# maniFEM

## USER MANUAL

author of this manual : **Cristian Barbarosie**
version of this manual : **21.02**
describes maniFEM version **21.02**

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 has been 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 has been 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 uses `Eigen` for storing matrices and solving systems of linear equations, see `http://eigen.tuxfamily.org/index.php?title=Main_Page`. Many drawings in this manual have been produced by `gmsh`, see `http://gmsh.info/`

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++`.

In its current version, 21.02, maniFEM works quite well for mesh generation. Quotient manifolds (section 5) and anisotropic meshing (paragraph 3.24) are not yet implemented. Variational formulations (section 6) are not yet implemented. Finite elements (section 7) work in a rather rudimentary manner for now. 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 9.15.

*Manifem* is being developed by Cristian Barbarosie,* Sérgio Lopes and Anca-Maria Toader. This work 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.**



The home page of *manifem* is `https://webpages.ciencias.ulisboa.pt/~cabarbarosie/manifem/` (where this manual can be found).

To use *manifem*, visit `https://github.com/cristian-barbarosie/manifem` and copy all files under `src/` to some directory in your computer. You can then run the examples in this manual : just `make run-1.1` for the example in paragraph 1.1, `make run-1.3` for the example in paragraph 1.3, and so on. 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/`. For some examples, the `Eigen` library is necessary; just copy its source tree from `http://eigen.tuxfamily.org/index.php?title=Main_Page` 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 `http://gmsh.info/`, for visualization purposes.

This manual is divided in sections describing *manifem* with increasing degree of technical detail. Section 1 is a quick overview. Section 2 shows meshes built by joining simple shapes, like patches, some of them on manifolds. Section 3 shows how to build meshes starting from their boundary alone, some of them on manifolds. Section 5 describes meshes on quotient manifolds (code not working). Section 6 (on functions and variational formulations) and section 7 (on finite elements) are still very incipient. These sections should be accessible to readers who have some knowledge of `C++` but are not necessarily experts in `C++` programming. Section 8 gives some insight on the implementation of cells and meshes in *manifem* (it should be useful for users who want finer control on the mesh, e.g. for implementing their own remeshing algorithms). Sections 9 and 10 give technical details, mainly for those interested in developing and extending *manifem*.

* `cristian.barbarosie@gmail.com`

** project UID/MAT/04561/2020

ISBN 978-972-8394-30-1

Universidade de Lisboa
Faculdade de Ciências
Departamento de Matemática

first edition, February 2021

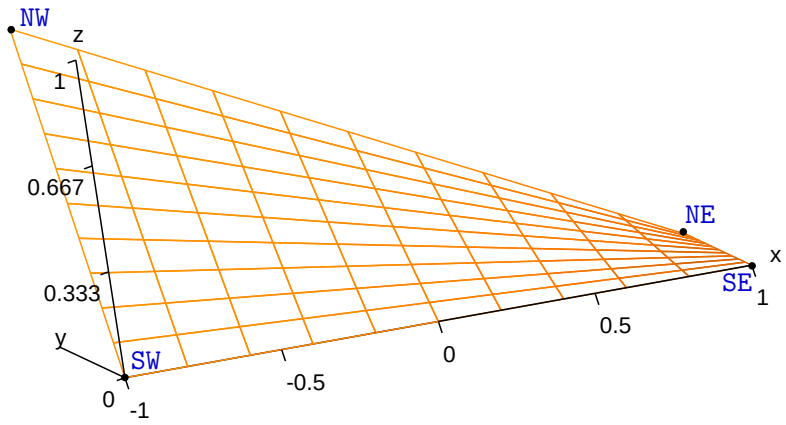
Table of contents

# 1. General description

This section is a quick tour through *maniFEM*'s capabilities.

## 1.1. An elementary example

In this paragraph, we show how to build a rectangular mesh on a surface in $\mathbb{R}^3$ and then compute the integral of a given function. Paragraph 1.3 shows a purely two-dimensional example.




```
#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;

   // 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_msh ("rectangle.msh");

   // Let's define a symbolic function to integrate
   Function f = x*x+1/(5+y);
   // and compute its integral on the rectangle,
   // using Gauss quadrature with 9 points

   // code below does not work yet

   // Integrator integ ( tag::Gauss, tag::Q9 );
   // cout << "integral = " << f.integrate ( rect_mesh, integ ) << endl;

}  // end of main
```

After running this program (through `make run-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 9.2 for some details.

