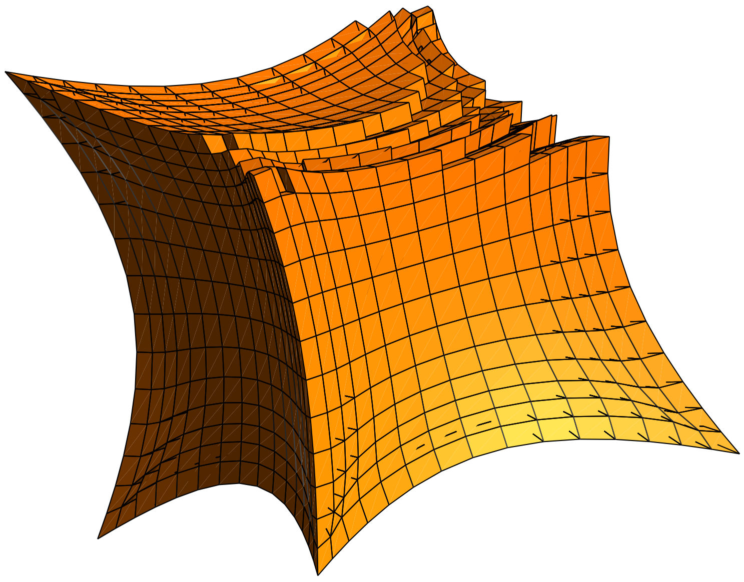


Figure 2.14.: a 3D diamond

For very curved meshes, the approach described in paragraph 2.22 may be better.

2.15. An organic shape

This paragraph describes a surface meant to mimic the shape of a physalis fruit.

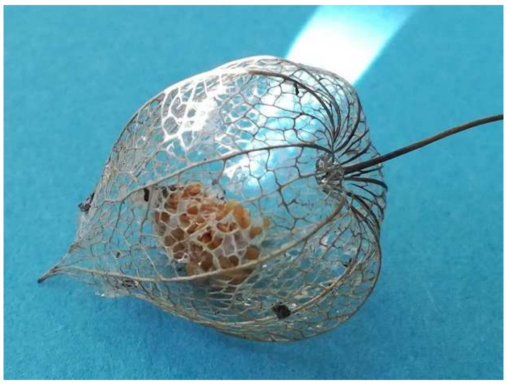


Figure 2.15.: dry physalis, with fruit inside

We begin by defining a revolution surface in $\mathbb{R}^3$:

```
parag-2.15.cpp
Function r2 = x*x + y*y + z*z;
const double pi = 3.1415926536;
Manifold apple = RR3 .implicit ( power(r2,0.5) * sin(r2-pi/6.) == z );
```

This surface has the shape shown in figure 2.16, which does not resemble a physalis fruit.

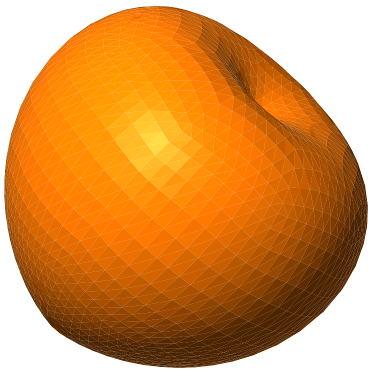


Figure 2.16.: a compact manifold resembling an apple

Instead of meshing the `apple` surface, we just build eight curves immersed in it (each curve joins `A` to `D` and is made of three shorter curves) :

```
parag-2.15.cpp
Cell A ( tag::vertex );  x(A) = 0.;  y(A) = 0.;  z(A) = std::sqrt ( 2.*pi/3. );
Cell B1 ( tag::vertex );  x(B1) = 1.;  y(B1) = 0.;  z(B1) = 1.;
Cell C1 ( tag::vertex );  x(C1) = 1.;  y(C1) = 0.;  z(C1) = 0.;
apple .project (B1);  apple .project (C1);
Cell D ( tag::vertex );  x(D) = 0.;  y(D) = 0.;  z(D) = 0.;

Mesh AB1 ( tag::segment, A .reverse(), B1, tag::divided_in, 10 );
Mesh B1C1 ( tag::segment, B1 .reverse(), C1, tag::divided_in, 10 );
Mesh C1D ( tag::segment, C1 .reverse(), D, tag::divided_in, 10 );
// and so on ...
```

Then we switch back to RR3 (thus leaving the apple manifold) and build transversal segments, as well as triangular and quadrangular patches :

parag-2.15.cpp

```
RR3 .set_as_working_manifold();
Mesh B1B2 ( tag::segment, B1 .reverse(), B2, tag::divided_in, 10 );
Mesh B2B3 ( tag::segment, B2 .reverse(), B3, tag::divided_in, 10 );
// and many other segments ...

Mesh AB1B2 ( tag::triangle, AB1, B1B2, AB2 .reverse() );
Mesh AB2B3 ( tag::triangle, AB2, B2B3, AB3 .reverse() );
// and other triangular patches ...

Mesh B1C1C2B2 ( tag::quadrangle, B1C1, C1C2, B2C2 .reverse(), B1B2 .reverse(),
                tag::with_triangles                                            );
Mesh B2C2C3B3 ( tag::quadrangle, B2C2, C2C3, B3C3 .reverse(), B2B3 .reverse(),
                tag::with_triangles                                            );
// and other quadrangular patches ...
```

We then join all patches :

parag-2.15.cpp

```
Mesh sect1 ( tag::join, AB1B2, B1C1C2B2, C1DC2 );
Mesh sect2 ( tag::join, AB2B3, B2C2C3B3, C2DC3 );
// more sectors ...
std::vector < Mesh > lm { sect1, sect2, sect3, sect4, sect5, sect6, sect7, sect8 };
Mesh fisalis ( tag::join, lm );
```

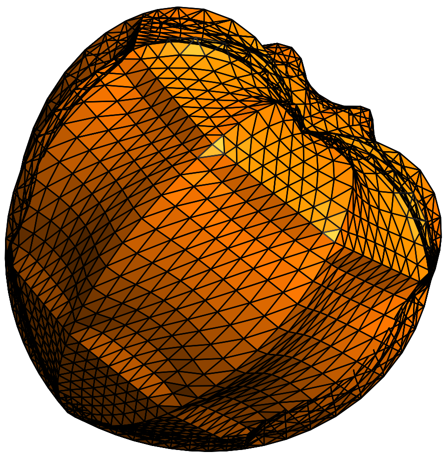


Figure 2.17.: not yet a physalis

This shape (shown in figure 2.17) is still not satisfactory, so we apply a deformation in $\mathbb{R}^3$ in order to get a sharper tip. The apple manifold is no longer relevant.

parag-2.15.cpp

```
Mesh::Iterator it = fisalis .iterator ( tag::over_vertices );
for ( it .reset(); it .in_range(); it++ )
{  Cell P = *it;
   x(P) *= 0.8;  y(P) *= 0.8;
   if ( z(P) > 1.3 )
   {  x(P) /= 1. + 300. * std::pow ( z(P) - 1.3, 3. );
      y(P) /= 1. + 300. * std::pow ( z(P) - 1.3, 3. );
      z(P) *= 1. + 10. * ( z(P) - 1.3 ) * ( z(P) - 1.3 );  }
   if ( z(P) > 0. ) z(P) *= 0.8;                             }
```

Paragraph 9.3 explains the notion of an iterator over cells.

We add a sequence of barycenter operations for smoothening the shape. Note that each barycenter operation is relative to a sector and applies only to inner vertices, so the boundary of the sector is not changed. Thus, the eight rims defined in the beginning are kept unchanged.

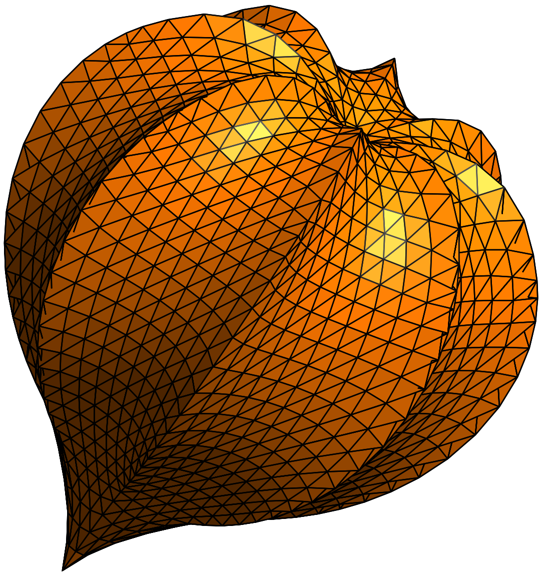


Figure 2.18.: meshing a chinese lantern

parag-2.15.cpp
```
std::vector < Mesh > ::iterator it1;
for ( it1 = lm .begin(); it1 != lm .end(); it1++ )
{  Mesh sect = *it1;
   Mesh::Iterator it2 = sect .iterator ( tag::over_vertices );
   for ( int i = 1; i < 20; i++ )
   for ( it2 .reset(); it2 .in_range(); it2++ )
   {  Cell ver = *it2;
      if ( ver .is_inner_to ( sect ) )  sect .barycenter ( ver );  }  }
```

And we are happy with the final result, shown in figure 2.18.

An animation related to this example is available at
`https://maniFEM.rd.ciencias.ulisboa.pt/apple-to-physalis.gif`

2.16. A manifold defined by two equations

We can define a one-dimensional submanifold of $\mathbb{R}^3$ by two implicit equations:

parag-2.16.cpp
```
Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xyz[0], y = xyz[1], z = xyz[2];
```

```
Manifold circle_manifold = RR3 .implicit ( x*x + y*y == 1., x*y == 4.*z );

Cell S ( tag::vertex );  x(S) =  0.;  y(S) = -1.;  z(S) = 0.;
Cell E ( tag::vertex );  x(E) =  1.;  y(E) =  0.;  z(E) = 0.;
Cell N ( tag::vertex );  x(N) =  0.;  y(N) =  1.;  z(N) = 0.;
Cell W ( tag::vertex );  x(W) = -1.;  y(W) =  0.;  z(W) = 0.;
// these four points already belong to 'circle_manifold', no projection needed

Mesh SE ( tag::segment, S .reverse(), E, tag::divided_in, 5 );
Mesh EN ( tag::segment, E .reverse(), N, tag::divided_in, 5 );
Mesh NW ( tag::segment, N .reverse(), W, tag::divided_in, 5 );
Mesh WS ( tag::segment, W .reverse(), S, tag::divided_in, 5 );

Mesh circle ( tag::join, SE, EN, NW, WS );
```

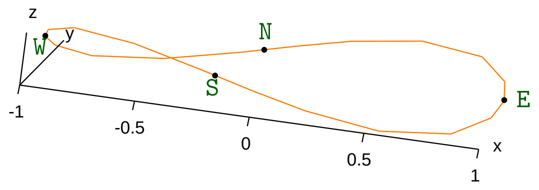


Figure 2.19.: one-dimensional mesh in $\mathbb{R}^3$

Paragraph 3.4 shows another way of meshing the same loop.

2.17. Exercise

In paragraph 2.14 we have built twelve segments embedded in two-dimensional manifolds. This means that, a priori, we do not control precisely the shape of these segments, they are projections of straight segments onto the respective surface. Change the code by defining precisely the shape of these segments (defining one-dimensional manifolds which the segments must follow), while obtaining exactly the same 3d star-like shape.

2.18. A submanifold of a submanifold

An implicit manifold has submanifolds. For instance, we can improve the look of the "bumpy hemisphere" in paragraph 2.8 by building its base (a circle-like closed curve) inside a one-dimensional manifold. For the rest of the surface, we switch back to the two-dimensional manifold `nut` :

parag-2.18.cpp

```
Manifold nut = RR3 .implicit ( x*x + y*y + z*z + 1.5*x*y*z == 1. );
int n = 10;

// first build the base (a closed curve)
Manifold base = nut .implicit ( x*x + 3.*z == 0. );

// define some points :
Cell S ( tag::vertex );  x(S) =  0.;  y(S) = -1.;  z(S) =  0.;
Cell E ( tag::vertex );  x(E) =  1.;  y(E) =  0.;  z(E) =  0.;
Cell N ( tag::vertex );  x(N) =  0.;  y(N) =  1.;  z(N) =  0.;
Cell W ( tag::vertex );  x(W) = -1.;  y(W) =  0.;  z(W) =  0.;
```

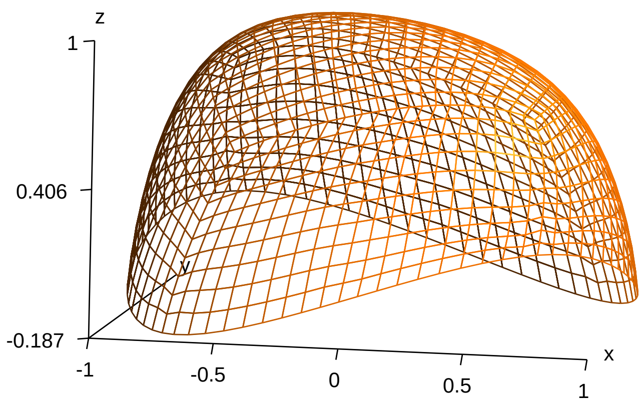


Figure 2.20.: a shell

```
  // no need to project S and N, they are already on 'base'
  base .project (E);  base .project (W);
  Cell mSW ( tag::vertex );  x (mSW) = -1.;  y (mSW) = -1.;  z (mSW) = 0.;
  base .project ( mSW );  // midway between S and W
  // define similarly mSE, mNE and mNW

  // now build eight segments, forming the base
  Mesh S_mSW ( tag::segment, S .reverse(), mSW, tag::divided_in, n );
  Mesh S_mSE ( tag::segment, S .reverse(), mSE, tag::divided_in, n );
  Mesh E_mSE ( tag::segment, E .reverse(), mSE, tag::divided_in, n );
  Mesh E_mNE ( tag::segment, E .reverse(), mNE, tag::divided_in, n );
  Mesh N_mNE ( tag::segment, N .reverse(), mNE, tag::divided_in, n );
  Mesh N_mNW ( tag::segment, N .reverse(), mNW, tag::divided_in, n );
  Mesh W_mSW ( tag::segment, W .reverse(), mSW, tag::divided_in, n );
  Mesh W_mNW ( tag::segment, W .reverse(), mNW, tag::divided_in, n );

  // we are done with the base, now switch back to 'nut'
  nut .set_as_working_manifold();

  // more points :
  Cell up ( tag::vertex );  x (up) = 0.;  y (up) = 0.;  z (up) = 1.;
  // no need to project 'up', it is already on 'nut'
  Cell mSup  ( tag::vertex );  x (mSup) = 0.;  y (mSup) = -1.;  z (mSup) = 1.;
  nut .project ( mSup );  // midway between S and up
  Cell mSWup ( tag::vertex );  x (mSWup) = -1.;  y (mSWup) = -1.;  z (mSWup) = 1.;
  nut .project ( mSWup );  // somewhere between S, W and up
  // ... and so forth ...

  // more segments :
  Mesh W_mWup ( tag::segment, W .reverse(), mWup, tag::divided_in, n );
  Mesh mSW_mSWup ( tag::segment, mSW .reverse(), mSWup, tag::divided_in, n );
  Mesh mWup_mSWup ( tag::segment, mWup .reverse(), mSWup, tag::divided_in, n );
  // ... and so forth ...
```

If we wanted a flat base, we could have defined

```
Manifold base = nut .implicit ( z == 0. );
```

Paragraph 3.9 shows a way to mesh the same surface using fewer lines of code.

2.19. Parametric manifolds – a curve

Paragraphs 2.5 – 2.18 describe manifolds defined through implicit equations, that is, level sets in $\mathbb{R}^2$ or $\mathbb{R}^3$. Another way of defining a submanifold is through a parametrization. Below is an example.

parag-2.19.cpp
```
// at the beginning, we define 'spiral' as a straight line
Manifold spiral ( tag::Euclid, tag::of_dim, 1 );
Function t = spiral .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );

// now build 'arc_of_spiral' merely as a segment from pi/2 to 5 pi
const double pi = 3.1415926536;
Cell A ( tag::vertex );  t (A) = pi/2.;
Cell B ( tag::vertex );  t (B) = 5.*pi;

Mesh arc_of_spiral ( tag::segment, A .reverse(), B, tag::divided_in, 30 );
// not very interesting for now

// but now define functions x and y as expressions of t :
Function x = t * cos(t), y = t * sin(t);
// and declare them to be the new coordinates
Manifold RR2 ( tag::Euclid, tag::of_dimension, 2 );
RR2 .set_coordinates ( x && y );

// in future statements (e.g. for graphical representation)
// x and y will be used, not t :
arc_of_spiral .export_to_file ( tag::eps, "spiral.eps" );
```

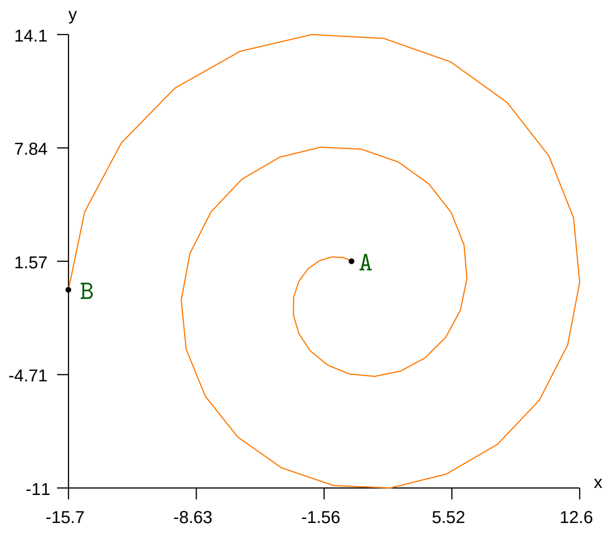


Figure 2.21.: one-dimensional mesh in $\mathbb{R}^2$

The operator `&&` joins two functions into one vector-valued function.

Note that, when defining points `A` and `B`, we only set the value of `t`. Functions `x` and `y` are defined later, as arithmetic expressions in terms of `t`; their values will be computed "on-the-fly" when needed. In the drawing above, we note that the generated points are not equidistant in the sense of the Euclidian distance in $\mathbb{R}^2$. They correspond to values of `t` which are uniformly distributed between $\pi/2$ (at `A`) and 5π (at `B`).

The approach described above has the disadvantage that, if we want to subsequently change the distribution of nodes along the `arc_of_spiral`, we must switch back to the original `t` coordinate.

Paragraph 3.5 shows another way of meshing a curve, producing equidistant vertices.

2.20. Closing a circle

In the approach of paragraph 2.19, it is possible but cumbersome to build a closed curve:

parag-2.20.cpp

```
Manifold circle_manif ( tag::Euclid, tag::of_dim, 1 );
Function t = circle_manif .build_coordinate_system
             ( tag::Lagrange, tag::of_degree, 1 );

// build 'circle' merely as a segment from 0 to 1.9 pi
const double pi = 3.1415926536;
Cell A ( tag::vertex );  t(A) =  0.;
Cell B ( tag::vertex );  t(B) =  1.9*pi;
Mesh incomplete_circle ( tag::segment, A .reverse(), B, tag::divided_in, 19 );

// now close the curve in a not very elegant manner
Mesh small_piece ( tag::segment, B .reverse(), A, tag::divided_in, 1 );
Mesh circle ( tag::join, incomplete_circle, small_piece );

// define new coordinates x and y as expressions of t :
Function x = cos(t), y = sin(t);
Manifold RR2 ( tag::Euclid, tag::of_dimension, 2 );
RR2 .set_coordinates ( x && y );

// in future statements (e.g. for graphical representation)
// x and y will be used, not t :
circle .export_to_file ( tag::eps, "circle.eps" );
```

Rather than declaring `small_piece` as a `Mesh` (made of one cell only), we may treat it as a `Cell`:

```
Mesh circle ( tag::segment, A .reverse(), B, tag::divided_in, 19 );
Cell small_piece ( tag::segment, B .reverse(), A );
small_piece .add_to_mesh ( circle );
```

Paragraph 7.2 shows a more elegant way to close a curve in itself.

On the other hand, if we only want a visual illusion of a closed circle, we may use the code below.

```
Manifold circle_manif ( tag::Euclid, tag::of_dim, 1 );
Function t = circle_manif .build_coordinate_system
   ( tag::Lagrange, tag::of_degree, 1 );

// build 'circle' merely as a segment from 0 to 2 pi
const double pi = 3.1415926536;
Cell A ( tag::vertex );  t (A) = 0.;
Cell B ( tag::vertex );  t (B) = 2.*pi;
Mesh circle ( tag::segment, A .reverse(), B, tag::divided_in, 20 );
// gives the illusion of a closed circle
// define new coordinates x and y as expressions of t :
Function x = cos(t), y = sin(t);
Manifold RR2 ( tag::Euclid, tag::of_dimension, 2 );
RR2 .set_coordinates ( x && y );
```

2.21. Parametric manifolds – a surface

Here is an example of a parametrized surface:

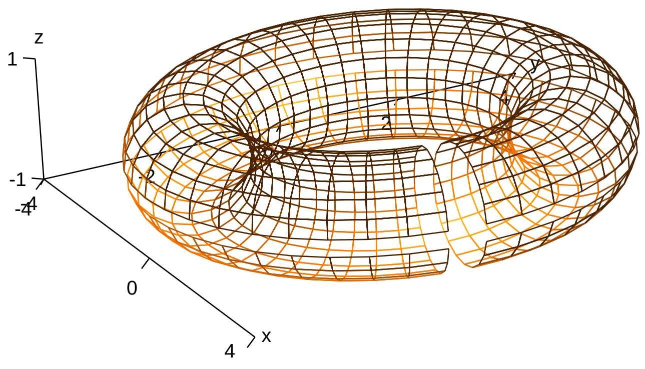


Figure 2.22.: incomplete torus

parag-2.21.cpp

```
Manifold torus ( tag::Euclid, tag::of_dim, 2 );
Function alpha_beta =
   torus .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
// extract components of alpha_beta :
Function alpha = alpha_beta [0], beta = alpha_beta [1];
// build a rectangle in the alpha-beta plane
const double pi = 3.1415926536;
Cell A ( tag::vertex );  alpha (A) = 0.;       beta (A) = 0.;
Cell B ( tag::vertex );  alpha (B) = 0.;       beta (B) = 1.9*pi;
Cell C ( tag::vertex );  alpha (C) = 1.95*pi;  beta (C) = 1.9*pi;
Cell D ( tag::vertex );  alpha (D) = 1.95*pi;  beta (D) = 0.;

// four almost-closed circles :
Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 19 );
Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in, 39 );
Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, 19 );
Mesh DA ( tag::segment, D .reverse(), A, tag::divided_in, 39 );

Mesh ABCD ( tag::rectangle, AB, BC, CD, DA );  // an almost-closed torus

// parametrize the torus
const double big_radius = 3., small_radius = 1.;
Function x = ( big_radius + small_radius * cos(beta) ) * cos(alpha),
         y = ( big_radius + small_radius * cos(beta) ) * sin(alpha),
         z = small_radius * sin(beta);

Manifold RR3 ( tag::Euclid, tag::of_dimension, 3 );
RR3 .set_coordinates ( x && y && z );  // forget about alpha and beta

// in future statements (e.g. for graphical representation)
// x, y and z will be used, not alpha nor beta :
ABCD .export_to_file ( tag::gmsh, "torus.msh" );
```

Closing the torus in the cumbersome manner shown in paragraph 2.20 is possible but not practical. Paragraph 7.5 shows a more elegant solution.

If we only want a visual illusion of a closed surface, we may change the numerical values as shown in paragraph 4.4.

Another way of building a torus is to define it as a submanifold of $\mathbb{R}^3$ through an implicit equation, then apply frontal mesh generation, in the spirit of paragraps 3.7, 3.24 and 4.5.

2.22. A cylindrical shell

The approach described in paragraph 2.21 can be used to build extruded shapes:

parag-2.22.cpp

```
Manifold RR3_param ( tag::Euclid, tag::of_dim, 3 );
Function zrt =
   RR3_param .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function z = zrt [0], r = zrt [1], theta = zrt [2];

Cell A ( tag::vertex );  r(A) = 1. ;  theta(A) = 0.;  z(A) = -1.;
Cell B ( tag::vertex );  r(B) = 1.2;  theta(B) = 0.;  z(B) = -1.;
// ... six more vertices ...

Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 5 );
Mesh GH ( tag::segment, G .reverse(), H, tag::divided_in, 5 );
// ... more segments ...

Mesh AEFB ( tag::rectangle, AE, EF, BF .reverse(), AB .reverse() );
Mesh CGHD ( tag::rectangle, CG, GH, DH .reverse(), CD .reverse() );
Mesh ABCD ( tag::rectangle, AB, BC, CD, DA );
Mesh EFGH ( tag::rectangle, EF, FG, GH, HE );
Mesh DHEA ( tag::rectangle, DH, HE, AE .reverse(), DA .reverse() );
Mesh BFGC ( tag::rectangle, BF, FG, CG .reverse(), BC .reverse() );

Mesh shell ( tag::cube, ABCD, EFGH .reverse(), BFGC, DHEA, CGHD, AEFB );

Function x = r * cos(theta), y = r * sin(theta);
Manifold RR3 ( tag::Euclid, tag::of_dimension, 3 );
RR3 .set_coordinates ( x && y && z );  // forget about r and theta

shell .export_to_file ( tag::gmsh, "shell.msh" );
```

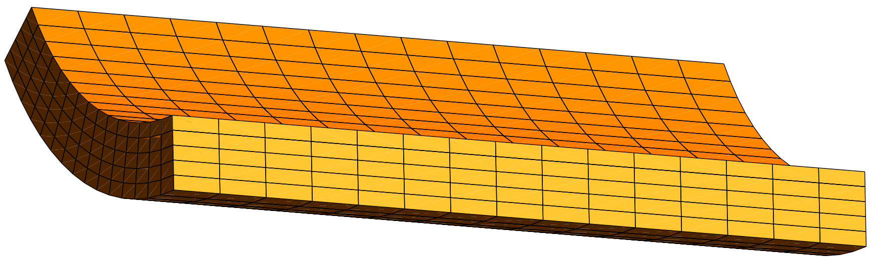


Figure 2.23.: cylindrical shell

2.23. Starting with a high-dimensional manifold

Instead of starting with a manifold having only the parameter(s), we may start with a high-dimensional manifold containing both the geometric coordinates and the parameter(s), then define the parametrization through equation(s). There is a disadvantage however, regarding performance: maniFEM will reserve, for each vertex, space in the computer's memory for five `double` values.

parag-2.23.cpp
```
Manifold RR5 ( tag::Euclid, tag::of_dim, 5 );
Function xyzab = RR5 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
// extract components of xyzab :
Function x = xyzab [0], y = xyzab [1], z = xyzab [2],
         alpha = xyzab [3], beta = xyzab [4];

// define a torus as a submanifold of RR5 :
const double big_radius = 3, small_radius = 1;
Manifold torus = RR5 .parametric
   ( x == ( big_radius + small_radius * cos(beta) ) * cos(alpha),
     y == ( big_radius + small_radius * cos(beta) ) * sin(alpha),
     z == small_radius * sin(beta)                               );

// define four corners :
const double pi = 3.1415926536;
Cell A ( tag::vertex );  alpha (A) = 0.;       beta (A) = 0.;
Cell B ( tag::vertex );  alpha (B) = 0.;       beta (B) = 1.9*pi;
Cell C ( tag::vertex );  alpha (C) = 1.95*pi;  beta (C) = 1.9*pi;
Cell D ( tag::vertex );  alpha (D) = 1.95*pi;  beta (D) = 0.;
torus .project (A);   torus .project (B);  // the projection operation computes
torus .project (C);   torus .project (D);  // coordinates x, y and z of each vertex

// four almost-closed circles :
Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 19 );
Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in, 39 );
Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, 19 );
Mesh DA ( tag::segment, D .reverse(), A, tag::divided_in, 39 );

// build a rectangle
Mesh ABCD ( tag::rectangle, AB, BC, CD, DA );  // an almost-closed torus
// internally, each newly built vertex is projected on the current working manifold
// thus, the coordinates x, y and z of that vertex are updated

ABCD .export_to_file ( tag::gmsh, "torus.msh" );
```

The `Manifold::parametric` method is similar to `Manifold::implicit`, presented in paragraphs 2.5 – 2.18. The only difference is that by using `parametric` we declare an explicit dependence[1] of the coordinates (here, `x`, `y` and `z`) upon the parameters (here, `alpha` and `beta`). This endows the manifold `torus` with a different projection operator. The projection of a vertex from `RR5` onto `torus` is done by merely updating the values of `x`, `y` and `z`, while keeping `alpha` and `beta` constant.

`Mesh` constructors with `tag::segment`, `tag::triangle` and `tag::rectangle`, when building each new vertex, interpolate the coordinates `alpha` and `beta` using several vertices from the boundary, then project the vertex, thus updating the values of `x`, `y` and `z`. On the other hand, method `Mesh::export_to_file` uses only coordinates `x`, `y` and `z`.

[1] Perhaps `explicit` would be a better name than `parametric`; unfortunately, that word is reserved in C++.

3. Frontal mesh generation; knitting

Section 2 shows how to build meshes by joining together several regular meshes (triangles or quadrangles), like patches. The present section explains how to build a mesh by gradually adding cells, one by one. We call this approach "frontal mesh generation" because, for building a 2D mesh, we gradually propagate a front of segments; for building a 3D mesh we gradually advance a front of triangles. In some cases, the process starts with a given interface and adds cells, one by one, moving and deforming the interface, until it shrinks and disappears. In other cases (like in paragraphs 3.2, 3.4, 3.6, 3.7) we begin with nothing at all; maniFEM finds a starting point by itself.

The frontal mesh generation algorithm behaves differently according to the topological dimension of the current working manifold (the most recently declared `Manifold` object). If the current working manifold is one-dimensional (which could be $\mathbb{R}$ or a curve in $\mathbb{R}^2$ or in $\mathbb{R}^3$), the frontal mesh generation algorithm builds segments, as shown in paragraphs 3.2 – 3.5, 3.9, 3.17 – 3.20, 3.23 – 3.30. If the current working manifold is two-dimensional (which could be $\mathbb{R}^2$ or a surface in $\mathbb{R}^3$), the frontal mesh generation algorithm builds triangles, as shown in paragraphs 3.1 – 3.3, 3.6 – 3.9, 3.16 – 3.20, 3.23 – 3.30. If the current working manifold is $\mathbb{R}^3$, the frontal mesh generation algorithm builds tetrahedra, as shown in paragraphs 3.8, 3.25.

The algorithm follows the shape of the current working manifold; when necessary, it `projects` each newly constructed vertex on that manifold. Recall that in this manual we use the term "manifold" to mean a manifold without boundary.

Paragraphs 3.23 – 3.26 show examples of non-uniform meshes; paragraphs 3.27 - 3.30 show examples of anisotropic meshes.

3.1. Filling a disk

Paragraph 2.10 shows how to build a mesh over a disk, but the quality of the mesh is quite poor. This is so because the `Mesh` constructor with `tag::quadrangle` treats the disk as a (much) deformed rectangle.

We can ask maniFEM to mesh the disk, starting from its boundary (a circle) and adding triangles, one by one, until the disk is completely covered :

parag-3.1.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

Manifold circle_manif = RR2 .implicit ( x*x + y*y == 1. );

Cell N ( tag::vertex );  x(N) =  0.;   y(N) =  1.;
Cell W ( tag::vertex );  x(W) = -1.;   y(W) =  0.;
Cell S ( tag::vertex );  x(S) =  0.;   y(S) = -1.;
Cell E ( tag::vertex );  x(E) =  1.;   y(E) =  0.;

Mesh NW ( tag::segment, N .reverse(), W, tag::divided_in, 10 );
Mesh WS ( tag::segment, W .reverse(), S, tag::divided_in, 10 );
```

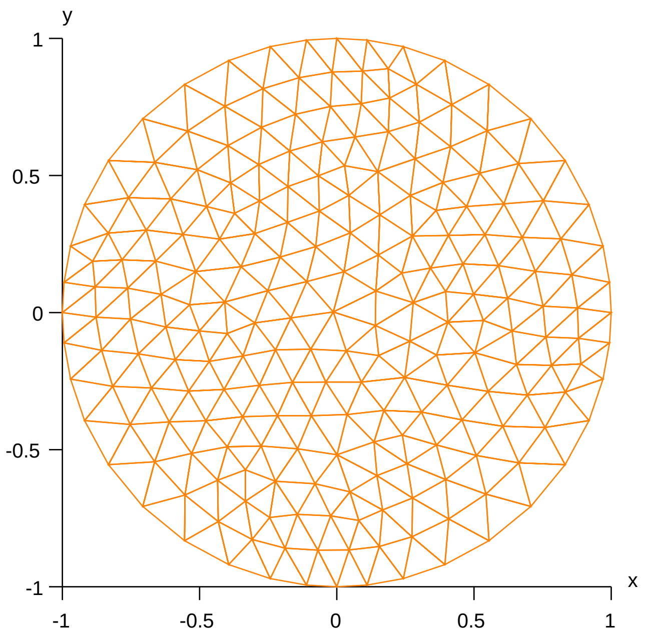


Figure 3.1.: circle built from patches, interior of disk meshed with frontal method

```
   Mesh SE ( tag::segment, S .reverse(), E, tag::divided_in, 10 );
   Mesh EN ( tag::segment, E .reverse(), N, tag::divided_in, 10 );
   Mesh circle ( tag::join, NW, WS, SE, EN );

   RR2 .set_as_working_manifold();
   Mesh disk ( tag::frontal, tag::boundary, circle, tag::desired_length, 0.157 );
```

We provide the desired length of the segments of the future mesh as an argument to the constructor. Of course the length of the segments inside the mesh will vary slightly. We must take care to give as boundary a curve with segments of length approximatively equal to the desired length (paragraph 3.15 discusses this requirement).

Paragraph 11.2 explains the coloring conventions observed in this manual for C++ code. Paragraph 11.3 gives details about tags.

3.2. Meshing a circle

Instead of building the circle by joining four (curved) segments, we can mesh directly the circle manifold, then mesh the disk :

parag-3.2.cpp

```
   Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
   Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
   Function x = xy[0], y = xy[1];

   Manifold circle_manif = RR2 .implicit ( x*x + y*y == 1. );
   Mesh circle ( tag::frontal,
                 tag::entire_manifold, circle_manif, tag::desired_length, 0.2 );
   // we can omit the manifold; maniFEM will take the current working manifold :
   // Mesh circle ( tag::frontal, tag::desired_length, 0.2 );

   RR2 .set_as_working_manifold();
```

```
  Mesh disk ( tag::frontal, tag::boundary, circle, tag::desired_length, 0.2 );
  disk .export_to_file ( tag::eps, "disk.eps" );
```

The code above is quite comfortable for the user; he or she only needs to define the manifold(s) to be meshed and provide the desired (average) length of segments in the future mesh. However, this comfort comes at the price of a significant computational effort. For building the `circle`, maniFEM must first find a starting point for the process of frontal mesh generation. In extreme cases, the algorithm may fail to find a starting point on the given manifold.

The user may choose to be more specific in order to save computation time, by providing a starting point :

```
Cell A ( tag::vertex );  x(A) = 1.;  y(A) = 0.;
Mesh circle ( tag::frontal, tag::start_at, A, tag::desired_length, 0.2 );
```

Also, maniFEM must infer the right orientation of the `circle` as explained in paragraphs 3.10 and 3.12. Choosing the other orientation would result in an endless process of meshing the exterior of the disk. The process of choosing the right orientation implies some computational effort. The user can make things easier for maniFEM either by attaching an orientation to the manifold `circle_manif` (this feature is not implemented yet) or by providing the initial direction as shown in paragraph 3.11.

3.3. Inner boundaries

Inner boundaries must have the reverse orientation :

parag-3.3.cpp

```
  Manifold circle = RR2 .implicit ( x*x + y*y == 1. );
  Mesh outer ( tag::frontal, tag::desired_length, 0.1 );
  Manifold ellipse = RR2 .implicit ( x*x + (y-0.37)*(y-0.37) + 0.3*x*y == 0.25 );
  Mesh inner ( tag::frontal, tag::desired_length, 0.1 );
  Mesh bdry ( tag::join, outer, inner .reverse() );
  RR2 .set_as_working_manifold();
  Mesh disk ( tag::frontal, tag::boundary, bdry, tag::desired_length, 0.1 );
```

Paragraphs 1.2 and 1.4 explain the relation between the orientation of a mesh and the orientation of its boundary.

Paragraphs 3.10 and 3.12 explain how maniFEM chooses the orientation of closed curves.

3.4. Meshing a three-dimensional loop

We may apply the same `frontal` algorithm for meshing the circle in $\mathbb{R}^3$ introduced in paragraph 2.16 :

parag-3.4.cpp

```
  Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
  Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
  Function x = xyz[0], y = xyz[1], z = xyz[2];
  Manifold circle_manif = RR3 .implicit ( x*x + y*y == 1., x*y == 4.*z );

  Mesh circle ( tag::frontal, tag::desired_length, 0.1,
                tag::orientation, tag::random          );
```

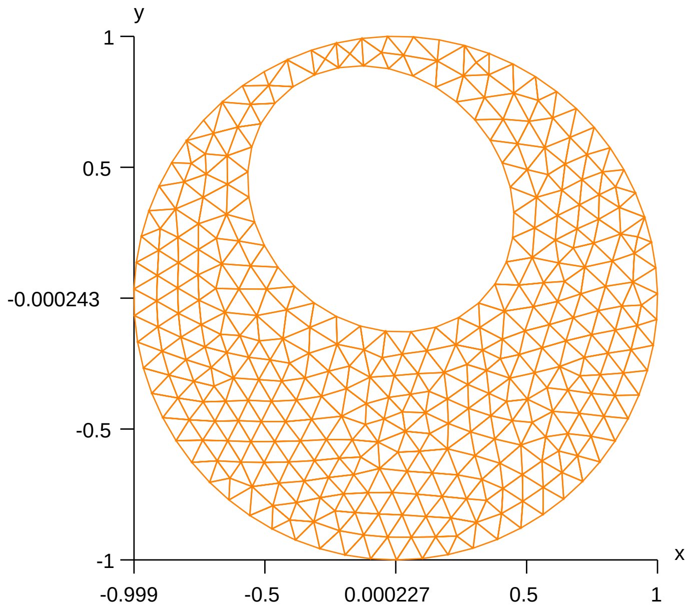


Figure 3.2.: disk with a hole

Unlike for the `circle` in paragraph 3.2, there is no way to choose between the two possible orientations of this `circle`. No one is more "correct" than the other. This is why maniFEM requires supplementary arguments `tag::orientation, tag::random` for the `Mesh` constructor. In other cases (e.g. for closed curves in $\mathbb{R}^2$) these supplementary arguments are not mandatory; however, even then the user may choose to provide them, thus sparing the computer from the burden of finding the right orientation. Paragraphs 3.10 and 3.12 give more details.

The term "random" should not be interpreted in the probabilistic sense. The orientation of such a mesh is not a random variable, not even a pseudo-random one. Is it just difficult to predict.

On the other hand, if the orientation matters for you, you can either attach an orientation to the manifold `circle_manif` (this feature is not implemented yet) or provide a starting point and an initial direction as shown in paragraph 3.11.

3.5. Starting and stopping points

In paragraphs 3.2 and 3.4 we have meshed the entire closed curve `circle`. If we only want a piece of a curve, we must specify two points, one for starting and the other one for stopping.

Looking at the example in paragraph 2.19, let us define a spiral with a slightly different look and switch to frontal mesh generation.

parag-3.5.cpp
```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];
Function r = power ( x*x + y*y, 0.25 );
const double pi = 3.14159;

RR2 .implicit ( x*sin(r) == y*cos(r) );
// we don't need to give a name to the implicit manifold
// the Manifold constructor sets the manifold it builds as working manifold
// after that, many methods use this working manifold by default
```

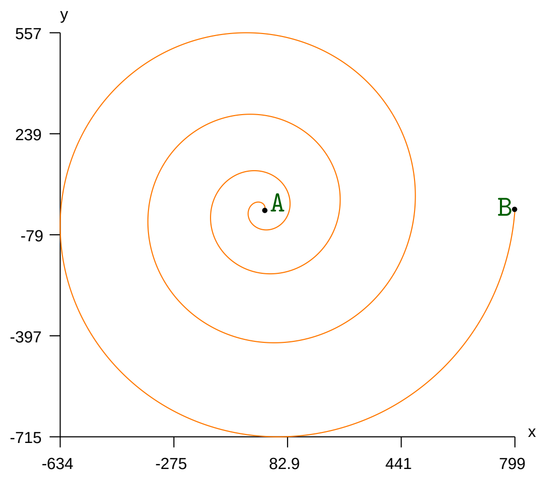


Figure 3.3.: spiral, meshed by frontal method

```
Cell A ( tag::vertex );  x(A) =     pi*pi;   y(A) = 0.;
Cell B ( tag::vertex );  x(B) = 81.*pi*pi;   y(B) = 0.;
Mesh spiral ( tag::frontal, tag::start_at, A, tag::stop_at, B,
              tag::desired_length, 1., tag::shortest_path     );
```

Figure 3.3 shows the resulting mesh. In `gmsh`, you must select `Tools` → `Options` → `Mesh` → `1D elements` in order to see this mesh.

Unlike in paragraph 2.19, we define the spiral through an implicit equation. Had we chosen the (natural) definition `r = power ( x*x + y*y, 0.5 )`, the very same spiral as in paragraph 2.19 would have been obtained. For aesthetic reasons, we have chosen a different definition of `r`, thus obtaining a different spacing between the arcs of the spiral.

Another noteworthy difference from paragraph 2.19 is that vertices are distributed uniformly along the spiral (with respect to the distance in the surrounding space $\mathbb{R}^2$). The segments are too small to be seen in the figure.

Note that there are two ways to go along the spiral starting from `A`. One of them will eventually stumble upon the stopping point `B` while the other one will never meet `B`. ManiFEM has no means to guess which way it should start building the mesh; the `tag::shortest_path` instructs it to perform a preliminary search in both directions. The one that first meets `B` wins and the mesh is subsequently built along the winning direction.

Thus, if we want to mesh an arc of a circle, we can use a sequence of statements like

```
RR2 .implicit ( (x-x0)*(x-x0) + (y-y0)*(y-y0) == radius * radius );
Cell A ( tag::vertex );  // set x(A) and y(A)
Cell B ( tag::vertex );  // set x(B) and y(B)
Mesh arc_of_circle ( tag::frontal, tag::start_at, A, tag::stop_at, B,
                     tag::desired_length, some_value, tag::shortest_path );
```

However, if we choose `A` and `B` diametrally opposed, maniFEM will mesh unpredictably one half of the circle or the other.

Paragraph 3.11 shows how we can specify the direction we want to follow starting from a given point, thus avoiding ambiguities and also saving computational time.

Of course we must be careful to choose starting and stopping points belonging to the manifold (or to `project` them explicitly) otherwise the meshing algorithm will either fail to start or will spin for ever on the circle or spiral, hopelessly searching for `B`.

3.6. Meshing a compact surface

Recall that in this manual we use the term “manifold” to mean a manifold without boundary. So, the term “compact surface” should be understood as “compact surface without boundary” like the sphere or the torus.

Just like for the `circle` in paragraphs 3.2 and 3.4, if we want to mesh a compact surface entirely the mesh will have no boundary so we only need to provide the desired (average) length of segments. Code below builds a mesh on the sphere.

```
parag-3.6.cpp
  Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
  Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
  Function x = xyz[0], y = xyz[1], z = xyz[2];
  Manifold sphere_manif = RR3.implicit ( x*x + y*y + z*z == 1. );
  Mesh sphere ( tag::frontal, tag::desired_length, 0.1 );
```

Again, a significant computational effort is made for finding a starting point for the meshing process. We can alleviate this burden simply by providing a starting point on the sphere :

```
Cell A ( tag::vertex );  x(A) = 1.;  y(A) = 0.;  z(A) = 0.;
Mesh sphere ( tag::frontal, tag::start_at, A, tag::desired_length, 0.1 );
```

Also, some computational effort is made for choosing the right orientation of the sphere; paragraphs 3.10 and 3.12 give more details.

3.7. A more complicated surface

Here is an example of a compact surface given by a more complicated implicit equation. It can be vaguely described as a convolution between two tori.

Function `d1`, defined in the code below, represents the square of the distance to a circle. The implicit equation `d1 == 0.15` defines a torus (it shows up in paragraphs 3.24 and 4.5). Function `d2` represents the squared distance to another circle, orthogonal to the former.

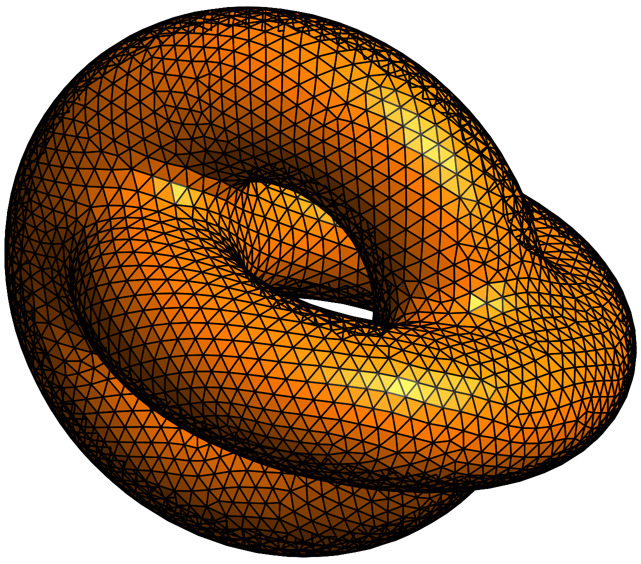


Figure 3.4.: convolution between two tori

parag-3.7.cpp

```
Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xyz[0], y = xyz[1], z = xyz[2];

Function f1 = max ( x*x + y*y, 0.1 );  // we cut by 0.1 to avoid singularities
Function f2 = 1. - power ( f1, -0.5 );
Function d1 = z*z + f1 * f2 * f2;  // squared distance to a circle in the xy plane
Function f3 = max ( (x-0.4)*(x-0.4) + z*z, 0.1 );
Function f4 = 1. - power ( f3, -0.5 );
Function d2 = y*y + f3 * f4 * f4;  // squared distance to a circle in the xz plane

Manifold tori_manif = RR3 .implicit
   ( smooth_min ( d1, d2, tag::threshold, 0.2 ) == 0.15 );

Mesh two_tori ( tag::frontal, tag::desired_length, 0.09 );
```

Function `smooth_min` gives a smooth approximation of the minimum between two or more `Function` objects. It is equal to the true minimum if the difference between the two values is higher, in absolute value, than the provided threshold. When the difference is smaller than the threshold, it gives a C^1 interpolation between the two values.

Comments at the end of paragraph 3.6 apply here, too : maniFEM must find a starting point and the right orientation.

If we want sharp edges, we must build the edges first as explained in paragraphs 3.18, 3.19 and 3.24.

3.8. Filling a volume with tetrahedra

If the current working manifold has dimension three, the frontal mesh generation algorithm builds tetrahedra.

In order to fill, for instance, the sphere bulit in paragraph 3.6 with tetrahedra, it suffices to add two lines of code :

parag-3.8.cpp

```
RR3 .set_as_working_manifold();
Mesh ball ( tag::frontal, tag::boundary, sphere, tag::desired_length, 0.11 );
```

In figure 3.5 below we only show a slice of the `ball`, in order to see the interior.

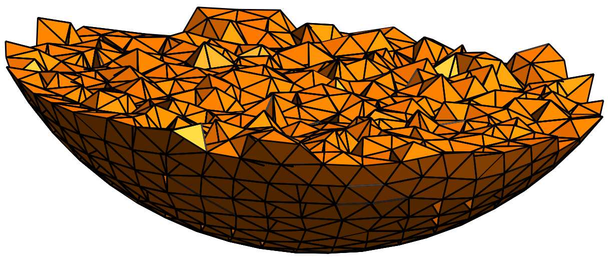


Figure 3.5.: slice of a ball made of tetrahedra

The same statements above can be used to fill any volume defined by a compact surface with no boundary, like `two_tori` in paragraph 3.7 or 3.24, or `perf_sphere` in paragraph 3.19. In paragraph 3.18, we must close the mesh `sphere_and_cylinder` by adding the `disk` before calling frontal mesh generation with tetrahedra.

3.9. A bumpy hemisphere

We now look again at the surface in paragraph 2.18 and build it with frontal meshing, using fewer lines of code.

parag-3.9.cpp
```
Manifold nut = RR3 .implicit ( x*x + y*y + z*z + 1.5*x*y*z == 1. );
Manifold base = nut .implicit ( x*x + 3.*z == 0. );
Mesh circle  // 'base' is used by default as working manifold
   ( tag::frontal, tag::desired_length, 0.1, tag::orientation, tag::random );

nut .set_as_working_manifold();
Mesh bumpy  // 'nut' is used as working manifold
   ( tag::frontal, tag::boundary, circle, tag::desired_length, 0.1,
     tag::orientation, tag::random                                 );
```

In this example, the issue of the orientation can be really tricky. An orientation of the `circle` is chosen at random. Also, maniFEM will make a random choice of starting to grow the mesh on one or the other side of `circle`, so we will get unpredictably the upper or the lower hemisphere. Paragraphs 3.10, 3.12 and 3.13 give more details.

3.10. How the orientation is chosen

In maniFEM, cells and meshes are oriented (paragraphs 1.2 and 9.5 provide details). For building a mesh with frontal method, maniFEM needs to know which orientation we want.

Manifolds may be oriented or not. Euclidian manifolds (that is, the space $\mathbb{R}^n$) have an intrinsic orientation, given by the natural order of the n coordinates. Parametric manifolds inherit the intrinsic orientation from the space of parameters. Implicit manifolds, however, have no intrinsic orientation. We can specify an orientation of a manifold by attaching an outer form to it (this feature is not implemented yet); otherwise, it will be considered not oriented.

In the example in paragraph 3.1, the boundary `circle` is oriented according to the four segments composing it, `NW`, `WS`, `SE` and `EN`. ManiFEM uses the intrinsic orientation of the surrounding space `RR2` in order to determine the desired orientation of the mesh `disk`. The orientation of the boundary `circle` is important; together with the intrinsic orientation of the surrounding space, it determines on which side of the circle the mesh will start to grow. Had we built the segments `NW`, `WS`, `SE` and `EN` in the opposite direction (that is, `WN`, `SW`, `ES` and `NE`) the program would try to mesh the exterior of the disk, never stopping.

We said before that implicit manifolds have no intrinsic orientation. Compact manifolds of co-dimension one (closed curves in $\mathbb{R}^2$, compact surfaces in $\mathbb{R}^3$) are a special case.[1] There is a "privileged" orientation of a compact manifold of co-dimension one, which is compatible with the intrinsic orientation of the surrounding space. We call this orientation `inherent`; paragraph 3.12 discusses this topic.

Open curves (like the `spiral` in paragraphs 3.5 and 3.11), as well as non-compact manifolds (like the `cylinder` in paragraphs 3.18 and 3.19), have no privileged orientation. These manifolds are considered not oriented, unless we attach to them an outer form (this feature is not implemented yet). Arguments `tag::orientation, tag::random` instruct maniFEM to choose an arbitrary orientation. For curves, we can provide `tag::shortest_path` and maniFEM will perform a preliminary search in both directions then choose the one that first meets the stopping point.

[1] Recall that in this manual we use the term "manifold" to mean a manifold without boundary.

Curves in $\mathbb{R}^3$, either closed like the `circle` in paragraphs 3.4 and 3.9 or open, have no privileged orientation either (because they have co-dimension two). There is no way to decide which of its two possible orientations should be chosen; no one is more "correct" than the other. This is why maniFEM will reject a tentative to define `circle` without `tag::orientation`, `tag::random` or `tag::shortest_path`. A statement like `Mesh circle ( tag::frontal, tag::desired_length, some_value )` will produce a run-time error if the current working manifold has dimension 1 and the geometric dimension (that is, the number of coordinates) is 3.

The options given to the constructor of `Mesh`es `circle` and `bumpy` in paragraph 3.9 are discussed in paragraph 3.13.

3.11. Specifying the direction

For one-dimensional manifolds, we can specify, along with the starting point, a starting direction, thus saving some computational time. The code producing the spiral in paragraph 3.5 can be changed as below.