When declaring a segment `Mesh`, we must `reverse` the first vertex (paragraph 1.2 discusses the `reverse` method). Paragraph 1.3 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. It is 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 domain 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.

A mesh is roughly a collection of cells of the same dimension. Internally, *manifem* keeps 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 this is what makes the classes fast, especially for remeshing.

A cell of dimension higher than zero is defined by its boundary, which in turn is a mesh of lower dimension. 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 (of course these orientations must be mutually compatible). Although this is not how the orientation is implemented internally (see paragraph 9.6), 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.

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 8.7 gives more details about the orientation of cells. See also paragraph 9.3.

Cells are topological entities; they carry no geometric information. In particular, points do not have coordinates. Coordinates are stored externally, see paragraph 7.1.

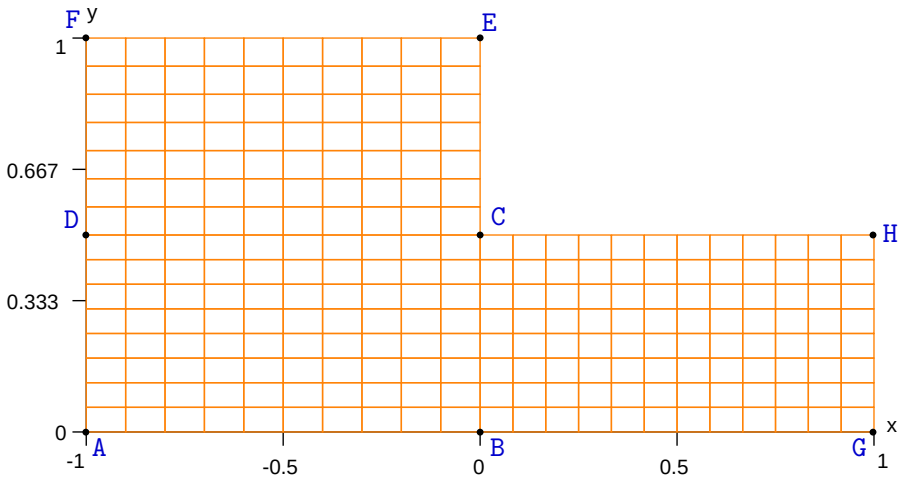
`Mesh`es have a `reverse` method, too. It is used mainly when we want to `join` meshes having a common piece of boundary; see e.g. paragraph 1.3.

## 1.3. 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 :

```
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 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) =  0.;  y(C) = 0.5;
Cell D ( tag::vertex );  x(D) = -1.;  y(D) = 0.5;
Cell E ( tag::vertex );  x(E) =  0.;  y(E) = 1.;
Cell F ( tag::vertex );  x(F) = -1.;  y(F) = 1.;
Cell G ( tag::vertex );  x(G) =  1.;  y(G) = 0.;
Cell H ( tag::vertex );  x(H) =  1.;  y(H) = 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 );
```




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

```
L_shaped.export_msh ("L-shaped.msh");
L_shaped.draw_ps ("L-shaped.eps");
```

Note that, in *manifem*, cells and meshes are oriented. To build `CEFD` one must use not `CD` but its reverse; to build `BGHC` one must use not `BC` but its reverse.

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

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

cannot be joined* 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).

See paragraph 8.2 for a more complex use of the `Mesh` constructor with `tag::join`.

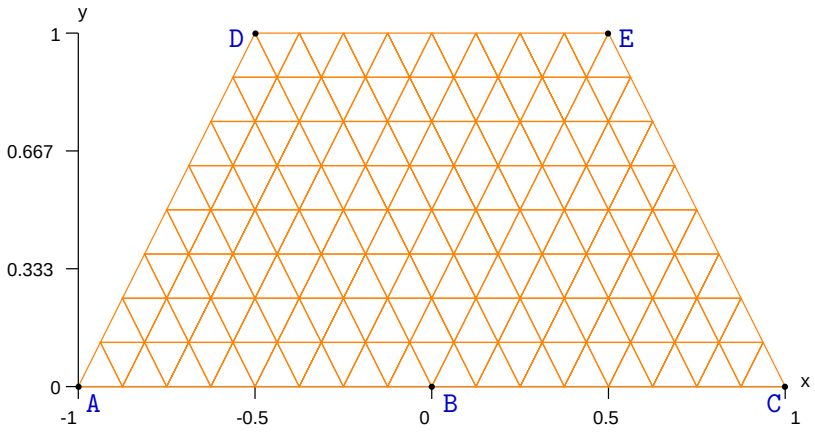
Incidentally, note that the `Mesh` constructor with `tag::rectangle` accepts any position for the vertices. Thus, you can use it to build any quadrilateral; the inner vertices' coordinates are simply interpolated from the coordinates of vertices on the boundary, as shown in paragraphs 1.5 and 2.1. This can be done even in more than two (geometric) dimensions, like in paragraphs 1.1 and 2.6. Tags `rectangle`, `quadrilateral` and `quadrangle` can be used interchangeably.