```
Cell A ( tag::vertex );  x(A) =    pi*pi;  y(A) = 0.;
Cell B ( tag::vertex );  x(B) = 81*pi*pi;  y(B) = 0.;
std::vector < double > direc = { 0., 1. };
Mesh spiral ( tag::frontal, tag::start_at, A, tag::towards, direc,
              tag::stop_at, B, tag::desired_length, 1.           );
```

Note that the vector `direc` is associated to the point `A` (is tangent to the manifold at point `A`).

Recall that open curves have no "default" (or "inherent") orientation. ManiFEM uses the vector `direc` as an indication about where to go.

It is the user's responsibility to ensure that, going in the specified direction, the algorithm will eventually meet the stopping point `B`. Otherwise, we may end up spinning endlessly along the spiral.

We could specify in a similar fashion the orientation of `circle` in paragraph 3.2, thus saving some computational time :

```
Cell A ( tag::vertex );  x(A) = 1.;  y(A) = 0.;
Mesh circle ( tag::frontal, tag::start_at, A,
              tag::towards, { 0., 1. }, tag::desired_length, 0.2 );
```

If, in the above, we choose the opposite `direction`, maniFEM will later try to mesh the exterior of the disk, never stopping.

The same technique can be used for building the `circle` in paragraph 3.4, thus avoiding maniFEM's random choice of the orientation :

```
Cell S ( tag::vertex );  x(S) = 0.;  y(S) = -1.;  z(S) = 0.;
Manifold circle_manif = RR3 .implicit ( x*x + y*y == 1., x*y == 4.*z );
Mesh circle ( tag::frontal, tag::start_at, S,
              tag::towards, { 1., 0., 0. }, tag::desired_length, 0.2 );
```

This also applies to the example in paragraph 3.9. However, because a surface `bumpy` is subsequently built, the topologic considerations are more complex, so we postpone this discussion to paragraph 3.13.

3.12. The intrinsic and inherent orientations

Euclidian manifolds have an intrinsic orientation given by the natural order of the coordinates. ManiFEM will always use this orientation when the working manifold is Euclidian. We can enforce the requirement of using the intrinsic orientation by providing `tag::orientation, tag::intrinsic` as arguments to the `Mesh` constructor. However, these arguments are not necessary; maniFEM will consider them by default. This happens for the disk in paragraphs 3.1 and 3.2, for the diamond in paragraph 3.16 and for the annulus in paragraphs 3.3, 3.23 and 3.27.

Compact manifolds of co-dimension one (closed curves in $\mathbb{R}^2$, compact surfaces in $\mathbb{R}^3$) have a privileged orientation, compatible with the intrinsic orientation of the surrounding space; we call this orientation "inherent". Such a manifold cuts the surrounding space in two regions, one of them being bounded; the "inherent" orientation is the one pointing towards the unbounded region; intuitively speaking, it points "outwards". ManiFEM will choose it in some situations, unless we provide `tag::orientation, tag::random` as arguments to the `Mesh` constructor. We can enforce the requirement of finding the inherent orientation by providing `tag::orientation, tag::inherent` as arguments to the `Mesh` constructor; however, maniFEM will consider these arguments by default when we provide no stopping point (for one-dimensional meshes) or we provide no boundary (for two-dimensional meshes). In these cases, maniFEM will assume that the manifold is compact and, if it has co-dimension one, the inherent orientation will be used. This happens for the `circle` in paragraph 3.2, for the `sphere` in paragraph 3.6 and for the `tori` in paragraph 3.7.

If we do provide a stopping point, or a boundary, maniFEM has no means to know in advance whether the manifold is compact or not, so it will not try to find the inherent orientation. This happens for the `spiral` and for the `arc_of_circle` in paragraph 3.5, for the `bumpy` hemisphere in paragraph 3.9, for the `piece_of_cyl` and `piece_of_sph` in paragraphs 3.18 and 3.19 and others. In these situations, we may require specifically the inherent orientation by providing the `tag::orientation, tag::inherent` arguments to the `Mesh` constructor. Do not misuse this option; if you ask maniFEM to find the inherent orientation of an open curve or of a non-compact surface, it will not complain but will hopelessly try to mesh the entire manifold, never stopping. For instance, in paragraph 3.5 it makes sense to enforce the inherent orientation for the `arc_of_circle` but not for the `spiral`. Also, in paragraphs 3.18 and 3.19 it makes sense to enforce the inherent orientation for the `piece_of_sph` but not for the `piece_of_cyl`.

Determining the intrinsic orientation of a Euclidian manifold is computationally very cheap. However, determining the inherent orientation of a manifold of co-dimension one is not a trivial computational process. ManiFEM is unable to determine this orientation by looking at the equations defining the manifold; it needs a mesh on the entire manifold.

So, in paragraph 3.2 we obtain a correctly oriented mesh on the disk, with no need to provide any specific information about the orientation of `circle_manif`. Behind the curtains, maniFEM builds a mesh on `circle_manif` with an initially arbitrary orientation, then checks its consistency within the surrounding manifold and, if necessary, switches the orientation of the produced mesh. Later, when we ask maniFEM to mesh the `disk`, the orientation of the boundary (together with the intrinsic orientation of the surrounding space `RR2`) defines whether the interior or the exterior of the disk is to be meshed.

A similar process happens for compact surfaces in $\mathbb{R}^3$ like those considered in paragraphs 3.6 and 3.7, except that checking the orientation of a compact surface has a computational cost considerably higher than for a closed curve in $\mathbb{R}^2$.

You can save some computing time if you specify the orientation, either by providing more information to the `Mesh` constructor as shown in paragraph 3.14 or by attaching this information to the manifold itself by means of an outer form (this feature is not implemented yet).

If the orientation is not important for you, you can add to the Mesh constructor the arguments tag::orientation, tag::random and maniFEM will not waste computing time to find the inherent orientation.

3.13. Revisiting the bumpy hemisphere

Let us have a closer look at the code in paragraph 3.9. The manifold nut has a unique orientation consistent with the surrounding space, just like those in paragraphs 3.6 and 3.7, but the base within it has not. This is why, when we build the circle, we must provide tag::orientation, tag::random as last arguments to the Mesh constructor, or specify the orientation by providing a starting point and a starting direction as described in paragraph 3.11.

When we build bumpy, since we are providing a boundary, maniFEM does not treat the current working manifold nut as compact, that is, it does not try to determine an inherent orientation. If arguments tag::orientation, tag::random are provided, maniFEM will start the meshing process on an arbitrary side of circle, thus meshing unpredictably the upper or the lower bumpy hemisphere.

If we enforce the orientation by providing tag::orientation, tag::inherent as last arguments to the Mesh constructor of bumpy, then maniFEM will mesh (behind the curtains) both halves of the nut; let's call them msh1 and msh2. Note that the circle has already been built, so its orientation is given. Both msh1 and msh2 have circle as boundary, which means that they have mutually incompatible orientations; they cannot be joined. But the reverse of any of them can be joined with the other one, the result being a mesh on the entire bumpy sphere. Two choices are possible: either msh1.reverse() is joined with msh2 or msh1 is joined with msh2.reverse(), resulting in two meshes on the same compact surface with opposite orientations. One of these orientations is compatible with the intrinsic orientation of the surrounding space $\mathbb{R}^3$ and is called inherent orientation. If the first one is correctly oriented, msh2 will be returned; if the second one is correctly oriented, msh1 will be returned.

Thus, in the code below we will obtain the upper hemisphere.

```
Cell S ( tag::vertex );  x(S) = 0.;  y(S) = -1.;  z(S) = 0.;
Mesh circle ( tag::frontal, tag::start_at, S, tag::towards, { 1., 0., 0. },
              tag::desired_length, 0.1                                     );
nut .set_as_working_manifold();
Mesh bumpy ( tag::frontal, tag::boundary, circle,
             tag::desired_length, 0.1, tag::orientation, tag::inherent );
```

Code below will produce the lower hemisphere.

```
Cell S ( tag::vertex );  x(S) = 0.;  y(S) = -1.;  z(S) = 0.;
Mesh circle ( tag::frontal, tag::start_at, S, tag::towards, { -1., 0., 0. },
              tag::desired_length, 0.1                                      );
nut .set_as_working_manifold();
Mesh bumpy ( tag::frontal, tag::boundary, circle,
             tag::desired_length, 0.1, tag::orientation, tag::inherent );
```

Code below will mesh unpredictably either the upper or the lower hemisphere, but only due to the random choice in the orientation of circle. The construction of bumpy is uniquely determined by the orientation of circle.

```
Manifold nut = RR3.implicit ( x*x + y*y + z*z + 1.5*x*y*z == 1. );
Manifold base = nut.implicit ( x*x + 3.*z == 0. );
```

```
Mesh circle
   ( tag::frontal, tag::desired_length, 0.1, tag::orientation, tag::random );
nut .set_as_working_manifold();
Mesh bumpy ( tag::frontal, tag::boundary, circle,
             tag::desired_length, 0.1, tag::orientation, tag::inherent );
```

3.14. Specifying the direction

In paragraph 3.11 we have seen how we can specify the direction of propagation of the mesh for one-dimensional manifolds. A similar approach can be taken for two-dimensional meshes. For instance, code below will mesh the upper half of the "bumpy hemisphere" already considered in paragraphs 2.18, 3.9 and 3.13, oriented upwards.

parag-3.14.cpp

```
Cell S ( tag::vertex );  x(S) = 0.;  y(S) = -1.;  z(S) = 0.;
std::vector < double > tau { 1., 0., 0. };
Mesh circle ( tag::frontal, tag::start_at, S, tag::towards, tau,
              tag::desired_length, 0.1                          );
nut .set_as_working_manifold();
std::vector < double > N { 0., 0., 1. };
Mesh bumpy ( tag::frontal, tag::boundary, circle,
             tag::start_at, S, tag::towards, N,
             tag::desired_length, 0.1            );
```

If we define `N` as `{ 0., 0., -1. }`, we will obtain the lower half of the "sphere", still oriented upwards.

If we define `tau` as `{ -1., 0., 0. }`, we will get a mesh oriented downwards.

3.15. Geometric limitations

Frontal meshing has its limits. maniFEM cannot mesh a manifold whose curvature is too high when compared to the length of the segments. The example in paragraph 3.7 is "on the edge", that is, if we increase the segment size the meshing process will probably get stuck somewhere. On the other hand, decreasing the segment size will make the meshing process easier.

Also, we should avoid domains whose boundary has components too close to each other, when compared to the segment size. In this respect, the example in paragraph 3.3 is also "on the edge". In contrast, maniFEM is "at ease" when meshing the same domain with other options, like in paragraphs 3.23, 3.26 or 3.27.

Sharp edges and singular points must be dealt with as described in subsequent paragraphs.

The example in paragraph 3.1 is also "on the edge" for a different reason. While maniFEM tries to build a mesh with segments of constant given length, the length of the segments on the boundary of the disk varies (paragraphs 2.5 and 2.6 explain why). This contradiction puts the algorithm in a difficult position. If the variation were larger (or sharper) the algorithm might stop with some error message. In contrast, maniFEM is "at ease" when meshing the disk in paragraph 3.2 because there the segments on the boundary have (approximately) constant length.

When a `Function` is provided as desired length (paragraphs 3.23 and 3.24), it is the user's resposibility to ensure it takes always positive values and that it is reasonably smooth (has no sharp variations). The same holds if you define a Riemann metric (paragraphs 3.26 – 3.30). You should be careful to provide a definite positive matrix if you use the approach in paragraph

3.27 or 3.28. You should be careful to provide a positive principal part and a semi-positive deviatoric part if you use the approach in paragraph 3.29 or 3.30.

3.16. Sharp angles

maniFEM can deal with non-smooth domains. The user must define separately smooth pieces of the boundary and then join them.

Here is how to mesh the diamond-shaped domain in paragraph 2.12.

parag-3.16.cpp

```
// we define the arcs NW, WS, SE and EN either as segments, like in
RR2 .implicit ( x*y + x - y == -1. );
Mesh NW ( tag::segment, N .reverse(), W, tag::divided_in, 12 );
// or with tag::frontal like in
RR2 .implicit ( x*y - x - y ==  1. );
Mesh WS ( tag::frontal, tag::start_at, W, tag::stop_at, S,
          tag::desired_length, 0.1                        );
// ... //

Mesh bdry ( tag::join, NW, WS, SE, EN );
RR2 .set_as_working_manifold();
Mesh diamond ( tag::frontal, tag::boundary, bdry, tag::desired_length, 0.1 );
```

3.17. Intersection of manifolds

For computing the position of corners, we can project on zero-dimensional manifolds (as already done in paragraph 2.13) :

parag-3.17.cpp

```
Manifold big_circle_left  = RR2 .implicit ( (x-2.)*(x-2.) + y*y == 4. );
Manifold big_circle_right = RR2 .implicit ( (x+1.)*(x+1.) + y*y == 4. );
Manifold two_points ( tag::intersect, big_circle_left, big_circle_right );

Cell A ( tag::vertex, tag::of_coords, { 0.,  1. }, tag::project );
Cell B ( tag::vertex, tag::of_coords, { 0., -1. }, tag::project );
```

3.18. Sharp edges

We can mesh surfaces with sharp edges, by building individually smooth pieces of surface and then `joining` them. The edges must be built first; thus, some previous knowledge about the geometry of the surface is needed.

parag-3.18.cpp

```
const double rs = 1.;    // radius of the sphere
const double rc = 0.45;  // radius of the cylinder
const double seg_size = 0.1;

Manifold cylinder = RR3 .implicit ( y*y + (z-0.5)*(z-0.5) == rc*rc );
Manifold sphere = RR3 .implicit ( x*x + y*y + z*z == rs*rs );

cylinder .implicit ( x == 1.5 );  // we cut the cylinder on its right side
Cell start_1 ( tag::vertex );
x ( start_1 ) = 1.5;  y ( start_1 ) = 0.;  z ( start_1 ) = 0.5 + rc;
```

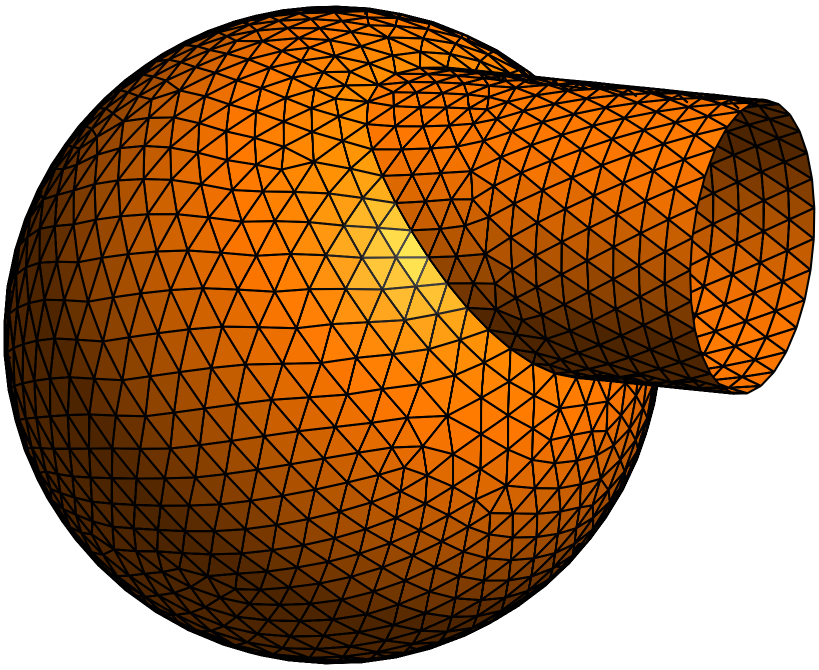


Figure 3.6.: sphere glued to cylinder

```
Mesh circle_1 ( tag::frontal, tag::start_at, start_1,
                tag::towards, { 0., 1., 0. }, tag::desired_length, seg_size );

// we cut the cylinder on its left side with a sphere
Manifold interface ( tag::intersect, cylinder, sphere );
Cell start_2 ( tag::vertex );
x ( start_2 ) = 1.;  y ( start_2 ) = 0.;  z ( start_2 ) = 0.5 - rc;
interface .project ( start_2 );
Mesh circle_2 ( tag::frontal, tag::start_at, start_2,
                tag::towards, { 0., -1., 0. }, tag::desired_length, seg_size );

Mesh two_circles ( tag::join, circle_1, circle_2 .reverse() );

cylinder .set_as_working_manifold();
Mesh piece_of_cyl ( tag::frontal, tag::boundary, two_circles,
                    tag::start_at, start_1, tag::towards, { -1., 0., 0. },
                    tag::desired_length, seg_size                         );

sphere .set_as_working_manifold();
Mesh piece_of_sph ( tag::frontal, tag::boundary, circle_2,
                    tag::start_at, start_2, tag::towards, { 0., 0., -1. },
                    tag::desired_length, seg_size                         );

Mesh sphere_and_cylinder ( tag::join, piece_of_sph, piece_of_cyl );
```

Note how the orientation is important (see paragraphs 1.2 and 1.4). For meshing the piece of cylinder, we provide its future boundary which is the union of `circle_1` with `circle_2.reverse()`. For meshing the sphere, we provide `circle_2`. In figure 3.7, where a crack has been opened artificially, we see that the common boundary, `circle_2`, has a certain orientation when seen from the cylinder and has the opposite orientation when seen from the sphere. If we do not respect these orientations, maniFEM will be unable to `join` the two meshes.

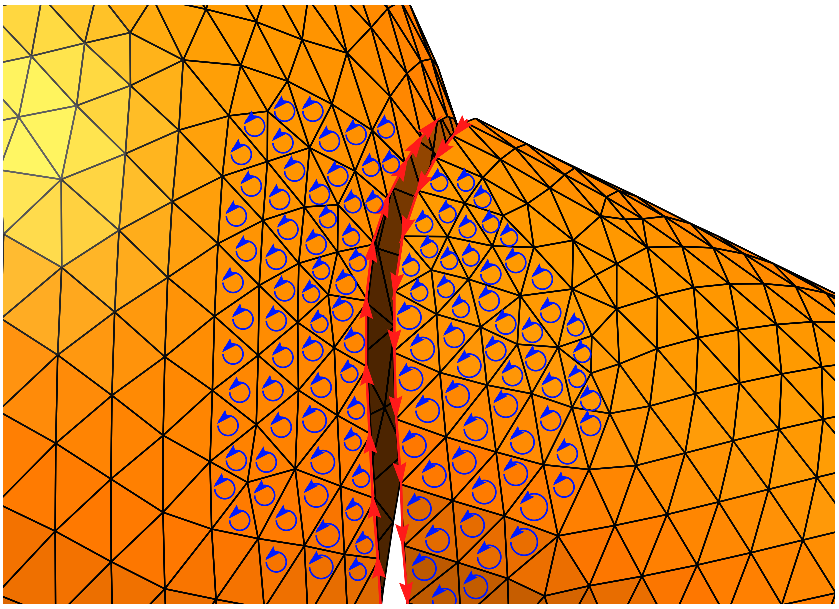


Figure 3.7.: zoom around the joint

Note that `interface` is a disconnected manifold, made of two closed loops, one of them being `circle_2` (the other one shows up in paragraph 3.19 under the name `circle_3`). We must provide detailed information to the `Mesh` constructor in order to obtain the correct loop with the correct orientation. A statement like

```
Mesh circle_2 ( tag::frontal, tag::desired_length, seg_size );
```

would produce a run-time error asking the user to specify the orientation (see comments in paragraph 3.10 about curves in $\mathbb{R}^3$). If we didn't care about the orientation (but we do care in this example), we could use a constructor like

```
Mesh circle_2 ( tag::frontal, tag::desired_length, seg_size,
                tag::orientation, tag::random               );
```

and maniFEM would mesh one connected component of the manifold `intersection` (we would have no control on which component maniFEM happens to find).

If we want to close the extremity of the cylinder, it suffices to add a few lines of code :

```
RR3 .implicit ( x == 1.5 );
Mesh disk ( tag::frontal, tag::boundary, circle_1 .reverse(),
            tag::start_at, start_1, tag::towards, { 0., 0., -1. },
            tag::desired_length, seg_size                         );
Mesh sphere_and_cylinder ( tag::join, piece_of_sph, piece_of_cyl, disk );
```

We have chosen the diameter of the cylinder slightly smaller than 1. Had we chosen a diameter equal to 1, the cylinder would be tangent to the sphere at point $(0, 0, 1)$. At that point, the two equations defining the `intersection` manifold would be degenerate (the Jacobian matrix would not have full rank). This is a situation which maniFEM cannot handle yet; it is discussed in paragraph 3.22.

3.19. Sharp edges, again

With a few changes to the code in paragraph 3.18, we can cut two holes in the sphere and create a tunnel using the cylinder's wall.

parag-3.19.cpp

```
Manifold cylinder = RR3 .implicit ( y*y + (z-0.5)*(z-0.5) == rc*rc );
Manifold sphere = RR3 .implicit ( x*x + y*y + z*z == rs*rs );

Manifold interface ( tag::intersect, cylinder, sphere );

// we choose names start_2 and circle_2 for consistency with previous paragraph
Cell start_2 ( tag::vertex );
x ( start_2 ) = 1.;  y ( start_2 ) = 0.;  z ( start_2 ) = 0.5 - rc;
interface .project ( start_2 );
Mesh circle_2 ( tag::frontal, tag::start_at, start_2,
                tag::towards, { 0., 1., 0. }, tag::desired_length, seg_size );

Cell start_3 ( tag::vertex );
x ( start_3 ) = -1.;  y ( start_3 ) = 0.;  z ( start_3 ) = 0.5 - rc;
interface .project ( start_3 );
Mesh circle_3 ( tag::frontal, tag::start_at, start_3,
                tag::towards, { 0., 1., 0. }, tag::desired_length, seg_size );

Mesh two_circles ( tag::join, circle_2 .reverse(), circle_3 );

cylinder .set_as_working_manifold();
Mesh piece_of_cyl ( tag::frontal, tag::boundary, two_circles,
                    tag::start:at, start_2, tag::towards, { -1., 0., 0. },
                    tag::desired_length, seg_size                        );
```

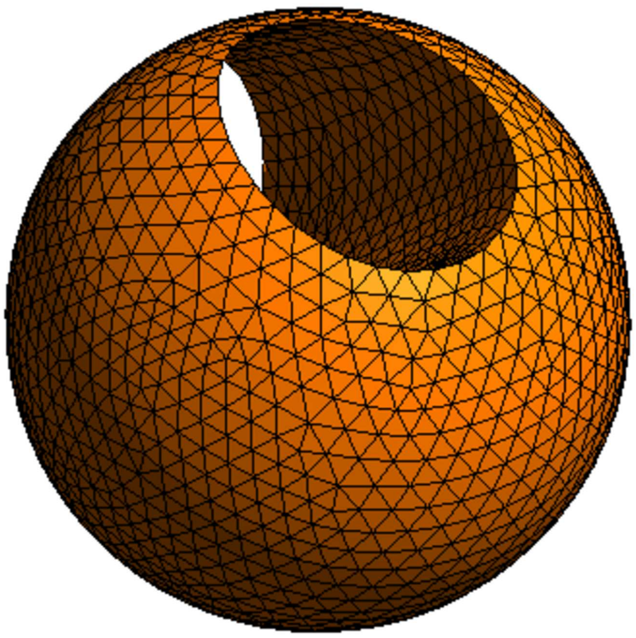


Figure 3.8.: sphere with tunnel

parag-3.19.cpp
```
sphere .set_as_working_manifold();
Mesh piece_of_sph ( tag::frontal, tag::boundary, two_circles .reverse(),
                    tag::start:at, start_2, tag::towards, { 0., 0., -1. },
                    tag::desired_length, seg_size                          );

Mesh perf_sphere ( tag::join, piece_os_sph, piece_of_cyl );
```

maniFEM deals well with a disconnected manifold like `interface`. Each of the constructors `Meshcircle_2` and `Meshcircle_3` meshes only a connected component of `interface`, depending on the starting point we provide.

Had we chosen a cylinder of diameter 1, a singular point would appear; the two "circles" would touch at point $(0, 0, 1)$. This is a situation which *maniFEM* cannot handle yet; it is discussed in paragraph 3.22.

3.20. Overlapping meshes

In *maniFEM*, cells (including vertices) are independent of their geometric coordinates. We can build two distinct vertices `O1` and `O2`, both occupying the origin of `RR2`. Similarly, `A1` and `A2` have the same geometric coordinates, although they are different `Cell` objects (with different cores). Also, segments `O1A1` and `O2A2` are superimposed.

parag-3.20.cpp
```
Cell O1 ( tag::vertex );  x(O1) = 0.;  y(O1) = 0.;
Cell A1 ( tag::vertex );  x(A1) = 1.;  y(A1) = 0.;
Mesh O1A1 ( tag::segment, O1 .reverse(), A1, tag::divided_in, n );
Cell O2 ( tag::vertex );  x(O2) = 0.;  y(O2) = 0.;
Cell A2 ( tag::vertex );  x(A2) = 1.;  y(A2) = 0.;
Mesh O2A2 ( tag::segment, O2 .reverse(), A2, tag::divided_in, n );
```

We can take advantage of this peculiarity of *maniFEM* and build meshes which pass more than once in the same region of the space (they overlap themselves). In this paragraph, we build a mesh by `join`ing four half-annuli.

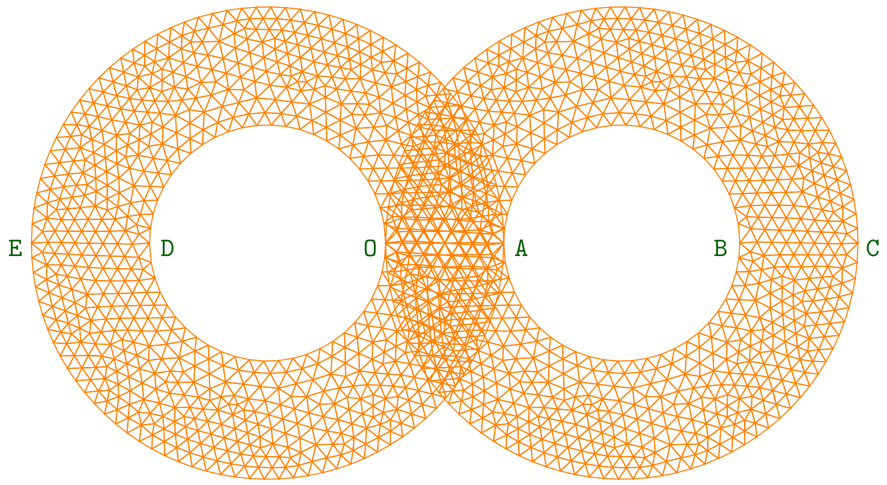


Figure 3.9.: a self-overlapping mesh

The result, shown in figure 3.9, is not two overlapping annuli, it is an infinity-shaped strip. The boundary of this mesh has two connected components. Each of these components is a (self-intersecting) smooth curve. Due to the overlapping, it is impossible to distinguish between an "inner" boundary and an "outer" one. In contrast, the mesh built in paragraph 3.17 has a boundary made of three connected components : two inner circles and an outer boundary with two re-entrant corners.

3.21. Singularities

The code described in this paragraph does not work yet. It should be regarded as a mere declaration of intentions.

If we want to mesh (a part of) a surface with singular points, we must give to maniFEM knowledge about these singularities. A typical example is the vertex of a cone.

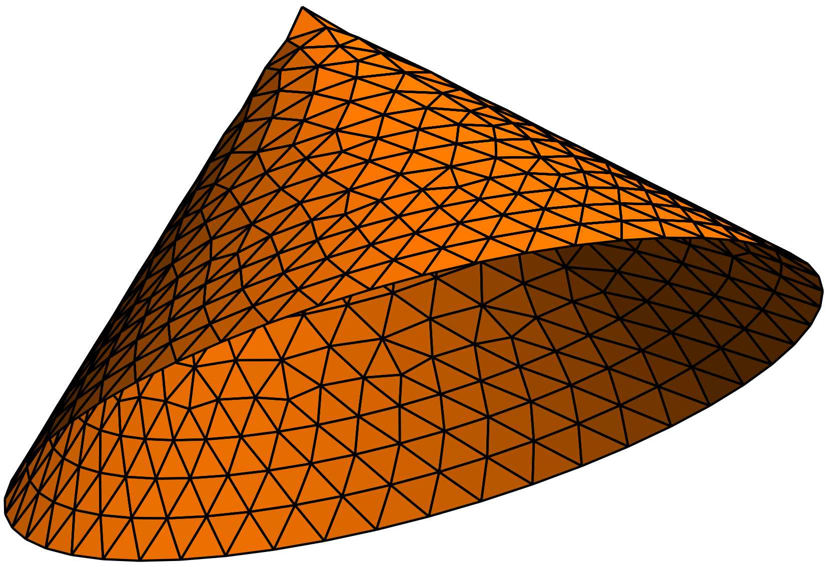


Figure 3.10.: a singularity (vertex)

code not working

```
Manifold cone_manif = RR3 .implicit ( x*x + y*x == z*z );

cone_manif .implicit ( z == 1. );
Cell A ( tag::vertex );  x(A) = 1.;  y(A) = 0.;  z(A) = 1.;
Mesh circle ( tag::frontal, tag::start_at, A,
              tag::towards, { 0., -1., 0. },
              tag::desired_length, 0.1       );

cone_manif .set_as_working_manifold();
Cell V ( tag::vertex );  x(V) = 0.;  y(V) = 0.;  z(V) = 0.;
Mesh cone ( tag::frontal, tag::boundary, circle,
            tag::start_at, A, tag::towards, { -1., 0., -1. },
            tag::singular, V, tag::desired_length, 0.1       );
```

3.22. Singularities, again

The code described in this paragraph does not work yet. It should be regarded as a mere declaration of intentions.

Besides vertices like the one described in paragraph 3.21, another kind of singularity appears when we intersect two manifolds which have a tangency point. This happens if we choose `rc = 0.5` in paragraph 3.18 or 3.19.

We focus on the piece of cylinder from paragraph 3.19 (which is to be subsequently `joined` with a sphere with two holes).

code not working

```
  Manifold cylinder = RR3 .implicit ( y*y + (z-0.5)*(z-0.5) == 0.25 );
  Manifold infinity = cylinder .implicit ( x*x + y*y + z*z == 1. );

  Cell V ( tag::vertex );  x(V) = 0.;  y(V) = 0.;  z(V) = 1.;
  Mesh circle_1 ( tag::frontal,
                  tag:: start_at, V, tag::towards, { 1., 1., 0. },
                  tag::singular, V, tag::desired_length, 0.1      );
  Mesh circle_2 ( tag::frontal,
                  tag:: start_at, V, tag::towards, { -1., -1., 0. },
                  tag::singular, V, tag::desired_length, 0.1        );

  cylinder .set_as_working_manifold();
  Cell W = circle_1 .cell_in_front_of (V) .tip();
  Mesh piece_of_cyl ( tag::frontal, tag::boundary, circle_1, circle_2,
                      tag::start_at, W, tag::towards, { -1., 1., 0. },
                      tag::singular, V, tag::desired_length, 0.1      );
```

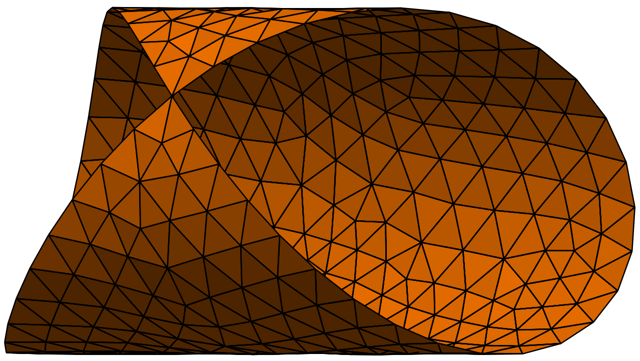


Figure 3.11.: a singularity (napkin)

ManiFEM accepts as `Manifold` a self-intersecting set like `infinity`. In maniFEM, an implicit manifold is simply a level set of one or two symbolic expressions involving the coordinates of the surrounding, Euclidean manifold. However, in the neighbourhood of a singularity, the equation(s) are degenerate and certain parts of maniFEM do not work as expected. In particular, the projection algorithm onto the manifold may fail to converge. By specifying (in the `Mesh` constructor) `V` as singular point, we tell maniFEM that it must take special care in that neighbourhood.

Unlike in paragraph 3.19, here we cannot `join` the two meshes `circle_1` and `circle_2`. ManiFEM is unable to join two closed polygonal lines having a vertex in common; in other words, it does not handle self-intersecting `Mesh`es. This is why we provide the two pieces of boundary separately.

3.23. Non-uniform meshing

The desired length may be a non-constant function.

parag-3.23.cpp
```
Function d = 0.03 + 0.04 * ( (x+0.3)*(x+0.3) + (y-0.9)*(y-0.9) );

Manifold circle = RR2 .implicit ( x*x + y*y == 1. );
Mesh outer ( tag::frontal, tag::desired_length, d );
Manifold ellipse = RR2 .implicit ( x*x + (y-0.37)*(y-0.37) + 0.3*x*y == 0.25 );
Mesh inner ( tag::frontal, tag::desired_length, d );

Mesh bdry ( tag::join, outer, inner .reverse() );

RR2 .set_as_working_manifold();
Mesh disk ( tag::frontal, tag::boundary, bdry, tag::desired_length, d );
```

It is the user's responsibility to provide a length `d` which takes positive values at all points of the future mesh. Also, `d` should be a reasonably smooth function; sharp variations should be avoided.

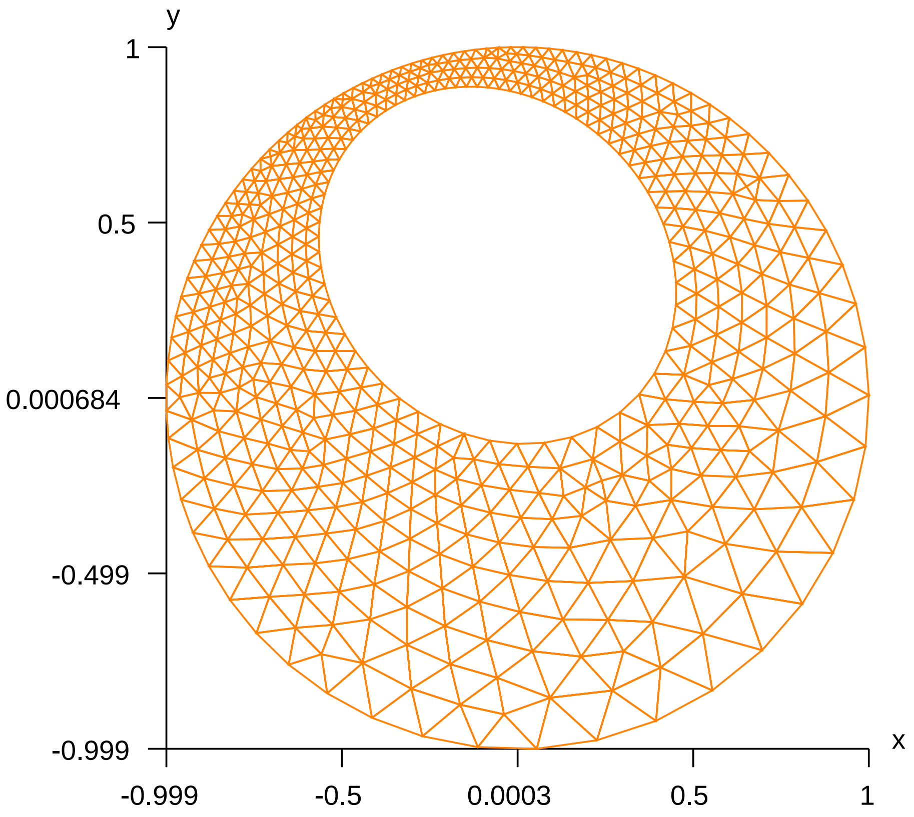


Figure 3.12.: non-uniform mesh

Compare with the mesh in paragraph 3.3.

3.24. Two intersecting tori

Going back to the two tori in paragraph 3.7, we now build the edges first and obtain a sharp, precise cut (forget about `smooth_min`).

parag-3.24.cpp
```
// unlike in paragraph 3.7, here we don't need to cut f1 and f3 with 0.1
// (the shape of each torus avoids the singularities)
// we change slightly f3 (replace 0.4 by 0.5) to highlight the intersection
Function f1 = x*x + y*y;
Function f2 = 1. - power ( f1, -0.5 );
Function d1 = f1 * f2 * f2 + z*z;  // squared distance to a circle in the xy plane
Function f3 = (x-0.5)*(x-0.5) + z*z;
Function f4 = 1. - power ( f3, -0.5 );
Function d2 = y*y + f3 * f4 * f4;  // squared distance to a circle in the xz plane
```

We also define the desired length of segments as a `Function` in order to get a finer mesh around the joints :

parag-3.24.cpp
```
Function seg_size = 0.1 * power ( 0.15 + abs ( d2 - d1 ), 0.4 );
```

We begin by meshing each connected component of the intersection between the two tori, then `join` them in a disconnected one-dimensional mesh :

parag-3.24.cpp
```
Manifold intersection = RR3 .implicit ( d1 == 0.15, d2 == 0.15 );

Cell start_1 ( tag::vertex, tag::of_coords, { 1.3, -0.4, 0. }, tag::project );
Mesh loop_1 ( tag::frontal, tag::start_at, start_1,
              tag::towards, { 0., 0., 1. }, tag::desired_length, seg_size );

Cell start_2 ( tag::vertex, tag::of_coords, { -0.9, 0., 0.4 }, tag::project );
Mesh loop_2 ( tag::frontal, tag::start_at, start_2,
              tag::towards, { 0., 1., 0. }, tag::desired_length, seg_size );

Mesh two_loops ( tag::join, loop_1, loop_2 );
```

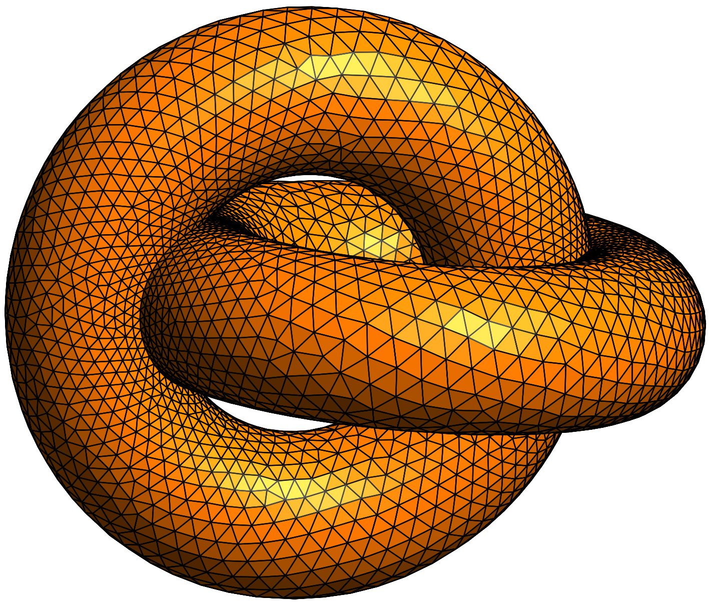


Figure 3.13.: sharp edges

Starting from this disconnected mesh, we build two tori, then `join` them together. We start the frontal mesh generation process at the same location, `start_1`, but provide different directions. These directions are approximately tangent to the manifold and approximately orthogonal to the boundary, at point `start_1`.

parag-3.24.cpp
```
RR3 .implicit ( d1 == 0.15 );
Mesh torus_1 ( tag::frontal, tag::boundary, two_loops,
               tag::start_at, start_1, tag::towards, { -0.2, -1., 0. },
               tag::desired_length, seg_size                             );
RR3 .implicit ( d2 == 0.15 );
Mesh torus_2 ( tag::frontal, tag::boundary, two_loops .reverse(),
               tag::start_at, start_1, tag::towards, { 1., -0.2, 0. },
               tag::desired_length, seg_size                            );

Mesh two_tori ( tag::join, torus_1, torus_2 );
```

Figure 3.13 shows the resulting mesh.

3.25. Filling tori with tetrahedra

We fill the volume bounded by the surface `two_tori` (built in the previous paragraph) with tetrahedra. Like in paragraph 3.8, it suffices to switch to the manifold `RR3` and invoke the frontal mesh generation algorithm :

parag-3.25.cpp
```
RR3 .set_as_working_manifold();
Mesh two_tori_filled ( tag::frontal, tag::boundary, two_tori,
                       tag::desired_length, seg_size         );
```

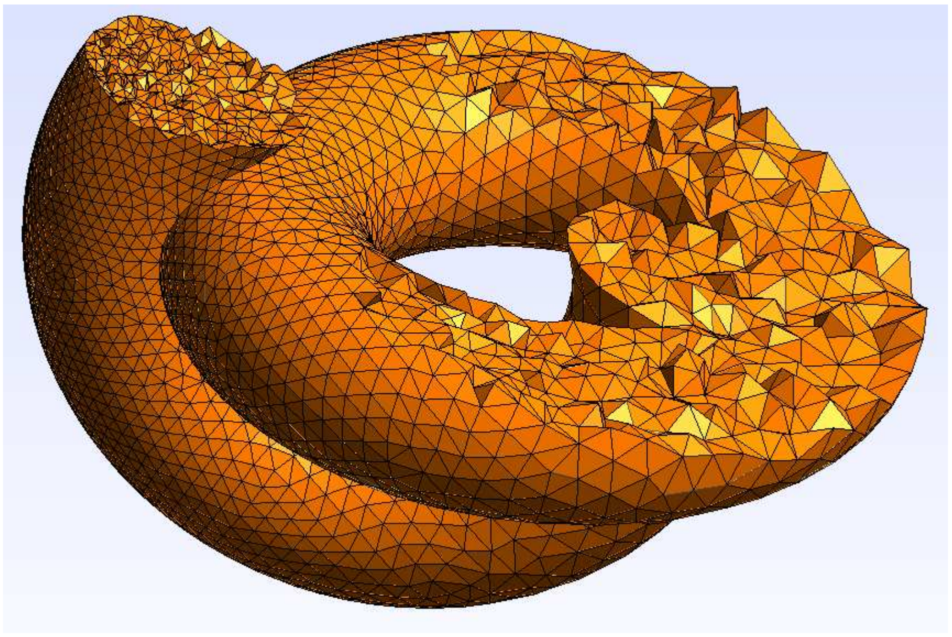


Figure 3.14.: filling two tori with tetrahedra

We have chosen a different `seg_size` from paragraph 3.24 because that function has singularities. Those singularities are avoided in paragraph 3.24 because the frontal mesh generation happens on the surface `two_tori`. In the present paragraph, the mesh generation happens inside the volume.

In figure 3.14, a cut allows the user to see inner cells. This was done using the following options of `gmsh`: `Tools` → `Clipping` → `Mesh` → `Planes` → `Keep whole elements`, $A = -0.25, B = -1., C = 0., D = 0.3$.

3.26. Changing the Riemann metric

We can attach a non-uniform metric to our manifold; as a consequence, for a constant `desired_length`, the apparent segment length will vary from zone to zone. For instance, in the code below we set a metric which increases the measured length of the segments close to the narrow part of the domain. As a consequence, the meshing algorithm, while trying to build a mesh of constant `desired_length`, will choose shorter segments in the proximity of the narrow zone (because the length measured by the Riemann metric will be larger than the length we see in the drawing), thus producing the same result as in paragraph 3.23.

parag-3.26.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

Function d = 0.3 + 0.5 * ( (x+0.3)*(x+0.3) + (y-0.9)*(y-0.9) );
RR2 .set_metric ( tag::isotropic, 1. / d );
// implicit manifolds 'circle' and 'ellipse' will inherit this metric

Manifold circle = RR2 .implicit ( x*x + y*y == 1. );
Mesh outer ( tag::frontal, tag::desired_length, 0.1 );
Manifold ellipse = RR2 .implicit ( x*x + (y-0.37)*(y-0.37) + 0.3*x*y == 0.25 );
Mesh inner ( tag::frontal, tag::desired_length, 0.1 );

Mesh bdry ( tag::join, outer, inner .reverse() );

RR2 .set_as_working_manifold();
Mesh disk ( tag::frontal, tag::boundary, bdry, tag::desired_length, 0.1 );
```

These two approaches (the one described in paragraph 3.23 and the one described here) can be used interchangeably. It is possible to use both in the same code, but the code will become rather obscure.