See also paragraph 9.7.

## 1.4. Triangular meshes

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

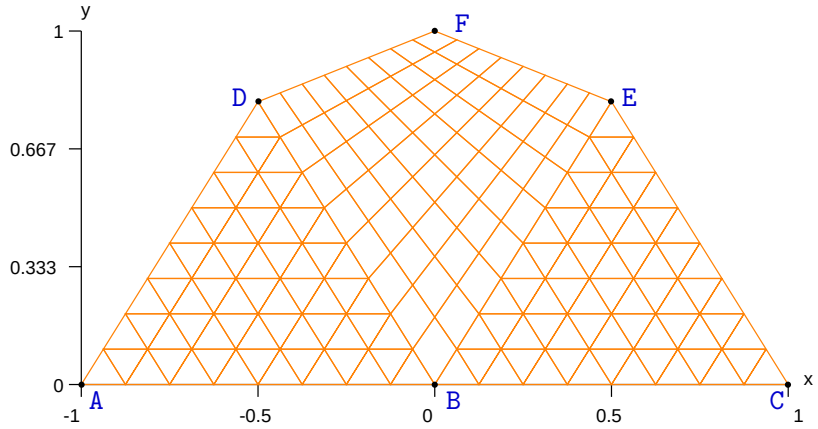
```
Cell A ( tag::vertex );  x(A) = -1. ;  y(A) = 0.;
Cell B ( tag::vertex );  x(B) =  0. ;  y(B) = 0.;
Cell C ( tag::vertex );  x(C) =  1. ;  y(C) = 0.;
Cell D ( tag::vertex );  x(D) = -0.5;  y(D) = 1.;
Cell E ( tag::vertex );  x(E) =  0.5;  y(E) = 1.;
Mesh AB ( tag::segment, A.reverse(), B, tag::divided_in, 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 );
```




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

## 1.5. Mixing triangles and rectangles

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




```
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 );
```

Paragraph 2.2 shows another example of mixed mesh.

## 1.6. Functions

Objects like `xyz` and `x` encountered in previous paragraphs are `Function` objects. They allow for arithmetic expressions like in

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

We access the value of a `Function` at a cell like in `double n = norm(A)`. We can also set this value but not for arithmetic expressions like `norm`. Thus, pieces of code like `x(A) = 1.` work fine, while `norm(A) = 1.` produces a run-time error.

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

```
Function norm_x = norm.deriv ( x );
Function norm_y = norm.deriv ( y );
```

# 2. Meshes and manifolds

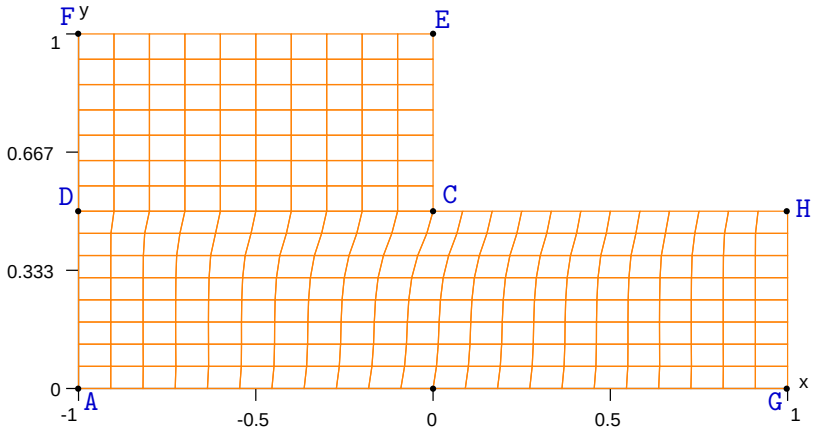
This section describes several examples of meshes, some of them built on specific manifolds. Note that in this manual we use the term "manifold" to mean a manifold without boundary.