3.27. Anisotropic metric

The technique described in paragraph 3.26 can be generalized to an anisotropic Riemann metric. We define the metric by means of a square matrix M which must be symmetric positive definite. The arguments of `set_metric` are, besides a descriptive `tag`, m_{11}, m_{12} and m_{22} for two dimensions, m_{11}, m_{12}, m_{13}, m_{22}, m_{23} and m_{33} for three dimensions.

The matrix M should be interpreted in the following sense: for any two vectors v and w tangent to the manifold at a given point P, their inner product is defined as $\langle v, w \rangle_a = \sum_{i,j} m_{ij}(\mathtt{P})\, v_j w_i$. As a consequence, the norm of a vector v is given by $\|v\|_a = \sqrt{\sum_{i,j} m_{ij}(\mathtt{P})\, v_j v_i}$ (the subscript $_a$ stands for "anisotropic").

In the code below, we define M as the sum between the identity matrix and a perturbation depending on a quantity `beta` which is very small at points far from the narrow part of the domain. We have chosen this rather complicated expression for `beta` in order for the matrix M to be the square of another matrix, showing up in paragraph 3.28.

parag-3.27.cpp
```
  Function alpha = 5. + 10. * ( (x+0.3)*(x+0.3) + (y-0.9)*(y-0.9) );
  Function beta = ( 1. + 5. / alpha ) / alpha;
  RR2 .set_metric ( tag::m11_m12_m22, 1. + 2.*beta, -6.*beta, 1. + 18.*beta );
  // implicit manifolds 'circle' and 'ellipse' will inherit this metric
```

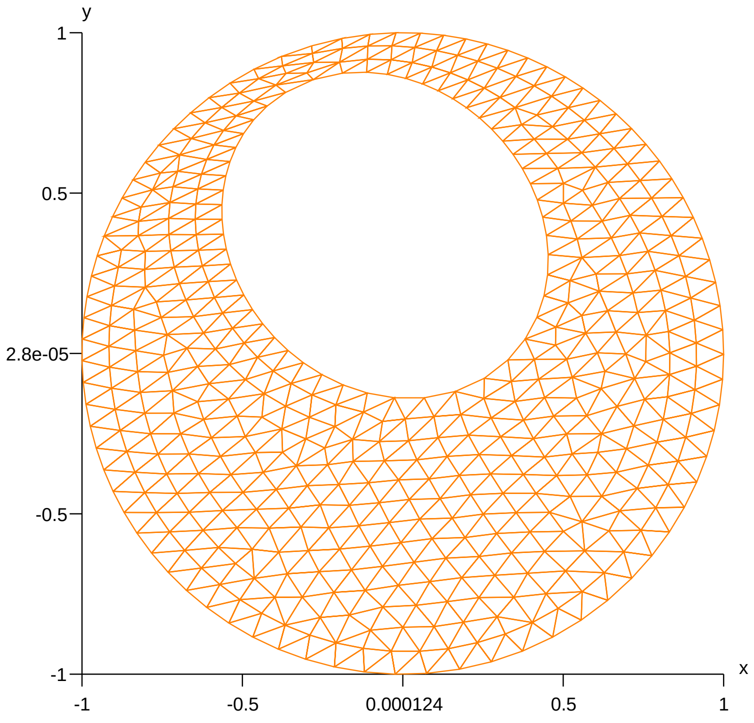


Figure 3.15.: anisotropic mesh

Compare with the mesh in paragraph 3.3.

This result cannot be achieved using the approach of paragraph 3.23 (by providing a non-constant desired length).

3.28. Anisotropic metric, square root of the matrix

Instead of describing the metric as in paragraph 3.27, we can provide another symmetric positive definite matrix A and define the inner product as $\langle v, w\rangle_a = \langle Av, Aw\rangle$, where $\langle\cdot,\cdot\rangle$ stands for the usual Euclidian product. Thus, the norm of a vector is given by $\|v\|_a = \|Av\|$, where $\|\cdot\|$ stands for the usual Euclidian norm.

The matrices M from paragraph 3.27 and A here are related by $M = A^2$, so A may be viewed as the "square root" of M.

For providing this information to *maniFEM* we use a `tag::square_root`. You may also replace `tag::m11_m12_m22` by `tag::a11_a12_a22`, they are interchangeable.

parag-3.28.cpp
```
  Function alpha = 5. + 10. * ( (x+0.3)*(x+0.3) + (y-0.9)*(y-0.9) );
  RR2 .set_metric ( tag::square_root,
                    tag::m11_m12_m22, 1. + 1./alpha, -3./alpha, 1. + 9./alpha );
```

The code above will produce the same mesh as in paragraph 3.27. Providing the square root of M rather than M itself turns the frontal mesh generation process more efficient (maniFEM can avoid computing many square roots, thus gaining speed).

However, if the matrix A is given by expressions (e.g. involving square roots) which are more complicated than those defining M, there is no advantage in using the approach presented here (with `tag::square_root`). In these cases, we recommend using the direct approach described in paragraph 3.27.

3.29. Inner spectral radius

During frontal mesh generation with anisotropic Riemann metric, following either of the paths outlined in paragraph 3.27 or 3.28, maniFEM must compute, when building each new vertex, the inner spectral radius of the matrix A, or, equivalently, compute the square root of the inner spectral radius of M. The reasons are explained in paragraph 12.15.

By "inner spectral radius" we simply mean the lowest eigenvalue of the matrix (which should be positive).

In order to make the frontal mesh generation process more efficient, the user may provide (a lower bound on) the inner spectral radius of the matrix M as below.

parag-3.29.cpp
```
  Function alpha = 5. + 10. * ( (x+0.3)*(x+0.3) + (y-0.9)*(y-0.9) );
  Function beta = ( 1. + 5. / alpha ) / alpha;
  RR2 .set_metric ( tag::inner_spectral_radius, 1.,
                    tag::m11_m12_m22, 2.*beta, -6.*beta, 18.*beta );
```

So, we provide M as the sum between a "principal part" σI and a "deviatoric part" D. The scalar σ should be either a positive constant or a `Function` taking positive values (in the above, σ is 1) . The deviatoric matrix D should be symmetric positive semi-definite. For maximum efficiency, D should be singular but this is not mandatory. You may replace `tag::m11_m12_m22` by `tag::d11_d12_d22`, they are interchangeable.

3.30. Inner spectral radius, square root

We can combine both advantages of paragraphs 3.28 and 3.29 by providing the matrix $A = \sqrt{M}$ as the sum between a "principal part" σI and a "deviatoric part" :

parag-3.30.cpp
```
  Function alpha = 5. + 10. * ( (x+0.3)*(x+0.3) + (y-0.9)*(y-0.9) );
  RR2 .set_metric ( tag::square_root, tag::inner_spectral_radius, 1.,
                    tag::m11_m12_m22, 1./alpha, -3./alpha, 9./alpha  );
```

Paragraphs 3.27 – 3.30 build the same mesh shown in figure 3.15, with increasing degree of efficiency.

3.31. Future work

It would be nice to define domains using inequalities between `Function`s.

The stability of frontal mesh generation with tetrahedra should be improved, as well as the quality of the produced mesh.

4. Exchanging information with third-party software

4.1. Composite meshes

In this paragraph and the next one we show how maniFEM can use the `msh` format (see `https://gmsh.info`) for writing and reading back complex meshes.

To fix ideas, we focus on the example in paragraph 3.18, where a mesh called `sphere_and_cylinder` is built. We can save this mesh using method `Mesh::export_to_file`, then use `gmsh` to draw it. But if we want to re-use this mesh in another maniFEM program, the declaration

```
Mesh sphere_and_cylinder ( tag::import, tag::gmsh, "filename.msh" );
```

may be insufficient for our needs. You see, in the original program (the one presented in paragraph 3.18) we had at our disposal not only the `Mesh` object `sphere_and_cylinder` but also other `C++` objects like `piece_of_sph`, `piece_of_cyl`, `two_circles`, `circle_1` and `circle_2`. Depending on our intentions, these objects may be useful. For instance, we may want to iterate over all segments of `circle_1` in order to impose some boundary condition. Or, we may want to `join` the mesh `sphere_and_cylinder` to some other mesh (say, another sphere). In order for the `join` operation to work well, we must build that other mesh starting from `circle_1` as boundary. If you don't understand the previous statement, you may want to read again paragraphs 1.4 and 3.18.

The problem is, in the second maniFEM program, the one which `imports` the mesh `sphere_and_cylinder` from an `msh` file, we do not have access to the `circle_1` object. True, we can recover it by testing each segment of `sphere_and_cylinder` (or rather, both extremities of each segment) against the equation `x == 1.5`, but this not elegant. It would be nice to extract, from one `msh` file, several related `Mesh` objects. And this is only possible if we save, in the first maniFEM program, more precise information than merely one large `Mesh` object.

Luckily, the `msh` file format allows one to specify regions of a mesh either through `EntityTags` (which are integer numbers) or through `PhysicalNames` (which are strings). ManiFEM takes advantage of the latter feature to save (and later recover) several `Mesh`es, components of one large `Mesh`, in (and from) one `msh` file.

The class `Mesh::Composite` allows the user to save, using a clear and elegant syntax, components of a `Mesh` and later retrieve them from an `msh` file, using strings to identify them. A `Mesh::Composite` object is simply a collection of cells and meshes, endowed with methods for adding and retrieving such components.

Here is how one builds a composite mesh and saves it in an `msh` file :

parag-4.1.cpp

```
Mesh::Composite sphere_cyl;
sphere_cyl .add_cell ( "start 1", start_1 );
sphere_cyl .add_mesh ( "circle 1", circle_1 );
sphere_cyl .add_mesh ( "circle 2", circle_2 );
sphere_cyl .add_mesh ( "cylinder", cyl );
```

```
   sphere_cyl .add_mesh ( "sphere", sph );
   sphere_cyl .export_to_file ( tag::gmsh, "sphere-cyl-composite.msh" );
```

In the above, we save the vertex `start_1` because it may be needed for building another mesh from `circle_1`. We choose not to keep `start_2` because it is unlikely to be needed for later use.

For geometric dimension higher than 3, the exported file will not be readable by `gmsh`. However, it can still be used by maniFEM to read `Mesh` and `Cell` objects.

4.2. Recovering composite meshes

In a separate program, we retrieve each component of the mesh from the file created in the previous paragraph, using methods `retrieve_cell` and `retrieve_mesh`.

parag-4.2.cpp

```
   Mesh::Composite cyl_sph
      ( tag::import_from_file, tag::gmsh, "sphere_cyl_composite.msh" );
   Cell S        = cyl_sph .retrieve_cell ( "start 1" );
   Mesh circle_1 = cyl_sph .retrieve_mesh ( "circle 1" );
   Mesh circle_2 = cyl_sph .retrieve_mesh ( "circle 2" );
   Mesh cylinder = cyl_sph .retrieve_mesh ( "cylinder" );
   Mesh sphere   = cyl_sph .retrieve_mesh ( "sphere" );
   Mesh all ( tag::join, cylinder, sphere );
```

4.3. Exercise

Change the code in paragraph 3.8 to save the mesh of tetrahedra as a `Mesh::Composite`. In a separate program, read this mesh and extract separately the `sphere` and the `ball` and obtain the same results as in paragraph 6.10.

4.4. Fancy coloring in `gmsh`

Composite meshes can be used to obtain colorful drawings using `gmsh` (see `https://gmsh.info`). This technique has already been used for the colorful box in paragraph 2.3; here we show further examples and provide more details.

We go back to the imperfect torus in paragraph 2.21. We stress once more that this is a mere rectangle in the `alpha_beta` plane, twisted into the `xyz` space to the shape of a doughnut :

parag-4.4.cpp

```
   Manifold torus ( tag::Euclid, tag::of_dim, 2 );
   Function alpha_beta =
      torus .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
   Function alpha = alpha_beta [0], beta = alpha_beta [1];

   // build a rectangle in the alpha-beta plane
   const double pi = 3.1415926536;
   Cell A ( tag::vertex );  alpha (A) = 0.;     beta (A) = 0.;
   Cell B ( tag::vertex );  alpha (B) = 0.;     beta (B) = 2.*pi;
   Cell C ( tag::vertex );  alpha (C) = 2.*pi;  beta (C) = 2.*pi;
   Cell D ( tag::vertex );  alpha (D) = 2.*pi;  beta (D) = 0.;
   Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 20 );
   Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in, 40 );
   Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, 20 );
```

```
Mesh DA ( tag::segment, D .reverse(), A, tag::divided_in, 40 );
Mesh ABCD ( tag::rectangle, AB, BC, CD, DA );

// parametrize the torus
const double big_radius = 3., small_radius = 1.;
Function x = ( big_radius + small_radius * cos(beta) ) * cos(alpha),
         y = ( big_radius + small_radius * cos(beta) ) * sin(alpha),
         z = small_radius * sin(beta);
Manifold RR3 ( tag::Euclid, tag::of_dimension, 3 );
RR3 .set_coordinates ( x && y && z );  // forget about alpha and beta
```

The mesh `ABCD` has a boundary made of four segments; the opposite sides overlap in the `xyz` space.

Below we save this imperfect torus as a composite mesh. We want to keep the surface `ABCD` and two of its four sides, for instance `AB` and `BC`, for graphics purposes.

parag-4.4.cpp

```
Mesh::Composite comp_msh;
comp_msh .add_mesh ( "big loop", BC );
comp_msh .add_mesh ( "small loop", AB );
comp_msh .add_mesh ( "torus", ABCD );

comp_msh .export_to_file ( tag::gmsh, "composed.msh" );
```

We can then choose options in `gmsh` like `Tools` → `Options` → `Mesh` → `General` → unselect `2D element edges`, then `Aspect` → `Line width` 5, then `Color` → `Smooth normals`, then choose colors `By elementary entities`, under `One`, `Two`, `Three` and so on. You can find out the correspondence between these numerals and the strings provided to maniFEM through the `add_cell` and `add_mesh` methods by opening the file `composed.msh` and looking for the `PhysicalNames` section. Or, you can just guess which are the corresponding meshes by trial and error. For exporting the file in the `eps` format, you may also want to disable `Color` → `Use two-side lighting`, then go to `General` → `Color` and change the `Light position`, e.g. by switching the signs of all coordinates. You may also want to go to `Axes` and disable `Show small axes`. The resulting drawing is shown in figure 4.1.

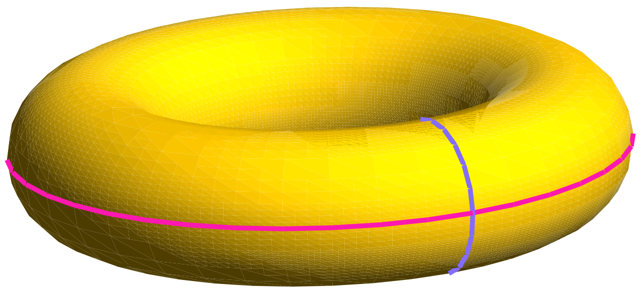


Figure 4.1.: a torus with two loops

4.5. Another torus with loops

We may prefer to mesh the torus using frontal mesh generation, in the spirit of paragraphs 3.7 and 3.24. The torus is defined as a submanifold of $\mathbb{R}^3$ through the implicit equation `d == 0.15`, where

parag-4.5.cpp
```
   Function f1 = x*x + y*y;
   Function f2 = 1. - power ( f1, -0.5 );
   Function d = f1 * f2 * f2 + z*z;  // squared distance to a circle in the xy plane
```

The one-dimensional meshes intended to be visible in the drawing (the two loops) must be built first. Actually, we need to build the torus from four parts, and for this we need to define eight segments. Only four segments are kept for graphic purposes, gathered as `big_loop` and `small_loop`.

In the future, the construction described above (torus made of four pieces, from eight segments) will be simplified by defining new `Mesh` constructors with `tag::frontal` and `tag::embed` which will embed a given interface in the mesh being built.

4.6. A sphere with three handles

This is an example of a more complicated surface with embedded curves (figure 4.2).

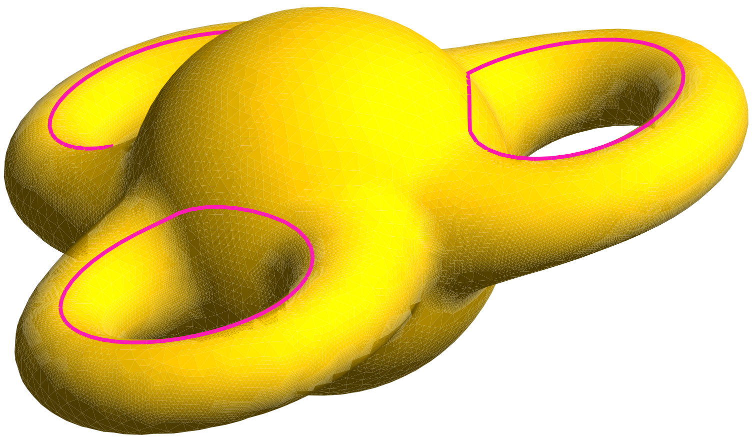


Figure 4.2.: a sphere with three handles

Comments at the end of paragraph 4.5 apply : in the future the user will be able to embed an interface by providing `tag::embed` as argument of the `Mesh` constructor.

5. Functions and variational formulations

5.1. Functions

As the reader may have already noted, all examples in maniFEM begin by declaring a Euclidian Manifold and then go on by building a coordinate system :

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];
```

See paragraph 11.3 for more details about tags.

In the above, xy is a vector-valued Function with two components, x and y. The method Manifold::build_coordinate_system invokes, invisibly to the user, statemets similar to [1]

```
Function xy ( tag::lives_on, tag::vertices, tag::has_size, 2 );
```

The above declaration of xy starts a complex process; behind the curtains, maniFEM builds a Field object associated to xy.[2] The Field object changes the behaviour of maniFEM in what regards initialization of Cells. Since xy is of type Lagrange of degree 1, each newly built vertex Cell will have memory space reserved for two double precision numbers. If A is a vertex Cell, an assignment like x(A) = 1.5 sets the value of the first component of the Field associated to xy at A. Note that negative vertices will have no coordinates, only positive vertices will.

If we declare xy to be of type Lagrange of degree 2, not only future vertices will have space reserved for two doubles, but also future segments. So, we may assign the value of y associated to a segment AB by using the syntax y(AB) = 0.75. This number may be interpreted as the value taken by y at the middle of segment AB. This feature is not yet implemented.

Other Functions are not related with space in the computer's memory (they have no subjacent Field). For instance, there are arithmetic expressions like

```
Function norm = power ( x*x + y*y + z*z, 0.5 );
```

We can still access the value of such a Function at a cell like in double n = norm(A); the computer will return the value of the corresponding arithmetic expression, replacing the symbols x, y and z by the corresponding values at cell A. But it makes no sense to change the value of norm at cell A. Thus, statements like x(A) = 1. work fine, while norm(A) = 1. produces a run-time error.

There are also abstract basis Functions which are mere placeholders, to be later replaced by specific shape functions in a FiniteElement, as explained in paragraphs 6.11 and 6.12. Similar abstract functions will be used for implementing VariationalFomrulations.

The deriv method performs symbolic differentiation :

```
Function d_norm_d_x = norm .deriv (x);
Function d_norm_d_y = norm .deriv (y);
```

Functions can also be integrated, see paragraph 6.2.

[1] Such a statement appears explicitly in paragraph 7.29.

[2] In order to avoid having too many names exposed in namespace maniFEM, we chose to hide the name Field inside the namespace tag::Util.

6. Finite elements and integrators

This section describes maniFEM's implementation of finite elements. Paragraph 6.1 discusses the concept of finite element. Paragraph 6.2 shows an examples of use of an `Integrator` without declaring a `FiniteElement`. Paragraphs 6.3 – 6.12 provide examples showing different types of `FiniteElement`s. There is a lot of ongoing work on this subject.

ManiFEM distinguishes between cells (which are geometric entities) and finite elements. We may have different finite elements whose geometry is the same. For instance, we may have triangular Lagrange finite elemens of different degrees; they will all `dock_on` the same triangular cell. See e.g. paragraph 6.6.

Finite elements in maniFEM can be divided in two categories. Finite elements based on symbolic computations, using Gauss quadrature on a master element, are discussed in paragraphs 6.3 – 6.10. They are flexible (we can compute integrals of arbitrary expressions) but quite slow (because they rely strongly on symbolic manipulation of expressions). They can be interesting for dydactic purposes.[1] Finite elements described in paragraphs 6.11 and 6.12 are much faster at the cost of a tedious internal implementation. Also, from the point of view of the final user, their syntax is slightly less elegant.

Note that many examples in this section make use of the `Eigen` library; paragraph 11.14 gives details.

6.1. Finite elements

The notion of finite element is quite complex. At global level (that is, at the level of the entire mesh), the purpose of a `FiniteElement` is to build a list of functions, say, ψ, defined on our mesh. The linear span of these functions will be a finite-dimensional Hilbert space (the discretization of an infinite-dimensional Sobolev space). It is the `FiniteElement`'s job to replace, in the variational formulation, the unknown function by one ψ, the test function by another ψ and, by evaluating the integrals, obtain the coefficients of a system of linear equations. Some external solver will then solve the system, and it is the job of the finite element to transform back the vector produced by the solver into a function defined on our mesh.

Often, we view finite elements as acting at cell level. According to this view, the finite element's job is to build a local matrix, to be later assembled into a global matrix.

Computing each integral is a somewhat separate process; it's the job of an `Integrator` which could be a Gauss quadrature or some other procedure like symbolic integration. The separation between a `FiniteElement`'s job and the `Itegrator`'s job is not very sharp. For instance, the Gauss quadrature is often perfomed not on the physical cell but rather on a master element which is built and handled by the `FiniteElement`. The authors of maniFEM have tried to separate these two concepts as much as possible, especially because we may want to use a `FiniteElement` with no master element, or an `Integrator` acting directly on the physical cell.

An object belonging to the `class FiniteElement` is a mere wrapper for a `FiniteElement::Core` (see paragraph 11.5). Several subclasses of `FiniteElement::Core` exist for different kinds of finite elements. There is a `class FiniteElement::WithMaster` with subclasses like `FiniteElement::`

[1] They are also very useful for validating other types of finite elements.

`::WithMaster::Segment`, `Triangle` and `Quadrangle`. There is also a `class FiniteElement::StandAlone` for elements with no master, having its own subclasses for different geometries of the cell.

The final user does not need to worry about these details. He or she will declare `FiniteElement`s (wrappers) of different types by providing `tag`s to the constructor, as shown in subsequent paragraphs of this section. In some cases, when we only want to compute integrals of certain functions (like in paragraphs 1.1 and 6.2), it is enough to declare an `Integrator`. maniFEM will create, invisibly to the user, a finite element compatible with that integrator.

`FiniteElement`s with master keep as an attribute the map transforming the master element into the current physical cell. This map depends of course on the geometry of the cell and thus it must be computed from scratch each time we begin integrating on a new cell. We say that the `FiniteElement` is docked on a new `Cell`; the method `dock_on` performs this operation. This method is element-specific, each type of finite element having its own class. The docking may happen behind the scenes, like in paragraphs 1.1 and 6.2.

For instance, `FiniteElement`s declared with `tag::quadrangle` will only `dock_on` quadrilaterals (two-dimensional `Cell`s with four sides). When docking on a cell, the `FiniteElement` object will build four "shape functions" and a transformation map. The four shape functions can be accessed through the method `basis_function`, as shown in paragraphs 6.3, 6.4, 6.6 and 6.8. If the geometric dimension is two (that is, if the current cell lies within a two-dimensional Euclidian manifold), the above referred map will be a diffeomorphism between a master element occuppying the square $[-1, 1]^2$ and the cell. The jacobian of this transformation map is also computed, through symbolic differentiation. If the geometric dimension is three (like in paragraphs 1.1 and 6.2), the above referred map is an immersion and the set of available operations is more limited. For instance, we will not be able to differentiate a shape function with respect to a geometric coordinate because the shape functions are only defined in the current (two-dimensional) cell, not in the surrounding three-dimensional Euclidian manifold.

It should be stressed that the approach presented in this section is rather low-level. We are working hard to make maniFEM understand statements describing variational formulations given as `C++` objects. When this part of the code is done, the programming style will become much more elegant and compact.

6.2. Integrating functions

To begin with, let us simply compute integrals of functions defined on a spherical mesh.

parag-6.2.cpp

```
Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
Function xyz = RR3 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xyz[0], y = xyz[1], z = xyz[2];

Manifold sphere_manif = RR3.implicit ( x*x + y*y + z*z == 1. );
Mesh sphere ( tag::frontal, tag::desired_length, 0.1 );

Integrator integr ( tag::Gauss, tag::tri_6 );

std::cout << "integral of 1 = " << integr ( 1., tag::on, sphere ) << std::endl;
std::cout << "integral of x = " << integr ( x, tag::on, sphere ) << std::endl;
std::cout << "integral of x*x = " << integr ( x*x, tag::on, sphere ) << std::endl;
std::cout << "integral of x*y = " << integr ( x*y, tag::on, sphere ) << std::endl;
```

Invisibly to the user, maniFEM builds a `FiniteElement` compatible with the given `Integrator`, triangular Lagrange P1 in this case (recall that frontal meshing builds triangular cells). Paragraph 11.5 describes the lifecycle of a `FiniteElement` – `Integrator` pair.

6.3. Laplace operator, Dirichlet boundary conditions

Let's look at an example about the Laplace operator with non-homogeneous Dirichlet boundary conditions.

parag-6.3.cpp
```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

// declare the type of finite element
FiniteElement fe
   ( tag::with_master, tag::quadrangle, tag::Lagrange, tag::of_degree, 1 );
Integrator integ = fe .set_integrator ( tag::Gauss, tag::quad_4 );

// build a 10x12 mesh on a square domain
Cell A ( tag::vertex );  x(A) = 0.;   y(A) = 0.;
Cell B ( tag::vertex );  x(B) = 1.;   y(B) = 0.;
Cell C ( tag::vertex );  x(C) = 1.;   y(C) = 1.;
Cell D ( tag::vertex );  x(D) = 0.;   y(D) = 1.;

Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 10 );
Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in, 12 );
Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, 10 );
Mesh DA ( tag::segment, D .reverse(), A, tag::divided_in, 12 );

Mesh ABCD ( tag::rectangle, AB, BC, CD, DA );
```

Integrators that act on a quadrangular master cell are obtained with `tag::quad_4` (four points, exact for polynomials of degree 3 in each variable),[2] `tag::quad_9` (nine points, exact for polynomials of degree 5 in each variable).[2]

A finite element can handle by itself the numbering of vertices, as shown in paragraph 6.5. For now, we prefer the conceptually simpler approach of building by hand a `map` for numbering vertices.

parag-6.3.cpp
```
std::map < Cell, size_t > numbering;

Mesh::Iterator it = ABCD .iterator ( tag::over_vertices );
size_t counter = 0;

for ( it .reset() ; it .in_range(); it++ )
{  Cell V = *it;  numbering [V] = counter;  ++counter;  }
```

Paragraph 9.3 describes `Iterators` over cells of a `Mesh`.

The matrix of the linear system and the vector holding the free coefficients are declared as objects of the `Eigen` library.

parag-6.3.cpp
```
size_t size_matrix = ABCD .number_of ( tag::vertices );
assert ( size_matrix == numbering .size() );
Eigen::SparseMatrix < double > matrix_A ( size_matrix, size_matrix );
Eigen::VectorXd vector_b ( size_matrix );  vector_b .setZero();
```

[2] from the book E.B. Becker, G.F. Carey, J.T. Oden, Finite Elements, an introduction, vol. I, Prentice-Hall, 1981

We now run over all rectangular cells of ABCD, dock the finite element on the current cell, compute integrals of the form $\iint \frac{\partial \psi_i}{\partial x_\alpha} \frac{\partial \psi_j}{\partial x_\beta}\, dx$ and add the obtained values to the global matrix:

parag-6.3.cpp
```
// run over all square cells composing ABCD
Mesh::Iterator it = ABCD .iterator ( tag::over_cells, tag::of_max_dim );
for ( it .reset(); it .in_range(); it++ )
{  Cell small_square = *it;
   fe .dock_on ( small_square );

   // run twice over the four vertices of 'small_square'
   Mesh::Iterator it1 = small_square .boundary() .iterator ( tag::over_vertices );
   Mesh::Iterator it2 = small_square .boundary() .iterator ( tag::over_vertices );
   for ( it1 .reset(); it1 .in_range(); it1++ )
   for ( it2 .reset(); it2 .in_range(); it2++ )
   {  Cell V = *it1, W = *it2;
      // V may be the same as W, no problem about that
      Function psiV = fe .basis_function (V),
               psiW = fe .basis_function (W),
               d_psiV_dx = psiV .deriv (x),
               d_psiV_dy = psiV .deriv (y),
               d_psiW_dx = psiW .deriv (x),
               d_psiW_dy = psiW .deriv (y);

      // 'fe' is already docked on 'small_square'
      // so this will be the domain of integration
      matrix_A .coeffRef ( numbering[V], numbering[W] ) +=
         fe .integrate ( d_psiV_dx * d_psiW_dx + d_psiV_dy * d_psiW_dy );    }  }
```

In the above, coeffRef is the method used by Eigen to access elements of a sparse matrix.

We impose Dirichlet boundary conditions $u(x, y) = xy$ (this way, we know beforehand the exact solution will be $u(x, y) = xy$). We use a function impose_value_of_unknown which changes the matrix_A and the vector_b in order to impose the Dirichlet condition u(i) = some_value. See the source code in file examples-manual/parag-6.3.cpp in the distribution tree for the definition of the impose_value_of_unknown function.

parag-6.3.cpp
```
Mesh::Iterator it = BC .iterator ( tag::over_vertices );
for ( it .reset(); it .in_range(); it++ )
{  Cell P = *it;
   size_t i = numbering [P];
   impose_value_of_unknown ( matrix_A, vector_b, i, y(P) );  }
```

We then use Eigen to solve the system of linear equations:

parag-6.3.cpp
```
Eigen::ConjugateGradient < Eigen::SparseMatrix <double>,
                           Eigen::Lower | Eigen::Upper  > cg;
cg .compute ( matrix_A );
Eigen::VectorXd u = cg .solve ( vector_b );
```

And obtain the expected solution, shown in figure 6.1.

We stress again that the examples in this section show a rather rudimentary way of using finite elements in maniFEM. We are working hard to reach a more elegant, compact and high-level style. The user should have the possibility to declare the partial differential equation through a compact declaration of the corresponding variational formulation.

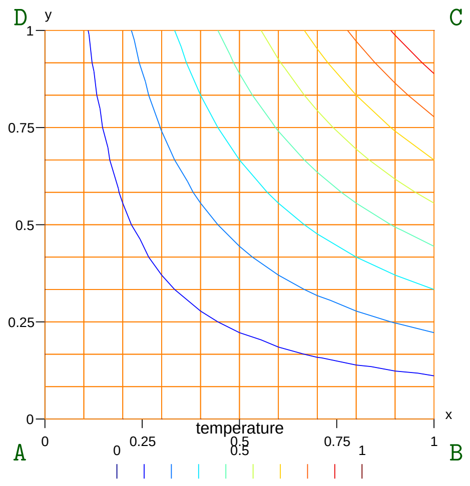


Figure 6.1.: level lines of u

Another limitation of *maniFEM* is that this type of finite element is rather slow. This is so because each docking operation implies a lot of symbolic calculations; the same happens when we differentiate base functions. Even the final operation, of finding the value of the integral by using Gauss quadrature, involves some symbolic calculus because the integrands are kept in the computer's memory as trees of symbolic expressions, so just calculating the value of such a function at a given point is a rather slow operation.

There are three ways of circumventing this limitation. The first one is : if your mesh is made of identical finite elements (like the example in the present paragraph), you can compute the elementary matrix just once and then assembly the global matrix from it. It will surely be much faster. A second way to speed up the code is to add options `-DNDEBUG` and `-O3` to the compilation command in your `Makefile` (see also paragraph 11.13).

The third (and soundest) solution is a deep optimization of the *maniFEM* library. A different type of finite elements is presented in paragraphs 6.11 and 6.12. They perform a few symbolic calculations just once, typically right after they are declared. Only raw arithmetic operations are performed when the user actually requests the values of integrals in order to assembly the matrix.

6.4. Neumann boundary conditions

Implementing zero Neumann boundary conditions does not imply any programming effort (we don't have to do anything). However, for imposing non-zero Neumann boundary conditions we need to compute one-dimensional integrals. We do this by declaring a 1D finite element which we then dock on each segment of the part of the boundary where we need to integrate.

parag-6.4.cpp

```
FiniteElement fe_bdry ( tag::with_master, tag::segment,
                        tag::Lagrange, tag::of_degree, 1 );
Integrator integ_bdry = fe_bdry .set_integrator ( tag::Gauss, tag::seg_3 );

// ... assemble the matrix ...

Function heat_source = y*y;
// impose Neumann boundary conditions du/dn = heat_source :
Mesh::Iterator it = DA .iterator ( tag::over_segments );
for ( it .reset(); it .in_range(); it++ )
```

```
{  Cell seg = *it;
   fe_bdry .dock_on ( seg );
   Cell V = seg .base() .reverse();
   assert ( V .is_positive() );
   size_t i = numbering [V];
   Function psiV = fe_bdry .basis_function (V);
   vector_b [i] += fe_bdry .integrate ( heat_source * psiV );
   Cell W = seg.tip();
   assert ( W .is_positive() );
   size_t j = numbering [W];
   Function psiW = fe_bdry .basis_function(W);
   vector_b [j] += fe_bdry .integrate ( heat_source * psiW );  }
```

One may object that there is no need for a 1D finite element, a 1D integrator should be enough. We agree; the user should have the possibility to attach a 1D integrator to a 2D finite element. This is object of on-going work.

Also, in this paragraph we switch to triangular finite elements (see the source code, file `examples-manual/parag-6.4.cpp` in the distribution tree of maniFEM).

Integrators that act on a triangular master cell are obtained with `tag::tri_3` (three points, exact for polynomials of degree 2),[3] `tag::tri_3_Oden` (three points, exact for polynomials of degree 2),[2] `tag::tri_4` (four points, exact for polynomials of degree 3),[3] `tag::tri_4_Oden` (four points, exact for polynomials of degree 3),[2] `tag::tri_6` (six points, exact for polynomials of degree 4).[3]

6.5. Automatic numberig of vertices

In previous examples we have built a `map` for handling numbering of vertices. In this paragraph we show how to use instead the automatic numbering associated to the finite element itself. The method `FiniteElement::build_global_numbering` creates an object which is similar to `std::map` but uses a different mechanism. It modifies the process of creating new cells (vertices, in this specfic case) by ading a new function to the list of functions `Cell::init_pos_cell[0]`, invoked by `Cell::Core` constructors (see paragraph 11.9). Space will be reserved for a `size_t` value for each new vertex, and this value will be used for numbering vertices.

parag-6.5.cpp

```
FiniteElement fe ( tag::with_master, tag::quadrangle,
                   tag::Lagrange, tag::of_degree, 1  );
Cell::Numbering & numbering = fe .build_global_numbering ( tag::vertices );
```

In this approach, the access to the index of a given vertex is very fast (constant time) because the index is stored within the `Cell::Core`, so no search is needed. In contrast, when using a `map` as shown in paragraph 6.3, a search is necessary through all vertices of the mesh. Due to the implementation of the `map` class in `STL`, this search takes logarithmic time. The only advantage of the approach described in paragraph 6.3 is that the user has full contol of the numbering (see below).

Using the numbering associated to the finite element itself has the following disadvantage. This automatic numbering process does not distinguish between vertices created at the user's request and other vertices used internally by different components of maniFEM; these vertices are

[3] from the book J.F. Flaherty, Finite Element Analysis, Lecture Notes: Spring 2000, Rensselaer Polytechnic Institute, New York

invisible to the user. As a consequence, in some cases the numbering may be not contiguous. Lines of the global matrix corresponding to "missing" vertices are not significant in any way. Columns corresponding to missing vertices are not significant either (because they will only have zero entries). However, the solver of the linear system will complain that the matrix is singular.

The user should check these two numbers : `numbering .size()` and `ABCD .number_of (tag::vertices)`. If they are not equal, measures must be taken. For instance, a non-zero value can be inserted in the diagonal of the global matrix at places corresponding to "missing" vertices. This is equivalent to imposing a Dirichlet boundary condition on those vertices. Since we have no way of running over the missing vertices, we first fill the entire diagonal with ones, then erase them for existing vertices.

parag-6.5.cpp

```
for ( size_t i = 0; i < size_matrix; i++ )
   matrix_A .insert ( i, i ) = 1.;
Mesh::Iterator it = ABCD .iterator ( tag::over_vertices );
for ( it .reset(); it .in_range(); it++ )
{  Cell P = *it;
   size_t i = numbering [P];
   matrix_A .coeffRef ( i, i ) = 0.;  }
```

The rest of the code is identical to paragraph 6.3.

A different way around can be used, which allows the use of a `matrix_A` with the correct dimension (equal to the number of vertices in the mesh `ABCD`). We simply assign the desired value to the numbering of each vertex :

```
Mesh::Iterator it = ABCD .iterator ( tag::over_vertices );
size_t counter = 0;
for ( it .reset(); it .in_range(); it++ )
{  Cell P = *it;
   numbering [P] = counter++;  }
```

This solution keeps the advantage of constant accesss time to the number of a vertex. However, the information from `numbering .size()` will be misleading (it still takes into account the "hidden" vertices). You should use instead `ABCD .number_of ( tag::vertices )`; this is the dimension you should use when declaring your global matrix.

Note that both numbering methods (presented in paragraphs 6.3 and 6.5) allow for retrieving an index of a given vertex but not the other way around. If you need to retrieve a vertex from its index you should probably use a vector of `Cell`s.

6.6. Triangular Lagrange finite elements of degree two

To get a quadratic Lagrange finite element, simply choose degree two; add also a `tag::straight` to inform *maniFEM* that you do not want a curved triangle.

parag-6.6.cpp

```
FiniteElement fe2 ( tag::with_master, tag::triangle,
                    tag::Lagrange, tag::of_degree, 2, tag::straight );
```

This finite element has six basis functions, associated to vertices and to sides. Note that, although the user may imagine the finite element having nodes at the middle of each side, the Cell on which the element is docked is still a regular triangle. This cell has only three vertices and three sides. This is why three of the six basis functions are associated to vertices (zero-dimensional cells) while the other three are associated to sides (one-dimensional cells). See also paragraph 5.1.

parag-6.6.cpp
```
      fe2 .dock_on ( small_tri );
      //  for loop
         Function psi_V = fe2 .basis_function ( V );   // V is a vertex of small_tri
         Function psi_seg = fe2 .basis_function ( seg );  // seg is a side of small_tri
```

The user must provide positive vertices V but oriented (possibly negative) segments seg belonging to small_tri .boundary().

Table below shows the values taken by the six shape functions at relevant points in the triangle.

	A	B	C	midpoint of AB	midpoint of BC	midpoint of CA
$\psi_{\texttt{A}}$	1	0	0	0	0	0
$\psi_{\texttt{B}}$	0	1	0	0	0	0
$\psi_{\texttt{C}}$	0	0	1	0	0	0
$\psi_{\texttt{AB}}$	0	0	0	1	0	0
$\psi_{\texttt{BC}}$	0	0	0	0	1	0
$\psi_{\texttt{CA}}$	0	0	0	0	0	1

It is worth noting that, although we intend to work with degree two finite elements, the coordinate system xy is still declared with tag::of_degree, 1 (see the source code in file parag-6.6.cpp in the distribution tree). This is so because we use straight triangles; if we wanted curved triangles, we should declare xy with tag::of_degree, 2. Paragraph 5.1 gives more details.

6.7. Triangular Lagrange, degree two, incremental basis

We can add a tag::incremental_basis:

parag-6.7.cpp
```
   FiniteElement fe2 ( tag::with_master, tag::triangle, tag::Lagrange,
                       tag::of_degree, 2, tag::straight, tag::incremental_basis );
```

This finite element is almost identical to the one presented in paragraph 6.6. The six basis functions are not the same, but they span the same linear function space.

The set of basis functions of this element contains the three basis functions of the Lagrange P1 element (associated to the three vertices). In addition, there are three basis functions associated to the three sides.

The six basis functions of this finite element take the following values

	A	B	C	midpoint of AB	midpoint of BC	midpoint of CA
$\psi_{\texttt{A}}$	1	0	0	0.5	0	0.5
$\psi_{\texttt{B}}$	0	1	0	0.5	0.5	0
$\psi_{\texttt{C}}$	0	0	1	0	0.5	0.5
$\psi_{\texttt{AB}}$	0	0	0	0.25	0	0
$\psi_{\texttt{BC}}$	0	0	0	0	0.25	0
$\psi_{\texttt{CA}}$	0	0	0	0	0	0.25

The difference relatively to the "usual" finite element described in paragraph 6.6 lies in the interpretation of the coefficients when we write a function u as a linear combination of basis functions. If $u = \sum_{\texttt{V}} u_{\texttt{V}}\psi_{\texttt{V}} + \sum_{\texttt{s}} u_{\texttt{s}}\psi_{\texttt{s}}$ (`v` being vertices, `s` being segments), then $u_{\texttt{V}} = u(\texttt{v})$ but $u_{\texttt{s}}$ is not the value of u at `m`; rather, $u_{\texttt{s}} = 4u(\texttt{m}) - 2(u(\texttt{A}) + u(\texttt{B}))$ (assuming `m` is the middle point of segment `s` = `AB`). This interpretation is used when imposing the Dirichlet boundary condition in the main file `parag-6.7.cpp`.

This finite element is especially useful in the context of quotient manifolds.

6.8. Quadrangular Lagrange finite elements of degree two

To get a bi-quadratic Lagrange finite element, simply choose degree two; add also a `tag::straight` to inform maniFEM that you do not want a curved quadrilateral.

parag-6.8.cpp
```
  FiniteElement fe2 ( tag::with_master, tag::quadrangle,
                      tag::Lagrange, tag::of_degree, 2, tag::straight );
```

This finite element has nine basis functions, associated to vertices, to sides, and to the quadrangular cell itself:

parag-6.8.cpp
```
     fe2 .dock_on ( small_square );
     //  for loop
        Function psi_V = fe2 .basis_function ( V );   // V is a vertex of small_square
        Function psi_seg = fe2 .basis_function ( seg );  // seg is a side of small_square
     Function psi_central = fe2 .basis_function ( small_square );
```

The user must provide positive vertices `V` but oriented (possibly negative) segments `seg` belonging to `small_square .boundary()`.

Table below shows the values taken by the nine shape functions at relevant points in the quadrangle.

	A	B	C	D	mid of AB	mid of BC	mid of CD	mid of DA	center
$\psi_{\texttt{A}}$	1	0	0	0	0	0	0	0	0
$\psi_{\texttt{B}}$	0	1	0	0	0	0	0	0	0
$\psi_{\texttt{C}}$	0	0	1	0	0	0	0	0	0
$\psi_{\texttt{D}}$	0	0	0	1	0	0	0	0	0
$\psi_{\texttt{AB}}$	0	0	0	0	1	0	0	0	0
$\psi_{\texttt{BC}}$	0	0	0	0	0	1	0	0	0
$\psi_{\texttt{CD}}$	0	0	0	0	0	0	1	0	0
$\psi_{\texttt{DA}}$	0	0	0	0	0	0	0	1	0
$\psi_{\texttt{q}}$	0	0	0	0	0	0	0	0	1

It is worth noting that, although we intend to work with degree two finite elements, the coordinate system `xy` is still declared with `tag::of_degree, 1` (see the source code in file `parag-6.8.cpp` in the distribution tree). This is so because we use straight quadrangles; if we wanted curved quadrangles, we should declare `xy` with `tag::of_degree, 2`. Paragraph 5.1 gives more details.

6.9. Quadrangular Lagrange, degree two, incremental basis

We can add a `tag::incremental_basis`:

parag-6.9.cpp
```
FiniteElement fe2 ( tag::with_master, tag::quadrangle, tag::Lagrange,
                    tag::of_degree, 2, tag::straight, tag::incremental_basis );
```

This finite element is almost identical to the one presented in paragraph 6.8. The nine basis functions are not the same, but they span the same linear function space.

The set of basis functions of this element contains the four basis functions of the Lagrange Q1 element (associated to the four vertices). In addition, there are five basis functions associated to the four sides and to the quadrangular cell itself.

The nine basis functions of this finite element take the following values

	A	B	C	D	mid of AB	mid of BC	mid of CD	mid of DA	center
$\psi_{\mathtt{A}}$	1	0	0	0	0.5	0	0	0.5	0.25
$\psi_{\mathtt{B}}$	0	1	0	0	0.5	0.5	0	0	0.25
$\psi_{\mathtt{C}}$	0	0	1	0	0	0.5	0.5	0	0.25
$\psi_{\mathtt{D}}$	0	0	0	1	0	0	0.5	0.5	0.25
$\psi_{\mathtt{AB}}$	0	0	0	0	2	0	0	0	1
$\psi_{\mathtt{BC}}$	0	0	0	0	0	2	0	0	1
$\psi_{\mathtt{CD}}$	0	0	0	0	0	0	2	0	1
$\psi_{\mathtt{DA}}$	0	0	0	0	0	0	0	2	1
$\psi_{\mathtt{q}}$	0	0	0	0	0	0	0	0	1

The difference relatively to the "usual" finite element described in paragraph 6.8 lies in the interpretation of the coefficients when we write a function u as a linear combination of basis functions. If $u = \sum_{\mathtt{V}} u_{\mathtt{V}}\psi_{\mathtt{V}} + \sum_{\mathtt{S}} u_{\mathtt{S}}\psi_{\mathtt{S}} + u_{\mathtt{q}}\psi_{\mathtt{q}}$ (`v` being vertices, `s` being segments, `q` being the quandrangular cell itself), then $u_{\mathtt{V}} = u(\mathtt{v})$. However, $u_{\mathtt{S}}$ is not the value of u at `m`; rather, $u_{\mathtt{S}} = u(\mathtt{m})/2 - (u(\mathtt{A}) + u(\mathtt{B}))/4$ (assuming `m` is the middle point of segment $\mathtt{s} = \mathtt{AB}$). On the other hand, $u_{\mathtt{q}} = u(\mathtt{b}) - \sigma_{\mathtt{S}}/2 + \sigma_{\mathtt{V}}/4$, where `b` is the barycenter of cell `q`, $\sigma_{\mathtt{S}}$ is the sum of the values of u at middle point of sides of `q` and $\sigma_{\mathtt{V}}$ is the sum of the values of u at vertices of cell `q`. This interpretation is used when imposing the Dirichlet boundary condition in the main file `parag-6.9.cpp`.

This finite element is especially useful in the context of quotient manifolds.

6.10. Tetrahedral Lagrange finite elements of degree one

A tetrahedral finite element is declared with `tag::tetrahedron`:

parag-6.10.cpp
```
FiniteElement fe
   ( tag::with_master, tag::tetrahedron, tag::Lagrange, tag::of_degree, 1 );
Integrator integ = fe .set_integrator ( tag::Gauss, tag::tetra_4 );
```

This finite element must be `docked`, unsurprisingly, on tetrahedral cells. It has four basis functions, associated to the four vertices of the tetrahedron.

Integrators that act on a tetrahedral master cell are obtained with `tag::tetra_4` (four points, exact for polynomials of degree 2),[3] `tag::tetra_5` (five points, exact for polynomials of degree

3),[3] `tag::tetra_11` (eleven points, exact for polynomials of degree 4),[3] `tag::tetra_15` (fifteen points, exact for polynomials of degree 5).[3]

Figure 6.2 shows the solution of the Laplace problem in the unit ball, subject to non-homogeneous Dirichlet boundary condition $u = xyz$. The exact solution is, of course, $u = xyz$. We use the mesh built in paragraph 3.8; see also paragraph 4.3.

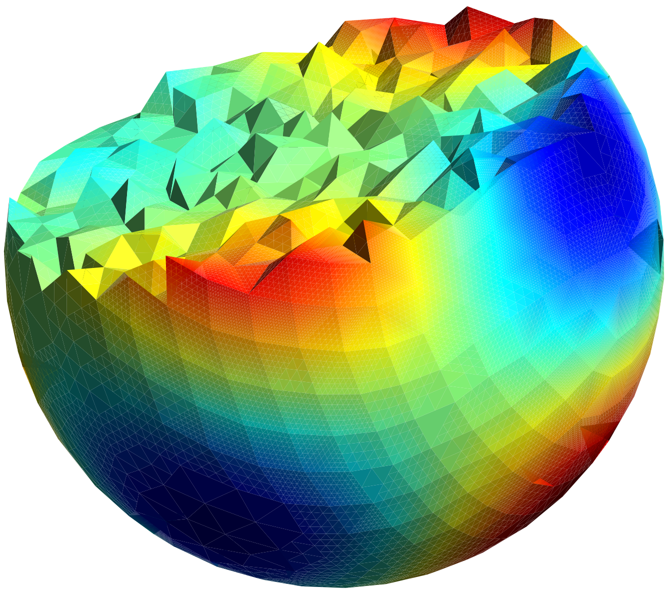


Figure 6.2.: harmonic function in a ball, equal to xyz on the boundary

In order to export such a drawing from `gmsh`, you must go to `Tools` → `Options` → `Mesh` and deselect both `3D element edges` and `3D element faces`. You may also want to disable `Color` → `Use two-side lighting`, then go to `General` → `Color` and change the `Light position`, e.g. by switching the signs of all coordinates. You may also want to go to `Axes` and disable `Show small axes`, and also go to `View [0]` → `Visibility` and disable `Show value scale`. To make the cut, go to `Tools` → `Clipping` → `View [0]` → `Planes`, choose values for parameters A, B, C, D; also, select `Keep whole elements`.

6.11. Hand-coded finite elements

Paragraphs 6.3 – 6.10 show finite element computations where we simply take the shape functions provided by the method `FiniteElement::basis_function`, differentiate them symbolically and then integrate their product through a Gauss quadrature. This process is simple and clear from the user's viewpoint. However, it involves a lot of symbolic computations at each docking operation and aftewards; thus, it is quite slow.

In this paragraph we present another type of finite elements (or rather, integrators). They are much faster at the cost of a slightly less elegant syntax. Also, they are less flexible; finite elements described in previous paragraphs can be used to compute integrals of very general expressions, e.g. `fe .integrate ( sin (x-y) * psi_V.deriv(x) + cos (x-y) * psi_V.deriv(y) )`. Finite elements described in this paragraph only compute integrals of certain expressions (linear or bi-linear with respect to the base functions); paragraph 6.12 gives more details. They do not keep a diffeomorphism to a master element. The quantities relevant for finite element analysis are hard-wired in the source code as arithmetic expressions of the coordinates of the vertices of the cell where the finite element will later dock.

The finite element is declared by simply omitting the tag::with_master; the integrator gets a tag::hand_coded.

parag-6.11.cpp
```
   FiniteElement fe ( tag::rectangle, tag::Lagrange, tag::of_degree, 1 );
   Integrator integ = fe .set_integrator ( tag::hand_coded );
```

Here we use a tag::rectangle instead of a tag::quadrangle; these two tags produce different integrators; see paragraph 6.12.

A hand-coded integrator requires an early declaration of the integrals we intend to compute. One or two abstract basis functions must be declared, then the method Integrator::pre_compute is called with a list of symbolic expressions involving these abstract basis functions and possibly their derivatives. Later, the user instructs the integrator to replace each of these abstract basis function by one of the shape functions provided by the method FiniteElement::basis_function.

parag-6.11.cpp
```
   Function bf1 ( tag::basis_function, tag::within, fe ),
            bf2 ( tag::basis_function, tag::within, fe );
   integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                        tag::integral_of, { bf1 .deriv(x) * bf2 .deriv(x) +
                                            bf1 .deriv(y) * bf2 .deriv(y)   } );
```

The symbolic expressions provided to pre_compute must obey to rather strict rules, described in paragraph 6.12.

The dock_on operation has the same syntax as before :

parag-6.11.cpp
```
      fe .dock_on ( small_square );
```

When we call the method dock_on, the integrator computes all integrals previously required by the user by means of pre_compute. Actually, it sometimes computes other quantities, not required by the user. For instance, if we ask for just one quantity tag::integral_of, { bf1 .deriv(x) }, *maniFEM* will compute integrals of all partial derivatives (in y and, if the geometric dimension is 3, also in z). If we ask for something like tag::integral_of, { bf1 .deriv(x) * bf2 .deriv(x) }, *maniFEM* will compute integrals of products of all derivatives of all pairs of base functions. These values are kept in an internal buffer; later, when we call the method integrate, the computer simply retrieves the corresponding values from the internal buffer.

The integrate method has a syntax which is more complicated and less intuitive than the one we have seen in paragraphs 6.3 – 6.10. We do not provide an expression; this has already been done by invoking method pre_compute. We just specify the tag::pre_computed and provide information about which shape function should replace bf1 and bf2 in the symbolic expressions previously declared through pre_compute. If a list of several expressions has been provided to pre_compute, they will be retrieved all at the same time. Thus, we must store the result in an std::vector rahter than as a double. We then recover each value as components of this vector (in this example there is only one component).

parag-6.11.cpp
```
         Function psi_V = fe .basis_function (V),
                  psi_W = fe .basis_function (W);
         std::vector < double > result = fe .integrate
            ( tag::pre_computed, tag::replace, bf1, tag::by, psi_V,
                                 tag::replace, bf2, tag::by, psi_W );
         assert ( result .size() == 1 );
         matrix_A .coeffRef ( numbering[V], numbering[W] ) += result [0];
```

You can use a hand-coded integrator for performing certain computations, probably within a loop over the cells of your mesh, and then use the same integrator in another loop to perform different computations, involving integrals of other symbolic expressions. To achieve this, you simply call again the method `Integrator::pre_compute` and provide another list of quantities whose integral you will need. You may use the same abstract basis functions, they are mere place-holders which will be replaced by shape functions after the docking of the finite element on a cell. However, for maximum efficiency, if you need to integrate several different quantities, you should compute them simultaneously (in one `pre_compute` call). For instance, code below is correct but inefficient.

```
Function bf1 ( tag::basis_function, tag::within, fe ),
         bf2 ( tag::basis_function, tag::within, fe );

integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 .deriv(x) * bf2 .deriv(x) +
                                         bf1 .deriv(y) * bf2 .deriv(y)   } );
// loop over cells of the mesh, assemble rigidity matrix
Mesh::Iterator it = ABCD .iterator ( tag::over_cells, tag::of_max_dim );
for ( it .reset(); it .in_range(); it++ )
{  Cell small_square = *it;
   fe .dock_on ( small_square );
   Mesh::Iterator it1 = small_square .boundary() .iterator ( tag::over_vertices );
   Mesh::Iterator it2 = small_square .boundary() .iterator ( tag::over_vertices );
   for ( it1 .reset(); it1 .in_range(); it1++ )
   for ( it2 .reset(); it2 .in_range(); it2++ )
   {  Cell V = *it1, W = *it2;
      Function psi_V = fe .basis_function (V), psi_W = fe .basis_function (W);
      std::vector < double > result = fe .integrate
         ( tag::pre_computed, tag::replace, bf1, tag::by, psi_V,
                              tag::replace, bf2, tag::by, psi_W );
      assert ( result .size() == 1 );
      matrix_A .coeffRef ( numbering[V], numbering[W] ) += result [0];      }  }

integ .pre_compute ( tag::for_a_given, tag::basis_function, bf1,
                     tag::integral_of, { bf1 }                    );
// loop over cells of the mesh, assemble vector of loads
for ( it .reset(); it .in_range(); it++ )
{  Cell small_square = *it;
   fe .dock_on ( small_square );
   Mesh::Iterator it1 = small_square .boundary() .iterator ( tag::over_vertices );
   for ( it1 .reset(); it1 .in_range(); it1++ )
   {  Cell V = *it1;
      Function psi_V = fe .basis_function (V);
      std::vector < double > result = fe .integrate
         ( tag::pre_computed, tag::replace, bf1, tag::by, psi_V );
      assert ( result .size() == 1 );
      vector_b ( numbering[V] ) += load * result [0];                        }  }
```

Code below achieves the same result but is (in most cases) more efficient because it only calls `dock_on` once for each cell. The computationally heavy part of the program, that is, the floating point arithmetic operations providing the values of the integrals, happens in `dock_on`. By asking `dock_on` to compute all the integrals simultaneously, some common code is executed only once, thus gaining execution speed.

```
Function bf1 ( tag::basis_function, tag::within, fe ),
         bf2 ( tag::basis_function, tag::within, fe );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1, bf1 .deriv(x) * bf2 .deriv(x) +
                                              bf1 .deriv(y) * bf2 .deriv(y)   } );

// loop over cells of the mesh, assemble both matrix and vector
Mesh::Iterator it = ABCD .iterator ( tag::over_cells, tag::of_max_dim );
for ( it .reset(); it .in_range(); it++ )

{  Cell small_square = *it;
   fe .dock_on ( small_square );
   Mesh::Iterator it1 = small_square .boundary() .iterator ( tag::over_vertices );
   Mesh::Iterator it2 = small_square .boundary() .iterator ( tag::over_vertices );

   for ( it1 .reset(); it1 .in_range(); it1++ )
   for ( it2 .reset(); it2 .in_range(); it2++ )
   {  Cell V = *it1, W = *it2;
      Function psi_V = fe .basis_function (V), psi_W = fe .basis_function (W);
      std::vector < double > result = fe .integrate
         ( tag::pre_computed, tag::replace, bf1, tag::by, psi_V,
                              tag::replace, bf2, tag::by, psi_W );
      assert ( result .size() == 2 );
      matrix_A .coeffRef ( numbering[V], numbering[W] ) += result [1];      }

   for ( it1 .reset(); it1 .in_range(); it1++ )
   {  Cell V = *it1;
      Function psi_V = fe .basis_function (V);
      std::vector < double > result = fe .integrate
         ( tag::pre_computed, tag::replace, bf1, tag::by, psi_V,
                              tag::replace, bf2, tag::by, psi_V );
      // in the above, 'bf2' is irrelevant,
      // we only use 'result[0]' which gives the integral of 'bf1'
      assert ( result .size() == 2 );
      vector_b ( numbering[V] ) += load * result [0];             }           }
```

As an exception to the above rule, let us note that maniFEM is still under construction and thus some integrators are incomplete. If you get an error when calling `pre_compute` with the complete list of integrals that you need, then you should use the first approach with two separate calls to `dock_on`. You may inquire the development status of the integrator by calling the `info` method, described at the end of paragraph 6.12.

6.12. More details on hand-coded integrators

Integrators declared with `tag::hand_coded` are fast at the cost of a tedious internal implementation (relevant quantities are hard-wired in the code as arithmetic expressions involving the coordinates of the vertices of the cell where the finite element will later be docked) and also at the cost of a slightly less elegant syntax.

They work in a three-step process which we describe below.

The first step is `pre_compute`, where the user provides a list of symbolic expressions to be later integrated. These expressions involve one or two abstract basis functions which have no special meaning; they are mere place-holders for shape functions in the finite element and will be replaced by specific shape functions when we invoke the `integrate` method.

The expressions listed in the last argument of `pre_compute` must obey to rather strict rules. They must be either linear expressions in one abstract basis function, possibly differentiated (first order derivatives only), or bi-linear expressions in two abstract basis functions, possibly differentiated (first order derivatives only). In the former case, we only provide one basis function, in the latter we provide two basis functions as arguments. We may mix the two, including linear and bi-linear expressions in the same list; in this case, we must provide two basis functions as arguments; the linear expressions must depend only on the first one.

Below are some example invocations of `pre_compute` with one basis function :

```
Function bf ( tag::basis_function, tag::within, fe );
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
                     tag::integral_of, { bf }                 );
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
                     tag::integral_of, { bf .deriv(x) }        );
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
                     tag::integral_of, { bf .deriv(x), bf .deriv(y) } );
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
                     tag::integral_of, { bf .deriv(y), bf }    );
```

Below are some example invocations of `pre_compute` with two basis functions :

```
Function bf1 ( tag::basis_function, tag::within, fe ),
         bf2 ( tag::basis_function, tag::within, fe );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 * bf2 }                );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 * bf2, bf1 .deriv(x) * bf2 } );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 * bf2, bf1 .deriv(x) * bf2 .deriv(x) } );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 * bf2, bf1 .deriv(x) * bf2 .deriv(y) } );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 .deriv(x) * bf2 .deriv(x),
                                         bf1 * bf2, bf1 * bf2 .deriv(x) } );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 .deriv(x) * bf2 .deriv(x),
                                         bf1 .deriv(x) * bf2 .deriv(y),
                                         bf1 .deriv(y) * bf2 .deriv(x),
                                         bf1 .deriv(y) * bf2 .deriv(y) } );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 .deriv(x) * bf2 .deriv(x) +
                                         bf1 .deriv(y) * bf2 .deriv(y)  } );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1, bf1 * bf2 }           );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1, bf1 .deriv(x), bf1 * bf2,
                                         bf1 .deriv(x) * bf2 .deriv(x) +
                                         bf1 .deriv(y) * bf2 .deriv(y)  } );
```

Below are some examples of invocations of `pre_compute` which maniFEM does not accept, for a variety of reasons :

```
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
   tag::integral_of, { bf + 1. } );  // WRONG, symbolic expression not allowed
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
   tag::integral_of, { 2.* bf } );  // WRONG, symbolic expression not allowed
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
   tag::integral_of, { bf + bf .deriv(x) } );  // WRONG, symbolic expression not allowed
```

```
integ .pre_compute ( tag::for_a_given, tag::basis_function, bf,
   tag::integral_of, { bf .deriv(x) + bf .deriv(y) } );  // WRONG, expr not allowed
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
   tag::integral_of, { bf2 * bf1 } );  // WRONG, bf1 must be first
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
   tag::integral_of, { bf1 - bf2 } );  // WRONG, symbolic expression not allowed
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
   tag::integral_of, { bf2, bf1 * bf2 } );  // WRONG, bf2 cannot appear in a linear expr
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
   tag::integral_of, { bf1 * bf2 * bf2 } );  // WRONG, symbolic expression not allowed
```

ManiFEM is under construction and many types of finite elements and integrators are still missing. At present, hand-coded integrators can be used in association with finite elements listed below

```
FiniteElement fe ( tag::triangle,    tag::Lagrange, tag::of_degree, 1 );
FiniteElement fe ( tag::quadrangle,  tag::Lagrange, tag::of_degree, 1 );
FiniteElement fe ( tag::rectangle,   tag::Lagrange, tag::of_degree, 1 );
FiniteElement fe ( tag::square,      tag::Lagrange, tag::of_degree, 1 );
FiniteElement fe ( tag::tetrahedron, tag::Lagrange, tag::of_degree, 1 );
```

Note that `tag::quadrangle` and `tag::quadrilateral` are synonymous. However, `tag::rectangle` and `tag::square` produce different integrators; they take advantage of the particular geometry of the cell to perform efficient computations; they return of course the same values when docked on a cell of the right geometry. In the future, there will also be a finite element with `tag::parallelogram`.

Some integrators' implementation is still incomplete and they may reject some of the expressions listed above as correct. For instance, in paragraph 6.11 we have used a finite element declared with `tag::rectangle` which accepts the expression `bf1.deriv(x) * bf2.deriv(x) + bf1.deriv(y) * bf2.deriv(y)`. Suppose our mesh was made of irregular quadrangular cells (not rectangular). Then we should use a `tag::quadrangle`. However, the computer would reject this expression because the implementation of this integrator is still incomplete. The only solution would be to use a `tag::quadrangle` and a vector `result` with two components :

```
FiniteElement fe ( tag::quadrangle, tag::Lagrange, tag::of_degree, 1 );
Integrator integ = fe .set_integrator ( tag::hand_coded );

Function bf1 ( tag::basis_function, tag::within, fe ),
         bf2 ( tag::basis_function, tag::within, fe );
integ .pre_compute ( tag::for_given, tag::basis_functions, bf1, bf2,
                     tag::integral_of, { bf1 .deriv(x) * bf2 .deriv(x),
                                         bf1 .deriv(y) * bf2 .deriv(y) } );

// run over all square cells composing ABCD
Mesh::Iterator it = ABCD .iterator ( tag::over_cells_of_dim, 2 );
for ( it .reset(); it .in_range(); it++ )
{  Cell small_square = *it;
   fe .dock_on ( small_square );

   // run twice over the four vertices of 'small_square'
   Mesh::Iterator it1 = small_square .boundary() .iterator ( tag::over_vertices );
   Mesh::Iterator it2 = small_square .boundary() .iterator ( tag::over_vertices );
   for ( it1 .reset(); it1 .in_range(); it1++ )
   for ( it2 .reset(); it2 .in_range(); it2++ )
   {  Cell V = *it1, W = *it2;
```

```
        Function psi_V = fe .basis_function(V),
                 psi_W = fe .basis_function (W);
        std::vector < double > result = fe .integrate
           ( tag::pre_computed, tag::replace, bf1, tag::by, psi_V,
                                tag::replace, bf2, tag::by, psi_W );
        assert ( result .size() == 2 );
        matrix_A .coeffRef ( numbering[V], numbering[W] ) += result[0] + result[1];
  }  }  // end of for loops over vertices, end of for loop over cells
```

This approach implies a slight loss in efficiency: it would probably be computationally cheaper to compute the integral of `bf1.deriv(x) * bf2.deriv(x) + bf1.deriv(y) * bf2.deriv(y)` than to integrate separately `bf1.deriv(x) * bf2.deriv(x)` and `bf2.deriv(x) * bf2.deriv(y)` and then add the two values. Besides, in the code above maniFEM will compute integrals of all products between derivatives of `bf1` and `bf2`, including e.g. `bf1.deriv(x) * bf2.deriv(y)` which we haven't asked for and don't need at all.

A second step in using a hand-coded integrator is the `dock_on` operation. When we call this method, the integrator receives information about the current cell (particularly, the coordinates of the vertices) and computes the values of the expressions (hard-wired in the source code) giving the integrals requested by the user in the `pre_compute` step. It computes these values for all shape functions of that finite element (if `pre_compute` was called with one abstract basis function) or for all pairs of shape functions (if `pre_compute` was called with two abstract basis functions). These values are kept in an internal buffer for later use.

The arithmetic expressions hard-wired in the source code are, in some cases, inspired in the syntax of UFL [4] and the output of FFC.[5] See files `form-*-*-c??.txt` at `https://codeberg.org//cristian.barbarosie/manifem-manual`. In other cases, the expressions are bluntly and tediously computed by hand. Method `FiniteElement::info()` returns a multi-line string with information about the internal implementation of that finite element. You may call this method just after the declaration of the element to get generic information, or after `pre_compute` to get information specific to the expressions listed in the last argument of `pre_compute`. The `info` method is only available in `DEBUG` mode (described in paragraph 11.13).

A third step in using a hand-coded integrator is the invocation of the `integrate` method which simply retrieves (from the internal buffer) values associated with the provided shape functions. The return value is a vector of `doubles` rather that a `double` in order to attend to the case when the user requests more than one expression in the invocation of `pre_compute`.

[4] https://fenics.readthedocs.io/projects/ufl/en/latest/
[5] https://fenics.readthedocs.io/projects/ffc/en/latest/

7. Quotient manifolds

The roots of maniFEM go back to a PhD thesis in 2002 where finite elements on a torus were implemented in `FORTRAN`.[1] The torus is meant as a mere quotient manifold between $\mathbb{R}^2$ and a group of translations of $\mathbb{R}^2$ with two generators; you may think of it as $\mathbb{R}^2/\mathbb{Z}^2$. It should be stressed that this manifold is not the usual "doughnut" built in paragraph 2.21. These two manifolds are homeomorphic (topologically equivalent) but they are not isometric, that is, their geometries differ. The quotient torus is a Riemann manifold with no curvature; it is locally Euclidian (that is, locally isometric to open sets of $\mathbb{R}^2$); we may call it "flat torus". It cannot be embedded in $\mathbb{R}^3$, much less be represented graphically. An unfolded mesh in $\mathbb{R}^2$ can be represented graphically, where vertices and segments from the torus are drawn more than once.

One of the goals of maniFEM is to deal with meshes on quotient manifolds. Different quotient operations can be used, with groups of translations of $\mathbb{R}^2$ or $\mathbb{R}^3$ but also with other groups of transformations. In paragraphs 7.1 and 7.2 we deal with a one-dimensional manifold, equivalent to a circle. In paragraphs 7.3, 7.19 and 7.26 the quotient manifold is isometric to a cylinder. In paragraphs 7.14 and 7.17 we get a manifold homeomorphic (topologically equivalent) to a cylinder if we eliminate a neigbourhood of the origin of the plane. In paragraphs 7.15 and 7.16 the mesh contains the origin which is a singular point, thus the manifold reaches the shape of a cone. In paragraphs 7.4 – 7.10, 7.20, 7.21 and 7.22 the quotient manifold is homeomorphic to a torus. The same holds for paragraph 7.18 if we eliminate a neigbourhood of the origin of the plane. Paragraphs 7.11 – 7.13 deal with three-dimensional examples (volumic meshes). Note that in paragraphs 7.17 and 7.18 the metric is not well-defined, so the manifold we are dealing with is not a Riemannian manifold.

In paragraphs 7.3 – 7.18, a quotient manifold is declared and then a mesh is built directly on this manifold. It is like holding a cylinder or a torus or a cone in your hand and drawing the mesh directly on the cylinder or torus or cone. In paragraphs 7.19 – 7.22, a mesh is built in the usual Euclidian plane and then is folded. It is like drawing the mesh on a plane sheet of paper, then cutting the paper with a pair of scissors along the boundary of the periodicity cell and then folding the resulting piece of paper by glueing opposite faces. In paragraphs 7.23 – 7.25, a boundary is built in the Euclidian plane then is folded around the torus, after which a mesh of triangles is built directly on the torus manifold. Paragraph 7.26 follows a mixed approach; part of the boundary is built in the Euclidian plane then folded around a cylinder while another part of the boundary is built directly on the cylinder manifold.

Paragraphs 7.27 – 7.31 dwell on multi-functions.

[1] see the paper C. Barbarosie, Shape optimization of periodic structures, Computers & Structures 30, 2003

7.1. A circle with no curvature

Here is the closed curve $\mathbb{R}/\mathbb{Z}$. We define it as a segment from `A` to `A`, with a specified `winding` number.

parag-7.1.cpp

```
// begin with the one-dimensional line
Manifold RR ( tag::Euclid, tag::of_dim, 1 );
Function x = RR .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );

// define an action on RR (a translation)
Manifold::Action g ( tag::transforms, x, tag::into, x+1. );
Manifold circle = RR .quotient (g);

// one vertex is enough to start the process
Cell A ( tag::vertex );  x(A) = 0.02;

// with this vertex, we build a segment
Mesh seg ( tag::segment, A .reverse(), A, tag::divided_in, 10, tag::winding, g );
```

We do not bother with the graphical representation of this one-dimensional mesh.

This mesh has 10 segments and 10 vertices. It has no boundary. In contrast, if we mesh a segment in the usual Euclidian space $\mathbb{R}$, we get 10 segments and 11 vertices (and we get a boundary made of two vertices).

Paragraph 11.2 explains the coloring conventions observed in this manual for `C++` code. Paragraph 11.3 gives details about `tags`.

In an effort not to expose too many names in the `namespace maniFEM`, we have chosen to hide the class name `Action` inside the namespace of `class Manifold`. See paragraph 11.1.

7.2. A curved circle

We are now in a position to resume the example in paragraph 2.20, this time in a less cumbersome manner.

parag-7.2.cpp

```
Manifold RR ( tag::Euclid, tag::of_dim, 1 );
Function theta = RR .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );

const double pi = 3.1415926536;
Manifold::Action g ( tag::transforms, theta, tag::into, theta + 2.*pi );
Manifold circle = RR .quotient (g);

Cell A ( tag::vertex );  theta ( A ) = 0.;
Mesh loop ( tag::segment, A.reverse(), A, tag::divided_in, 20, tag::winding, g );

// define new coordinates x and y as arithmetic expressions of theta
Function x = cos (theta), y = sin (theta);

// forget about theta; in future statements, x and y will be used
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
RR2 .set_coordinates ( x && y );
loop .export_to_file ( tag::gmsh, "circle.msh" );
```

In `gmsh`, you must select `Tools` → `Options` → `Mesh` → `1D Elements` in order to see this mesh.

7.3. A cylinder

Here is how to build a (part of a) cylinder in $\mathbb{R}^3$:

parag-7.3.cpp
```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function theta_z =
   RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function theta = theta_z [0], z = theta_z [1];

const double pi = 3.1415926536;
Manifold::Action g ( tag::transforms, theta_z, tag::into, (theta+2*pi) && z );
Manifold cylinder_manif = RR2 .quotient (g);

Cell A ( tag::vertex );  theta (A) = 0.;  z(A) = -1.;
Cell B ( tag::vertex );  theta (B) = 0.;  z(B) =  1.;
Mesh AA ( tag::segment, A .reverse(), A, tag::divided_in, 20, tag::winding,  g );
Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 15 );  // no winding
Mesh BB ( tag::segment, B .reverse(), B, tag::divided_in, 20, tag::winding, -g );

Mesh cylinder ( tag::rectangle, AA, AB, BB, AB .reverse(), tag::winding );
// define new coordinates x and y as arithmetic expressions of theta
Function x = cos (theta), y = sin (theta);

// forget about theta; in future statements, x, y and z will be used
Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
RR3 .set_coordinates ( x && y & z );
cylinder .export_to_file ( tag::gmsh, "cylinder.msh" );
```

The `tag::winding` in the constructor `Mesh cylinder` provides no specific information, it just warns maniFEM that we are on a quotient manifold and that it must take winding numbers into account. Specific information about these winding numbers is included in the three segments `AA`, `AB` and `BB`.

Note the additive syntax used to represent composition of actions. The expression `-g` stands for the inverse of `g`. In subsequent paragraphs, we shall encounter expressions like `g2 - g1` which stands for the composition between `g2` and the inverse of `g1` (we assume the action group is commutative).

Some people may prefer a multiplicative notation like `pow ( g, -1 )` or `g2 / g1`; we have chosen the additive syntax because it allows for expressions like `-5*g2 + 3*g1` which become somewhat cumbersome if we use the multiplicative notation. Besides that, the composition of translations corresponds to the sum of the vectors defining the translations.

7.4. A flat torus

Here is the classical example of $\mathbb{R}^2/\mathbb{Z}^2$.

parag-7.4.cpp
```
// begin with the usual two-dimensional space
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

// define two actions on RR2 (translations)
Manifold::Action g_horiz ( tag::transforms, xy, tag::into, (x+1.) && y ),
                 g_vert  ( tag::transforms, xy, tag::into, x && (y+1.) );
```

```
    // divide the Euclidian plane RR2 by these two actions
    Manifold torus_manif = RR2 .quotient ( g_horiz, g_vert );

    // one vertex is enough to start the process
    Cell A ( tag::vertex );  x(A) = 0.02;  y(A) = 0.02;

    // with this vertex, we build two segments
    Mesh seg_horiz ( tag::segment, A .reverse(), A,
                     tag::divided_in, 10, tag::winding, g_horiz );
    Mesh seg_vert  ( tag::segment, A .reverse(), A,
                     tag::divided_in, 10, tag::winding, g_vert  );
    // and a rectangle
    Mesh torus ( tag::rectangle, seg_horiz, seg_vert,
                 seg_horiz .reverse(), seg_vert .reverse(), tag::winding );
    // we could export 'torus' as msh file, taking advantage of the $Periodic section
    // (this is object of current work)
    // we can directly draw an unfolded mesh :
    torus .export_to_file ( tag::eps, "torus.eps", tag::unfold,
                     tag::over_region, -0.5 < x < 1.5, -0.2 < y < 1.2 );
```

The `tag::winding` in the constructor `Mesh torus` provides no specific information, it just warns *maniFEM* that we are on a quotient manifold and that it must take winding numbers into account. Specific information about the winding numbers is included in the two segments `seg_horiz` and `seg_vert`.

This quotient torus is a Riemann manifold with no curvature; it is locally Euclidian (that is, locally isometric to open sets of $\mathbb{R}^2$); we may call it "flat torus". It cannot be embedded in $\mathbb{R}^3$, much less be represented graphically. An unfolded mesh in $\mathbb{R}^2$ can be represented graphically, as in figure 7.1, where vertices and segments from the torus are drawn more than once.

The mesh `torus` has 100 squares, 200 segments and 100 vertices. It has no boundary, it is closed in itself, it covers entirely $\mathbb{R}^2/\mathbb{Z}^2$ which is a compact manifold with no boundary. In contrast, if we mesh the square $[0,1]^2$ in the usual Euclidian plane we get a mesh with 100 squares, 220 segments and 121 vertices (and of course we get a boundary).

In figure 7.1 we have added a shadow representing the periodicity cell $[0,1]^2$. This gives a hint about the repeated structure of the the unfolded mesh. We see that, for instance, the vertex `A` shows up four times. Each segment originating from `A` is also drawn four times. Other vertices and segments are drawn only twice, or only once, depending on the position of the vertex or segment and on the shape and size of the viewing window.

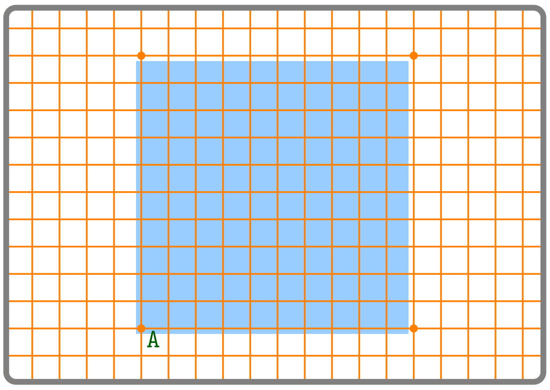


Figure 7.1.: a flat torus

Figure 7.1 is rather dull because the mesh is very regular. In paragraphs 7.6 – 7.9 we introduce an inhomogeneity (a small hole) in the mesh in order to turn the repetitive pattern more visible.

We could export `torus` in `msh` format, by taking advantage of the `$Periodic` section (this is object of current work). Another possibility is to build an unfolded mesh which lives in `RR2` rather than in `torus_manif`, and this unfolded mesh accepts all methods of two-dimensional meshes, including `export_to_file`. Paragraph 7.6 shows an example.

If you feel uncomfortable with this minimalist approach (of defining a square with only one vertex), you may think in terms of four squares instead :

```
Cell A ( tag::vertex );  x(A) = 0. ;  y(A) = 0. ;
Cell B ( tag::vertex );  x(B) = 0.5;  y(B) = 0. ;
Cell C ( tag::vertex );  x(C) = 0.5;  y(C) = 0.5;
Cell D ( tag::vertex );  x(D) = 0. ;  y(D) = 0.5;

Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 5 ),
     BC ( tag::segment, B .reverse(), C, tag::divided_in, 5 ),
     CD ( tag::segment, C .reverse(), D, tag::divided_in, 5 ),
     DA ( tag::segment, D .reverse(), A, tag::divided_in, 5 ),
     BA1 ( tag::segment, B .reverse(), A, tag::divided_in, 5, tag::winding, g_horiz ),
     CD1 ( tag::segment, C .reverse(), D, tag::divided_in, 5, tag::winding, g_horiz ),
     CB2 ( tag::segment, C .reverse(), B, tag::divided_in, 5, tag::winding, g_vert ),
     DA2 ( tag::segment, D .reverse(), A, tag::divided_in, 5, tag::winding, g_vert );

Mesh sq1 ( tag::rectangle, AB, BC, CD, DA ),
     sq2 ( tag::rectangle, BA1, DA .reverse(), CD1 .reverse(),
                           BC .reverse(), tag::winding        ),
     sq3 ( tag::rectangle, DA2 .reverse(), CD .reverse(), CB2,
                           AB .reverse(), tag::winding        ),
     sq4 ( tag::rectangle, CD1, DA2, BA1 .reverse(), CB2 .reverse(), tag::winding );

Mesh torus ( tag::join, sq1, sq2, sq3, sq4 );
```

For those readers who want to understand what's happening behind the curtains, it is worth mentioning that, after the declaration of the `Manifold torus_manif`, the current working manifold is no longer `RR2` (`torus_manif` becomes the working manifold). However, `x` and `y` are the same, coordinates defined on `RR2` (not on `torus_manif`). Coordinates on `torus_manif` are multi-functions for which inequalities like those provided in the last statement (invocation of `draw_ps`) would make no sense.

7.5. A curved torus

We are now in a position to resume the example in paragraph 2.21, this time by using the quotient manifold $\mathbb{R}^2/\mathbb{Z}^2$ introduced in paragraph 7.4.

parag-7.5.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function ab = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function alpha = ab [0], beta = ab [1];

const double pi = 3.1415926536;
Manifold::Action g1 ( tag::transforms, ab, tag::into, (alpha+2.*pi) && beta );
Manifold::Action g2 ( tag::transforms, ab, tag::into, alpha && (beta+2.*pi) );
Manifold torus_manif = RR2 .quotient ( g1, g2 );
```

```
Cell A ( tag::vertex );  alpha (A) = 0.;  beta (A) = 0.;
Mesh seg_horiz ( tag::segment, A .reverse(), A,
                 tag::divided_in, 40, tag::winding, g1 );
Mesh seg_vert  ( tag::segment, A .reverse(), A,
                 tag::divided_in, 20, tag::winding, g2 );
Mesh torus ( tag::rectangle, seg_horiz, seg_vert,
             seg_horiz .reverse(), seg_vert .reverse(), tag::winding );

// parametrize the doughnut
const double big_radius = 3., small_radius = 1.;
// define x, y and z as functions of alpha and beta
Function x = ( big_radius + small_radius*cos(beta) ) * cos(alpha),
         y = ( big_radius + small_radius*cos(beta) ) * sin(alpha),
         z = small_radius*sin(beta);
// forget about alpha and beta :
Manifold RR3 ( tag::Euclid, tag::of_dim, 3 );
RR3 .set_coordinates ( x && y && z );

// in future statements (e.g. for graphical representation)
// x, y and z will be used, not alpha nor beta :
torus .export_to_file ( tag::gmsh, "torus.msh" );
```

7.6. A flat torus with a hole

In this paragraph, we perforate the flat torus. This small inhomogeneity makes the drawing more meaningful. We also change the shape of the viewing window, just for the fun of it.

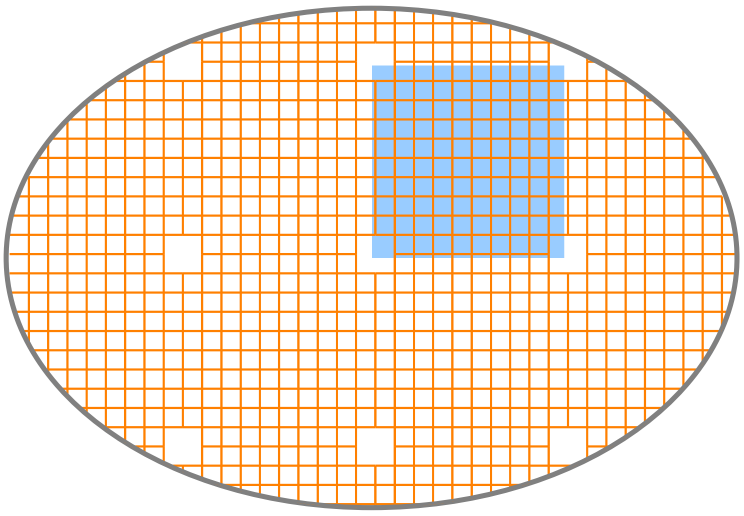


Figure 7.2.: a flat torus with a hole

We stress that this mesh has only one hole; the unfolded mesh, shown in figure 7.2, has several holes.

parag-7.6.cpp

```
std::vector < Cell > vec;
Mesh::Iterator it = torus .iterator
   ( tag::over_cells, tag::of_dim, 2, tag::around, A );
for ( it .reset(); it .in_range(); it++ ) vec .push_back ( *it );
```

```
std::vector < Cell > ::iterator itv;
for ( itv = vec .begin(); itv != vec .end(); itv++ )
{  Cell sq = *itv;  sq .remove_from_mesh ( torus );  }

// we can draw an unfolded mesh :
torus .export_to_file ( tag::eps, "torus.eps", tag::unfold,
                        tag::over_region, x*x + 2.*y*y < 3.5 );

// or we can build a new, unfolded mesh and subsequently export as msh file
Mesh unfolded = torus .unfold ( tag::over_region, x*x + 2.*y*y < 3.5 );
unfolded .export_to_file ( tag::gmsh, "torus.msh" );
```

Paragraph 9.8 describes iterators around a vertex.

As explained in paragraph 10.3, it is not safe to modify a mesh while iterating over its cells. This is why we use a vector of cells (in paragraph 10.3 a list of cells is used, both are OK).

In the spirit of the comment at the end of paragraph 7.4, it is worth explaining that, when we declare the `Manifold torus_manif`, this becomes the current working manifold. Later, when we invoke the `unfold` method, `RR2` becomes again the current working manifold, so the last statement (`export_to_file`) acts on `RR2`.

7.7. A skew flat torus

We can build a skew torus by simply choosing other actions on `RR2`.

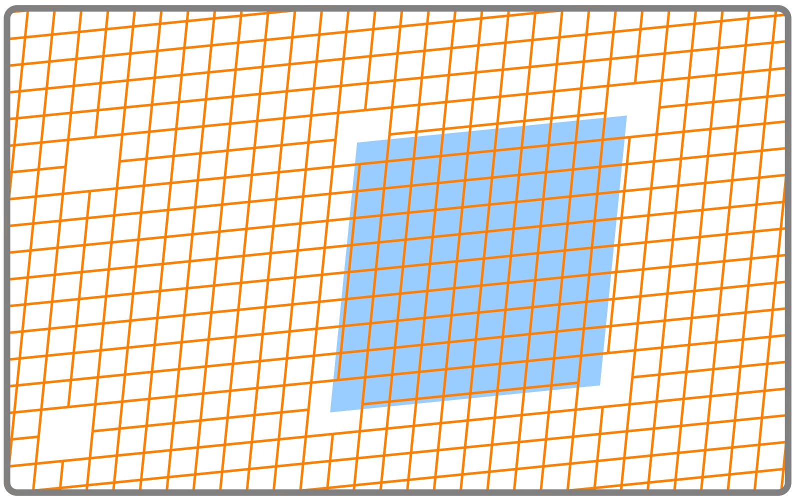


Figure 7.3.: a skew flat torus

parag-7.7.cpp

```
Manifold::Action g1 ( tag::transforms, xy, tag::into, (x+1.) && (y+0.1) );
Manifold::Action g2 ( tag::transforms, xy, tag::into, (x+0.1) && (y+1.) );

Manifold torus_manif = RR2 .quotient ( g1, g2 );
```

Again, we have added a shadow representing the periodicity cell, this time a parallelogram.

7.8. A skew torus, again

Here is a different example of a skew torus, this time meshed with square cells rather than skew parallelograms.

parag-7.8.cpp

```
Manifold::Action g1 ( tag::transforms, xy, tag::into, (x+1.) && y ),
                 g2 ( tag::transforms, xy, tag::into, (x+0.5) && (y+1.) );
Manifold torus_manif = RR2 .quotient ( g1, g2 );
Cell A ( tag::vertex );  x(A) = 0. ;  y(A) = 0.;
Cell B ( tag::vertex );  x(B) = 0.5;  y(B) = 0.;
Mesh AB       ( tag::segment, A .reverse(), B, tag::divided_in, 5 );
Mesh BA_horiz ( tag::segment, B .reverse(), A, tag::divided_in, 5,
                tag::winding, g1                                 );
Mesh BA_vert  ( tag::segment, B .reverse(), A, tag::divided_in, 10,
                tag::winding, g2                                  );
Mesh AB_vert  ( tag::segment, A .reverse(), B, tag::divided_in, 10,
                tag::winding, g2 - g1                             );
Mesh sq1 ( tag::rectangle, AB, BA_vert, BA_horiz .reverse(), AB_vert .reverse(),
           tag::winding                                                         );
Mesh sq2 ( tag::rectangle, BA_vert .reverse(), BA_horiz, AB_vert, AB .reverse(),
           tag::winding                                                         );
Mesh torus ( tag::join, sq1, sq2 );
```

Note the additive syntax for composing actions; comments in paragraph 7.3 apply.

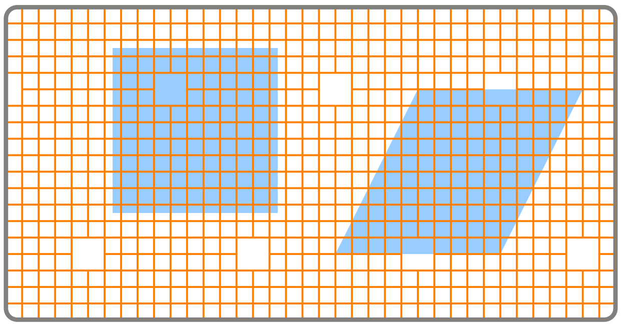


Figure 7.4.: a different torus

In figure 7.4 we have drawn two shadows. The skew parallelogram can be called “periodicity cell”; we call the other one “representative domain”.

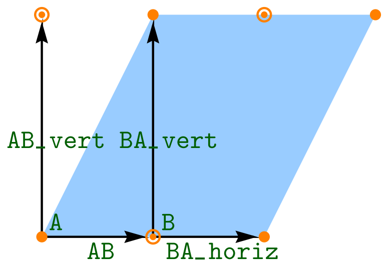


Figure 7.5.: segment meshes used for building the torus

Figure 7.5 shows the four segment meshes `AB`, `BA_horiz`, `AB_vert`, `BA_vert` used for building this torus.

7.9. Using triangular patches

Here is an example of a torus built with two triangular patches.

parag-7.9.cpp
```
Manifold::Action g1 ( tag::transforms, xy, tag::into, (x+1.) && y ),
                 g2 ( tag::transforms, xy, tag::into, (x+0.5) && (y+0.866) );
Manifold torus_manif = RR2 .quotient ( g1, g2 );
Cell A ( tag::vertex );  x(A) = 0.;  y(A) = 0.;
Mesh seg_horiz ( tag::segment, A .reverse(), A,
                 tag::divided_in, 10, tag::winding, g1 ),
     seg1      ( tag::segment, A .reverse(), A,
                 tag::divided_in, 10, tag::winding, g2 ),
     seg2      ( tag::segment, A .reverse(), A,
                 tag::divided_in, 10, tag::winding, g2 - g1 );
Mesh tri1 ( tag::triangle, seg_horiz, seg2, seg1 .reverse(), tag::winding ),
     tri2 ( tag::triangle, seg_horiz .reverse(), seg2 .reverse(), seg1,
            tag::winding                                                );
Mesh torus ( tag::join, tri1, tri2 );
```

Note the additive syntax for composing actions; comments in paragraph 7.3 apply.

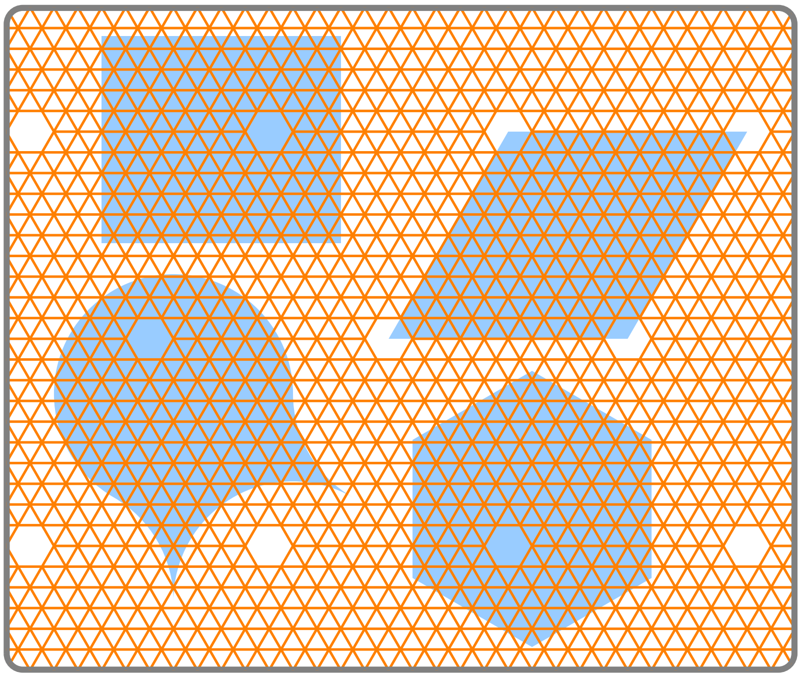


Figure 7.6.: a torus meshed with triangles

In figure 7.6 we have drawn four shadows. The skew paralellogram can be called "periodicity cell"; the other three are "representative domains". Due to the shape of one of these shadows, this periodic arrangement is often called "hexagonal periodicity".

7.10. Exercise

Build the mesh shown in figure 7.6 (the same as in paragraph 7.9) using only one quadrangular patch. (Hint: have a look at paragraphs 2.4 and 2.9.)

7.11. Three dimensions, one periodicity direction

If a two-dimensional flat torus is already difficult to imagine, a three-dimensional one is a real challenge to our capacity of abstraction. Just like a two-dimensional flat torus is no more than a parallelogram whose opposite faces have been identified, a three-dimensional one is a cube whose opposite faces have been identified.

To begin with, let us focus on one pair of opposite faces only. We will obtain a structure repeated in only one direction of the space. This is achieved by defining a periodicity group with one generator only.

parag-7.11.cpp
```
Manifold::Action gx ( tag::transforms, xyz, tag::into, (x+2.) && y && z );
Manifold torus_manif = RR3 .quotient ( gx );
```

We mesh a cube with a spheric hole in the center. We do this by `joining` six 3D meshes (with cubic cells).

For building the six initial vertices on the sphere (corners of the future meshes), we build zero-dimensional `Manifold`s and then use a `Cell` constructor with `tag::project`:

parag-7.11.cpp
```
const double r2 = 0.2;
Manifold sphere = RR3 .implicit ( x*x + y*y + z*z == r2 );

Manifold points_AA_GG = sphere .implicit ( x == y, x == z );
Cell AA ( tag::vertex, tag::of_coords, { -0.5, -0.5, -0.5 }, tag::project );
Cell GG ( tag::vertex, tag::of_coords, {  0.5,  0.5,  0.5 }, tag::project );
```

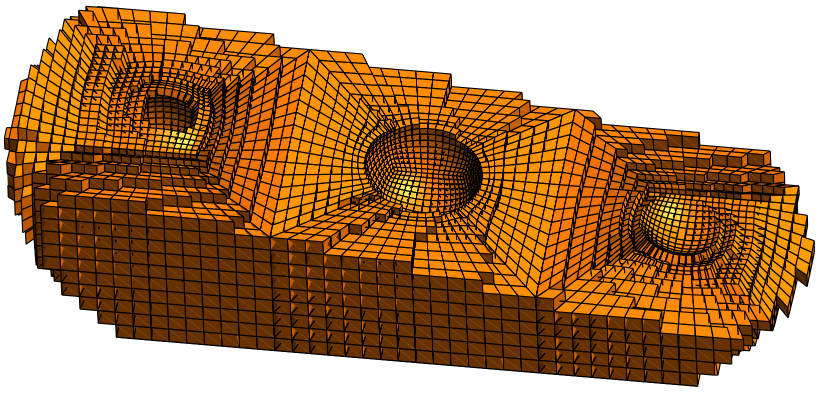


Figure 7.7.: three-dimensional cube repeated in one direction

The resulting mesh cannot be represented graphically; an unfolded mesh is shown in figure 7.7. Note that the unfolded mesh would be unbounded had we not cut it with two inequalities :

parag-7.11.cpp
```
Mesh torus_unfolded = torus .unfold
   ( tag::over_region, x*x + 2.*y*y + 3.*z*z < 10., 5.*z + x < 0.4 );
```

Comments at the end of paragraph 7.13 about the early declaration of the `Manifold torus_manif` apply here, too.