Paragraphs 2.3, 2.4, 2.8, and 2.9 deal with one-dimensional meshes (curves) in $\mathbb{R}^2$, paragraphs 2.11, 2.12, 2.13 and 2.14 show curves in $\mathbb{R}^3$, paragraphs 2.1, 2.2, 2.8 and 2.9 show plane domains (two-dimensional meshes in $\mathbb{R}^2$), while paragraphs 2.5, 2.6, 2.7, 2.11, 2.12, 2.15 and 2.16 focus on two-dimensional meshes in $\mathbb{R}^3$ (surfaces).

Paragraphs 2.3 – 2.12 are about manifolds defined implicitly as level sets; paragraphs 2.13 – 2.16 describe parametric manifolds.

## 2.1. Joining segments

Here is another way of meshing the same L-shaped domain as in paragraph 1.3 :




```
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 an the one presented in paragraph 1.3 is a slight distortion in the lower half of the domain, due to the non-uniform distribution of the vertices along `HD`.

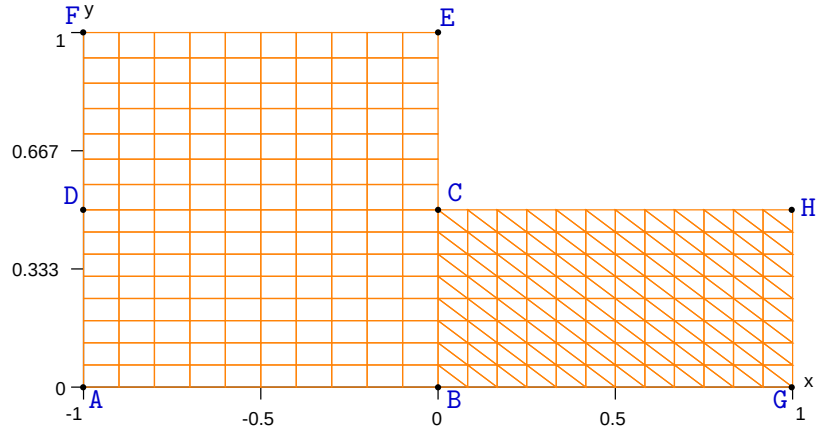
See paragraph 9.2 for more details about tags. See also paragraph 9.7.

## 2.2. 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 the example 1.3, 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 below.




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 1.4 could have been built like this :

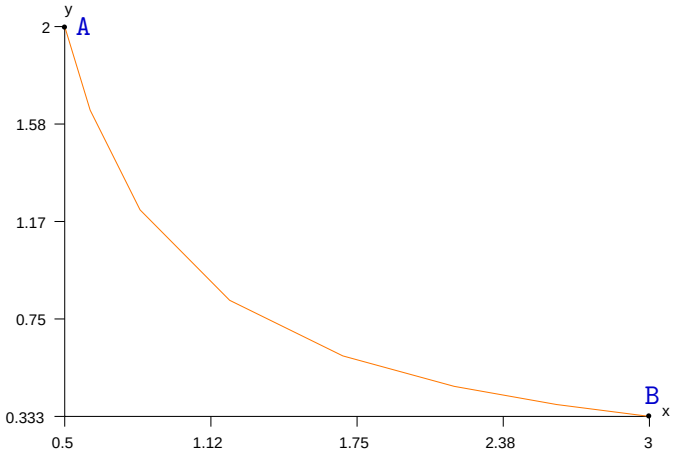
```
Mesh ABD ( tag::triangle, AB, BD, AD.reverse() );
Mesh BCED ( tag::quadrangle, CE, ED, BD.reverse(), BC, tag::with_triangles );
Mesh one_tri_one_rect ( tag::join, ABD, BCED );
```

## 2.3. 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, n )`. 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).

```
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.draw_ps ("hiperbola.eps");
arc_of_hiperbola.export_msh ("hiperbola.msh");
```




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

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.

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);
```

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

The projection is done by applying a few steps of Newton's method for under-determined (systems of) equations.* Thus, it only works as expected for a point not too far from the manifold.