7.12. Three dimensions, two periodicity directions

By defining a periodicity group with two generators, we obtain a structure repeated in the horizontal plane :

parag-7.12.cpp
```
Manifold::Action gx ( tag::transforms, xyz, tag::into, (x+2.) && y && z );
Manifold::Action gy ( tag::transforms, xyz, tag::into, x && (y+2.) && z );
Manifold torus_manif = RR3 .quotient ( gx, gy );
```

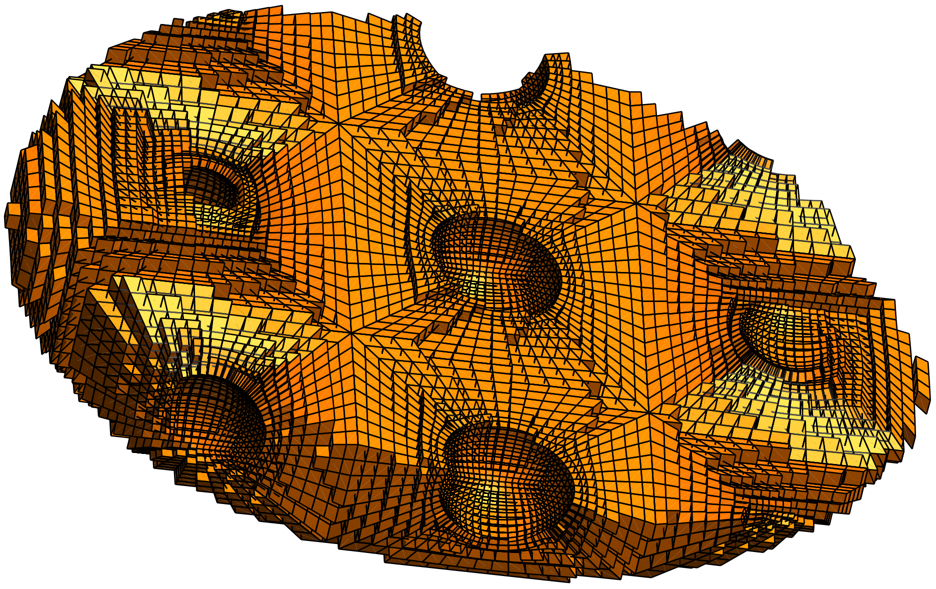


Figure 7.8.: three-dimensional cube repeated in two directions

Comments at the end of paragraph 7.13 about the early declaration of the `Manifold torus_manif` apply here, too.

7.13. Three dimensions, three periodicity directions

Finally, with a periodicity group generated by three translations, we obtain a periodic structure which fills the entire three-dimensional space :

parag-7.12.cpp
```
Manifold::Action gx ( tag::transforms, xyz, tag::into, (x+2.) && y && z );
Manifold::Action gy ( tag::transforms, xyz, tag::into, x && (y+2.) && z );
Manifold::Action gz ( tag::transforms, xyz, tag::into, x && y && (z+2.) );

Manifold torus_manif = RR3 .quotient ( gx, gy, gz );
```

One may ask : why do we declare the `Manifold torus_manif` first, then we build the 2D meshes on the inner sphere (the boundary of the hole) with no winding ? Since these segments have no winding, couldn't we postpone the declaration of `torus_manif` for later, after having meshed the six quadrangles composing the sphere ?

The answer to the above question is negative, but the reason is not easy to explain. Although segments on the sphere (like the ones composing the polygonal line `AABB`) have zero winding

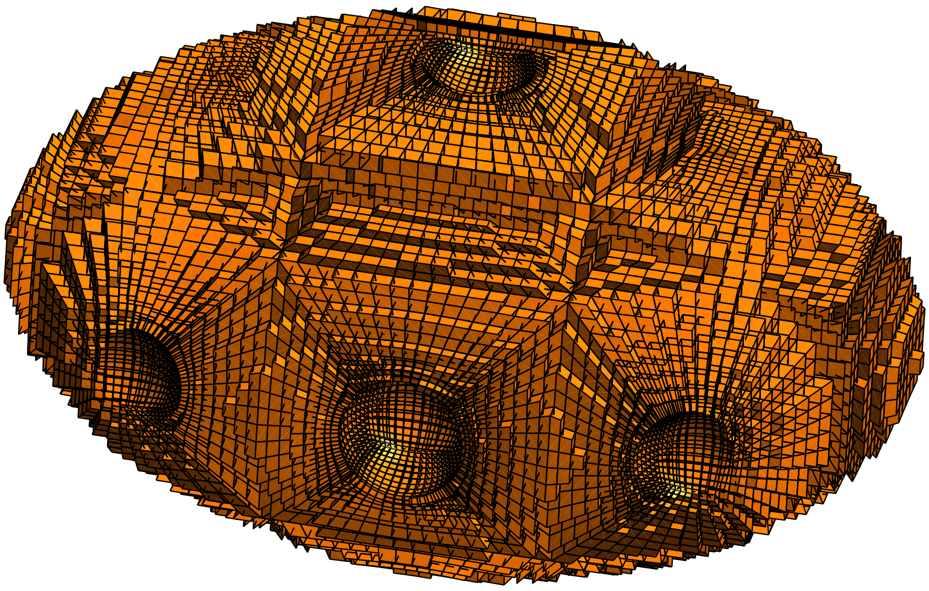


Figure 7.9.: three-dimensional cube repeated in three directions

number, they must nevertheless be treated as lying in the quotient manifold rather than in RR3 or in some submanifold of it like the sphere. This is necessary because Mesh constructors like

```
Mesh cube_ABCD ( tag::cube, ABCD, ABCD_s .reverse(),
                            ABBA .reverse(), CDDC .reverse(),
                            DAAD .reverse(), BCCB .reverse(), tag::winding );
```

treat all faces they receive as arguments as lying in a quotient manifold (last argument tag::winding instructs them so). So, they expect all segments to have a winding number, be it zero or not.

The declaration of a quotient manifold changes the behaviour of Cell constructors. After such a declaration, the constructor of a segment (which is a one-dimensional Cell) will reserve space in the computer memory for a winding number associated to that segment. By "winding number" we mean one short int if the periodicity group has one generator, two short ints if the periodicity group has two generators, three short ints in the present case.

Another legitimate question is : how can we look at equations like x*x + y*y + z*z == r2 (in the declaration of the sphere manifold) if the current working manifold is not RR3 but its quotient torus_manif ? The coordinates on a quotient manifold are multi-functions for which such an equation does not make sense. First, note that we define the sphere manifold by invoking the implicit method of RR3, not of torus_manif. Moreover, the symbols x, y and z do not change their meaning when we declare a new manifold, they are still the functions initially declared as coordinates on the Euclidian manifold RR3. This has already been explained at the end of paragraph 7.4. See also comments at the end of paragraph 7.6.

7.14. A rotation

Actions are not limited to translations. Here is an example of a quotient manifold using one rotation. Note that theta is not a divisor of 2π; more precisely, no multiple of theta is a multiple of 2π. See comments in paragraph 8.2.

It is the user's responsibility to provide an invertible transformation. In contrast, translations (used in previous paragraphs) are always invertible.

parag-7.14.cpp

```
const double theta = 1.;  // radians
const double cos_theta = std::cos ( theta ), sin_theta = std::sin ( theta );

// define an action on RR2 (a rotation)
Manifold::Action rot ( tag::transforms, xy, tag::into,
      ( cos_theta * x - sin_theta * y ) && ( sin_theta * x + cos_theta * y ) );

// and divide RR2 by the group generated by 'rot'
Manifold RR2_q = RR2 .quotient ( rot );

Cell A ( tag::vertex );  x(A) = 0.;  y(A) = 1.;
Cell B ( tag::vertex );  x(B) = 0.;  y(B) = 2.;

Mesh AA ( tag::segment, A.reverse(), A, tag::divided_in, 10, tag::winding, -rot );
Mesh AB ( tag::segment, A.reverse(), B, tag::divided_in, 7 );  // no winding
Mesh BB ( tag::segment, B.reverse(), B, tag::divided_in, 10, tag::winding, rot );

Mesh sector ( tag::rectangle, AA, AB, BB, AB .reverse(), tag::winding );
```

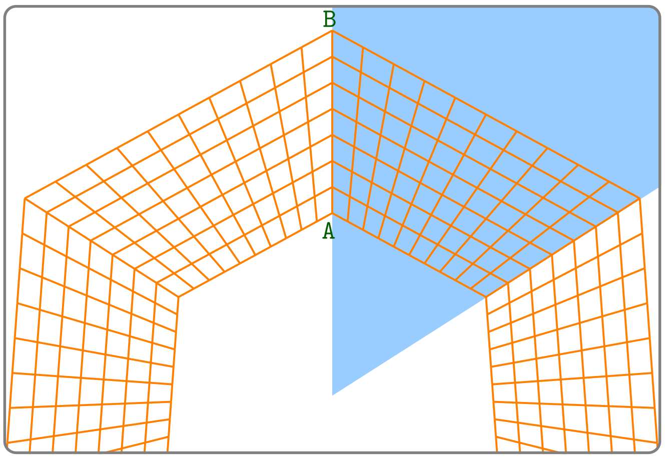


Figure 7.10.: one rotation

In figure 7.10 we have drawn a "representative domain" but this is not entirely correct since rotations of this region with multiples of `theta` close to $2k\pi$ will overlap with the region.

This process is very different from the one described in paragraph 9.2 where many meshes are built in a `for` loop then are joined to form an annulus.

7.15. A singular point

An interesting question is whether we can extend the mesh up to the origin (see figure 7.10). It should be stressed that, when dividing the plane $\mathbb{R}^2$ by a rotation, the origin represents a singularity. The rotation transforms an ordinay point into a different one, but the origin is mapped into itself. This quotient manifold is not locally Euclidian in the neighbourhood of the origin. The shape of this manifold around the origin is similar to the tip of a cone. When building a mesh of triangles having the origin as vertex, we must provide the `tag::singular` and maniFEM will take special care with that vertex.

In this paragraph, we build two triangles and then join them; figure 7.11 shows the unfolded mesh. The tag::singular is necessary when building tri_2 but not for tri_1 because the sides of tri_1 have no winding. In paragraph 7.16, we use only one triangular mesh.

parag-7.15.cpp

```
const double theta = pi/2.;  // radians
const double cos_theta = std::cos ( theta ), sin_theta = std::sin ( theta );

// define an action on RR2 (a rotation)
Manifold::Action rot ( tag::transforms, xy, tag::into,
      ( cos_theta * x - sin_theta * y ) && ( sin_theta * x + cos_theta * y ) );
// and divide RR2 by the group generated by 'rot'
Manifold RR2_q = RR2 .quotient ( rot );

Cell O ( tag::vertex );  x(O) = 0. ;  y(O) = 0. ;
Cell A ( tag::vertex );  x(A) = 1. ;  y(A) = 0. ;
Cell B ( tag::vertex );  x(B) = 1.2;  y(B) = 1.2;

Mesh OA ( tag::segment, O .reverse(), A, tag::divided_in, 10 );  // no winding
Mesh OB ( tag::segment, O .reverse(), B, tag::divided_in, 10 );  // no winding
Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 10 );  // no winding
Mesh BA ( tag::segment, B .reverse(), A, tag::divided_in, 10, tag::winding, rot );

Mesh tri_1 ( tag::triangle, OA, AB, OB .reverse() );  // no winding
Mesh tri_2 ( tag::triangle, OB, BA, OA .reverse(),
             tag::winding, tag::singular, O       );
// on the last triangular cell, the one having O as vertex,
// the sum of windings is not zero

Mesh sector ( tag::join, tri_1, tri_2 );
```

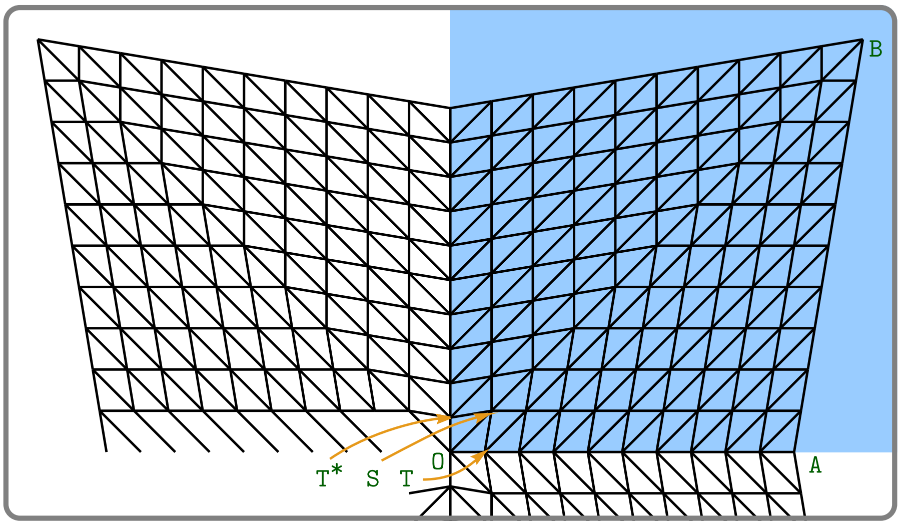


Figure 7.11.: a vertex similar to that of a cone

Usually, the sum of the winding numbers of the three sides of any triangle is zero. The triangle last built in this paragraph, OST^*, does not obey to this rule.

7.16. Avoiding degenerate segments

In paragraph 7.15 we have built two triangular meshes and then we have `joined` them.
It is possible to build a similar shape with only one triangular mesh. However, *maniFEM* cannot build a segment like `TT`*, having the same vertex as base and as tip. This is why a new vertex is introduced near the origin and the two adjacent triangles are split (see figure 7.12).

parag-7.16.cpp
```
const double theta = pi/3.;  // radians
const double cos_theta = std::cos ( theta ), sin_theta = std::sin ( theta );
// define an action on RR2 (a rotation)
Manifold::Action rot ( tag::transforms, xy, tag::into,
      ( cos_theta * x - sin_theta * y ) && ( sin_theta * x + cos_theta * y ) );
// and divide RR2 by the group generated by 'rot'
Manifold RR2_q = RR2 .quotient ( rot );
Cell O ( tag::vertex );  x(O) = 0.;  y(O) = 0.;
Cell A ( tag::vertex );  x(A) = 1.;  y(A) = 0.;
Mesh OA ( tag::segment, O .reverse(), A, tag::divided_in, 10 );  // no winding
Mesh AA ( tag::segment, A .reverse(), A, tag::divided_in, 10, tag::winding, rot );
Mesh sector ( tag::triangle, OA, AA, OA .reverse(),
              tag::winding, tag::singular, O       );
```

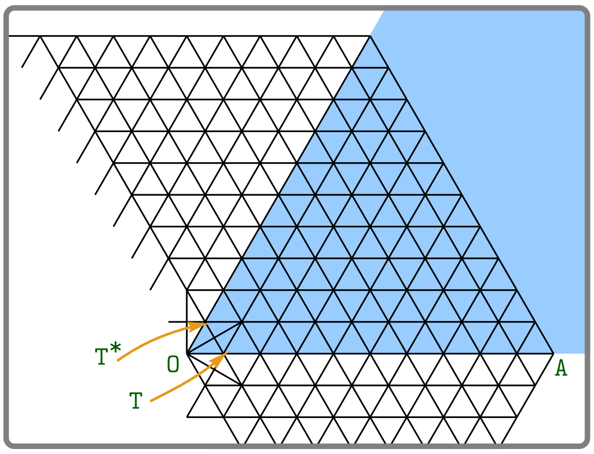


Figure 7.12.: a cone-like shape built with one triangular patch

7.17. A linear transformation

Actions can be arbitrary linear transformations. It is the user's responsibility to provide an invertible transformation. In contrast, translations are always invertible.

parag-7.17.cpp
```
// define an action on RR2 (a linear map)
Manifold::Action g ( tag::transforms, xy, tag::into,
   ( 0.8 * cos_theta * x - 0.8 * sin_theta * y ) &&
   ( 0.8 * sin_theta * x + 0.8 * cos_theta * y )    );

// and divide RR2 by the group generated by g
Manifold RR2_q = RR2 .quotient (g);
```

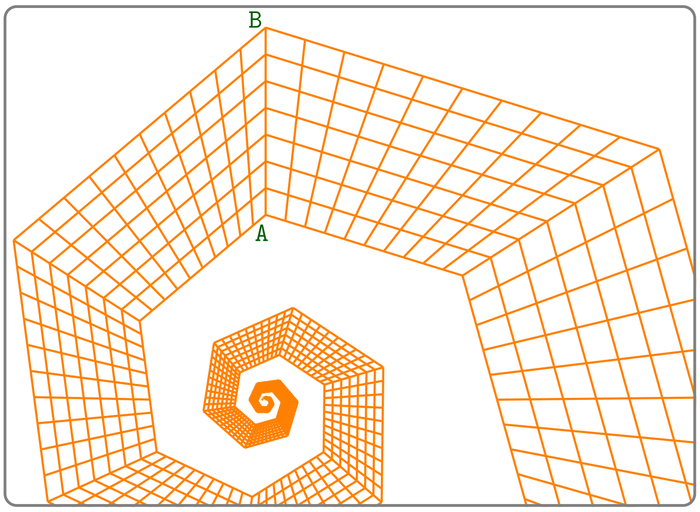


Figure 7.13.: one linear transformation

Note that in this example the metric is not well-defined so the manifold we are dealing with is not Riemannian. See comments in paragraph 8.2.

7.18. Two linear transformations

Here is an example with two linear transformations. It is the user's responsibility to provide invertible transformations and to make sure that they commute. In contrast, translations are always invertible and they commute.

parag-7.18.cpp

```
// define two actions on RR2
// a rotation
Manifold::Action rot ( tag::transforms, xy, tag::into,
   ( cos_theta * x - sin_theta * y ) && ( sin_theta * x + cos_theta * y ) );
// and a zoom
Manifold::Action zoom ( tag::transforms, xy, tag::into, ( 2.*x ) && ( 2.*y ) );

// divide RR2 by the group generated by { rot, zoom }
Manifold torus_manif = RR2 .quotient ( rot, zoom );

// one vertex is enough to start the process
Cell A ( tag::vertex );  x(A) = 0.;  y(A) = 1.;
Mesh AA ( tag::segment, A.reverse(), A, tag::divided_in, 10, tag::winding, -rot );
Mesh AA_vert ( tag::segment, A .reverse(), A,
               tag::divided_in, 7, tag::winding, zoom );

Mesh sector ( tag::rectangle, AA, AA_vert, AA .reverse(), AA_vert .reverse(),
              tag::winding                                                    );
```

Again, the metric is not well-defined (due to the zoom) so the manifold we are dealing with is not Riemannian. See comments in paragraph 8.2.

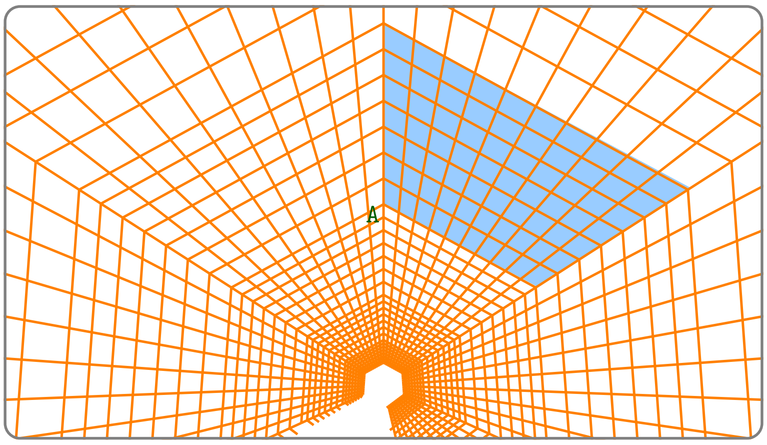


Figure 7.14.: two linear transformations : a rotation and a zoom

7.19. Wrapping a mesh around a cylinder

We can start with a usual mesh in the Euclidian plane and wrap it around a cylinder. Or we may call this operation "wind" or "fold". The outer boundary of the mesh must have two parallel segments of equal length and with the same number of segments (to be identified). The mesh may have many other details (like the circular hole shown in figure 7.15) which do not interfere in the process.

parag-7.19.cpp

```
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

size_t n = 20;
double d = 2.6 / double(n);

Cell A ( tag::vertex );  x(A) = -1.3;  y(A) = -1.3;
Cell B ( tag::vertex );  x(B) =  1.3;  y(B) = -1.3;
Cell C ( tag::vertex );  x(C) =  1.3;  y(C) =  1.3;
Cell D ( tag::vertex );  x(D) = -1.3;  y(D) =  1.3;

Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, n );
Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in, n );
Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, n );
Mesh DA ( tag::segment, D .reverse(), A, tag::divided_in, n );

Manifold circle = RR2 .implicit ( x*x + y*y == 1. );
Mesh inner
   ( tag::frontal, tag::entire_manifold, circle, tag::desired_length, d );

Mesh bdry ( tag::join, AB, BC, CD, DA, inner .reverse() );

RR2.set_as_working_manifold();
Mesh square ( tag::frontal, tag::boundary, bdry, tag::desired_length, d );

Mesh cyl = square .fold
   ( tag::identify, BC, tag::with, DA .reverse(), tag::use_existing_vertices );
```

The method Mesh::fold (which can also be used under the name Mesh::wrap) uses information from the geometry of the mesh to identify an action defining a periodicity group, builds a quotient manifold using this action then builds a new mesh on this quotient manifold by identifying the specified pair of segment meshes. This (nameless) quotient manifold becomes the current working manifold. As explained in paragraph 7.6, the unfold method does the opposite : it sets the Euclidian manifold as working.

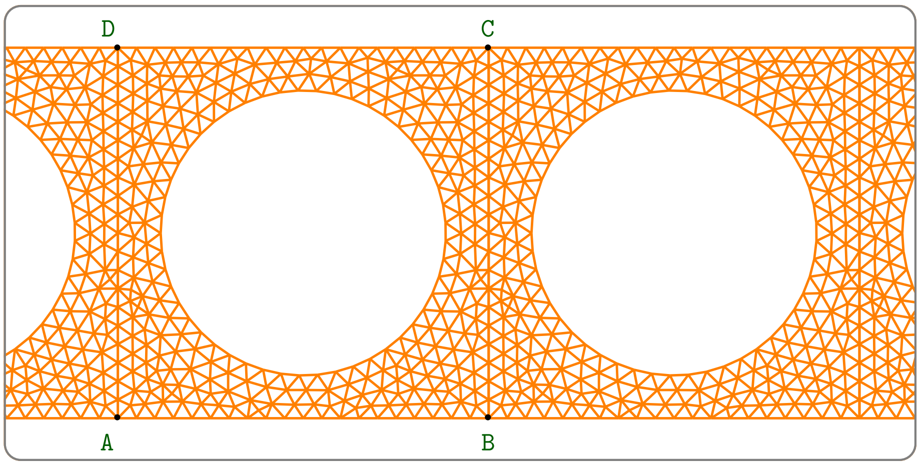


Figure 7.15.: identifying a pair of opposite faces

The mesh cyl is entirely new and distinct from square, with the exception of vertices, which can be shared between the two meshes if the user so desires. The last argument of method Mesh::fold controls this detail. A tag::build_new_vertices says that new vertices must be built and geometric coordinates copied from the old vertices to the new ones. A tag::use_existing_vertices says that old vertices should be used for the new mesh; this can be useful if the vertices carry some information (e.g. values of a solution of a PDE) and also saves space in the computer's memory. Note that vertices belonging to DA will not show up in the new mesh; corresponding vertices from BC will be used instead.

We must identify BC with the reverse of DA (which we may call AD if we wish); trying to identify BC with DA should result in a twisted cylinder (a Möbius strip) but maniFEM will reject such a construction. This event is discussed in paragraph 13.5.

A different approach for meshing the same domain is presented in paragraph 7.26.

7.20. Wrapping a mesh around a torus

A process similar to the one described in paragraph 7.19 can be used to "wrap" a mesh (having a parallelogram as outer boundary) around a torus. It suffices to identify two pairs of opposite faces.

parag-7.20.cpp

```
Mesh torus = square .fold ( tag::identify, AB, tag::with, CD .reverse(),
                            tag::identify, BC, tag::with, DA .reverse(),
                            tag::use_existing_vertices                  );
```

7.21. Identifying three pairs of sides

If we want to obtain the hexagonal periodicity referred in paragraph 7.9, we can literally take a hexagon and identify three pairs of opposite faces. In this paragraph we begin by meshing a hexagonal domain with a star-shaped hole, as shown in figure 7.16.

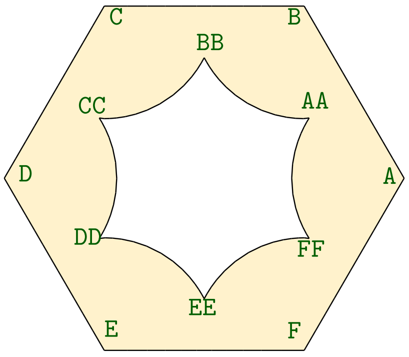


Figure 7.16.: a star-shaped hole

We provide a `Function` as desired length (rather than a constant `double` value) in order to obtain a finer mesh near the re-entrant corners `AA`, `BB` and so on (see also paragraph 3.23).

Then we invoke method `Mesh::fold` and identify opposite faces :

parag-7.21.cpp

```
   Mesh torus = hexa .fold ( tag::identify, AB, tag::with, DE .reverse(),
                             tag::identify, BC, tag::with, EF .reverse(),
                             tag::identify, CD, tag::with, FA .reverse(),
                             tag::use_existing_vertices                  );
```

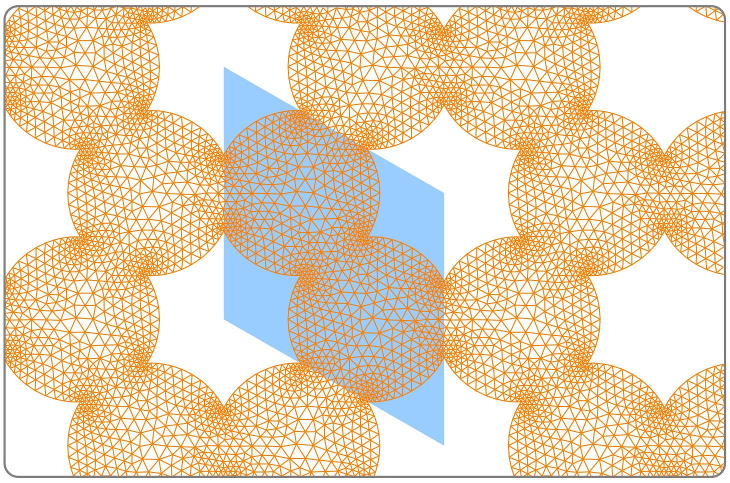


Figure 7.17.: hexagonal arrangement of a star-shaped hole

Figure 7.17 shows an unfolding of the resulting mesh, together with the periodicity cell. We recall that the same periodicity group (i.e., periodic arrangement) may be generated by different sets of generators and, consequently, may be viewed as originating from different periodicity cells.

7.22. Treating a rectangle as a hexagon

Actually, we can even take a rectangle[2] and treat it as if it were a hexagon :

parag-7.22.cpp

```
const double d = 0.13;
const size_t n = 1.5 / d, m = 2.6 / d;

Cell A ( tag::vertex );  x(A) = -1.5;  y(A) = -1.3;
Cell B ( tag::vertex );  x(B) =  0. ;  y(B) = -1.3;
Cell C ( tag::vertex );  x(C) =  1.5;  y(C) = -1.3;
Cell D ( tag::vertex );  x(D) =  1.5;  y(D) =  1.3;
Cell E ( tag::vertex );  x(E) =  0. ;  y(E) =  1.3;
Cell F ( tag::vertex );  x(F) = -1.5;  y(F) =  1.3;

Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, n );
Mesh BC ( tag::segment, B .reverse(), C, tag::divided_in, n );
Mesh CD ( tag::segment, C .reverse(), D, tag::divided_in, m );
Mesh DE ( tag::segment, D .reverse(), E, tag::divided_in, n );
Mesh EF ( tag::segment, E .reverse(), F, tag::divided_in, n );
Mesh FA ( tag::segment, F .reverse(), A, tag::divided_in, m );

Manifold circle = RR2 .implicit ( x*x + y*y == 1.);
Mesh inner ( tag::frontal, tag::entire_manifold, circle,
             tag::desired_length, d                     );
```

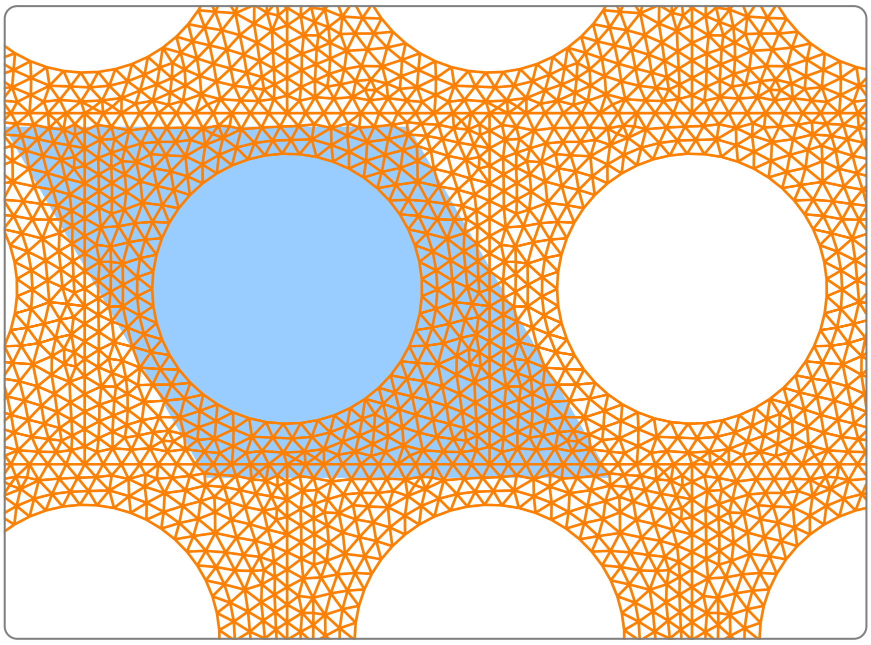


Figure 7.18.: hexagonal arrangement of a circular hole

[2] This rectangle is somewhat similar to the square in paragraph 7.8. It is different in that it has a round hole and different aspect ratio – it has the proportions of the rectangular shadow in figure 7.6.

parag-7.22.cpp
```
Mesh bdry ( tag::join, { AB, BC, CD, DE, EF, FA, inner .reverse() } );

RR2.set_as_working_manifold();
Mesh rectangle ( tag::frontal, tag::boundary, bdry, tag::desired_length, d );

Mesh torus = rectangle .fold ( tag::identify, CD, tag::with, FA .reverse(),
                               tag::identify, BC, tag::with, EF .reverse(),
                               tag::identify, AB, tag::with, DE .reverse(),
                               tag::use_existing_vertices                   );
```

Figure 7.18 shows the resulting mesh.

A different approach for meshing the same domain is presented in paragraph 7.23.

7.23. Frontal meshing on quotient manifolds

The algorithm for frontal meshing (described in section 3) works on quotient manifolds if we understand their topology and orientation. The simplest example would be on a one-dimensional manifold (isomorphic to a circle), but this example is rather dull so we skip it.

In this paragraph, we build a circle in $\mathbb{R}^2$, then fold it onto a flat torus. In mathematical language, we simply project the circle from one manifold to another.

parag-7.23.cpp
```
Manifold circle_manif = RR2 .implicit ( x*x + y*y == 1. );
Mesh circle ( tag::frontal, tag::desired_length, 0.2 );

Manifold::Action gx ( tag::transforms, xy, tag::into, ( x + 3. ) && y );
Manifold::Action gy ( tag::transforms, xy, tag::into, ( x + 1.5 ) && ( y + 2.6 ) );
RR2 .quotient ( gx, gy );

Mesh circle_fold = circle .fold ( tag::use_existing_vertices );
```

Calling the `Mesh::fold` method with only one argument is quite different from what we have seen in paragraphs 7.19 – 7.22. There, the geometry of the manifold (the actions defining the group) were computed from the geometry of the given mesh. Here, it is the other way around: a quotient manifold has already been declared and the folding operation will use the geometry of this torus.

We then use `circle_fold` as a starting point for meshing the flat torus with a circular hole. The algorithm starts from the given boundary and creates new triangles until it meets the mesh on the other side of the torus.

parag-7.23.cpp
```
Mesh perf_torus ( tag::frontal, tag::boundary, circle_fold .reverse(),
                  tag::desired_length, 0.2                           );
```

The resulting mesh is quite similar to the one in paragraph 7.22 (figure 7.18).

7.24. Exercise

What happens if, the previous paragraph, we use `circle_fold` as boundary, rather than `circle_fold .reverse()` ?

Try to answer the question before running the code on the computer.

7.25. A perforated plate

We present here an example which is conceptually similar to the one in paragraph 7.23 but exhibits more complicated geometrical details.[3]

We begin by defining, in $\mathbb{R}^2$, four (arcs of) very elongated ellipses, then join them to build the two V-shaped loops represented in figure 7.19.

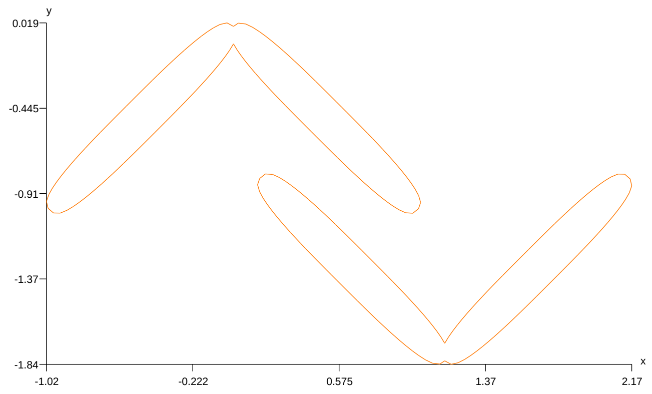


Figure 7.19.: two V-shaped loops

parag-7.25.cpp
```
Mesh two_loops ( tag::join, ellipse_1, ellipse_2, ellipse_3, ellipse_4 );
```

Then, define a torus manifold by means of two appropriate translations.

parag-7.25.cpp
```
Manifold::Action g1 ( tag::transforms, xy, tag::into, (x+2.3) && y ),
                 g2 ( tag::transforms, xy, tag::into, x && (y+1.4) );
Manifold torus_manif = RR2 .quotient ( g1, g2 );
```

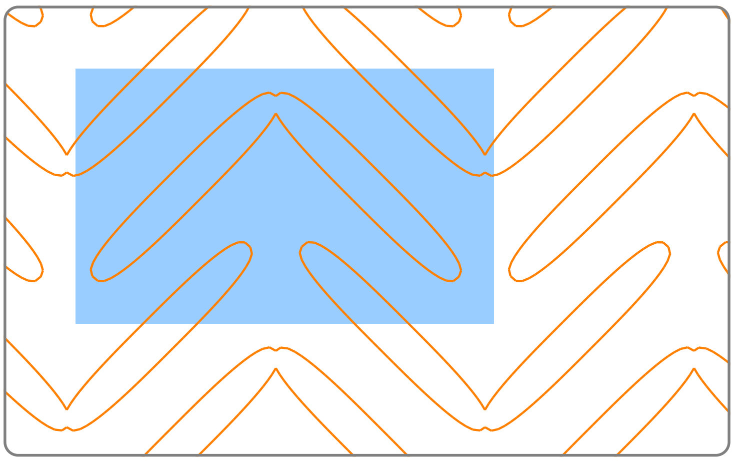


Figure 7.20.: two V-shaped loops, folded then unfolded

Fold the two loops around this torus; the result is shown (after unfolding) in figure 7.20.

[3] This example is inspired in figures 24 and 25 in the paper C. Barbarosie, Shape optimization of periodic structures, Computational Mechanics 30, 2003.

parag-7.25.cpp
```
Mesh folded_loops = loops .fold ( tag::use_existing_vertices );

folded_loops .export_to_file ( tag::eps, "VV.eps", tag::unfold,
                          tag::over_region, -1.5 < x < 2.5, -2. < y < 0.5 );
```

When we declare the `torus_manif`, we must use translations chosen with care, avoiding intersections of these curves with themselves or with each other during the subsequent `folding`. *ManiFEM* does not check for such intersections, it is the user's resposibility to avoid them.

Now call the frontal meshing algorithm. We must reverse the two loops (see paragraph 7.24).

parag-7.25.cpp
```
Mesh perforated ( tag::frontal, tag::boundary, folded_loops .reverse(),
                  tag::desired_length, 0.05                            );
```

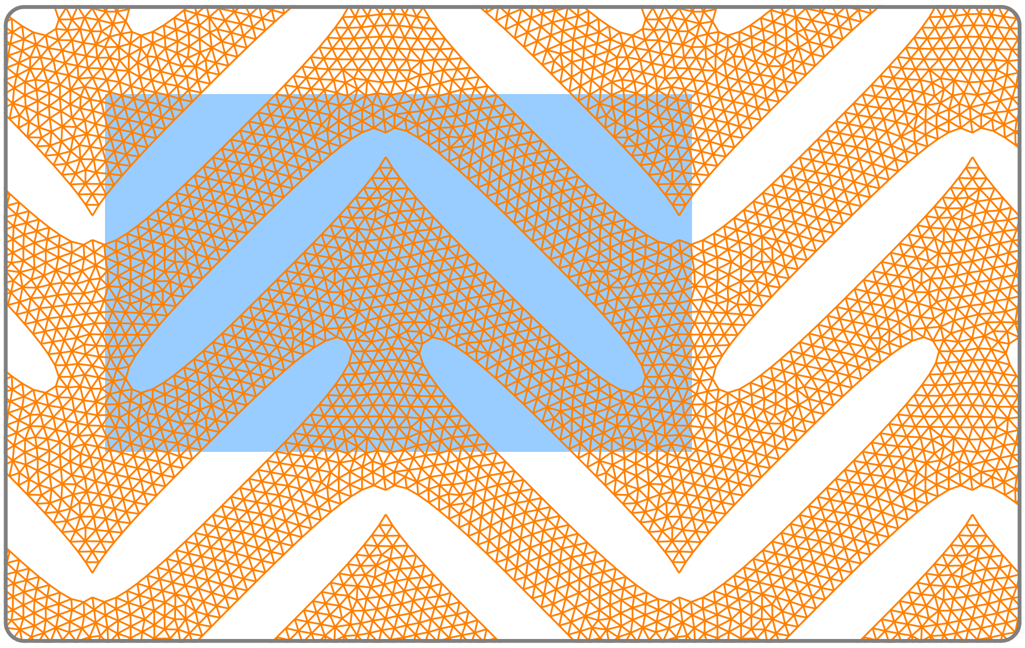


Figure 7.21.: periodically perforated plane, two V-shaped holes

The resulting structure, shown (unfolded) in figure 7.21, represents an example of a perforated material whose effective elastic behaviour exhibits a negative Poisson coefficient.

7.26. Frontal meshing on a cylinder

We now mesh the region of the cylinder from paragraph 7.19 using frontal mesh generation.

We need three curves for defining the region of the cylinder that we want to mesh, a circle and two horizontal “segments” `AB` and `CD`. Actually, these two “segments” are closed loops around the cylinder. There is no need for vertical segments.

We build first the `circle` in $\mathbb{R}^2$ then `fold` it around the cylinder, in the spirit of paragraph 7.23.

parag-7.26.cpp
```
size_t n = 20;
double d = 2.6 / double(n);
Manifold circle_manif = RR2 .implicit ( x*x + y*y == 1. );
Mesh circle ( tag::frontal, tag::desired_length, d );

Manifold::Action gx ( tag::transforms, xy, tag::into, ( x + 2.6 ) && y );
RR2 .quotient ( gx );
Mesh circle_fold = circle .fold ( tag::use_existing_vertices );
```

We then build two horizontal segments `AB` and `CD` as done in paragraph 7.3 :

parag-7.26.cpp
```
Cell A ( tag::vertex );  x(A) = -1.3;  y(A) = -1.3;
Cell C ( tag::vertex );  x(C) =  1.3;  y(C) =  1.3;
Mesh AB ( tag::segment, A .reverse(), A, tag::divided_in, n, tag::winding,  gx );
Mesh CD ( tag::segment, C .reverse(), C, tag::divided_in, n, tag::winding, -gx );
```

Starting from these three curves, we invoke the frontal mesh generation algorithm :

parag-7.26.cpp
```
Mesh bdry ( tag::join, circle_fold .reverse(), AB, CD );

Mesh perf_cyl ( tag::frontal, tag::boundary, bdry, tag::desired_length, d );
```

7.27. Multi-functions, motivation

Even if your mesh does not seem related to any quotient manifold, you may want to use multi-functions.

Consider the mesh `circle` and a pair of functions (u, v) as defined below [4]

parag-7.27.cpp
```
Manifold circle_manif = RR2 .implicit ( x*x + y*y == 1. );
Mesh circle ( tag::frontal, tag::desired_length, 0.2 );

Function r = sqrt ( x*x + y*y );
Function u = sqrt ( (x+r) / 2. / r );
Function v = y / sqrt ( 2. * r * (x+r) );
```

The above definition of (u, v) is equivalent to taking the vector which points from the origin to the current point (x, y), writing it in the trigonometric form and dividing the angle by two. The resulting vector field is shown in figure 7.22.

We see that, after a complete loop around the circle, the vector meets its previous value with a change in sign.

It is easy to imagine the same vector field defined in an annulus instead of a circle (see e.g. paragraph 9.2).

[4] This example is inspired in exercise 2.4 from the book : C. Conca, J. Planchard, M. Vanninathan, Fluids and periodic structures, Wiley, 1995. See also figure 8 in the paper : O. Pantz, K. Trabelsi, A post-treatment of the homogenization method for shape optimization, SIAM Journal on Control and Optimization 47, 2008.

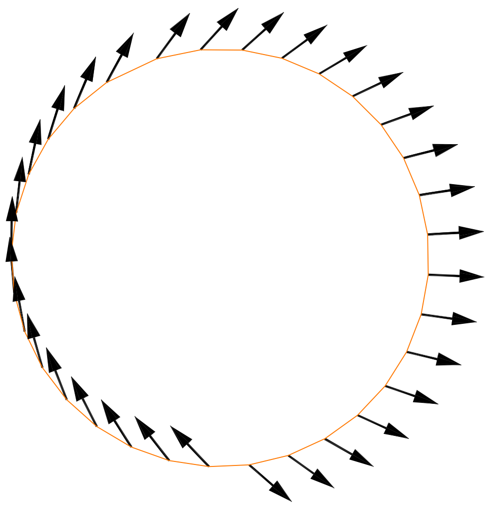


Figure 7.22.: a rotating vector

The above code works fine, but there is a disadvantage. If we want to differentiate this vector field, we will get a very high value of the derivative at the point where the vector changes its sign. Next paragraphs show how multi-functions can help overcome this dilemma.

7.28. Multi-functions

Let us define the circle with the aid of a quotient manifold, just like in paragraph 7.2.

parag-7.28.cpp
```
Manifold RR ( tag::Euclid, tag::of_dim, 1 );
Function theta = RR .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );

const double pi = 3.1415926536;
Manifold::Action g ( tag::transforms, theta, tag::into, theta + 2.*pi );
Manifold circle_manif = RR .quotient (g);

Cell A ( tag::vertex );  theta (A) = 0.;
Mesh circle ( tag::segment, A .reverse(), A, tag::divided_in, 20, tag::winding, g );
```

We then define two functions `u` and `v` by dividing `theta` by `2`:

parag-7.28.cpp
```
Function u = cos ( theta / 2. );
Function v = sin ( theta / 2. );
```

and build multi-functions by specifying that, after a complete loop `g`, they change sign:

parag-7.28.cpp
```
Function u_mv = u .make_multivalued ( tag::through, g, tag::becomes, -u );
Function v_mv = v .make_multivalued ( tag::through, g, tag::becomes, -v );
```

We compute the derivatives of `u` and `v` by finite differences. When we do not take windings into accout, we observe (as expected) a concentration at the point where the vector changes sign. With multi-functions, the computed derivative has constant magnitude.

7.29. Eigenvectors as multi-functions

Perhaps more interesting than defining u and v through half of θ is to compute directly the pair (u, v) as eigenvector of the matrix [5]

$$A = \begin{pmatrix} x & y \\ y & -x \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & -\cos\theta \end{pmatrix}$$

This time, the `Function`s `u` and `v` will not be arithmetic expressions involving `x` and `y` (like in paragraph 7.27) or involving `theta` (like in paragraph 7.28). They will be `Function`s holding `double` values at each vertex :

parag-7.29.cpp
```
Function uv ( tag::lives_on, tag::vertices, tag::has_size, 2 );
Function u = uv[0], v = uv[1];
```

The declaration of `uv` must occur before any vertex is built. This declaration changes the behaviour of constructors of zero-dimensional positive `Cell`s; from now on, space for two more `double` values will be reserved for each vertex, besides the space for the coordinate `theta`. Paragraph 5.1 gives more details.

For a given 2×2 matrix, choosing an eigenvector implies two arbitrary choices : one between the eigenvectors, a second one between the signs (any eigenvector is subject to an arbitrary change in sign). At each vertex, we choose among these four candidates the one which is closest to the eigenvector chosen for the previous vertex. Without taking `windings` into account, this procedure does not work because, at the last segment, the signs are not compatible.

With multi-functions, everything gets into place.

7.30. Swapping eigenvectors

Another interesting example is the matrix

$$A = \begin{pmatrix} 1+x & y \\ y & -1-x \end{pmatrix} = \begin{pmatrix} 1+\cos\theta & \sin\theta \\ \sin\theta & -1-\cos\theta \end{pmatrix}$$

The eigenvectors of this matrix rotate more slowly than the ones in paragraph 7.29 (see figure 7.23). This matrix is null at $(x, y) = (-1, 0)$, so it has a double eigenvalue; this makes it possible for the eigenvectors to swap places.

The behaviour of the vector field after a complete loop is now more complicated and cannot be described separately for `u` and for `v`; we must prescribe it for the pair `uv` :

parag-7.30.cpp
```
Function uv_mv = uv .make_multivalued ( tag::through, g, tag::becomes, (-v) && u );
```

7.31. Jump given by a rotation

In paragraphs 7.28 and 7.29, the jump of the vector field after a complete loop was a change in sign, which we may view as a 180° rotation. In paragraph 7.30 there is a 90° rotation. We can choose any other angle. By doing this, the vector field can no longer be viewed as the eigenvector of a matrix.

[5] taken from exercise 2.4 in the book : C. Conca, J. Planchard, M. Vanninathan, Fluids and periodic structures, Wiley, 1995

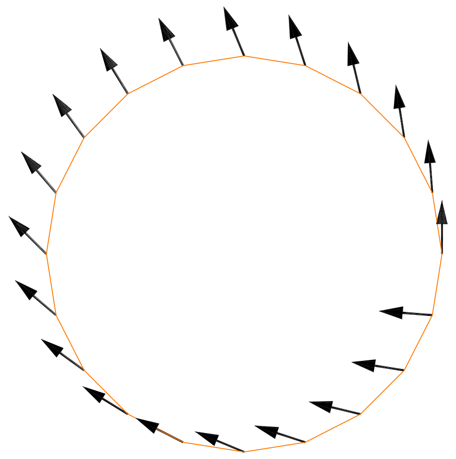


Figure 7.23.: a rotating eigenvector

parag-7.31.cpp
```
const double coef = 0.3333;
Function u = cos ( coef * theta );
Function v = sin ( coef * theta );
Function uv = u && v;
```

Just like in paragraph 7.30, we must prescribe the behaviour of the vector field for the pair uv, not for its components:

parag-7.31.cpp
```
const double c = std::cos ( coef * 2.* pi ), s = std::sin ( coef * 2.* pi );
Function uv_mv = uv .make_multivalued
   ( tag::through, g, tag::becomes, ( c*u - s*v ) && ( s*u + c*v ) );
```

Figure 7.24 shows the resulting vector field for a coefficient of one-third.