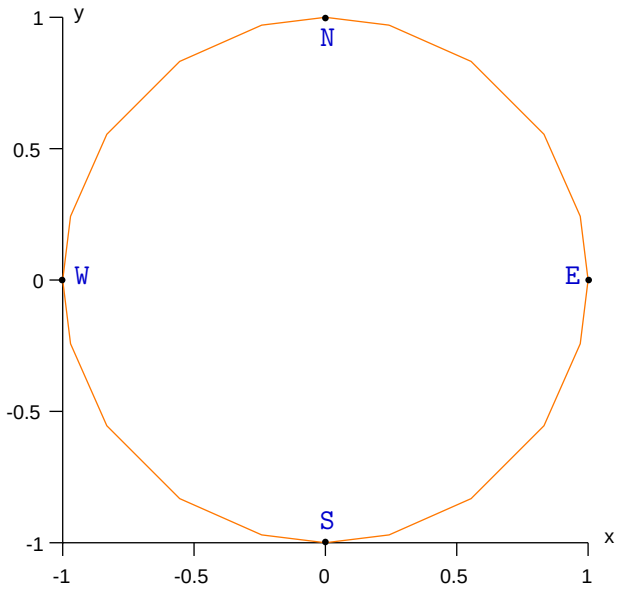
Paragraph 3.5 shows another way of meshing a curve, producing equidistant vertices.

## 2.4. A circle defined by four curved segments

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

```
Manifold circle_manifold = 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, N.reverse(), tag::divided_in, 5 );
Mesh circle ( tag::join, NW, WS, SE, EN );
```

* See e.g. Appendix B in 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




Again, the vertices are not perfectly uniformly distributed along the circle because they are obtained as projections (on the circle) of points along straight segments `NW`, `WS` and so forth.

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 paragraph 2.8.

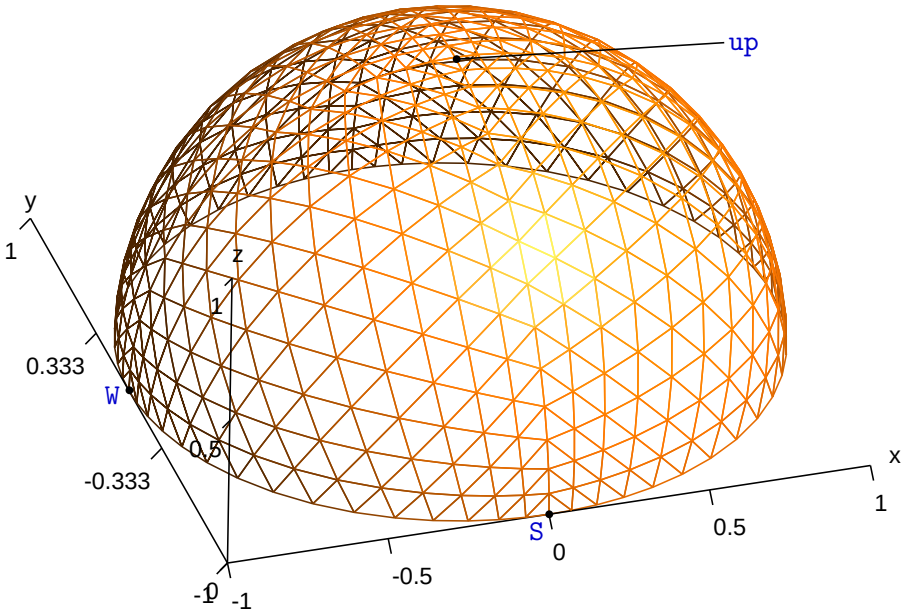
Paragraph 3.2 shows another way of meshing a closed curve, producing equidistant vertices.