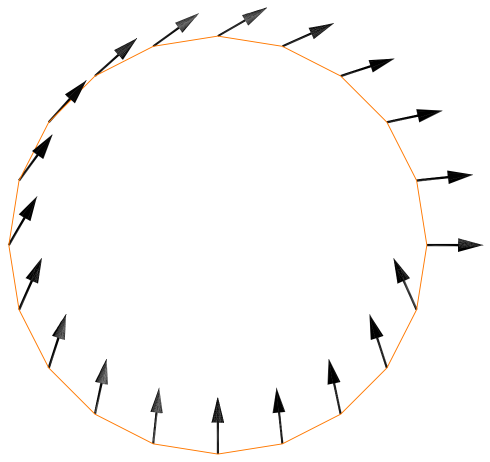


Figure 7.24.: a rotating vector

8. Miscellanea

This section gives informations which do not fit in other sections of the manual.

8.1. Projecting a vertex on a manifold

We recall that, in this manual, we use the term "manifold" to refer to a manifold without boundary. Of course we will only mesh a bounded domain of the manifold, so most meshes will have a boundary. We may also mesh an entire compact manifold like a circle or a sphere or a torus; such a mesh will have no boundary.

In many examples we consider a submanifold of $\mathbb{R}^n$ defined by an implicit equation (or by several implicit equations). When meshing (a bounded domain of) such a manifold, we often need to project points in the surrounding space onto our manifold. This projection operation is achieved by solving the equation (or system of equations) defining the submanifold.

If the manifold is defined by one or two implicit equations, these equations are solved numerically by using a Newton-like algorithm.[1] For two equations, the inverse of the Jacobian matrix is computed by simply applying the definition (quotients of 2x2 determinants). If the manifold is defined by three implicit equations (often, this will be a zero-dimensional manifold), we prefer not to compute the inverse of a 3x3 matrix, so we apply a conjugate gradient method instead. These algoritms converge only if the initial guess (that is, the coordinates of the point we are trying to project) is close enough to the manifold. The user should take this limitation into account.

The case of four or more implicit equations is not implemented.

Many `Mesh` constructors project vertices on the current working manifold `Manifold::working` wihtout the user's assistance. On the other hand, vertices can be explicitly `projected` if necessary, either directly within the `Cell` constructor or by invoking the method `Mesh::project`. Paragraphs 2.5 – 2.18, 3.2 – 3.9 show such examples.

8.2. Quotient manifolds

Section 7 is devoted to meshes on quotient manifolds.

Usually, a quotient manifold is defined by means of orbits of the "periodicity" group. The manifolds considered in paragraphs 7.14 – 7.18 cannot be defined this way if their angle is not a divisor of 2π. This can be easily circumvented; it suffices to consider only a neighbourhood of a representative domain instead of the entire $\mathbb{R}^2$. In the case of a singular point (paragraphs 7.15 and 7.16) the above refered neighbourhood should not contain the singularity.

When we divide a Riemannian manifold (in particular, the Euclidian space $\mathbb{R}^n$) by a group of isometries, the resulting (quotient) manifold inherits naturally a Riemann metric. This is not true for the manifolds considered in paragraphs 7.17 and 7.18 – the actions are not isometries. In other words, we identify a pair of sides (of the representative domain) having

[1] see e.g. appendix B in the paper: C. Barbarosie, A.M. Toader, S. Lopes, A gradient-type algorithm for constrained optimization with application to microstructure optimization, Discrete and Continuous Dynamical Systems series B, 25, p. 1729-1755, 2020

different lengths, so the quotient operation is not compatible with the metric in $\mathbb{R}^2$. Thus, these quotient manifolds do not inherit a Riemann metric. They are valid as differentiable manifolds but not as Riemann manifolds. Length and area are not well-defined in these examples. Since they are compact, these manifolds can be viewed as uniform spaces.[2]

[2] see e.g. corollary 30 in chapter 6 of J.L. Kelley, General Topology, Van Nostrad, 1955

Part II.

Advanced usage

Sections 9 and 10 show how manifEM can be used for controlling the details of the mesh. We hope it will be particularly useful for people interested in implemented their own meshing or remeshing algorithms.

9. Cells, meshes, iterators

This section gives details about cells and meshes.

9.1. Building cells and meshes

As we have already seen in examples in sections 2 and 3, cells and meshes are created by declaring them as `Cell` or `Mesh` objects and by providing specific options to their constructor, by means of `tag`s. For instance :

```
Cell SW ( tag::vertex ); // and the same for vertices SE, NE, NW
Mesh south ( tag::segment, SW .reverse(), SE, tag::divided_in, 10 );
// similar declarations of east, north, west
Mesh rectangle ( tag::rectangle, south, east, north, west );
```

Paragraph 11.2 explains the coloring conventions observed in this manual for `C++` code. Paragraph 11.3 gives more details about `tag`s.

Internally, maniFEM implements cells and meshes as persistent objects, built using the `new` operator and thus having no syntactic scope. Objects belonging to classes `Cell` and `Mesh` are just a wrapper around a persistent core (cell or mesh). When they go out of scope, the wrappers are destroyed; the core may remain alive or not, depending on its relationships with other cells and meshes. This is explained in paragraph 11.5.

Cells and meshes are unique objects, it makes no sense to copy them. A statement like `Cell copy_of_A = A` will make a copy of the wrapper but it will refer to the same cell `A`. If you change e.g. a coordinate of `copy_of_A`, the coordinate of `A` will also change. That is, wrapper classes `Cell` and `Mesh` can be viewed as customized pointers. This is useful if we need to create many meshes in a loop, as shown in paragraph 9.2. However, there are operations which do create a new cell or mesh. There are also operations which create a new cell or mesh only if necessary, otherwise they will return an existing cell or mesh. Paragraph 9.9 gives a complete list.

Recall that, in maniFEM, cells and meshes are oriented. When a cell is declared, it is built as positive and has no reverse. Its reverse is a negative cell and will be built only if necessary. Cells have a method `reverse` which, invoked without arguments, does the following. It checks if the reverse object has already been built; if yes, it returns that object; otherwise, it builds the reverse cell on-the-fly and returns it. Paragraph 9.5 gives a more detailed explanation about orientation of cells and meshes.

`Mesh`es have also a `reverse` method. Note that reverse meshes exist always (negative meshes are temporary objects built on-the-fly).

At a basic level, the only situation when you need the `reverse` method for cells is when you declare a segment `Mesh` (you must provide a negative `Cell` as starting point). You will occasionaly need to use the `reverse` method for meshes (for instance, if you intend to `join` two meshes, their common boundary must have a certain orientation when seen from a mesh and the opposite orientation when seen from the other mesh). In the example in paragraph 1.4, reverses of meshes `CD` and `BC` are used.

9.2. A ring-shaped mesh

For creating many meshes within a cycle, we can view a `Cell` object as a (customized) pointer to a persistent core cell, and the same for `Mesh`es. They are cheap to store and to copy. We call these customized pointers “wrappers”; their behaviour is described in some detail in paragraph 11.4.

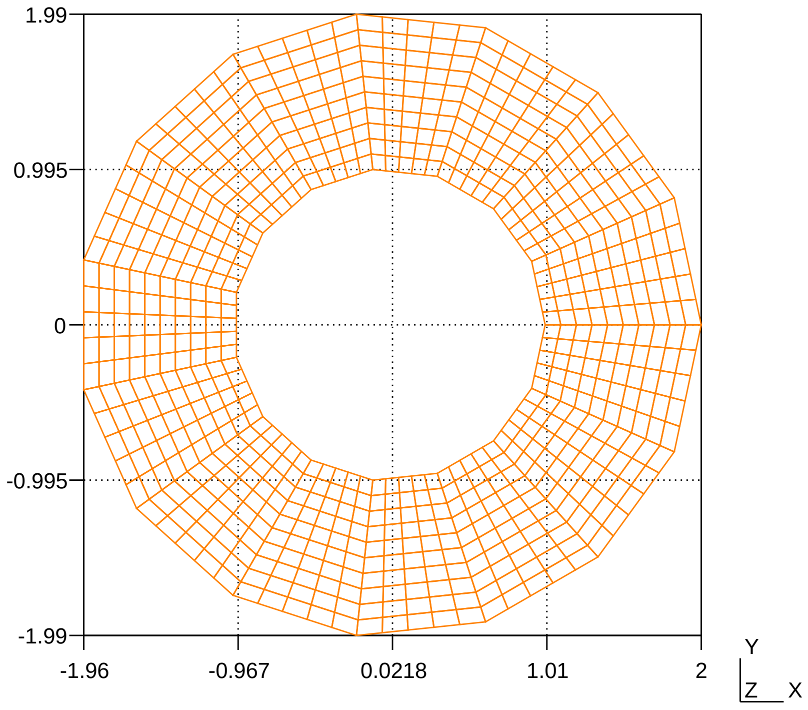


Figure 9.1.: a ring-shaped mesh

parag-9.2.cpp
```
FManifold RR2 ( tag::Euclid, tag::of_dim, 2 );
Function xy = RR2 .build_coordinate_system ( tag::Lagrange, tag::of_degree, 1 );
Function x = xy[0], y = xy[1];

short int n_sectors = 15;
double step_theta = 8 * atan(1.) / n_sectors;
short int radial_divisions = 10;
short int rot_divisions = 5;

// start the process by building a segment
Cell ini_A ( tag::vertex );  x ( ini_A ) = 1.;  y ( ini_A ) = 0.;
Cell ini_B ( tag::vertex );  x ( ini_B ) = 2.;  y ( ini_B ) = 0.;
Mesh ini_seg ( tag::segment, ini_A .reverse(), ini_B,
               tag::divided_in, radial_divisions     );
Mesh prev_seg = ini_seg;
Cell A = ini_A,  B = ini_B;
std::vector < Mesh > sectors;
```

```
  for ( short int i = 1; i < n_sectors; i++ )
  {  double theta = i * step_theta;
     // we build two new points
     Cell C ( tag::vertex );
     x(C) =    std::cos(theta);  y(C) =    std::sin(theta);
     Cell D ( tag::vertex );
     x(D) = 2.*std::cos(theta);  y(D) = 2.*std::sin(theta);
     // and three new segments
     Mesh BD ( tag::segment, B .reverse(), D, tag::divided_in, rot_divisions );
     Mesh DC ( tag::segment, D .reverse(), C, tag::divided_in, radial_divisions );
     Mesh CA ( tag::segment, C .reverse(), A, tag::divided_in, rot_divisions );
     Mesh quadr ( tag::quadrangle, prev_seg, BD, DC, CA );
     sectors .push_back ( quadr );
     prev_seg = DC .reverse();
     A = C;  B = D;                                                    }

  // we now build the last sector, thus closing the ring
  // prev_seg, A and B have rotated during the construction process
  // but ini_seg, ini_A and ini_B are the same, initial, ones
  Mesh outer ( tag::segment, B .reverse(), ini_B, tag::divided_in, rot_divisions );
  Mesh inner ( tag::segment, ini_A .reverse(), A, tag::divided_in, rot_divisions );
  Mesh quadr ( tag::quadrangle, outer, ini_seg .reverse(), inner, prev_seg );
  sectors.push_back ( quadr );

  Mesh ring ( tag::join, sectors );
  ring .export ( tag::gmsh, "ring.msh" );
```

Note how we use a version of the `Mesh` constructor with `tag::join` taking as argument a vector of `Mesh`es; we have already seen it in paragraphs 2.8 and 2.15.

We might have set curved boundaries by using a submanifold of $\mathbb{R}^2$, like in paragraph 2.10. See also paragraph 11.4.

9.3. Iterators over cells

As explained in paragraph 1.2, a `Mesh` is roughly a list of cells. Internally, *maniFEM* keeps lists of cells of each dimension, up the the maximum dimension which is the dimension of the mesh. Thus, if `msh` is a `Mesh` object, modelling a mesh of triangles, then `msh.core->cells[0]` is a list of (wrappers) of each vertex of that mesh, `msh.core->cells[1]` is a list of (wrappers) of each segment and `msh.core->cells[2]` is a list of (wrappers) of each triangle.

Thus, if we want to iterate over all segments of a mesh, we could use an `std::list<Cell>::` `::iterator`. If you are not familiar with the notion of iterator over a list (or over other containers) in `C++`, this may be a good time for you to read an introductory book on the `C++` Standard Template Library (STL).

On the other hand, a cell is roughly defined by its boundary which in turn is a mesh of lower dimension. Thus, the same type of `std::list<Cell>::iterator` could be used to implement a loop, say, over all vertices of a cube.

However, implementation details make this solution unpractical. For instance, negative meshes do not keep any list, they just hold a pointer towards their positive counterpart. Recall that the boundary of a negative cell is a negative mesh. Even among positive meshes, some are implemented with the goal of saving space in the computer's memory and store less information (connected one-dimensional meshes are an example). Paragraph 11.6 describes the implementation of different types of meshes.

For these reasons, maniFEM provides objects which iterate over cells. Suppose we have a mesh `msh` of rectangles. If we want to do something to each rectangle, that is, to each two-dimensional cell, we can use a code like

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 2 );
// or, equivalently :  Mesh::Iterator it = msh .iterator ( tag::over_cells, tag::of_max_dim )
for ( it .reset(); it .in_range(); it++ )
{  Cell cll = *it;  do_something_to ( cll );  }
```

Paragraph 11.3 gives some details about `tag`s.

In the above code you may note that these iterators obey to syntactic conventions slightly different from the ones in the Standard Template Library. We have chosen that, when we want to start an iteration process, we set the iterator in a starting configuration by using a `reset` method rather than through an assignment like `it = container.begin()`. Similarly, when an iterator has offered access to all cells of a mesh and cannot find other cells, it goes into a state which can be checked using its `in_range` method rather than by testing equality with some abstract object like `container.end()`. We have kept the syntax `it++` for advancing an iterator in the process of running over cells, offering also the equivalent alteratives `++it` and `it.advance()`. We have also kept the notation `*it` for dereferencing a `Mesh::Iterator`; this operation returns a `Cell` object (a wrapper). Of course, dereferencing a `Mesh::Iterator` does not produce a new cell, just provides access to a previously built cell (with the slight exception described in paragraph 9.8; see also paragraph 9.9).

Often, we `reset` an iterator only once, use it in a loop and then discard it. For this situation, a handy shortcut is to add the `tag::reset` to the constructor itself, as in the code excerpt below. This has the advantage that the name of the iterator gains local scope. (this feature is not yet implemented)

code not working

```
for ( Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 2, tag::reset );
      it .in_range(); it++                                                          )
{  Cell cll = *it;  do_something_to ( cll );  }
```

If we want to iterate over all vertices of the mesh, we can use

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 0 );
// or, equivalently :  Mesh::Iterator it = msh .iterator ( tag::over_vertices )
for ( it .reset(); it .in_range(); it++ )
{  Cell P = *it;  do_something_to (P);  }
```

If we want to iterate over all segments of the mesh, we can use

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 1 );
// or, equivalently :  Mesh::Iterator it = msh .iterator ( tag::over_segments )
for ( it .reset(); it .in_range(); it++ )
{  Cell seg = *it;  do_something_to ( seg );  }
```

Note that an iterator running through cells of maximum dimension, that is, of dimension equal to the dimension of the mesh, may produce negative cells if the mesh contains them. Paragraph 9.5 discusses this possibility. Iterators over cells of lower dimension produce always positive cells.

We can force an iterator over cells of maximum dimension to produce only positive cells by adding a `tag::force_positive` as in

```
Mesh::Iterator it = msh .iterator ( tag::over_cells, tag::of_max_dim, tag::force_positive );
```

Note that it is not safe to modify a mesh while iterating over its cells (paragraph 10.3 shows an example). After modifying a mesh, you may re-use a previously declared iterator by `reset`ting it.

If we only want to know how many cells there are in a certain mesh, instead of using `msh.core->cells[d].size()` (`d` being the desired dimension of the cells) we may use the method `number_of` :

```
size_t n = msh .number_of ( tag::cells_of_dim, 2 );
```

Expression `msh.number_of(tag::vertices)` is equivalent to `msh.number_of(tag::cells_of_dim,0)`, while `msh.number_of ( tag::segments )` is equivalent to `msh.number_of ( tag::cells_of_dim, 1 )`.

Paragraph 9.4 describes iterators specific to one-dimensional meshes.

9.4. Iterators over chains of segments

One-dimensional meshes are peculiar, especially if they are connected. They can be traversed by simply following the natural order of the segments. Recall that meshes in maniFEM are oriented, so there is a direct order which can be followed. Connected one-dimensional meshes may be closed chains of segments or open ones. In the latter case there is a natural starting point.

Meshes in maniFEM come in several different flavors which differ mainly in their internal implementation, as explained in paragraph 11.6. One of these flavors, called `Mesh::Connected::OneDim`, provides specialized iterators over vertices and segments which obey to the natural order of the chain of segments and, for an open chain, begin at the natural starting point.

The syntax is the same as the one described in pragraph 9.3.

```
Mesh chain ( tag::segment, A .reverse(), B, tag::divided_in, n );
// A and B are (positive) vertices, n is an integer

Mesh::Iterator it0 = chain .iterator ( tag::over_cells_of_dim, 0 );
for ( it0 .reset(); it0 .in_range(); it0++ )
{  Cell P = *it0;  do_something_to (P);  }

Mesh::Iterator it1 = chain .iterator ( tag::over_cells_of_dim, 1 );
for ( it1 .reset(); it1 .in_range(); it1++ )
{  Cell seg = *it1;  do_something_to ( seg );  }

// or, equivalently,
// Mesh::Iterator it0 = chain .iterator ( tag::over_vertices )
// Mesh::Iterator it1 = chain .iterator ( tag::over_segments )
// Mesh::Iterator it1 = chain .iterator ( tag::over_cells, tag::of_max_dim )
```

If we want to start at a specific location, we can make a `reset` call with `tag::start_at` followed by a second argument of type `Cell`. For iterators over vertices, this second argument should be a positive vertex, while for iterators over segments this argument should be a (positive or negative) segment. In this case, the iteration process will begin at that particular vertex or segment. This special kind of `reset` can be used for an open chain or a closed one, but beware : if applied to an open chain, the vertices or segments previous to the provided argument will not show up in the iteration process.

Note that it0 produces positive points, while it1 produces oriented segments (positive or negative). We may enforce that we only want positive segments by adding the tag::force_positive, as shown in paragraph 9.5.

There are also reversed versions of these iterators (they go backwards), obtained by adding the tag::backwards:

```
Mesh::Iterator it1r = chain .iterator ( tag::over_segments, tag::backwards );
Mesh::Iterator it0r = chain .iterator ( tag::over_vertices, tag::backwards );
```

Connected one-dimensional meshes have methods first_vertex, last_vertex, first_segment and last_segment which return the cell described by their names. They should only be used for an open chain (not for a loop). Note that Mesh::first_vertex returns a negative vertex while Mesh::last_vertex returns a positive vertex, similarly to the Cell::base and Cell::tip methods (described in paragraph 1.2).

As explained in paragraph 9.9, de-referencing a Mesh::Iterator does not produce a new cell, just provides access to a previously built cell.

Sometimes it may be not obvious for the user whether a certain Mesh is internally a Mesh:: ::Connected::OneDim (see paragraph 11.6). An iterator declared as it = chain.iterator (tag::over_vertices) will obey to the natural order of the vertices if chain is a Mesh::Connected::OneDim; otherwise it will bring up vertices in a rather unpredictable order. If we want to make sure the vertices will come up in a linear order, we can add a tag::require_order:

```
Mesh::Iterator it1 = chain .iterator ( tag::over_segments, tag::require_order );
Mesh::Iterator it0 = chain .iterator ( tag::over_vertices, tag::require_order );
```

If chain does not support linear ordering, the above statements will produce (in DEBUG mode) a run-time error rather than returning unordered iterators. Specifying tag::backwards also informs maniFEM that you need linear ordering of cells, so you don't have to provide both tags.

9.5. Orientation of cells within a mesh

In maniFEM, all cells and meshes are oriented. This can be confusing sometimes, so let's have a closer look at a particular example.

Consider a mesh tri_mesh made of triangles, part of which is represented in figure 9.2. Unless requested otherwise, tri_mesh will be a positive mesh and all triangles composing it will also be positive. So, the triangles composing tri_mesh will have no reverse cell (there is no need for such).

The above assertion that all triangles are positive should not be interpreted geometrically. As an extreme example, there may be no coordinates at all in our program (see paragraph 1.2 and comments at the beginning of paragraph 1.3). Or, the mesh tri_mesh may live in a high dimensional space where it is impossible to distinguish between a "positively oriented" triangle and a "negatively oriented" one. When we say that a cell is positive we simply mean that it has been declared earlier than its reverse. A positive cell may have no reverse at all. A negative cell has been declared after its positive counterpart so it has necessarily a reverse. See also paragraph 9.9.

However, the segments in figure 9.2 must have reverse. Consider triangle ABC for instance. Its boundary is made of three segments; let's look at AB for example, a segment having A as base (or rather A.reverse()) and B as tip. Now, a triangle BAD (no offense intended) also exists as part of tri_mesh. The boundary of BAD is made of three segments, one of them being BA,

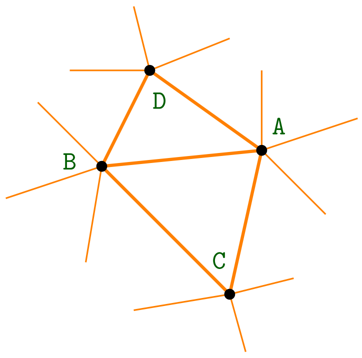


Figure 9.2.: a (part of a) triangular mesh

which has `B` as base and `A` as tip. `AB` and `BA` are different `Cell` objects; each is the reverse of the other. They are not two wrappers for the same core, they have different cores.[1] One of them is considered positive and the other is considered negative. Which is which depends on which one was built first. So, all inner segments must have a reverse; segments on the boundary of `tri_mesh` (not visible in figure 9.2) will probably have no reverse.

For points (vertices), the situation is even more complex. Segment `AB` sees `A` as negative because `A` is its base (or rather `A.reverse()`), but other segments like `CA` see `A` as positive.

Let's look again at iterators described in paragraph 9.3. We now understand that there is no point to have an iterator over oriented segments, or over oriented vertices, of `tri_mesh`. That's why iterators over cells of lower dimension always produce positive cells.

We also understand that usually there is no difference between these two iterators :

```
Mesh::Iterator it1 = tri_msh .iterator ( tag::over_cells, tag::of_max_dim );
Mesh::Iterator it2 =
   tri_msh .iterator ( tag::over_cells, tag::of_max_dim, tag::force_positive );
```

because all triangles composing `tri_mesh` should be positive (use `it1`, it is slightly faster). However, if some of the triangles are negative `it1` will behave differently from `it2`. For instance, if `tri_mesh` is the boundary of a polyhedron in $\mathbb{R}^3$ and this polyhedron touches other polyhedra (there are shared faces), then it is quite possible that some of the triangles in `tri_mesh` be negative. If you are aware that your mesh may contain negative cells but you want to iterate over their positive counterparts, use the `tag::force_positive`.

We now turn to iterators over one-dimensional meshes, described in paragraph 9.4. The two iterators below will probably have different behaviours, depending on which segments happen to be positive :

```
Mesh::Iterator it3 = ABC .boundary() .iterator ( tag::over_segments );
Mesh::Iterator it4 =
   ABC .boundary() .iterator ( tag::over_segments, tag::force_positive );
```

There is no difference between the two iterators below (both produce positive points).

```
Mesh::Iterator it5 = ABC .boundary() .iterator ( tag::over_vertices, tag::backwards );
Mesh::Iterator it6 = ABC .boundary() .reverse() .iterator ( tag::over_vertices );
```

However, the two iterators below are quite different.

[1] In contrast, negative meshes share the same core with their positive counterpart; see paragraph 11.4.

```
Mesh::Iterator it7 = ABC .boundary() .iterator ( tag::over_segments, tag::backwards );
Mesh::Iterator it8 = ABC .boundary() .reverse() .iterator ( tag::over_segments );
```

Iterator `it7` will produce segments `AB`, `CA`, `BC` (not necessarily beginning at `AB`), while `it8` will produce their reverses `BA`, `AC`, `CB` (not necessarily beginning at `BA`).

Incidentally, note that a segment, say, `BC`, may have no reverse, for instance if it is on the boundary of `tri_mesh`. However, its reverse `CB` will be built on-the-fly (and will stay persistent) as soon as you use the reverse mesh `ABC.boundary().reverse()` in the declaration of `it6` (or `it8`, whichever happens first in your code).

9.6. Navigating inside a mesh

Objects in class `Mesh` have two methods, `cell_behind` and `cell_in_front_of`, which provide access to the neighbours of a given cell within that mesh. Together with methods `base` and `tip` of class `Cell` (mentioned in paragraph 1.2), they allow us to navigate inside a mesh.

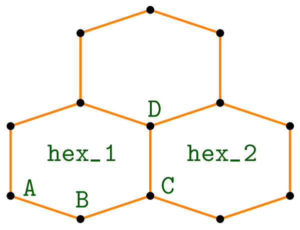


Figure 9.3.: a mesh made of hexagons

Consider the mesh `hex_msh` shown in figure 9.3, made of three hexagons. Pick one of them, at random :

```
Mesh::Iterator it1 = hex_msh .iterator ( tag::over_cells, tag::of_max_dim );
it1 .reset();  assert ( it1 .in_range() );
Cell hex_1 = *it1;
```

Now choose a random segment on the boundary of `hex_1` :

```
Mesh::Iterator it2 = hex_1 .boundary() .iterator ( tag::over_segments );
it2 .reset();  assert ( it2 .in_range() );
Cell AB = *it2;
```

Take its tip :

```
Cell B = AB .tip();
```

Suppose now we want the next segment, within the boundary of `hex_1` :

```
Cell BC = hex_1 .boundary() .cell_in_front_of (B);
```

And now we may continue by taking the tip of `BC` and then the segment following it :

```
Cell C = BC .tip();
Cell CD = hex_1 .boundary() .cell_in_front_of (C);
```

an so forth (this is how iterators over vertices and segments of connected one-dimensional meshes, described in paragraph 9.4, are implemented internally).

If you want to go backwards, you should be aware that the `base` of a segment is a negative vertex. Thus, an expression like `hex_1 .boundary() .cell_in_front_of ( BC .base() )` will produce an execution error; `BC.base()` is not equal to `B` but to `B.reverse()`. Thus, to obtain `AB` you should use `hex_1 .boundary() .cell_behind ( B )` or `hex_1 .boundary() .cell_behind ( BC .base() .reverse() )`. Recall that `hex_1 .boundary() .cell_in_front_of (B)` is `BC`.

Within the mesh `hex_msh`, we can navigate towards a neighbour hexagon :

```
Cell hex_2 = hex_msh .cell_in_front_of ( CD );
```

Since we have picked `hex_1` at random within `hex_msh`, as well as `AB` within the boundary of `hex_1`, there is no guarantee that we actually are in the configuration shown in Figure 9.3. That is, `CD` may be on the boundary of `hex_msh`; there may be no neighbour hexagon `hex_2`. If that is the case, the code above will produce an execution error. See paragraph 9.7 for a way to check whether there is actually a neighbour cell and thus avoid errors at execution time.

Note that faces point outwards. For instance, `CD` belongs to the boundary of `hex_1` and points outwards, towards `hex_2`. Thus, `hex_msh .cell_in_front_of ( CD )` produces `hex_2`. On the other hand, `hex_msh .cell_behind ( CD )` is `hex_1`.

Note also that `CD` does not belong to the boundary of `hex_2`. If we take `hex_2.boundary(). .cell_in_front_of (D)` we will obtain not `CD` but its reverse, a distinct cell which we may call `DC`. We have that `hex_msh .cell_in_front_of ( DC )` is `hex_1` and `hex_msh .cell_behind ( DC )` is `hex_2`. Between `CD` and `DC`, one is positive and the other is negative; which is which is not very important for the user.

9.7. Navigating at the boundary of a mesh

Consider again the mesh in paragraph 9.6. Suppose you try to get a neighbour hexagon which does not exist :

```
Cell no_such_hex = hex_msh .cell_in_front_of ( AB );
```

In `DEBUG` mode, you will get an `assertion error`. In `NDEBUG` mode, the behaviour is undefined (often, a `segmentation fault` will arise). The `DEBUG` mode is explained in paragraph 11.13.

You may check the existence of the neighbour cell by using a `tag::may_not_exist` and then the cell's method `exists` :

```
Cell possible_hex = hex_msh .cell_in_front_of ( AB, tag::may_not_exist );
if ( possible_hex .exists() ) do_something_to ( possible_hex );
else cout << "no neighbour !" << endl;
```

thus avoiding errors at execution time.

Paragraph 9.9 gives a complete list of operations which return an existing cell or build a new one.

9.8. Iterators around a cell

Suppose we are looking at a vertex within a mesh and we want to do something to all segments originating at that vertex, or to all squares touching that vertex, or (depending on the dimension of the mesh) to all cubes sharing that vertex.

Within this paragraph, in order to simplify the speech, we shall call "square" to a generic cell of dimension 2 (which may be a triangle or a hexagon or some other shape) and "cube" to a generic cell of dimension three (which may be a tetrahedron or some other shape).

Another similar situation is when we are looking at a segment in a three-dimensional mesh and we want to run over all squares having that segment as edge, or to all cubes having that edge. These situations require an iterator quite different from the ones described in paragraphs 9.3 and 9.4. We need an iterator which goes "around" a certain cell.

Note that we do not need an iterator to find, in a two-dimensional mesh, the two squares neighbour to a given segment, or, in a three-dimensional mesh, the two cubes neighbour to a given square. We do this by using methods `cell_in_front_of` and `cell_behind`, as explained in paragraphs 9.6 and 9.7. Thus, we shall not consider the case when the central cell has co-dimension one, that is, has dimension equal to the dimension of the mesh minus one.

So, we assume that the co-dimension is two or higher and we distinguish between the case when the central cell has co-dimension two and the case of co-dimension three or higher.

To begin with, suppose the central cell has co-dimension three or higher. The only natural example is a vertex in a three-dimensional mesh (that's co-dimension three). Less intuitive examples may appear if we play with meshes of (topological) dimension four or higher.

In code below, we iterate over all segments originating at a given vertex. The segments will show up in a rahter unpredictable order.

```
Mesh msh ( ... );  // a mesh of cubes for instance
Mesh::Iterator it =  // P is a positive vertex within msh
           msh .iterator ( tag::over_cells_of_dim, 1, tag::around, P.reverse() );
// or, equivalently :
// Mesh::Iterator it = msh .iterator ( tag::over_segments, tag::around, P.reverse() )
for ( it .reset(); it .in_range(); it++ )
{  Cell seg = *it;  do_something_to ( seg );  }
```

The orientation of the central cell is important in this case. In the above, we have provided a negative center, `P.reverse()`; thus, the iterator will produce segments `seg` having `P` as base (or rather `P.reverse()`) and pointing away from `P`. In contrast, the iterator declared below, having `P` as centrall cell, produces segments pointing towards `P`.

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 1, tag::around, P );
// or, equivalently :
// Mesh::Iterator it = msh .iterator ( tag::over_segments, tag::around, P )
```

In either case, since we are in a three-dimensional mesh, positive or negative segments `seg` may show up. If you want positive segments, add a `tag::force_positive` before `tag::around` in the declaration of `it`; in this case, some of the segments will point towards `P`, others will point away from `P`. If you provide a `tag::force_positive`, it makes no difference if you give `P` or `P.reverse()` as central cell in the constructor.

Note that `P` may be on the boundary of `msh`. The iterator will produce, of course, only cells belonging to the mesh, including those on the boundary.

If we want to run over squares or cubes having `P` as vertex, the orientation of the "center" `P` does not matter. We can provide a positive vertex `P` or a negative one `P.reverse()`, this will not change the behaviour of the iterator `it`:

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 2, tag::around, P );
for ( it .reset(); it .in_range(); it++ )
{  Cell cll = *it;  do_something_to ( cll );  }
```

Such an iterator will produce positive two-dimensional cells. Again, if `P` is the boundary of `msh`, the iterator will produce only cells belonging to the mesh, including those on the boundary.

If we ask for cubes, we will get cells oriented in accordance to the mesh `msh`. Except in exotic examples, these cells will be positive, but if you want to be sure you can always add the `tag::force_positive`:

```
Mesh::Iterator it = msh .iterator
   ( tag::over_cells_of_dim, 3, tag::force_positive, tag::around, P );
```

Consider now the case when the co-dimension is two, that is, when we are centered at a vertex within a two-dimensional mesh, or at a segment within a three-dimensional mesh.

In this case, the cells around that "center" have a natural linear order. If the "center" is at the boundary of `msh`, this order is like an open chain; if the "center" is in the interior of `msh`, these cells are organized in a closed loop. Hence, the constructor of an iterator centered at a cell of co-dimension two accepts tags like `tag::backwards` or `tag::require_order`. Also, the `reset` method accepts the `tag::start_at` followed by an argument of type `Cell`. Thus, many of the considerations in paragraph 9.4 apply to this type of iterators.

For example, if `msh` is a two-dimensional mesh, code below will produce all squares sharing the vertex `P`, linearly ordered.

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 2, tag::around, P );
```

If `P` is at the boundary of `msh`, the iteration process will begin with a square adjacent to the boundary and will end when it reaches again the boundary (which may happen at once if `P` is a corner of the mesh).

We can iterate over segments originating at `P` (now the orientation of `P` is important) as in:

```
Mesh::Iterator it =
            msh .iterator ( tag::over_cells_of_dim, 1, tag::around, P.reverse() );
// or, equivalently :
// Mesh::Iterator it = msh .iterator ( tag::over_segments, tag::around, P.reverse() )
```

If we want to start with a particular square `sq`, we specify this by giving supplementary arguments to the `reset` method :

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 2, tag::around, P );
for ( it .reset ( tag::start_at, sq ); it .in_range(); it++ )
{  Cell cll = *it;  do_something_to ( cll );  }
```

Just like in paragraph 9.4, beware : if `P` is at the boundary of `msh`, cells previous to `sq` will not show up.

There are also iterators over vertices around a given vertex. They simply iterate over the segments originating at that vertex and, when de-referenced, produce the tip of the current segment. They provide an ordered sequence of vertices if the surrounding mesh is two-dimensional; they produce vertices in a rather unpredictable order in a three-dimensional mesh :

```
Mesh::Iterator it = msh .iterator ( tag::over_vertices, tag::around, P );
for ( it .reset(); it .in_range(); it++ )
{  Cell Q = *it;  do_something_to ( Q );  }
```

If you need the cells to show up in a linear order but you are not familiar with the notion of co-dimension, you can add the `tag::require_order` to the `Mesh::Iterator` constructor; this way, if in `DEBUG` mode, *maniFEM* will produce a run-time error if that particular configuration is not compatible with linear ordering (that is, if the co-dimension of the central cell is not equal to two). Specifying `tag::backwards` also informs maniFEM that you want linear ordering of cells, so you don't have to provide both tags.

As a general rule, de-referencing a `Mesh::Iterator` does not build a new cell (it returns a new wrapper to an existing cell). Iterators around a cell, described in the present paragraph, are a slight exception to this rule. When they encounter positive cells having no reverse, they may choose to build the reverse, negative, cell. Actually, this may happen even without de-referencing the iterator; the constructor of the iterator may, in some cases, build a new negative cell (e.g., the reverse of a face on the boundary of the mesh). Negative cells are harmless, except that they occupy some space in the computer's memory.

Paragraph 10.5 shows an example of use of iterators around a vertex.

9.9. Declaring cells and meshes

As explained in paragraph 11.4, the `Cell` class is just a thin wrapper around a `Cell::Core`, and similarly for `Mesh`es.

Statements below build a new wrapper for an existing cell or mesh. No new cell or mesh is created :

```
Cell A = B;  // B is a Cell
Cell C = *it;  // 'it' is a Mesh::Iterator
Mesh msh_copy = msh;  // msh is a Mesh
Mesh bd = cll .boundary();  // cll is a Cell of dimension 1 or higher
```

Statements below search for an existing cell. If the respective cell exists, (a new wrapper for) it is returned. Otherwise, an `assertion error` will occur in `DEBUG` mode; in `NDEBUG` mode, the behaviour is undefined (often, a `segmentation fault` will arise). The `DEBUG` mode is explained in paragraph 11.13.

```
Cell A_rev = A .reverse ( tag::surely_exists );  //  A is a Cell
// or, equivalently :
Cell A_rev ( tag::reverse_of, A, tag::surely_exists );

Cell tri1 = msh .cell_behind ( CD );  //  CD is a Cell, a face within msh
Cell tri2 = msh .cell_in_front_of ( CD );
// or, equivalently :
Cell tri1 ( tag::behind_face, CD, tag::within_mesh, msh );
Cell tri2 ( tag::in_front_of_face, CD, tag::within_mesh, msh );
// or, equivalently :
Cell tri1 = msh .cell_behind ( CD, tag::surely_exists );
Cell tri2 = msh .cell_in_front_of ( CD, tag::surely_exists );
// or, equivalently :
Cell tri1 ( tag::behind_face, CD, tag::within_mesh, msh, tag::surely_exists );
Cell tri2 ( tag::in_front_of_face, CD,
            tag::within_mesh, msh, tag::surely_exists );
```

Note that reverse meshes exist always (negative meshes are temporary objects built on-the-fly).

```
Mesh rev_msh = msh .reverse();  // msh is a Mesh
```

Statements below search for an existing cell. If the respective cell exists, (a new wrapper for) it is returned. Otherwise, a non-existent cell is returned (an empty wrapper); the user has the possibility of inquiring the existence of the returned cell using its method `exists`, as illustrated in paragraph 9.7.

```
Cell A_rev = A .reverse ( tag::may_not_exist );  //  A is a Cell
// or, equivalently :
Cell A_rev ( tag::reverse_of, A, tag::may_not_exist );

Cell tri1 = msh .cell_behind ( CD, tag::may_not_exist );
//  CD is a Cell, a face within msh
Cell tri2 = msh .cell_in_front_of ( CD, tag::may_not_exist );
// or, equivalently :
Cell tri1 ( tag::behind_face, CD, tag::within_mesh, msh, tag::may_not_exist );
Cell tri2
   ( tag::in_front_of_face, CD, tag::within_mesh, msh, tag::may_not_exist );
```

Statements below return a previously built cell, if it exists. If that object does not exist, it is built on-the-fly.

```
Cell A_rev = A .reverse();  // A is some Cell
// or, equivalentely :
Cell A_rev ( tag::reverse_of, A );
// or, equivalentely :
Cell A_rev = A .reverse ( tag::build_if_not_exists );
// or, equivalentely :
Cell A_rev ( tag::reverse_of, A, tag::build_if_not_exists );
```

De-referencing a `Mesh::Iterator` returns (a new wrapper for) an existing cell; a slight exception to this rule is described in paragraph 9.8.

Statements below return a wrapper for a brand new cell or mesh :

```
Cell A ( tag::vertex );
Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 15 );
// B is another vertex Cell
Mesh ABC ( tag::triangle, AB, BC, CA );
//  BC and CA are segment Meshes, each having 15 segments Cells
// ... and many other shapes ...
```

10. Remeshing

This section is devoted to remeshing algorithms. Before proceeding please make sure you have read and understood paragraphs 9.6 and 9.7.

10.1. Cutting a square along a diagonal

Suppose we have a mesh of squares and for some reason we want to transform one of the squares in two triangles, cutting it along one of its diagonals.

One way to achieve this is

parag-10.1.cpp
```
    ABCD .remove_from_mesh ( msh );
    Cell AC ( tag::segment, A .reverse(), C );
    Cell ABC ( tag::triangle, AB, BC, AC .reverse() );
    Cell CDA ( tag::triangle, CD, DA, AC );
    ABC .add_to_mesh ( msh );
    CDA .add_to_mesh ( msh );
```

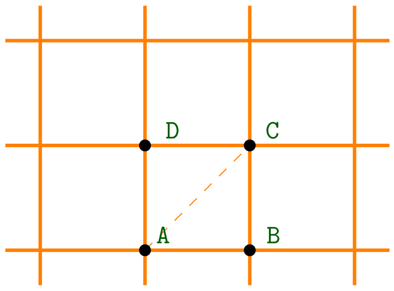


Figure 10.1.: a (part of a) mesh of squares

The only delicate point here is that the core of the `Cell` object `ABCD` will become unused (and inaccessible when the wrapper goes out of syntactic scope). If you have the garbage collector on (see paragraph 11.5), the core will be destroyed when the wrapper goes out of scope. Otherwise, the core will continue to occupy space in the computer's memory giving rise to an undesirable phenomenon known as "memory leak". If your program does this a lot, it will occupy increasingly more memory without reason.

10.2. Transforming a square into a triangle

Instead of discarding the square and building two new triangles, we can use the same `Cell` object `ABCD` by transforming it into a triangle, without removing it from `msh`, then add a new triangle to obtain the same result as in paragraph 10.1.

parag-10.2.cpp

```
CD .cut_from_bdry_of ( ABCD );
DA .cut_from_bdry_of ( ABCD );
// at this point, ABCD is a cell whose boundary is incomplete
// it has only two sides and an opening
// however, it still is part of 'msh'

Cell AC ( tag::segment, A .reverse(), C );
AC .reverse() .glue_on_bdry_of ( ABCD );
// at this point, and in spite of its name, ABCD is no longer a square
// it is a triangle, still part of 'msh'

Cell CDA ( tag::triangle, CD, DA, AC );
CDA .add_to_mesh ( msh );
```

10.3. Modifying the mesh within a loop

As is the case with many C++ containers, it is not safe to modify a mesh while iterating over its cells. For instance, code below gives undefined behaviour (often, a `segmentation fault` will occur).

incorrect code !