## 2.5. A hemisphere defined by four curved triangles

Let's look at a surface in $\mathbb{R}^3$ :

```
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. );

// 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 2.6). In contrast, the `Mesh` constructors with `tag::segment`, `tag::quadrangle` and `tag::triangle` build 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. The projection is done by applying a few steps of Newton's method for under-determined (systems of) equations.* 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 approximatins 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 2.6).

Section 3 shows other ways of meshing a surface.

## 2.6. A more complex surface

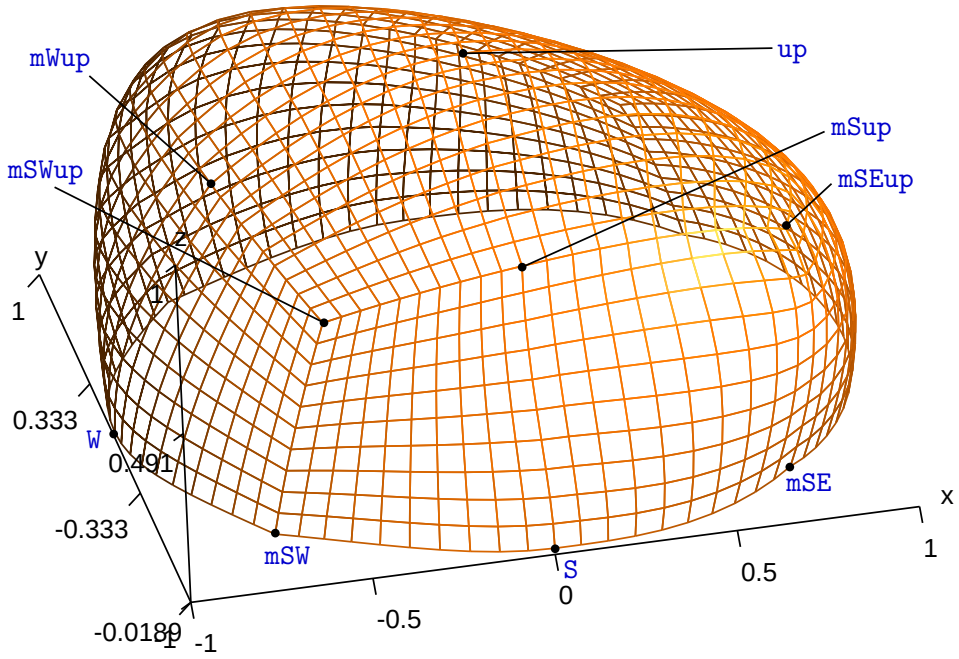
If the surface is more "bumpy", we must use smaller patches in order to get a mesh of good quality.

---

* See e.g. Appendix B in 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

Below we use twelve rectangles to get a bumpy hemisphere.

```
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_mWup ( tag::segment, W.reverse(), mWup, 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 list of `Meshes`; the same constructor is used in paragraph 8.2.

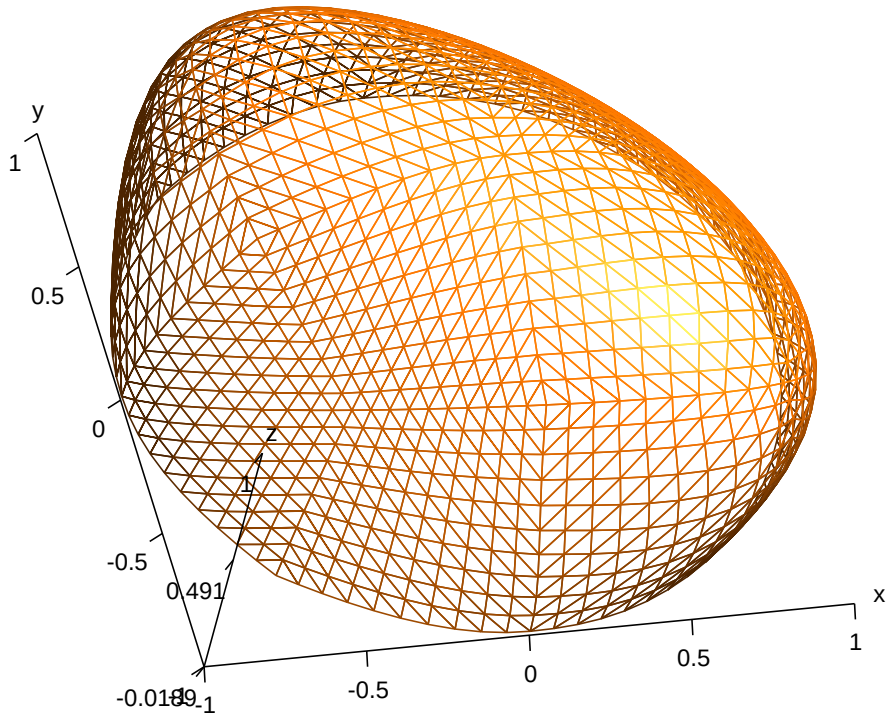
Unlike in paragraph 2.5, here we do not control the exact shape of the segments `S_mSW`, `S_mSup` and so on. They are projections of straight line segments onto our surface but since the equation of the surface is rather complicated we do not know the exact shape of these projections. Since points like `mSW` and `mSE` have been placed initially in `RR3` not belonging to the `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 2.12 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.

## 2.7. Exercise

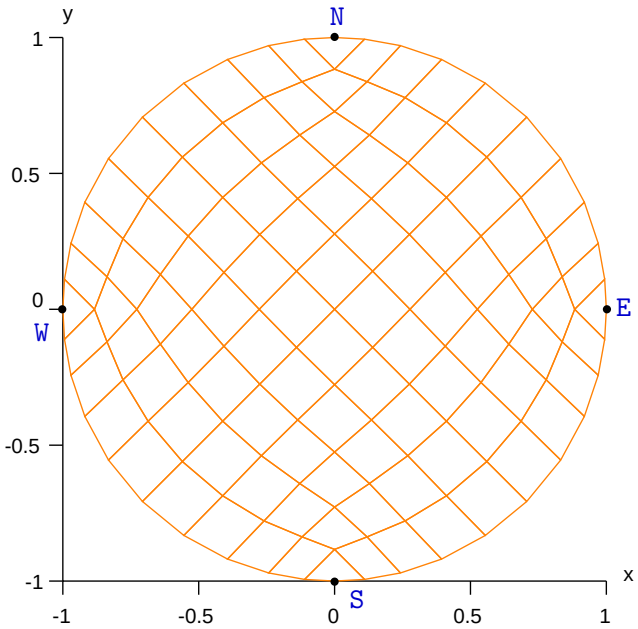
Slightly change the code in paragraph 2.6 in order to obtain the mesh below. (Hint: have a look at paragraph 2.2.)



## 2.8. Alternating between manifolds

Let's go back to the example in paragraph 2.4. Suppose we want to mesh the whole disk, not just its boundary. We can build the boundary of the disk just like in paragraph 2.4, by placing ourselves in the manifold `circle`. But if we want to 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.

```
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 );
```



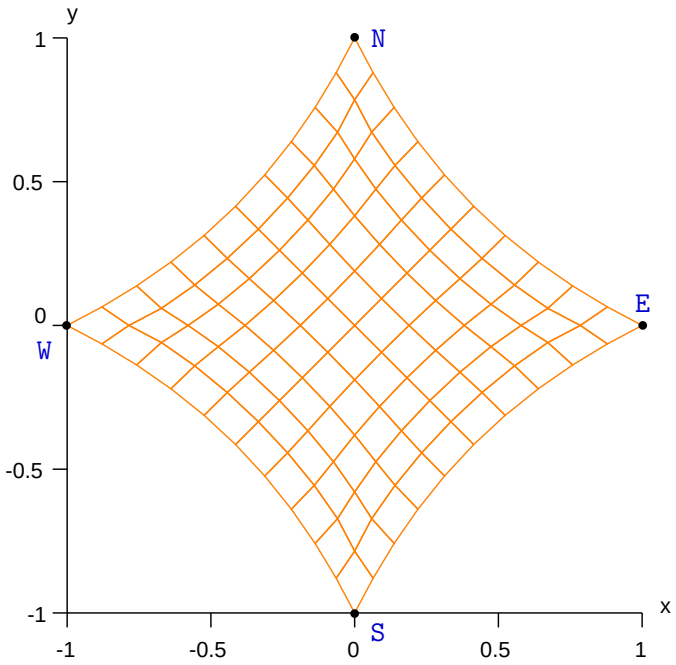

The mesh is of poor quality; we obtain quadrilaterals having a wide angle near `S`, `N`, `E` and `W`. Paragraphs 3.1 and 3.2 show another way of meshing a disk, having its boundary as starting point.

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.