```
Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 2 );
for ( it .reset(); it .in_range(); it++ )
{  Cell square = *it;
   if ( ... some criterion ... )
   {  square .remove_from_mesh ( msh );
      // statement above opens the path to disaster
      // next time 'it' is incremented, we get undefined behaviour
      // we may add other cells afterwards, at no good
      some_other_cell .add_to_mesh ( msh );     }   }
```

We can circumvent this problem by creating a list of cells to be cut.

parag-10.3.cpp

```
std::list < Cell > list_of_squares;

Mesh::Iterator it = msh .iterator ( tag::over_cells_of_dim, 2 );
for ( it .reset(); it .in_range(); it++ )
{  Cell square = *it;
   if (  some criterion  )  // we decide to cut this square in halves
      list_of_squares .push_back ( square );  }

for ( std::list < Cell > ::iterator it_square = list_of_squares .begin(),
      it_square != list_of_squares .end(); it_square               )
{  Cell square = * it_square;
   // according to the same criterion, choose a diagonal, here we cut along PR
   square .remove_from_mesh ( msh );
   Cell PR ( tag::segment, P .reverse(), R );
   Cell PQR ( tag::triangle, PQ, QR, PR .reverse() );
   PQR .add_to_mesh ( msh );
   Cell RSP ( tag::triangle, RS, SP, PR );
   RSP .add_to_mesh ( msh );                          }
```

10.4. Improving a mesh of triangles

Sometimes the frontal meshing algorithm, presented in section 3, produces imperfect meshes. Recall that this algorithm produces meshes of triangles. For such a mesh to be considered of good quality, one may require each inner vertex to have six neighbour triangles, and thus six neighbour segments. This is not realistic for arbitrary domains, but it is reasonable to require each vertex to have five, six or seven neighbours. In most cases, the algorithm fulfills this condition, but in rare cases we can see vertices with four or eight neighbours.

The code presented in this section searches for vertices with four (or three) neighbours, as well as vertices with eight (or more) neighbours. It then performs a "flip" operation in order to fix this undesired configuration. Figure 10.2 illustrates this operation. On the right hand side, we see that the red vertex has only four neighbours. After "flipping" the blue segment, the red vertex will have five neighbours.

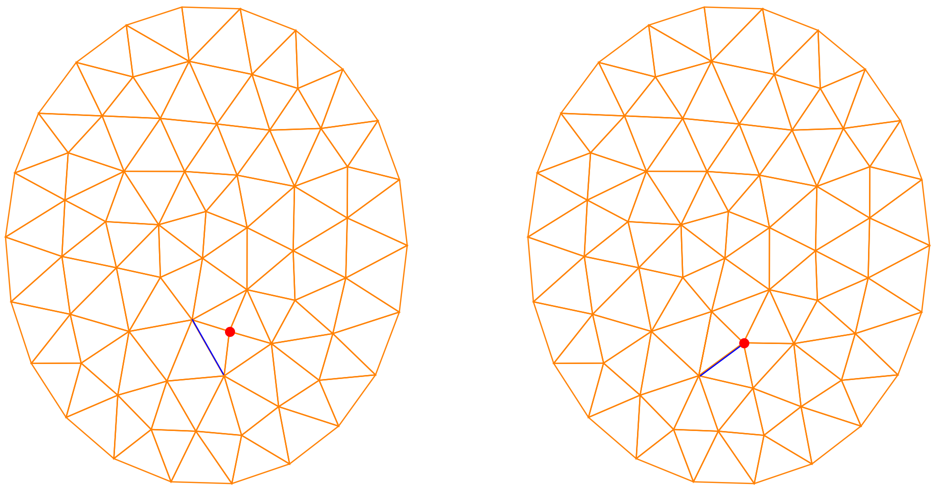


Figure 10.2.: left : before remeshing, right : after remeshing

Below is the code performing the "flip" operation.

parag-10.3.cpp

```
inline bool flip_segment ( Mesh & msh, Cell & seg )

// flip 'seg' if it is inner to 'msh'
// equilibrates (barycenter) the four neighbour vertices

// return true if the segment has been flipped, false if not

// assumes there are only triangular cells

// assumes there is no higher-dimensional mesh "above" 'msh'

{  Cell tri2 = msh .cell_in_front_of ( seg, tag::may_not_exist );
   if ( not tri2.exists() ) return false;
   Cell tri1 = msh .cell_behind ( seg, tag::may_not_exist );
   if ( not tri1 .exists() ) return false;
   // or, equivalently :  if ( not seg .is_inner_to ( msh ) ) return false

   Cell A = seg .base() .reverse();
   Cell B = seg .tip();
   Cell BC = tri1 .boundary() .cell_in_front_of ( B, tag::surely_exists );
   Cell CA = tri1 .boundary() .cell_behind ( A, tag::surely_exists );
   Cell AD = tri2 .boundary() .cell_in_front_of ( A, tag::surely_exists );
   Cell DB = tri2 .boundary() .cell_behind ( B, tag::surely_exists );
   Cell C = BC .tip();
   assert ( CA .base() .reverse() == C );
   Cell D = AD.tip();
   assert ( DB .base() .reverse() == D );

   B .cut_from_bdry_of ( seg, tag::do_not_bother );
   A .reverse() .cut_from_bdry_of ( seg, tag::do_not_bother );
   // at this point, 'seg' is a weird segment with no extremities at all

   CA .cut_from_bdry_of ( tri1, tag::do_not_bother );
   DB .cut_from_bdry_of ( tri2, tag::do_not_bother );
   // 'tri1' and 'tri2' now have as boundaries open chains of two segments only
   // even worse : one of these segments is 'seg' which is itself incomplete
   //              so these chains are actually disconnected

   C .reverse() .glue_on_bdry_of ( seg, tag::do_not_bother );
   D .glue_on_bdry_of ( seg, tag::do_not_bother );
   // 'seg' is complete again
   // boundaries of 'tri1' and 'tri2' are now connected chains of two segments each

   DB .glue_on_bdry_of ( tri1, tag::do_not_bother );
   CA .glue_on_bdry_of ( tri2, tag::do_not_bother );
   // 'tri1' and 'tri2' are complete again, their boundaries are closed loops

   tri1 .boundary() .closed_loop ( B );
   tri2 .boundary() .closed_loop ( A );

   if ( A .is_inner_to ( msh ) ) msh .barycenter ( A );
   if ( B .is_inner_to ( msh ) ) msh .barycenter ( B );
   if ( C .is_inner_to ( msh ) ) msh .barycenter ( C );
   if ( D .is_inner_to ( msh ) ) msh .barycenter ( D );

   return true;

}  // end of  flip_segment
```

Code above follows the approach, already used in paragraph 10.2, of destroying partially cells (segments and triangles) and then completing them later in a different configuration. In contrast, paragraphs 10.1 and 10.3 show code which simply removes cells from the mesh then builds entirely new cells and adds them to the mesh.

In the above code, we add a `tag::do_not_bother` as argument to methods `cut_from_bdry_of` and `glue_on_bdry_of`. This tag changes the behaviour of the respective methods when a `Mesh::Connected::OneDim` is involved (paragraph 11.6 gives details on different kinds of meshes). Recall that boundaries of triangles (and of other two-dimensional cells) are `Mesh::Connected::` `::OneDim` if built through the usual constructors.

Methods `cut_from_bdry_of` and `glue_on_bdry_of`, if invoked without the `tag::do_not_bother`, when trying to add or remove cells from a `Mesh::Connected::OneDim`, take care to leave that mesh in a consistent state. Removing a cell from a closed loop transforms it into an open chain of segments; the reverse may happen when adding a cell. Also, the number of component segments must be kept up-to-date.

If we provide the `tag::do_not_bother` to these methods, they will not waste computing time with such details. The respective meshes will be left in an inconsistent state. That's why we use the method `closed_loop` and provide a vertex; this method fixes the state of the mesh. We could also provide the updated number of segments as second argument to method `closed_loop`; this is not necessary in this case because the number of segments has not changed.

10.5. Evolution of an interface

This example is not directly related to remeshing; however, it is a good example of how the navigation in a mesh is important for shape optimization.

We consider a square body with a round inclusion. The thermal conductivity σ of the body is constant in each region (lower in the inclusion, higher in the bulk). We solve a non-homogeneous Dirichlet problem in this domain, then apply a few steps of shape optimization in order to maximize the functional

$$J = \iint_Q \sigma \frac{\partial u}{\partial x_i} \frac{\partial u}{\partial x_i} \, da$$

with a constraint on the volume of the bulk region, given by the variable `Lagr`.

It is a (difficult) mathematical exercise to show that, for an infinitesimal deformation of the geometry of our body, given by the vector field F, the corresponding variation in the value of J is

$$\delta J = \int_\Gamma \left[\sigma \frac{\partial u}{\partial x_i} \frac{\partial u}{\partial x_i} \right] F^k n_k \, ds - 2 \int_\Gamma \sigma \frac{\partial u}{\partial x_i} n_i \left[\frac{\partial u}{\partial x_k} \right] F^k ds$$

In the above, n is the normal vector to the interface Γ (it is irrelevant if n points inside the inclusion or towards the bulk) and the right parentheses stand for the jump of the quantity inside them, that is, the value on the side of the base of n minus the value on the side of Γ where n points.

Since maniFEM is built around the concept of oriented cell, it is easy to control the direction of the normal vector and the meaning of the jump. We take a vertex `Q` belonging to `circle` and we know exactly the segment "behind" `Q` and the one "in front of" `Q`:

parag-10.5.cpp

```
Mesh::Iterator it = circle .iterator ( tag::over_vertices );
for ( it .reset(); it .in_range(); it++ )
{  Cell Q = *it;
   Cell PQ = circle .cell_behind (Q);
   Cell QR = circle .cell_in_front_of (Q);
   Cell P = PQ .base() .reverse();
   Cell R = QR .tip();
   // compute normal vector, pointing inside 'disk'
   double n_x = y(P) - y(R), n_y = x(R) - x(P);
   double norm = std::sqrt ( n_x * n_x + n_y * n_y );
   n_x /= norm;  n_y /= norm;
```

For computing the jump of the derivatives of the solution u, we use the average value on triangles lying on each side of `circle`. To achieve this, we can use an iterator around a vertex, described in paragraph 9.8.

parag-10.5.cpp

```
   // compute the average value of grad temp on one side of 'circle'
   // (in 'disk') (in the inclusion)
   double avrg_dtemp_dx_incl = 0., avrg_dtemp_dy_incl = 0.;
   double area = 0.;
```

```
      Mesh::Iterator it_incl =
         disk .iterator ( tag::over_cells, tag::of_max_dim, tag::around, Q );
      for ( it_incl .reset(); it_incl .in_range(); it_incl ++ )
      {  Cell tri = * it_incl;
         fe .dock_on ( tri );
         double dtemp_dx_on_tri, dtemp_dy_on_tri;
         compute_integral_of_grad_on_cell_in_2D
            ( temperature, tri, fe, numbering, dtemp_dx_on_tri, dtemp_dy_on_tri );
         // 'fe' is already docked on 'tri' so this will be the domain of integration
         double area_tri = fe .integrate ( 1. );
         area += area_tri;
         avrg_dtemp_dx_incl += dtemp_dx_on_tri;
         avrg_dtemp_dy_incl += dtemp_dy_on_tri;
         seg = tri .boundary() .cell_in_front_of (Q) .reverse();                  }
      avrg_dtemp_dx_incl /= area;
      avrg_dtemp_dy_incl /= area;
```

Figure 10.3 shows five steps of the shape optimization process. In this paragraph, we choose to remesh from scratch after each deformation of the `circle`. Another possibility would be to deform the existing mesh by moving its vertices according to the vector field F. For this, one would need to extend F from the interface `circle` to the `entire_square`, typically by solving another, auxiliary, partial differential equation.

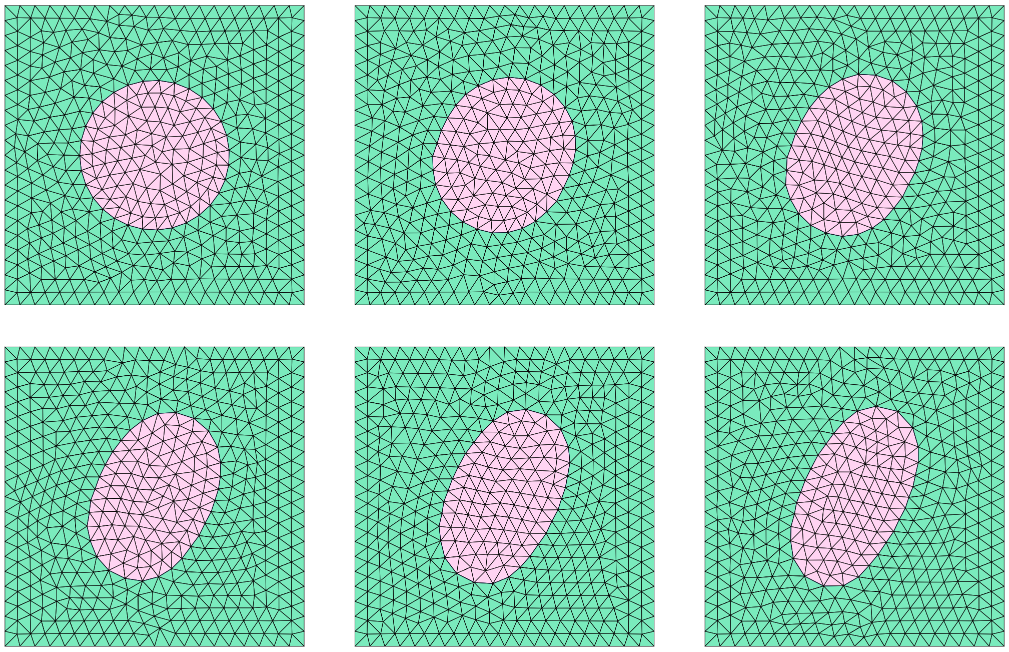


Figure 10.3.: five iterations of a shape optimization process

The inclusion becomes elongated in the direction $(1, 2)$ which is the average gradient of the temperature, due to the non-homogeneous Dirichlet boundary condition $u(x, y) = x + 2y$.

Note that some of the segments of `circle` are becoming longer than the average segment size of the mesh. This makes it difficult for maniFEM to build a mesh of triangles, starting from `circle`, with an approximately constant segment size. Thus, this approach is limited to just a few optimization steps; after some more steps, the frontal mesh generation process will stop with some error message.

Part III.

For programmers

Sections 11 and 12 are aimed at people interested in maintaining and developing maniFEM. Of course the ultimate documentation is the source code; these sections can be used as a guide through the source code.

11. Technical details

11.1. Namespaces and class names

All names in maniFEM are wrapped into the namespace maniFEM. We recommend using namespace maniFEM in your code, otherwise the text will become cumbersome. For instance, you will have to write maniFEM::Mesh::Iterator instead of Mesh::Iterator, and so on. We are using namespace maniFEM in the examples of this manual.

As a general rule, namespaces and class names are written with capital initial letter : Cell, Mesh, Integrator, FiniteElement, VariationalFormulation. Namespace tag (see paragraph 11.3) is an exception to the above rule. Namespace maniFEM itself is also an exception, for merely aestetic reasons.

In maniFEM, there are no private or protected class members or methods. Everything is public; the user can make use of any class member if he or she so chooses. This can be considered poor design; we endorse this criticism with no further comments.

However, some class members and methods are intended to be used by the final user, while others are used in the internal implementation of the former. Classes intended for basic usage are directly exposed in namespace maniFEM, for instance Cell, Mesh, Function, Manifold, VariationalFormulation, Integrator, FiniteElement. Some names, although not directly exposed, are intended for basic use : Mesh::Iterator, Manifold::Action, many tags (see paragraph 11.3). Classes not intended for the final user (at least not for the basic usage of maniFEM) have been hidden inside the above mentioned names, e.g. Mesh::Positive, Mesh::Iterator::Over::SegsOfPosLoop::Positive, Function::CoupledWithField::Scalar. Also, in the source code there are comments like

```
// do not use directly, let some other method do the job
```

In an effort not to expose too many names in the maniFEM namespace, we have used the namespace tag::Util to keep names which did not fit elsewhere. We have also used anonymous namespaces in order to hide names of functions which are only used in one source file.

In the namespace maniFEM there are also many overloaded functions. Because they take arguments of specific type, they will not clash with names defined by the user or by other libraries. Examples are : min, max, smooth_min, smooth_max, power, abs, sign, sin, cos.

11.2. Code coloring

Throughout this manual, example C++ code is coloured according to the following conventions.

Class names directly exposed in namespace maniFEM are shown in blue : Cell, Mesh, Function, Manifold, VariationalFormulation, Integrator, FiniteElement. There are also functions like smooth_min and also overloaded versions of common functions like power, sin, cos. The word tag deserves a special treatment. On one hand, it is directly exposed, on the other hand, we want it to be discrete. That's why we have chosen a blue grey for tags; see paragraph 11.3.

MetricTree is also written in blue, although it is not part of the namespace maniFEM.

Newly declared objects are shown in purple.
Numeric constants are orange.
Comments are grey.
Strings are dark green.

11.3. Tags

We use extensively tags. These are C++ structures gathered in the namespace tag. Most of them contain no data; only their type is useful, at compile time.

A tag is used to clearly distinguish between functions with the same name (overloaded functions). For instance, in the code excerpt below five Mesh::Iterators are defined by means of different (overloaded) methods having the same name Mesh::iterator; these five iterators behave very differently (see paragraphs 9.3 and 9.4).

```
// 'chain' is a one-dimensional Mesh
Mesh::Iterator it1 = chain .iterator ( tag::over_vertices );
Mesh::Iterator it2 = chain .iterator ( tag::over_vertices, tag::backwards );
Mesh::Iterator it3 = chain .iterator ( tag::over_segments );
Mesh::Iterator it4 = chain .iterator ( tag::over_segments, tag::backwards );
Mesh::Iterator it5 = chain .iterator ( tag::over_segments, tag::force_positive );
```

The above could be achieved by giving longer names to the functions, but we believe tags make the code more readable. Besides that, in the case of constructors, we do not have the choice of the name of the function (a constructor has the same name as its class). ManiFEM uses tags extensively for constructors, e.g.

```
Cell A ( tag::vertex );  Cell B ( tag::vertex );
Mesh AB ( tag::segment, A .reverse(), B, tag::divided_in, 10 );
```

There is a drawback to using tags instead of choosing different names for functions: compilation error messages may become quite long. A work-around is shown in paragraph 11.11.

In maniFEM, namespaces and class names begin with capital letter (see paragraph 11.1). The namespace tag (or maniFEM::tag if you are not using namespace maniFEM) is an exception to the above rule. We prefer its name to have only lower case letters because we want it to be discrete. For instance, if we wrote Tag::divided_in, the reader's eye would catch Tag much before divided_in; thus, tag::divided_in is more readable. Of course, the final user has the choice of using namespace tag but beware, it has many names which may conflict with the ones in your code.

Most tags have only lower-case letters and underscores. The exceptions are proper names: tag::Lagrange, tag::Euclid.

11.4. Cells and meshes revisited

Designing and implementing maniFEM has been a challenging endeavour. We had a lot of fun and we have learned much along the process.

Take cells and meshes, for instance. As explained in paragraph 1.2, at the conceptual level meshes are roughly collections of cells of the same dimension and cells are essentially defined by their boundary which is a lower-dimensional mesh. This conceptual simplicity does not survive to the demands of an efficient code (although it has been extremely useful in the process of designing maniFEM; it should also be useful for learning and using maniFEM).

Vertices (zero-dimensional cells) have no boundary at all. One-dimensional cells (segments) have all the same shape and have a rudimentary boundary (a zero-dimensional mesh consisting of a `base` and a `tip`). In order to save space in the computer's memory, specific classes have been created for vertices and for segments.

Also, there are positive cells and negative cells, positive meshes and negative meshes. A positive cell and its negative counterpart (its `reverse`) share some information. To save memory space, negative cells are implemented in different classes from positive cells, thus avoiding the storage of some of the redundant information.

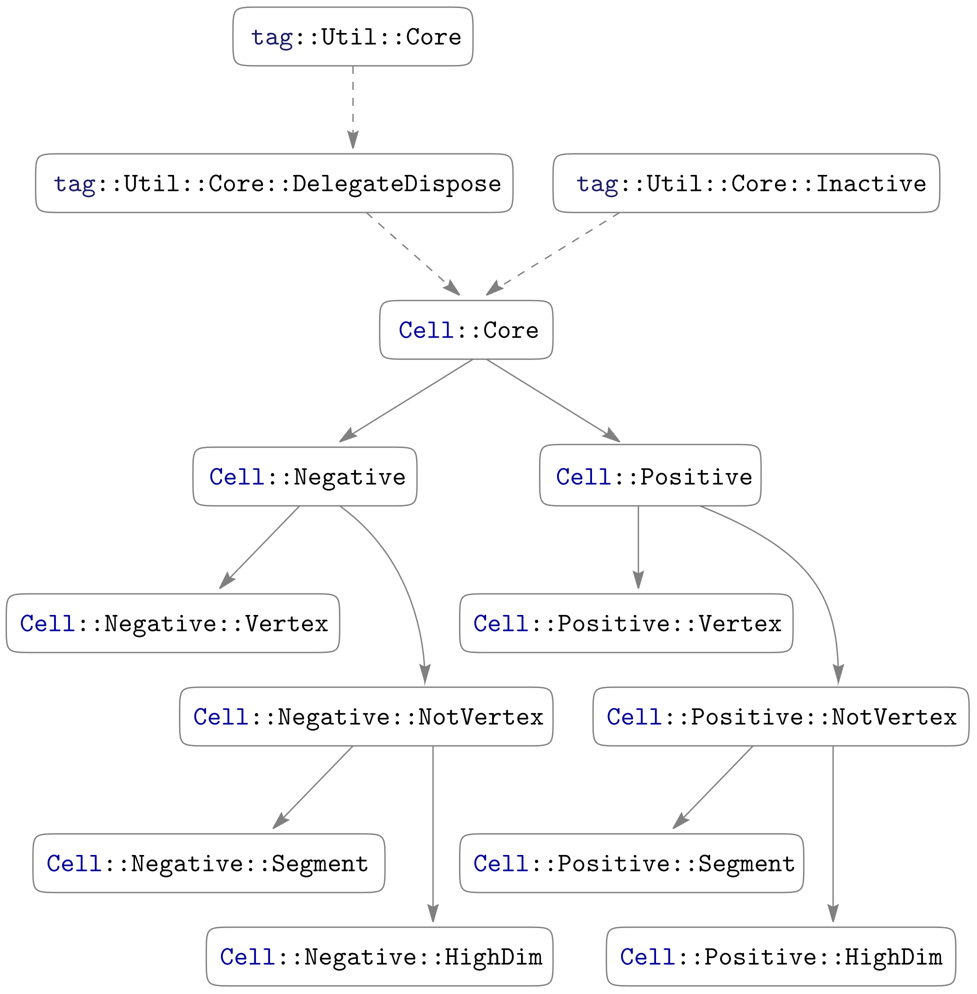


Figure 11.1.: inheritance diagram for `Cell` cores

We want, however, to manipulate all these different cells through a uniform interface, which is where `C++`'s inheritance and polymorphishm mechanisms come handy. Thus, there are classes `Cell::Positive::Vertex`, `Cell::Positive::Segment` and `Cell::Positive::HighDim`, all derived from `Cell::Positive`. And we have `Cell::Negative::Vertex`, `Cell::Negative::Segment` and `Cell::Negative::HighDim`, all derived from `Cell::Negative`. Both `Cell::Positive` and `Cell::Negative` are derived from `Cell::Core`. Figure 11.1 shows these inheritance relations; dotted lines represent relations subject to the `-DMANIFEM_COLLECT_CM` compilation option, explained in paragraph 11.5.

The following classes of core cells are abstract and cannot be instantiated : `Cell::Core`, `Cell::Positive`, `Cell::Core::Negative`, `Cell::Positive::NotVertex`, `Cell::Negative::NotVertex`. The following classes of core cells can be instantiated : `Cell::Positive::Vertex`, `Cell::Negative::::Vertex`, `Cell::Positive::Segment`, `Cell::Negative::Segment`, `Cell::Positive::HighDim`, `Cell::Negative::HighDim`.

On the other hand, we want cells and meshes to be persistent objects (not subject to syntactic scope). We create them within some function and we want them to remain alive after returning to the main program. Also, they are unique entities, it does not make sense to copy them. This is why we have implemented `Cell` as a thin wrapper around `Cell::Core` with most methods of `Cell` being delegated to `Cell::Core`. When it goes out of its syntactic scope, the wrapper is destroyed but the `Cell::Core` object inside remains intact (unless there are no more wrappers for that core, as explained in paragraph 11.5). Also, you can copy the wrapper as in `Cell A = B` or `Mesh BA = AB.reverse()` but these operations do not create new core objects, they just give new names to already existing cells or meshes (paragraph 9.9 gives more details). You can think of `Cell`s and `Mesh`es as customized pointers towards `Cell::Core`s and `Mesh::Core`s, respectively.

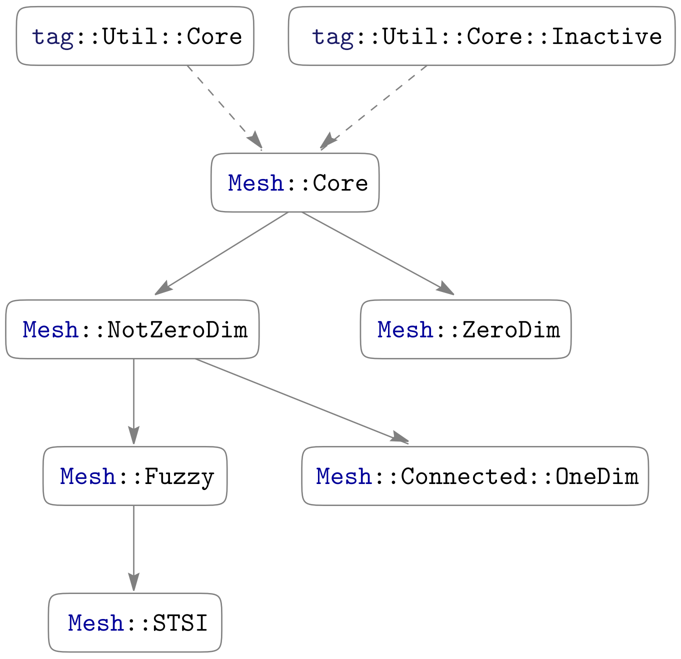


Figure 11.2.: inheritance diagram for `Mesh` cores

Negative meshes contain no useful information; they appear mostly as boundaries of negative cells, sometimes as interfaces between subdomains of our mesh. Thus, there is no such class as `Mesh::Negative`; all `Mesh::Core`s are positive. The wrapper class `Mesh` contains a pointer to a `Mesh::Core` and a flag telling it to reverse everything if the mesh is to be considered negative. Zero-dimensional meshes (boundaries of segments) are not stored at all. One-dimensional meshes have often a particular structure (they are chains of segments); when this happens they are implemented in a specialized class `Mesh::Connected::OneDim`. The graph in figure 11.2 shows these inheritance relations; dotted lines represent relations subject to the `-DMANIFEM_COLLECT_CM` compilation option, explained in paragraph 11.5.

Classes `Mesh::Core` and `Mesh::NotZeroDim` are abstract and cannot be instantiated. Classes `Mesh::ZeroDim`, `Mesh::Connected::OneDim`, `Mesh::Fuzzy` and `Mesh::STSI` can be instantiated.

Classes `Mesh::MultiplyConnected::OneDim`, `Mesh::Connected::HighDim` and `Mesh::` `::MultiplyConneted::HighDim` are object of on-going work.

To save memory, we don't even keep the dimension of a cell as an attribute, it's a static property for vertices and segments, while for higher-dimensional cells it's obtained from the boundary's dimension. The dimension of a mesh is not kept as an attribute either; it's computed on-the-fly by counting the levels of collections of cells the mesh is made of (and then substracting one). For instance, the mesh in paragraph 1.1 has three layers of cells : points, segments and squares.

Constructors for wrapper classes act as factory functions for the core object. According to their arguments, they build different core objects.

Other objects have been implemented using the same logic of wrappers and core objects : iterators, fields, functions, manifolds, integrators, finite elements.

11.5. Wrappers, cores and garbage collectors

Many objects in maniFEM are implemented using a wrapper-core model. Examples are `Cell`s, `Mesh`es, `Mesh::Iterator`s, `Manifold`s, `Function`s, `FiniteElement`s, `Integrator`s, `VariationalFormulation`s. A garbage collector is implemented; many of these classes inherit from `tag::Util::Wrapper` and their cores inherit from `tag::Util::Core`. Wrappers contain a pointer to a core; each core keeps a counter of wrappers pointing to it. The destructor of a wrapper decrements the counter and destroys the core if the counter reaches zero. It's like a `shared_ptr`.

`FiniteElement`s and `Integrator`s depend on each other in a complicated manner. The user may choose to declare an `Integrator`; a `FiniteElement` will be built behind the scenes; this happens in paragraphs 1.1 and 6.2. Or, they may choose to declare a `FiniteElement` and later build an `Integrator` by invoking the method `FiniteElement::set_integrator`; this happens in paragraphs 6.3 – 6.10. In the former case, the `Integrator` is responsible for keeping the `FiniteElement` alive and later for destroying it. In the latter case, it is the other way around. This is why we have implemented a dedicated garbage collector which behaves on most occasions like a `shared_ptr` but sometimes like a `weak_ptr`.

Cells and meshes depend on each other in a complicated manner, too. A cell keeps its boundary (which is a mesh) alive. A mesh keeps its component cells alive. A negative cell keeps alive its positive counterpart but the reverse does not hold; care must be taken to break dependency cycles.

Some programs consist only in building a mesh and never (or rarely) discarding cells. For such programs, there is no advantage in using a `shared_ptr`-like strategy. Thus, unlike for other maniFEM objects, the garbage collector for `Cell`s and `Mesh`es can be turned off by erasing the option `-DMANIFEM_COLLECT_CM` in the `Makefile`. Remember to `make clean` and re-build your program; see also paragraph 11.13. The graphs in figures 11.1 and 11.2 show dotted lines where the inheritance relationship is subject to the `-DMANIFEM_COLLECT_CM` option. If you turn off the garbage collector, `Cell`s and `Mesh`es will occupy less space in the computer's memory and your program will run slightly faster. But if you turn it off and your program does a lot of remeshing ghost cells and meshes will fill the computer's memory (this is known as "memory leak").

When the garbage collector for `Cell`s and `Mesh`es is on, you may notice a significant lag, especially as the program comes to an end. When a large mesh comes to the end of its life, it releases all of its component cells, thus allowing for their destructors to be invoked like in a chain reaction. This may take a long time, depending on the number of cells. Since the program is ending anyway, there is probably no advantage in invoking these destructors. If you end your program with a statement like `exit(0)`, the computer will skip this final cleanup, thus eliminating the lag at the end of the program.

11.6. Different kinds of meshes

A `Mesh` wrapper contains a pointer to a `Mesh::Core`. The latter is a polymorphic object; there are several different kinds of meshes which differ in their internal implementation; the wrapper provides a uniform interface to the user (making use of virtual methods of the core). The graph in figure 11.2, paragraph 11.4, shows the inheritance relations between different types of `Mesh::Cores`.

Negative meshes are temporary wrappers built on-the-fly. Their wrapper contains a pointer to a positive mesh core.

Zero-dimensional meshes appear only as boundaries of segment `Cell`s. They are temporary objects built on-the-fly.

Some meshes are implemented as `Mesh::Fuzzy`. They have as attributes lists of cells of different dimensions; these lists have no particular order. Iterators over cells of `Fuzzy` meshes simply run over the respective lists of cells.

Other meshes are assumed to be connected and keep no lists of cells. Iterators over such meshes traverse the mesh beginning at a starting point (which is an attribute of class `Mesh::` `::Connected::***Dim`) and then using the neighbourhood relations to move from one cell to another.

One-dimensional connected meshes are a special case because their topology is peculiar; segments and vertices composing such a mesh have a natural order (linear order). Recall that all meshes in *maniFEM* are oriented. A one-dimensional connected mesh can be a closed loop or an open chain of segments. In the latter case, there is a natural starting point (the first vertex or first segment). Iterators over such meshes are described in paragraph 9.4. The `Mesh` constructor with `tag::segment` builds an open `Mesh::Connected::OneDim`. Also, `Cell` constructors with `tag::triangle` and `tag::quadrangle` produce cells whose boundary is a closed `Mesh::Connected::OneDim`.

Connected meshes of dimension two or higher are not yet implemented; for now, `Mesh` constructors with `tag::rectangle` or `tag::triangle` build a `Fuzzy` mesh instead. A class `Mesh::` `::Connected::HighDim` is object of current work. We also intend to implement multiply connected meshes.

There are also `STSI` meshes; the name means “self-touching or self-intersecting” meshes. They are used internally for frontal meshing (described in section 3); they will also be used for implementing iterators over connected high-dimensional meshes.

Iterators over cells of a `Fuzzy` mesh are fast because they simply run over the list of cells of given dimension. Iterators over `Connected` meshes may be slower because they traverse the mesh using neighbourhood relations between cells. On the other hand, `Fuzzy` meshes require more space in the computer’s memory.

The kind of a certain mesh is not always completely obvious. For instance, when `joining` two meshes of the kind `Mesh::Connected::OneDim`, the result will be a `Mesh::Connected::OneDim` if the last vertex of one of the meshes is the same as the first vertex (reversed) of the other mesh. Otherwise, the result will be a `Mesh::Fuzzy`.

Methods `Cell::cut_from_bdry_of`, `Cell::glue_on_bdry_of`, `Cell::add_to_mesh` and `Cell::remove_from_mesh` accept as second argument a `tag::do_not_bother`. This tag has no effect for a fuzzy mesh. For a connected one-dimensional mesh, the method without tag will take measures to ensure the connectivity of the mesh and will update the total number of segments.

By providing a tag::do_not_bother, the user instructs maniFEM not to waste execution time with these details. The user assumes responsibility for updating data members nb_of_segs, first_ver and last_ver.. The same will apply to a high-dimensional connected mesh (object of current work).

11.7. Maximum topological dimension

maniFEM assumes you will not build meshes of topological dimension above 3. If you want to play with higher-dimensional meshes, you must relax this assumption through the statement Mesh::set_max_dim (some_integer).

On the other hand, you may want to decrease the expected dimension. Suppose you want to mesh surfaces in $\mathbb{R}^3$. This means your maximum topological dimension will be 2 (this has nothing to do with the geometric dimension, here 3). Then you may state your intention at the beginning of your program (before building any cell, before even declaring the first vertex) through the statement Mesh::set_max_dim (2). This will decrease the size of the Cell::Core objects in your code, thus saving some memory.

But beware, if you try to build a mesh of dimension higher than the one expected by maniFEM, you will get an assertion error at run-time in DEBUG mode, or some bizarre behaviour (often a segmentation fault) in NDEBUG mode. The DEBUG mode is explained in paragraph 11.13.

The geometric dimension is set by simply declaring a Euclidian manifold and building a coordinate system on it (see e.g. paragraphs 1.1, 2.23 and 5.1).

11.8. Adding and removing cells or faces

At a high level, we build cells and meshes using the appropriate Cell and Mesh constructors. At low level, cells and meshes are built by methods Cell::glue_on_bdry_of, Cell::cut_from_bdry_of, Cell::add_to_mesh and Cell::remove_from_mesh.

The inlined method Cell::glue_on_bdry_of is used for sticking a new face on the boundary of an existing cell cll (whose boundary is incomplete). It simply calls the inlined method Cell::Core::glue_on_bdry_of which in turn calls the virtual method cll->glue_on_my_bdry(face). Different methods Cell::***::glue_on_my_bdry exist for different kinds of cells but to summarize we state that glue_on_my_bdry first calls face->add_to_bdry in order to add the face to the boundary of cll then calls the inlined function add_cell_behind_above.

Method Cell::Core::add_to_bdry is virtual so it does different things for different kinds of cells but the common idea is that it calls another virtual method Mesh::Core::add_*** (the name depends on the type of cell being added). The latter calls an inlined function make_deep_connections (hidden in an anonymous namespace in mesh.cpp) which updates fields Cell::Positive::meshes, Cell::Positive***::meshes_same_dim and Mesh::Fuzzy::cells for several pairs cell-mesh (including cells of lower dimension and meshes of higher dimension). After that, Mesh::Core::add_*** updates neighbourhood relations coded in the map Cell::Core:: ::cell_behind_within.

The inlined function add_cell_behind_above deals with the case when the cell (on whose boundary we are trying to glue a new face) already belongs to some mesh(es). In that case, it updates the information in face->cell_behind_within.

If we want to add a cell to a mesh which is not a boundary of a cell (this happens frequently in Mesh constructors), we call the inlined method Cell::add_to which calls the virtual method Cell::Core::add_to_mesh which is similar to Cell::Core::add_to_bdry described above.

In the constructor of a cell, since we are sure there are no meshes containing that cell (the cell is just being built), we call method `add_to_mesh` instead of `glue_on_bdry_of` (we pretend the mesh is not a boundary).

Removing a face from the boundary of an existing cell, or removing a cell from a mesh which is not a boundary, implies a process similar to the one described above, with methods replaced by `Cell::cut_from_bdry_of`, `Cell::remove_from_mesh`, `Cell::Core::cut_from_bdry_of`, `Cell::Core::cut_from_my_bdry`, `Cell::Core::remove_from_bdry`, `Cell::Core::remove_from_mesh`, `Mesh::Core:remove_***`, `break_deep_connections`, `remove_cell_behind_above`.

Below is a very rough sketch of the process of adding a segment to the boundary of a triangle. Note that the triangle may belong to a mesh or even to the boundary of a three-dimensional cell which in turn may belong to a three-dimensional mesh (cells and meshes of dimension higher than three may also show up). Note also that the segment may be negative, just as the triangle may be negative. Three stars `***` stand for either `pos` or `neg`, one star `*` stands for a counter, `wh` stands for a pointer which for `Fuzyy` meshes points into the list of `cells` while for other types of meshes is meaningless. Often, `tri->boundary` is a `Mesh::Connected::OneDim`, not a `Mesh::Fuzzy`, but meshes `msh` of dimension 2 or higher are usually `Fuzzy`.

```
seg->glue_on_bdry_of ( tri )
   tri->glue_on_my_bdry ( seg )
      seg->add_to_bdry ( tri->boundary )
         tri->boundary->add_***_seg ( seg, tag::mesh_is_bdry )
            make_deep_connections_1d ( seg, tri->boundary, tag::mesh_is_bdry )
               add_link_same_dim ( seg, tri->boundary )
                  tri->boundary->cells[1] .push ( seg )  // if tri->boundary is Fuzzy
                  seg->meshes_same_dim [ tri->boundary ] = ( ± 1, wh )
               link_face_to_higher ( seg, tri, *, * )
                  for msh in {meshes of dimension d >= 2 containing tri}
                     add_link ( seg, msh, *, * )
                        if seg already belongs to msh
                           increment counters
                        else
                           msh->cells[1] .push ( seg )  // if msh is Fuzzy
                           seg->meshes[d] [ msh ] = ( *, *, wh )
               for ver in {two extremities of seg}
                  link_face_to_msh ( ver, seg, tri->boundary, *, * )
                     add_link ( ver, tri->boundary, *, * )
                        if ver already belongs to tri->boundary
                           increment one counter
                        else
                           tri->boundary->cells[0] .push ( ver )  // if Fuzzy
                           ver->meshes[1] [ tri->boundary ] = ( *, *, wh )
                  link_face_to_higher ( ver, tri, *, * )
                     for msh in {meshes of dimension d >= 2 containing tri}
                        add_link ( ver, msh, *, * )
                           if ver already belongs to msh
                              increment counters
                           else
                              msh->cells[0] .push ( ver )  // if msh is Fuzzy
                              ver->meshes[d] [ msh ] = ( *, *, wh )
            add_cell_behind_below_***_seg ( seg, tri->boundary )
               for ver in {two extremities of seg}
                  ver->cell_behind_within [ tri->boundary ] = seg
      add_cell_behind_above ( tri, seg )
         for msh in {meshes of dimension 2 containing tri}
            seg->cell_behind_within [ msh ] = tri
```

Below is a sketch of the reverse process of removing a segment from the boundary of a triangle.

```
seg->cut_from_bdry_of ( tri )
   tri->cut_from_my_bdry ( seg )
      remove_cell_behind_above ( tri, seg )
         for msh in {meshes of dimension 2 containing tri}
            seg->cell_behind_within .erase ( msh )
      seg->remove_from_bdry ( tri->boundary )
         tri->boundary->remove_***_seg ( seg, tag::mesh_is_bdry )
            for ver in {two extremities of seg}
               ver->cell_behind_within .erase ( tri->boundary )
            break_deep_connections_1d ( seg, tri->boundary, tag::mesh_is_bdry )
               for ver in {two extremities of seg}
                  unlink_face_from_msh ( ver, seg, tri->boundary, *, * )
                     remove_link ( ver, tri->boundary, *, * )
                        decrement one counter
                        if both counters zero
                           ver->meshes[1] .erase ( tri->boundary )
                           tri->boundary->cells[0] .remove ( ver )  // if Fuzzy
                  unlink_face_from_higher ( ver, tri, *, * )
                     for msh in {meshes of dimension d >= 2 containing tri}
                        remove_link ( ver, msh, *, * )
                           decrement counters
                           if both counters zero
                              ver->meshes[d] .erase ( msh )
                              msh->cells[0] .remove ( ver )  // if msh is Fuzzy
               unlink_face_from_higher ( seg, tri, *, * )
                  for msh in {meshes of dimension d >= 2 containing tri}
                     remove_link ( seg, msh, *, * )
                        decrement counters
                        if both counters zero
                           seg->meshes[d] .erase ( msh )
                           msh->cells[1] .remove ( seg )  // if msh is Fuzzy
               remove_link_same_dim ( seg, tri->boundary )
                  tri->boundary->cells[1] .remove ( seg )  // if tri->boundary is Fuzzy
                  seg->meshes_same_dim .erase ( tri->boundary )
```

11.9. About `init_***_cell`

The `Cell` class has static attributes `init_pos_cell`, `init_neg_cell`, `data_for_init_pos`, `data_for_init_neg`. The attribute `init_pos_cell` is a list of pointers to functions to be called by the constructor of a positive cell, while `data_for_init_pos` is a void pointer which can be used to pass supplementary information to `init_pos_cell`. Attributes `init_neg_cell` and `data_for_init_neg` fulfill a similar task when building negative cells.

For instance, the method `FiniteElement::build_global_numbering` builds a new `Cell::` `::Numbering::Field` whose constructor adds a call to `Cell::Numbering::Field::set_and_increment` to the list `Cell::init_pos_cell[0]`. This way, future vertices will receive automatically a `size_t` label, to be used by Lagrange finite elements.

11.10. Programming style

I (Cristian) have chosen some program-writing conventions which may seem unusual for some people. For instance, most programmers use braces like this

```
for ( ... ) {
   statement 1;
   statement 2;
   statement 3;
}
```

or perhaps like this

```
for ( ... )
{
   statement 1;
   statement 2;
   statement 3;
}
```

I just can't accept the idea that a brace opens more to the right than it closes, or at the same point. For me, a pair of braces should open at some point to the left and close at some point to the right, and the statements should be between them. That's how parentheses have been designed to be used. So I irreverently decided that my blocks will (often) look like this.

```
for ( ... )
{  statement 1;
   statement 2;
   statement 3;  }
```

I am aware I am violating conventions which are almost universally accepted. Sorry about that. For very large blocks, I kept the classical format.

On the other hand, I am very fussy about indentation. I guess my mind has been formatted by Python.

I do not use the `auto` keyword in the source code of maniFEM. I find that explicitly declaring the type of each variable makes the code more readable, at the price of increasing its size a little.

`Mesh::Iterators` obey to syntactic rules slightly different from the conventions for iterators in the Standard Template Library. See paragraph 9.3.

Also, the version numbering is somewhat unusual. The version consists merely of the year and month. Perhaps a nostalgic memory of my first serious programming language, `FORTRAN 77` ?

11.11. Frequent errors at compile time

```
variable var set but not used [-Wunused-but-set-variables]
func defined but not used [-Wunused-function]
```

These are harmless warnings.

Some variables are initialized but never used. When we create a `Manifold`, the constructor sets a global variable `Manifold::current`. Thus, maniFEM can remember at any time the geometry of the space and it can choose the right interpolation and projection operations. From the compiler's viewpoint, that `Manifold` oject is never used again and so it issues a warning.

Sometimes, an `Integrator` is declared but never formally used, the integration command being delegated to the associated `FiniteElement`.

Also, some functions are never used. They are there mainly for historical reasons. They have not been erased yet because part of their code may still be used in the future.

I get endless warnings !

Warnings are harmless, but when they are too many they can be a nuisance. On some computers, `Eigen` produces quite a lot of them. You can turn warnings off by removing `-W` options from your `Makefile`.

```
'class ManiFEM::tag::Util::CellCore' has no member named 'name'
-- or --
'class ManiFEM::tag::Util::CellCore' has no member named 'get_name'
```

It seems you are trying to compile your code in `NDEBUG` mode (see paragraph 11.13). In `NDEBUG` mode, cells and meshes do not have names.

```
cannot declare variable 'cll' to be of abstract type 'ManiFEM::tag::Util::CellCore'
-- or similar for ManiFEM::Cell::Positive or ManiFEM::Cell::Negative
  or ManiFEM::Cell::PositiveNotVertex or ManiFEM::Cell::NegativeNotVertex
  or tag::Util::MeshCore or ManiFEM::Mesh::NotZeroDim                    --
```

Classes `Cell::Core`, `Cell::Positive`, `Cell::Negative`, `Cell::Positive::NotVertex`, `Cell::Negative::NotVertex`, `Mesh::Core`, `Mesh::NotZeroDim` are abstract and cannot be instantiated. You must be more specific; `Cell::Positive::Vertex`, `Cell::Negative::Vertex`, `Cell::Positive::Segment`, `Cell::Negative::Segment`, `Cell::Positive::HighDim`, `Cell::Negative::HighDim`, `Mesh::ZeroDim`, `Mesh::Connected::OneDim`, `Mesh::Fuzzy`, `Mesh::STSI` can be instantiated. See paragraph 11.4.

```
error: no matching function for call to some function or method or constructor
followed by an extremely long list of candidates
```

As explained in paragraph 11.3, we use `tag`s for overloading function names (including methods and, especially, constructors). This can lead to very long compilation errors. If you have difficulties in viewing the beginning of the compilation message, where the most relevant information is, you may use a compilation command like `make run-3.15 2>&1 | less`. Another solution is `make run-3.15 > out.txt 2>&1` or `make run-3.15 2>&1 | tee out.txt`, then view the file `out.txt` using `less out.txt` or a text editor.

11.12. Frequent errors at run time

Many errors described in this paragraph will only show up in `DEBUG` mode, explained in paragraph 11.13.

Some errors give explicit messages. For instance, messages like

```
only one-dimensional meshes have first vertex
-- or --
virtual maniFEM::Cell maniFEM::tag::Util::CellCore::tip(): Only segments have extremities.
```

are self-explaining.

```
double& ManiFEM::OneDimField::operator()(ManiFEM::Cell&) const:
Assertion 'cll.real_heap_size() > index_min' failed.
```

You probably tried to access a coordinate (or some other value) at a cell to which no value has been associated. Either you picked a cell of a different dimension (e.g. a segment instead of a vertex) or you are looking at a negative cell. Values are usually stored at positive cells; negative cells have no information attached. You may check if a certain cell is positive or negative by using the method `is_positive`. Paragraph 9.5 gives more details about orientation of cells and meshes. See also paragraphs 9.9 and 11.4.

Note that, if `seg` is a segment (a one-dimensional cell), then `seg.tip()` is a positive cell but `seg.base()` is a negative cell. So, you probably need to use `seg.base().reverse()` instead. Iterators over cells of maximum dimension (that is, of dimension equal to the dimension of the mesh) produce oriented cells (which may be positive or negative). Consider using the `tag::force_positive`. See paragraphs 9.3 and 9.4.

```
assertion "this->core->reverse_attr.exists()" failed: file "mesh.h",
function: maniFEM::Cell maniFEM::Cell::reverse(const maniFEM::tag::SurelyExists&) const
-- or --
assertion "it != face.core->cell_behind_within.end()" failed: file "mesh.h",
function: maniFEM::Cell maniFEM::Mesh::cell_behind
(maniFEM::Cell, const maniFEM::tag::SurelyExists&) const
```

You may be navigating dangerously close to the boundary of a mesh. See paragraph 9.7.

```
assertion "cll->meshes.size() > dif_dim" failed
-- or --
static size_t ManiFEM::tag::Util::assert_diff(const size_t, const size_t):
Assertion 'a >= b' failed.
```

You are trying to build meshes of dimension higher than those maniFEM expects. Did you redefine this expectation through the statement `Mesh::set_max_dim` ? Along your program, you may have different maximum topological dimensions (that is, you may use `Mesh::set_max_dim` several times) but, at each moment, you can only build meshes of dimension up to the value most recently defined. See paragraph 11.7.

```
gmsh shows an empty drawing
```

Go to `Tools` → `Options` → `Mesh`. For viewing one-dimensional meshes, you need to select `1D Elements`.

```
assertion "( x > 0.04 ) and ( x < 5. )" failed: file "src/frontal.cpp"
```

Most likely, the frontal mesh generation algorithm has gone rogue. Try to use smoother data. "Data" may be the boundary of the domain being meshed, the desired segment length, the Riemann metric. See paragraph 3.15.

11.13. The `DEBUG` mode and other compilation flags

While you develop and test your program, you should compile it in `DEBUG` mode. To achieve this, simply remove any `-DNDEBUG` option from your compilation command (check your `Makefile`). Remember to `make clean` before re-building your application.

When you think your program is ready for shipping, you may want to speed it up by adding the `-DNDEBUG` option to your compilation command (check your `Makefile`). You may also want to add optimization options like `-O3`. Remember to `make clean` before re-building your application.

The garbage collector for cells and meshes (described in paragraph 11.5) can be turned on by adding the `-DMANIFEM_COLLECT_CM` compilation flag in your `Makefile`. To turn it off, remove the `-DMANIFEM_COLLECT_CM` compilation flag. Remember to `make clean` before re-building your application.

11.14. Compiling your own files

To use maniFEM, you should go to `https://codeberg.org/cristian.barbarosie/maniFEM`, choose a release and download all files to some directory in your computer. Current code may be unstable, releases are stable. You can then run the examples in this manual : just `make run-parag-1.1` for the example in paragraph 1.1, `make run-parag-2.7` for the example in paragraph 2.7, and so on. The source files (the `main` program) for the examples in this manual are grouped under the `examples-manual` folder.

You will need a recent `C++` compiler (we use `g++`) and the `make` utility. Under linux it should be easy to install them. It is not that easy to install and use them under Windows, but it is certainly possible, for instance by using `cygwin`, available at `https://www.cygwin.com/`. Some examples use the `Eigen` library; just copy its source tree from `https://eigen.tuxfamily.org/index.php?title=M` to some place in your computer and be sure that path is mentioned in your `Makefile` under the `-I` flag of your compiler. You may also want to use `gmsh`, available at `https://gmsh.info/`, for visualization purposes.

Frequent compilation errors are discussed in paragraph 11.11.

The source files of maniFEM are grouped in the directory `src`; the `Makefile` is in the root directory of the source tree, together with some other utility files. The source code of the `main` function for the examples in this manual can be found in the `examples-manual` directory. The `Makefile` expects the source files to be at these locations.

The user may naturally want to write their own `main` files. There are three ways to do this. A quick solution is : you create your own source files and save them in the root directory of maniFEM under a name matching `user-*.cpp`, then invoke `make run-user-*`.

The other two solutions consist in keeping maniFEM's files separate from the files written by the user. Here is how to use statically linked object files. You must create a static library file, in the maniFEM source code root directory, by issuing the command `make static-lib`. Then, in your working directory, you should create a `Makefile` similar to the one below (assuming your code is in a file `myprog.cpp`). After that, it suffices to issue the command `make run`.

Makefile
```
# C++ compiler
CC = g++

manifem_dir = the root directory of maniFEM

# compiler flags
# CFLAGS = -c -I . -I $(HOME)/include/ -I $(manifem_dir) -std=c++17
CFLAGS = -Wshadow -Wall -c -I . -I $(HOME)/include/  -I $(manifem_dir) -std=c++17
```

```
# CFLAGS = -DMANIFEM_COLLECT_CM -Wshadow -Wall -c -I . -I $(HOME)/include/  -I $(manifem_dir) -s
# CFLAGS = -DMANIFEM_COLLECT_CM -DNDEBUG -O3 -c -I . -I $(HOME)/include/ -I $(manifem_dir) -std=
# CFLAGS = -DNDEBUG -c -O3 -I . -I $(HOME)/include/ -I $(manifem_dir) -std=c++17
# CFLAGS = -DNDEBUG -c -I . -I $(HOME)/include/ -I $(manifem_dir) -std=c++17

%.o: %.cpp
	$(CC) $(CFLAGS) $^

a.out: myprog.o
	$(CC) $^ -L$(manifem_dir) -lmaniFEM -o a.out

run: a.out
	./$<

.SECONDARY:
```

You should be careful to use the same compilation options when compiling the library and when compiling your main file (see paragraph 11.13). You may build two (or more) versions of the library, compiled with different options, and keep them by copying the file `libmaniFEM.a` under appropriate names. Remember to `make clean` every time you want to re-build the library. Of course you should taylor your `Makefile` (in your own working directory) by replacing the option `-lmaniFEM` with the name of the particular version of the library you want to use.

You may want to copy or move files `*.h` to your personal (or system) `include` directory and `libmaniFEM.a` to some `lib` directory.

A third solution is to use dynamically linked object files, but this is slightly more complicated. See e.g. `https://iq.opengenus.org/create-shared-library-in-cpp`

12. Internal details

This section contains material intended to assist the developer in reading the source code of maniFEM. It contains mainly drawings which document specific parts of maniFEM.

12.1. Building a chain of segments

One of the simplest meshes is an open chain of segments. The `Mesh` constructor with `tag::segment` receives two vertices (a negative one and a positive one) and the desired number of segments and builds the chain. New vertices are built by using the `Cell` constructor with `tag::vertex`. Space coordinates of each new vertex are defined by interpolating the coordinates of the two extremities of the chain. New segments are built by means of constructor `Cell ( tag::segment, A, B )`, where `A` is a `Cell::Negative::Vertex` and `B` is a `Cell::Positive::Vertex`.

Note that the interpolation operation is a method belonging to `Manifold::working` which is the most recently declared `Manifold` object. For a Euclidian manifold, this is just a convex combination of the values of the coordinates. However, for other manifolds it may be a more complex operation. For an implicit manifold, it involves a projection operation, as explained in paragraph 8.1.

This constructor builds a `Mesh::Connected::OneDim` (see paragraph 11.6). Thus, we can later retrieve the starting vertex through methods `Mesh::first_vertex` (which provides a negative vertex `Cell`) and `Mesh::last_vertex` (which provides a positive vertex `Cell`).

For meshing curves on a quotient manifold, supplementary arguments must be given: a `tag::winding` followed by the winding number of the segment to be built. If this winding number is zero, these supplementary arguments may be omitted. Several examples are shown in section 7.

12.2. Building a mesh of rectangles

Constructor `Mesh::Mesh ( tag::rectangle,...)` builds a rectangular mesh from its four sides (which are one-dimensional meshes, more precisely, open chains of segments). No need to provide the number of divisions, the four sides already have their internal divisions. Of course, opposite sides should have the same number of (segment) cells.

Paragraphs 1.4, 1.5 and many others (e.g. in section 2) show the use of this constructor.

ManiFEM chooses the orientation of the mesh based on the orientation of the four sides, as provided by the user. These orientations must be mutually compatible, i.e. last vertex of a side must be equal to (the `reverse` of) the first vertex of the side following it. If we switch all four orientations, maniFEM will produce the same two-dimensional mesh with opposite orientation. If we switch the orientation of some, but not all, of the sides, maniFEM will complain that the orientations are not compatible with each other.

Actually, the name `rectangle` is misleading. Perhaps a better name would be `quadrangle` but even this is not general enough to describe the constructor's ability to build curved patches like the ones shown in paragraphs 1.1, 2.8 – 2.10, 2.12 or 2.13. Tags `rectangle`, `quadrangle` and `quadrilateral` can be used interchangeably. However, high distortions (e.g. high curvature) of the four sides should be avoided; you should use either smaller patches, to be subsequently `joined`, or frontal mesh generation if you are willing to accept triangles instead of quadrilaterals.

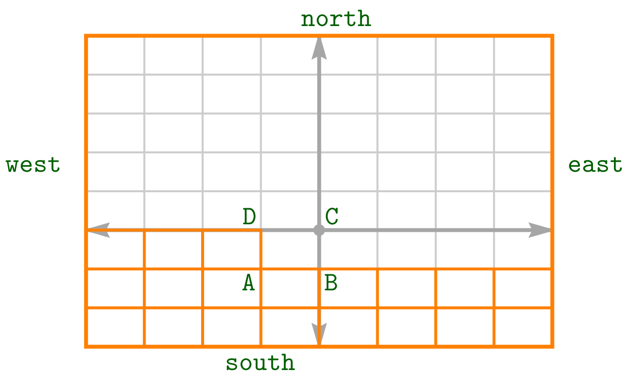


Figure 12.1.: meshing a rectangle

The coordinates of each new vertex are defined by interpolating the coordinates of four vertices on the four given sides, as shown in figure 12.1. The complicated formula for the coefficients (involving α^3 and β^3) is there to ensure a smooth transition for the distribution of inner vertices in the case of non-uniform distribution along one or several sides, or to the case of curved sides. Paragraphs 2.1, 2.10, 2.12 and 2.18 show examples where this is useful.

The remarks made in paragraph 12.1 about the interpolation operation hold here. Paragraphs 2.8, 2.12 and 2.15 show examples where the interpolation operation consists of a convex combination followed by a projection. For a manifold defined through an external parameter, the convex combination is performed on the external parameter and then the space coordinates are computed accordingly (paragraph 2.23 shows such a situation).

Providing a `tag::with_triangles` makes the constructor cut each rectangle in halves; results are shown in paragraphs 2.4, 2.9 and 2.15.

For meshing domains of a quotient manifold, a supplementary argument `tag::winding` must be given. No need to provide any winding number, this information is already contained in the four sides. Examples are shown in paragraphs 7.3 – 7.8, 7.10, 7.14, 7.17, 7.18.

12.3. Building a mesh of triangles

Building a triangular mesh involves an algorithm much alike the one for a rectangular mesh. A noteworthy difference is that the interpolation operation involves now six vertices on the boundary of the triangle, as shown in figure below.

The coefficients for the interpolation have simpler formulas when compared with the ones in the constructor of a rectangle (presented in paragraph 12.2); the fact that we interpolate from six vertices compensates that.

Examples are presented in paragraphs 1.5, 1.6, 2.2, 2.7, 2.11 and 2.15.

Just like for a rectangular mesh (see paragraph 12.2), maniFEM chooses the orientation of the mesh based on the orientation of the three sides, as provided by the user. Also, the number of divisions is inferred from the three sides.

These three sides may be curves in the plane $\mathbb{R}^2$ or in the space $\mathbb{R}^3$ (or even in higher dimensional spaces). However, high distortions (e.g. high curvature) of the sides should be avoided; you should use either smaller patches, to be susbequently `joined`, or frontal mesh generation.

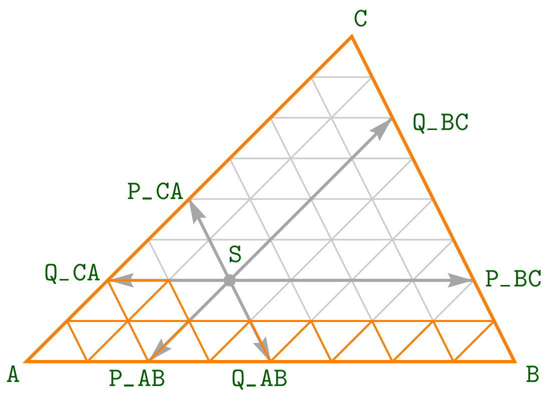


Figure 12.2.: meshing a triangle

For meshing domains of a quotient manifold, a supplementary argument `tag::winding` must be given. No need to provide any winding number, this information is already contained in the three sides. Examples are shown in paragraphs 7.9, 7.15, 7.16.

12.4. Building a mesh of cubes

The process of meshing a cube is similar to meshing a square; it is done layer by layer. Unlike for a square, where we have used an `std::list` of `Cell`s `horizon`, here we use a `Mesh` `front` which keeps the newly built faces. The advantage of using a `Mesh` instead of a `list` of `Cell`s is that we can navigate through it as described in paragraph 9.6. When we are on a quotient manifold (situation signaled by the `tag::winding`), the six faces provided as arguments may be in very awkward positions; this makes it impossible to use one `Mesh` `front` and we have to use three distinct meshes instead.

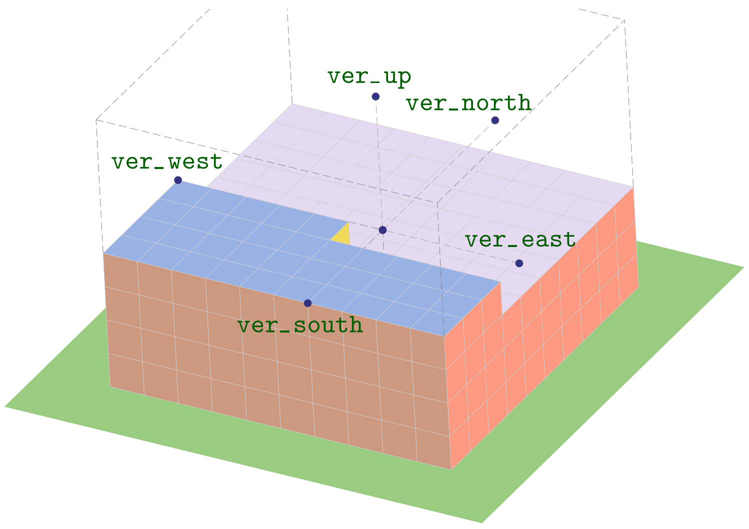


Figure 12.3.: meshing a cube

Just like for a rectangular mesh (see paragraph 12.2) or for a triangular one (see paragraph 12.3), *maniFEM* chooses the orientation of the mesh based on the orientation of the six faces, as provided by the user. Also, the number of divisions is inferred from the faces.

The six faces may be curved. However, faces with high distortions (e.g. high curvature) should be avoided; you should use smaller patches, to be subsequently `joined`. Frontal mesh generation is not yet implemented for volumes.

For meshing domains of a quotient manifold, a supplementary argument `tag::winding` must be given. No need to provide any winding number, this information is already contained in the six faces (or rather, in the twelve edges).

12.5. Building a mesh of tetrahedra

It is possible to fill a big triangle with many small identical triangles, as shown in paragraph 12.3. It is possible to fill a big rectangle with many small identical rectangles, as shown in paragraph 12.2. It is possible to fill a big cube with many small cubes, as shown in paragraph 12.4. But it is not possible to fill, say, a big regular tetrahedron with many small regular tetrahedra. This is why there is no `Mesh` constructor with `tag::tetrahedron`.

12.6. Frontal mesh generation

Several constructors `Mesh::Mesh ( tag::frontal,...)` are defined in `frontal.cpp`; they build a mesh on a given manifold starting with almost nothing. For a one-dimensional manifold, a mesh of segments is built. For a two-dimensional manifold, triangular cells are used. For a three-dimensional manifold, tetrahedral cells are used.

Frontal mesh generation on a two-dimensional manifold starts with the boundary of the future mesh, then moves this interface with small steps, building the mesh behind it like a spider. Actually, if the manifold is compact (like the sphere or the torus) and we want to mesh all of it, there will be no boundary. In this case we can start the process by providing nothing more than the manifold itself (or not even that – the current working manifold will be meshed in this case). The mesh follows the shape of the manifold; this is achieved by `projecting` newly created vertices on the working manifold, as explained in paragraph 8.1. Section 3 gives several examples.

This is a complex process, much more complex than building a regular mesh of rectangles (as described in paragraph 12.2) or of triangles (as described in paragraph 12.3).

The initial interface may be disconnected. Even if the initial interface is connected, it may become disconnected during the meshing process, if it touches itself (paragraph 12.11 discusses this event).

Detecting such touching points requires the evaluation of the distance between many pairs of points. This may become extremely time consuming, unless special care is taken. Points belonging to the interface are organized in a hierarchy allowing one to eliminate many pairs of points from the evaluation process. Paragraph 12.14 describes this hierarchy.

The next few paragraphs describe specific parts of this process of frontal mesh generation.

12.7. The metric

One of the first things the frontal meshing algorithm does is to change the metric of the current working manifold, by re-scaling it with the inverse of the desired segment length which was provided by the user after `tag::desired_length`. This way, the rest of the algorithm seeks to build segments of length as close to 1 as possible (measured by the modified Riemann metric).

One of the advantages of this approach is that the bulk of the mesh generation algorithm is independent of the type of metric. The code is the same for a uniform or a non-uniform metric, or even for an anisotropic metric.

12.8. Measuring angles

We often need to measure angles between adjacent segments of the interface (when building a 2D mesh) or between adjacent triangles of the interface (when building a 3D mesh). For efficiency reasons, we prefer to avoid invoking (inverses of) trigonometric functions. Thus, we define two functions `angle_2d` and `angle_3d` which give an indication of the angle by computing only inner products between vectors (tangent vectors and normal ones).

For a two-dimensional mesh (one-dimensional interface), we look at adjacent segments `AB` and `BC`. Denote by $\vec{\tau}_{\mathtt{AB}}$ and $\vec{\tau}_{\mathtt{BC}}$ the tangent vectors (of norm 1) and by $\vec{n}_{\mathtt{AB}}$ and $\vec{n}_{\mathtt{BC}}$ the normal vectors (or norm 1), as depicted in figure 12.4. The situation for a three-dimensional mesh (two-dimensional interface) is similar. Paragraph 12.9 explaines the orientation of normal vectors.

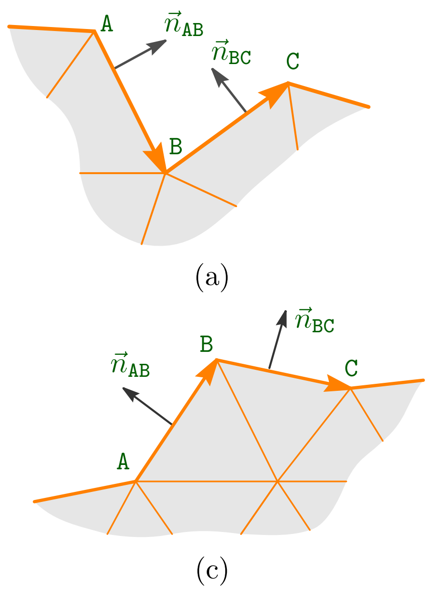


Figure 12.4.: measuring angles

We have chosen the quantity below (varying between −3 and 3) to describe the angle:

$$q = \begin{cases} \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} - 2\,, & \text{if } \vec{\tau}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} \leq 0 \text{ and } \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} \leq 0\,, \quad \text{figure 12.4 (a)} \\ \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} + \vec{\tau}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} - 1\,, & \text{if } \vec{\tau}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} \leq 0 \text{ and } \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} > 0\,, \quad \text{figure 12.4 (b)} \\ 1 - \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} + \vec{\tau}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}}\,, & \text{if } \vec{\tau}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} > 0 \text{ and } \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} > 0\,, \quad \text{figure 12.4 (c)} \\ 2 - \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}}\,, & \text{if } \vec{\tau}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} > 0 \text{ and } \vec{n}_{\mathtt{AB}} \cdot \vec{n}_{\mathtt{BC}} \leq 0\,, \quad \text{figure 12.4 (d)} \end{cases}$$

Figure 12.5 shows the behaviour of q as the angle between `AB` and `BC` widens; the origin corresponds to a flat boundary.

12.9. The normals

The meshing process described in paragraph 12.6 uses a set of normal vectors. Each cell in the interface has associated to it a vector tangent to the working manifold and normal to the interface. We can think of the interface as a curve embedded in the current working manifold if this manifold has two dimensions, or as an embedded surface if the current working manifold

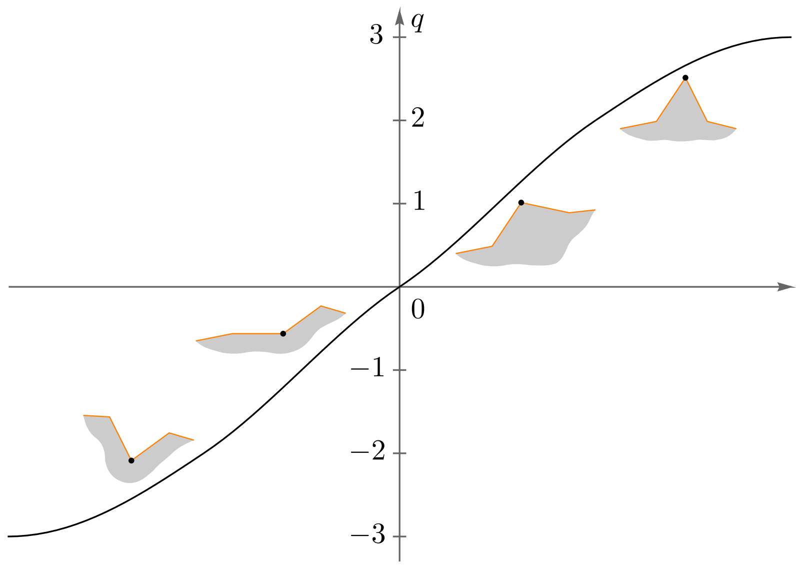


Figure 12.5.: behaviour of q

has three dimensions.[1] These normal vectors provide the sense in which we want the mesh to grow (towards one side or the other of the interface). They are used for defining the location of each new vertex.

For a two-dimensional mesh in $\mathbb{R}^2$ or a three-dimensional mesh in $\mathbb{R}^3$, the normals are built using the intrinsic orientation. In $\mathbb{R}^2$ a mere $90°$ rotation of the segment is enough. In $\mathbb{R}^3$ we need to use the exterior product of two vectors. If the metric is anisotropic, the normal thus obtained must be corrected (in three dimensions, even if the metric is isotropic, the exterior product must be rescaled). Classes `ExtProd2d` and `ExtProd3d` (hidden in anonymous namespace) deal with this construction through their method `get_normal_vector`.

The "correction" above referred (when the metric is anisotropic) consists of multiplying the rotated vector (or the exterior product), to the left, by $\det A \; A^{-2}$ (that is, by $\sqrt{\det M} \; M^{-1}$), using the notations in paragraphs 3.27 and 3.28.

Note that the interface is "looking outwards", towards the zone where we do not intend to build triangles/tetrahedra (either because this zone has already been meshed or because it is not part of the domain to be meshed). Method `get_normal_vector` returns the reversed normal, pointing towards the zone where we intend to build triangles/tetrahedra.

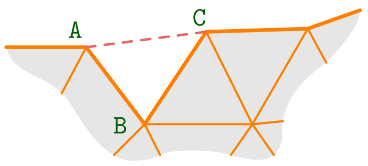
The process is quite different when building a two-dimensional mesh in $\mathbb{R}^3$ (we call this situation "co-dimension one"). Often, at the beginning of the process only one segment has an associated normal vector. maniFEM then propagates this normal vector to the neighbour segments, walking along the current connected component of the interface. This means that, if there are other connected components, they will have no normal vectors. When the current connected component of the interface touches other connected components (this event is discussed in paragraph 12.11), maniFEM propagates the normal vectors to the new connected component. Each time a new segment is added to the interface, the normal associated to the new segment must be computed (inferred from information at neighbour segments). The class `NormalContainer` (hidden in anonymous namespace) deals with this process.

[1] Not to be confused with the geometric dimension. For instance, if we are trying to mesh a surface in $\mathbb{R}^3$, the current working manifold has (topological) dimension two, while the geometric dimension (the number of coordinates) is three.

12.10. Filling triangles

Perhaps the simplest part of the frontal meshing process with triangles is just walking along the interface and adding new triangles.

A trivial case is when an angle is encountered which is close to 60° (see figure 12.6 left). Then we only have to fill the space with a new triangle. Of course, before creating this new triangle a new segment `AC` must be created (no new vertex is needed). Segments `AB` and `BC` are removed from the interface then `AC` is added. Its normal vector must be computed (see paragraph 12.9).



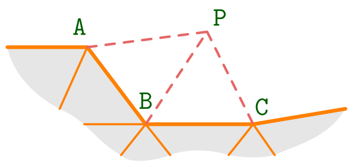
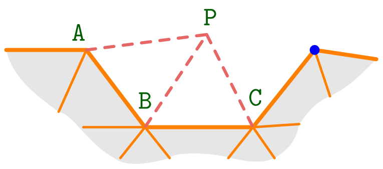


Figure 12.6.: filling a 60° angle and a 120° angle

Another situation is when an angle is encountered which is close to 120° (see figure 12.6 right). A new vertex `P` is created; its position is defined based on the two normals of the adjacent segments. Three new segments `AP`, `BP` and `CP` are created, then two new triangles are created and added to the mesh under construction. Segments `AB` and `BC` are removed from the interface then `AP` and `PC` are added to the interface (their normals must be computed).

The algorithm then searches for vertices close to `P`, building a list of them with the aid of the `MetricTree` described in paragraph 12.14. These may include, besides `A`, `B` and `C`, other vertices, marked by blue dots in figure 12.7. The algorithm improves the location of `P` by trying to make all these distances as close to 1 as possible (see paragraphs 12.7 and 12.12).



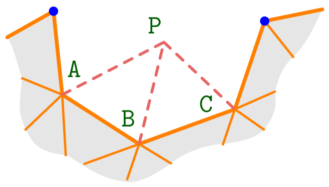


Figure 12.7.: blue dots show vertices close to `P` (besides `A`, `B`, `C`)

In certain instances, the algorithm decides to create a "bridge" between `P` and one of these blue-dotted vertices. This process is described in paragraph 12.11.

Finally, if all angles of the current connected component of the interface are wide, the algorithm creates a triangle "out of the blue", like in figure 12.8. Segment `AB` is removed from the interface then `AP` and `PB` are added to the interface (their normals must be computed).

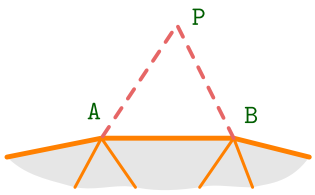


Figure 12.8.: creating a new triangle

Again, the position of the newly created angle is iteratively improved such that distances to nearby vertices are as close to 1 as possible (see paragraph 12.7); a bridge may be created (see paragraph 12.11).

12.11. Creating a bridge

After the creation of a new vertex `P` during one of the operations described in paragraph 12.10, the algorithm may detect vertices geometrically close to `P` but far from it from the viewpoint of the connectivity of the interface (see figure 12.9). They may even belong to new connected components of the interface, not yet dealt with. The `MetricTree`, described in paragraph 12.14, helps to search for these vertices in an efficient manner.

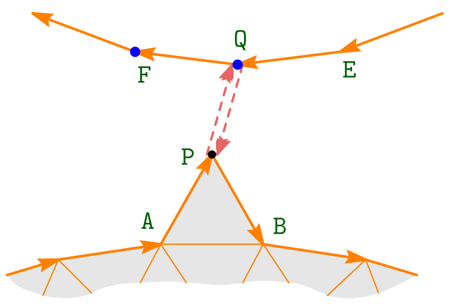


Figure 12.9.: creating a bridge

Among these vertices, the algorithm chooses the one closest to `P` (call it `Q`) and builds a bridge. By "bridge" we mean a pair of segments, one pointing from `P` to `Q` and another one pointing from `Q` to `P`. They are added to the interface which thus gains a strange configuration. Vertex `P` has four neighbour segments and the same holds for `Q`. These segments are interconnected in a specific order. Before the creation of the bridge, `AP` was followed by `PB` while `EQ` was followed `QF`. After the creation of the bridge, `AP` is followed by `PQ` then by `QF` while `EQ` is followed by `QP` then by `PB`. The class `Mesh::STSI`, described in paragraph 12.16, is able to deal with such a strange configuration.

Ideally, the segment `QP` should be the reverse of `PQ`. However, none of the types of meshes implemented in *maniFEM*, not even `Mesh::STSI`, allows this. This is why `QP` is initially created as a separate segment, sharing extremities with `PQ` but otherwise independent. This pair of segments is kept in the map `dummy_faces`. Later, when adjacent triangles are created, `QP` will be replaced by `PQ .reverse()`.

12.12. About `relocate`

The coordinates of each newly created vertex `P` are initially computed based on a number of faces in the interface (by "faces" we mean segments if we are building a mesh of triangles, triangles if we are building a mesh of tetrahedra). After that, a rather complicated algorithm is applied in order to relocate (improve the position of) the vertex. When building a mesh of tetrahedra, this process involves creating new tetrahedra around the vertex, while relocating it at the same time.

12.13. Filling tetrahedra

When meshing frontally a three-dimensional domain, the user must provide a two-dimensional mesh with no boundary (a closed surface) made of triangles; maniFEM will produce a mesh of tetrahedra filling the region of $\mathbb{R}^3$ bounded by that closed surface.

Of course the algorithm for building tetrahedra is more complex than the one for triangles. It looks for "cavities" in the interface and tries to fill them with tetrahedra. Newly created vertices are relocated in order for distances to other nearby vertices to be close to 1, but distances to triangles are also taken into account, as well as certain angles.

While in two dimensions we build bridges in order to allow for topological changes, in three dimensions we build filaments and membranes. A membrane is simply a pair of triangles sharing the same sides, with opposite orientations; it can be viewed as the 3D equivalent of a bridge. A filament is a pair of degenerated faces, having two sides. Just like for membranes, these sides are shared, with opposite orientations.

We do not include drawings about three-dimensional frontal meshing in this manual because such drawings are difficult to understand in static form. We prefer to keep files in the `msh` format within the repository `https://codeberg.org/cristian.barbarosie/manifem-manual`. The user can then interact with these files using e.g. the `gmsh` software. These files are refered to in the source code (file `frontal-3d.cpp`).

12.14. The `MetricTree`

During the mesh generation process described in paragraphs 12.6 – 12.11, we have to check frequently the distance between a newly built vertex and all other vertices of the evolving interface (including other connected components). A direct comparison with all vertices would be very time consuming, so we rely on a tree-like structure which we call `MetricTree` and which eliminates many vertices from the list of candidates.

The `MetricTree` is similar to quad- and oct-trees with two differences : there is no assumption on the geometric dimension and the zones overlap. It is similar to m-trees, just not balanced.

It works for a general metric space. Triangular inequality is assumed, as well as symmetry. Because it is not balanced, it deals well with non uniform clouds of points, that is, with clouds having zones with high density of points along with zones where the points are spread at large distances.

There is no upper limit on the number of children. Actually, the average number of children can be interpreted as a hint about the dimension of the metric space (in the spirit of Hausdorff dimension).

`MetricTree` has been implemented with the intent of having wide usability, independently[2] of maniFEM. It is templated over the type of `Points` (the metric space) and over a callable object returning the square of the distance between any two `Points`.

We focus on the square of the distance rather than on the distance itself because it is numerically cheaper (we prefer not to compute square roots). Of course there are parts of the code where the true distance must be used (e.g. when it comes to the triangular inequality). Even there, simple algebraic manipulations allow us to avoid computing any square root.

The user interacts with the `MetricTree` through four methods. The constructor sets the rank-zero distance and the ratio between successive distances (see below). Method `add` adds a `Point` to the cloud and returns a pointer to a `Node` which the user must keep and later provide to the `remove` method. The `remove` method removes a `Node` from the `MetricTree`. Finally, and most

[2] `MetricTree` is available separately at `https://codeberg.org/cristian.barbarosie/MetricTree`

importantly, method `find_close_neighbours_of` receives a `Point P` and a distance threshold and returns a list of `Nodes` near `P`. It is irrelevant whether `P` belongs or not to the cloud (if it belongs, it will show up in the returned list, disguised as a `Node`). The user can recover (a copy of) the `Point` from a `Node` using the attribute `point`. We have not implemented a method for finding the n nodes closest to a given point (we do not need such an operation for frontal mesh generation).

It is assumed that `Points` are cheap to copy. If this is not the case for your `Points`, use pointers. *maniFEM* uses wrappers, which are a sort of pointers (see paragraph 11.4).

Each node represents a point. Leaves have no special status. Each node has a rank which is an integer, possibly zero, possibly negative. Children of a node `N` have rank equal to `rank[N]-1`. Nodes with rank zero have no special status. Leaves may have any rank, positive, zero or negative. There is a `root` of course (a node with no parent). The root has the highest rank. Even the rank of the root may be negative (which means all nodes have negative ranks).

To each rank there is a distance `dist` associated. Children of a node `N` are no farther than `dist[rank[N]]` from `N`. Note that a point `P` being at distance less than `dist[rank[N]]` from `N` does not imply that `P` must be a child (not even an indirect descendant) of `N`. In other words: zones overlap.

`dist[k]` is a geometric sequence with ratio `ratio`. `ratio` must be greater than two; we recommend some value between five and ten. So, `dist[k] == ratio`$^{\texttt{k}}$ `dist[0]`. Recall that `k` may be negative.

Besides the `distance`, to each rank `k` we associate a `range` which represents the sum of distances of that rank and of all lower ranks. That is,

```
range[k] == dist[k] / ( 1 - 1 / ratio );  range[k] > dist[k].
```

This means that an indirect descendant of a node `N` cannot be farther than `range[rank[N]]` from `N`. Again, the reverse may be false.

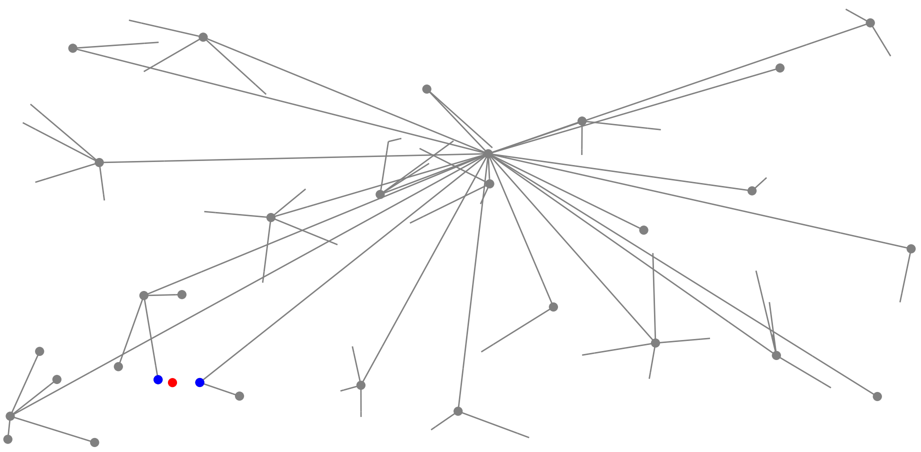


Figure 12.10.: metric tree

Figure 12.10 shows a cloud of (randomly generated) points in $\mathbb{R}^2$ organized in a `MetricTree`. It also illustrates the process of `find_close_neighbours_of` a given point. This method takes two arguments: a `Point` (drawn in red) and a distance. It returns a list (drawn in blue) of all `Nodes` in the cloud which are close enough to the red point (closer than the given distance). It is irrelevant whether the red point belongs to the cloud or not; in figure 12.10 it does not.

Figure 12.10 shows grey dots at nodes which have been analysed during the process of `find_close_neighbours` (their distance to the red dot has been computed). Lines without a dot represent nodes in the `MetricTree` which have not even been looked at because their parent has decided (based on the triangular inequality) that they cannot be close enough to the red dot. Their distance to the red point has not been computed, thus alleviating the computational burden.

If you want to run this example on your computer, it suffices to download three files (`Makefile`, `metric-tree-verbose.h` and `parag-12.14.cpp`) from
`https://codeberg.org/cristian.barbarosie/maniFEM`
then `make run-parag-12.14`

12.15. Triangular inequality

The implementation of the `MetricTree`, described in paragraph 12.14, assumes the triangular inequality holds. Without this assumption, there is no guarantee that a call to `find_close_neighbours_of` will find all requested `Node`s.

This rises a problem when non-uniform meshes are built, either by providing a `Function` as desired length, as illustrated in paragraphs 3.23 and 3.24, or by manipulating the Riemann metric of the manifold, as in paragraphs 3.26 – 3.30. The problem shows up when the metric depends on the point (is non-uniform); it is not due to the anisotropy on the metric. First, we shall explain the problem, then we shall explain how we circumvented it.

Specifying a "Riemann metric" on a differentiable manifold X means specifying an inner product on the tangent space of X, at every point $P \in X$. Such an inner product allows one to define the length of a curve Γ embeded in X: if $r : [a, b] \to X$ is a parametrization of Γ, the length of Γ is given by $\int_a^b \sqrt{\left\langle r'(t), r'(t) \right\rangle_{r(t)}}\, dt$, where $\langle u, v \rangle_P$ denotes the inner product between two vectors u and v in the tangent space of X at point P. Subsequently, one may regard X as a metric space by defining the distance between any two points as the infimum of the lengths of all curves joining the two points; this is called "geodesic distance".

The geodesic distance is a very elegant and convenient mathematical construction, but from the computational point of view it is useless. It is prohibitive to compute the distance between two points by measuring the length of all paths between those two points and then choosing the shortest one. This is why *maniFEM* uses a different "distance". Assuming X is a submanifold of $\mathbb{R}^n$, we define the "distance" between two points A and B as $\sqrt{\frac{1}{2}\left\langle \overrightarrow{AB}, \overrightarrow{AB} \right\rangle_A + \frac{1}{2}\left\langle \overrightarrow{AB}, \overrightarrow{AB} \right\rangle_B}$. This quantity is a reasonable approximation of the geodesic distance if the two points A and B are close enough to each other. However, there is no reason for it to satisfy the triangular inequality for points far from each other. Actually, it is easy to prove numerically that, if the metric $\langle \cdot, \cdot \rangle_P$ depends on the point $P \in X$, the above defined quantity infringes the triangular inequality (even if the metric is isotropic).

This means that, although it is OK to use the above defined "distance" in order to measure short segments, we cannot use it in the `MetricTree` (because the algorithm behind `MetricTree` is not local, it deals with points which are far away from each other, and it relies on the triangular inequality). This is why we provide the usual Euclidian distance (squared) when declaring the `MetricTree`.[3] And we must ensure the compatibility between this choice and the fact that the

[3] Ironically, `MetricTree` was designed with the goal of dealing with arbitrary metric spaces (and it does deal). The initial intention was to feed it directly the anisotropic Riemann metric. We later realized that this was not feasible for a non-uniform metric, as explained above, so we ended up using the `MetricTree` with the trivial Euclidian metric.

rest of the frontal mesh generation algorithm uses a modified Riemann metric.

It is easy to ensure this compatibility if the metric is isotropic. We simply provide a rescaled length when we invoke `find_close_neighbours_of`. To prevent any error, we check each point in the returned list with the Riemann metric.

But for an anisotropic mesh things are more complicated. We have chosen the following work-around. We use as scaling factor the inner spectral radius of the matrix M defining the Riemann metric (i.e. a lower bound on its eigenvalues, or, equivalently, an upper bound on the spectral radius of the inverse matrix). Providing separately the "principal part" and the "deviatoric part", as shown in paragraphs 3.29 and 3.30, helps. The principal part is a positive scalar σ times the identity matrix, while the deviatoric part is a matrix D responsible for the anisotropy; D should be semi-definite positive. Desirably, D should be a singular matrix (having a null eigenvalue); this way, the inner spectral radius is equal to σ.

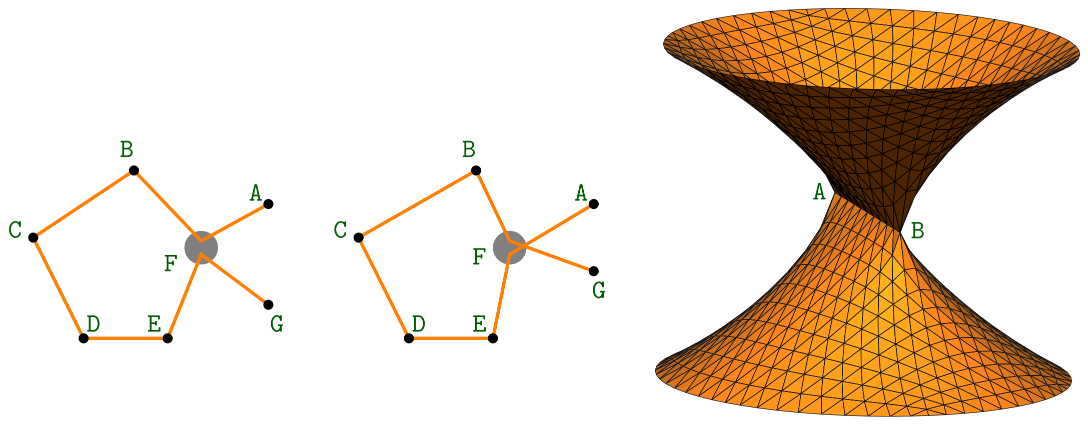
If we provide only the sum M as in paragraph 3.27 (or only $\sqrt{M}$ as in paragraph 3.28), maniFEM will compute, at each step, the inner spectral radius, which may be a heavy computational burden.

Again, we filter the list of points returned by `find_close_neighbours_of` by checking them against the Riemann metric. When the metric is highly anisotropic, the list returned by method `find_close_neighbours_of` will be singnificantly larger than the correct, filtered, list.

See also the limitations of the frontal mesh generation algorithm described in paragraph 3.15.

12.16. STSI meshes

Usually, meshes in maniFEM belong to one of the classes `Mesh::Connected::OneDim` or `Mesh::Fuzzy`. Neither of these classes is able to reach a configuration like `AFBCDEFG` in figure 12.11 (a) or like `AFEDCBFG` in figure 12.11 (b) or like the double coffee filter in figure 12.11 (c). The internal implementation of neighbourhood relations requires that each `face` in a mesh have exactly two neighbours, one behind `face` and another one in front of `face`. This does not happen for vertex `F` in figure 12.11 (a) (b), nor for segments along `AB` in figure 12.11 (c).



(a) (b) (c)

Figure 12.11.: self-crossing, self-touching meshes

We have implemented a different class called `Mesh::STSI` which is more flexible with respect to neighbourhood relations. The name comes from "self-touching, self-intersecting meshes". It can reach the shape of any of the meshes shown in figure 12.11.

This class is not intended for the final user of maniFEM. It is used internally in frontal mesh generation, see paragraph 12.11. It will also be used in the implementation of iterators over high-dimensional connected meshes.

If msh is a Mesh having a Mesh::STSI core, a statement like cll .add_to_mesh (msh) will do things that depend on the configuration of faces of cll. If these faces do not belong already to msh, or if they do belong but have only one neighbour, the same steps as for a Mesh::Fuzzy will be taken. If a face has already two neighbour cells within msh, that face will receive a special treatment. msh will be erased from face .core->cell_behind_within and an entry will be created for face in msh .core->singular. A statement like cll .remove_from_mesh (msh) will do the reverse. In “normal” situations (i.e. for faces having one or two neighbour cells within msh) it will behave just like for a Mesh::Fuzzy. If the face is already singular, the statement may revert it to the “normal” configuration. The above description is imprecise because it does not take orientations into account.

For instance, the configuration in figure 12.11 (a) can be obtained by

```
Mesh msh ( tag::STSI, tag::of_dimension, 1 );
AF .add_to_mesh ( msh );
FB .add_to_mesh ( msh );
BC .add_to_mesh ( msh );
CD .add_to_mesh ( msh );
DE .add_to_mesh ( msh );
EF .add_to_mesh ( msh );
FG .add_to_mesh ( msh );
```

After the first five insertions, msh has a structure identical to a Mesh::Fuzzy (in spite of having a Mesh::STSI core). That is, msh .core->singular is an empty map. After inserting EF, vertex F becomes singular; however, AF is still follwed by FB while EF stays “alone”. The last statement links FG to EF (rather than to AF) simply because EF has no successor and is eager to gain one.

Suppose we add the two statements below :

```
AF .remove_from_mesh ( msh );
FG .remove_from_mesh ( msh );
```

After removing AF, FB remains “alone”. After removing FG in the last statement, maniFEM links the two “orphans” EF and FB and the mesh becomes a closed path (a pentagon) and returns to a state similar to a Fuzzy mesh.

In other words, the connections between cells in a STSI mesh depend on the order of insertions and removals.

If we add tag::force as argument to the method add_to_mesh, maniFEM will break previously existing neighbourhood relations and will give priority to the newly inserted cell with respect to “glueing” to neighbour cells. For instance, the configuration in figure 12.11 (b) can be obtained by

```
Mesh msh ( tag::STSI, tag::of_dimension, 1 );
FB .add_to_mesh ( msh );
BC .add_to_mesh ( msh );
CD .add_to_mesh ( msh );
DE .add_to_mesh ( msh );
EF .add_to_mesh ( msh );
// at this point, 'msh' is a closed path, a pentagon
FG .add_to_mesh ( msh, tag::force );
AF .add_to_mesh ( msh );
```

When inserting FG, the existing neighbourhood relation between EF and FB will be broken and EF will be glued to FG; FB becomes orphan. In the last statement, AF will be linked to FB simply because FB has no predecessor and is eager to gain one.

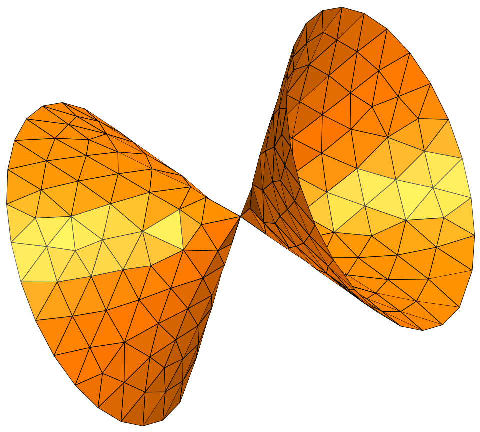


Figure 12.12.: double cone

Incidentally, note that `Fuzzy` meshes can take the shape shown in figure 12.12.

13. Frequently asked questions

13.1. How do I compile my own `main` files ?

See paragraph 11.14.

13.2. Is maniFEM thread-safe ? Does it support parallel processing ?

No special care has been taken with this regard.
If the problem at hand can be split into several smaller problems, then it is of course possible to launch several independent processes simultaneously. [1]

13.3. What's the difference between a mesh and a manifold ?

A `Mesh` is a collection of `Cell`s of the same dimension.
A `Manifold` is a `C++` object modelling an abstract mathematical concept. Recall that in this manual the term "manifold" means a differentiable manifold with no boundary like the straight line $\{(x,y) \in \mathbb{R}^2 : x+y=1\}$ or the paraboloid $\{(x,y,z) \in \mathbb{R}^3 : z = x^2+y^2\}$ or the sphere $S^2 = \{(x,y,z) \in \mathbb{R}^3 : x^2+y^2+z^2 = 1\}$. There are Euclidian `Manifold`s, there are implicitly defined sub-`Manifold`s, there are parametric `Manifold`s and also quotient `Manifold`s.
A `Mesh` may be the discretization of a `Manifold` or of a piece (region) of a `Manifold`.
A `Manifold` is smooth (unless it has singular points, like in paragraphs 3.21, 3.22, 7.15, 7.16), a `Mesh` can be thought of as a polygonal surface.
See paragraphs 1.1, 2.5, 2.12, 2.18, 2.19 and 8.2; see section 7.

13.4. What's the difference between methods `glue_on_bdry_of` and `add_to_mesh`, or between `cut_from_bdry_of` and `remove_from_mesh` ?

For adding a cell to a mesh which is the boundary of some other cell, you should use `glue_on_bdry_of` and `cut_from_bdry_of`. Otherwise, you should use `add_to_mesh` and `remove_from_mesh`.
In some cases (when you are sure there is no mesh "above" that other cell) it makes no difference, so you can use either. For instance, in `Cell` constructors and destructors we have chosen to use `add_to_mesh` and `remove_from_mesh` because they are slightly faster.

13.5. What happens if we try to identify a twisted pair of opposite faces of a parallelogram ?

If, in paragraph 7.19, we ask maniFEM to identify `BC` with `DA` (forgetting to reverse `DA`), maniFEM will complain about orientations.

[1] e.g. as described in section 5 of the paper: C. Barbarosie, A.M. Toader, Optimization of bodies with locally periodic microstructure, Mechanics of Advanced Materials and Strucures 19, p. 290-301, 2012

maniFEM has been designed around the concept of oriented mesh so it will bluntly refuse to mesh a non-orientable manifold. Had it been otherwise, identifying `BC` with `AD` would produce a Möbius strip.

Similarly, in paragraph 7.20, a statement like

```
Mesh Klein = square .fold ( tag::identify, AB, tag::with, CD.reverse(),
                            tag::identify, BC, tag::with, DA,
                            tag::use_existing_vertices                 );
```

would produce a Klein surface if maniFEM were not so stubbornly reliant on oriented meshes.

On the other hand, the statement

```
Mesh projective = square .fold ( tag::identify, AB, tag::with, CD,
                                 tag::identify, BC, tag::with, DA,
                                 tag::use_existing_vertices       );
```

would produce a projective plane if maniFEM were not so reluctant of non-oriented objects.

13.6. Why do we declare coordinates to be of type Lagrange, of degree 1 ?

That's because we want (positive) vertices to have coordinates. Other cells (e.g. segments) will have no coordinates. See paragraph 5.1.

13.7. Will maniFEM support parallel processing in the future ?

At the moment, this is not part of our plans. See also paragraph 13.2.

Changelog

22.01

Lagrange finite elements of degree two, paragraphs 6.6 – 6.9.

22.02

In progressive mesh generation (section 3), change `tag::inherent_orientation` into `tag::` `::orientation`, `tag::inherent` etc. Introduce `tag::shortest_path`.
Introduce constructor `Mesh msh ( tag::import, tag::msh, "filename.msh")`. Rename `Mesh::` `::export_msh` to `Mesh::export_to_file`. Switch to `gmsh` version 4 format.
Introduce constructor `Mesh msh ( tag::cube, ... )`, paragraphs 1.7, 2.14 and 2.17.
Gather source files under `src` directory.

22.03

Fix nasty bug in progressive mesh generation. Still not entirely stable, needs more work.
3D torus, made of cubic cells, paragraphs 7.11 – 7.13.
`Cell` constructor with `tag::vertex`, `tag::of_coords`, `tag::project`.
Zero-dimensional `Manifold`s, paragraphs 2.13 and 7.11.
Change `tag::progressive` into `tag::frontal`.
STSI meshes work in an incipient setting.

22.04

Specific garbage collector implemented for `FiniteElement`s and `Integrator`s. We can declare and use an integrator without bothering with the subjacent finite element, paragraphs 1.1 and 6.2.
Add `Mesh` constructor with `tag::quadrangle`, `tag::winding`, `tag::singular`, in the spirit of paragraph 7.16.

22.05

Multifunctions, paragraphs 7.27 – 7.31.
Intersection of `Manifold`s, paragraphs 2.13, 3.17, 3.18, 3.19.
Overlapping meshes, paragraph 3.20.

22.08

Migrate from `github.com` to `codeberg.org`.
New implementation of frontal mesh generation, more robust and easier to generalize for quotient manifolds and for 3D meshing.
Anisotropic Riemann metric, paragraph 3.27.
Frontal meshing on quotient manifolds, paragraphs 7.23 – 7.26.

22.09

Declare the `MetricTree` with the trivial Euclidian distance on $\mathbb{R}^n$, even if we work on a submanifold of $\mathbb{R}^n$ with a different Riemann metric, paragraph 12.15. Rescale each call to `find_close_neighbours_of` if the metric is isotropic, paragraphs 3.23 – 3.26. Use the inner spectral radius if the metric is anisotropic, paragraphs 3.27 – 3.30.
Hide name `Field` into namespace `tag::Util`.

23.10

Implement frontal meshing with tetrahedra.
Many changes in `frontal.cpp` not visible to the final user. In the case of co-dimension one, the normals are propagated from each segment to neighbour segments. In the pure 2D case, the normal is obtained simply by rotation of the segment. In the pure 3D case, the normal is computed by rescaling the exterior product of two segments. Anisotropic metrics are given a special treatment.

23.11

Fix bugs in frontal mesh generation with tetrahedra.
Implement class `Mesh::Composite`, section 4.

24.02

Fix bugs in frontal mesh generation with tetrahedra.
Improve class `Mesh::Composite`.

24.03

Fix bugs in frontal mesh generation with tetrahedra.

24.05

Add tetrahedral finite elements, Lagrange P1.
Improve hand-coded integrators by using one `std::array` instead of nested `std::vectors` as well as `constexpr size_t` indices, hopefully allowing the compiler to produce code optimized for speed.
Change method `Mesh::draw_ps` to `Mesh::export_to_file` with `tag::eps`.

To do

Increase efficiency of the frontal meshing algorithm with tetrahedra. Implement post-processing for improving mesh quality.
Implement intersection of `Mesh`es (whose dimension depends on the way the two meshes touch).
Add more types of finite elements (lots of them).
Implement Gauss quadrature on triangles with 7, 12 and 13 points.
High-dimensional connected meshes; iterators based on STSI meshes; similar to, but simpler than, frontal mesh generation.
Define regions in a manifold (i.e. manifolds with boundary) by means of inequalities between `Function`s (partially done, see methods `Mesh::unfold` and `Mesh::export_to_file` with `tag::eps`, `tag::unfold`).
Implement frontal mesh generation with triangles for geometric dimension higher than three.
Implement locally Delaunay mesh generation, starting from a cloud of points.

Index

A gray paragraph number indicates that that concept can be found not in the text of the manual but in the source code of that example.