## 2.9. Alternating between manifolds, again

Here is an example similar to the one in paragraph 2.8, this time with four arcs of hiperbola.

```
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 3.17 shows another way of meshing the same domain.

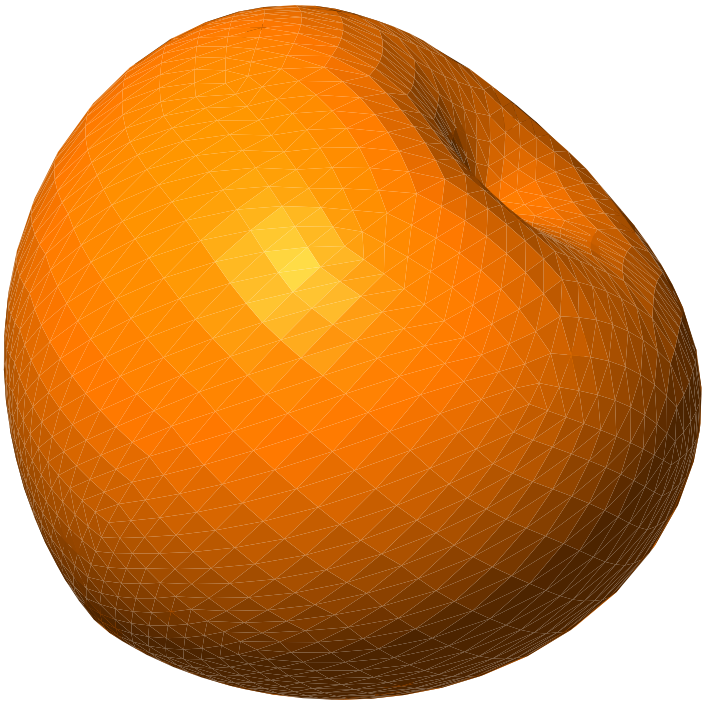
## 2.10. An organic shape

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

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

```
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 below, which does not resemble a physalis fruit.



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 segments) :

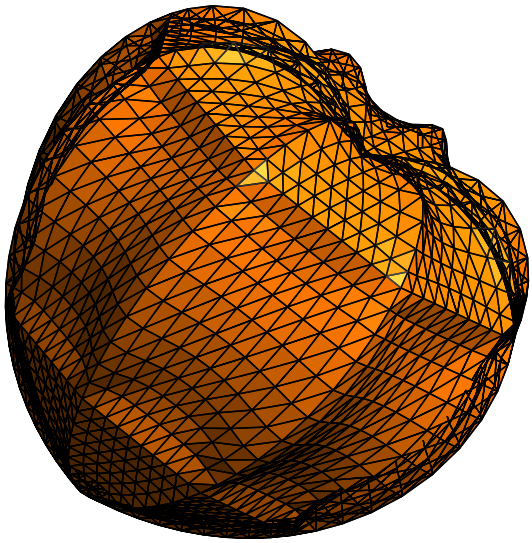
```
Cell A ( tag::vertex );  x(A) =  0.;  y(A) =  0.;  z(A) = std::sqrt ( 2.*pi/3. );
Cell B1 ( tag::vertex );  x(B1) =  1.;  y(B1) =  0.;  z(B1) =  1.;
Cell C1 ( tag::vertex );  x(C1) =  1.;  y(C1) =  0.;  z(C1) =  0.;
apple.project (B1);  apple.project (C1);
Cell D ( tag::vertex );  x(D) =  0.;  y(D)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

```
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 :

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



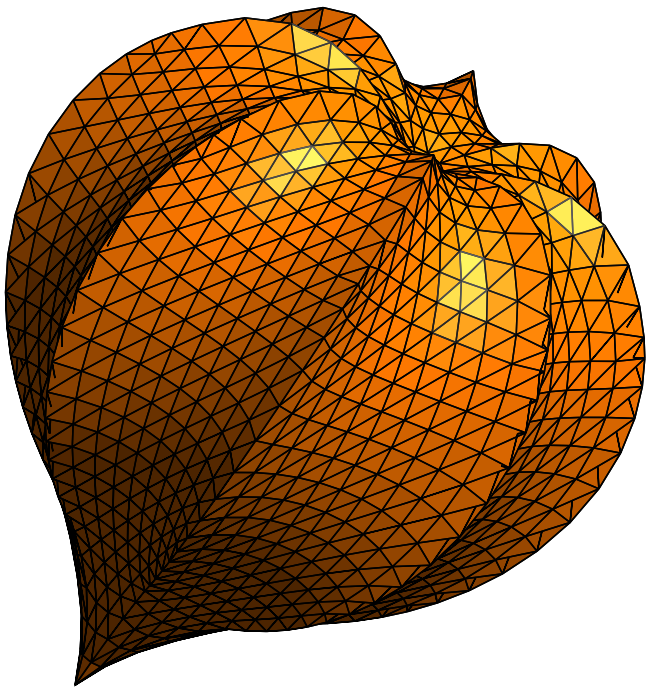
This shape is still not satisfactory, so we apply a deformation in $\mathbb{R}^3$ in order to get a sharper tip. The `apple` manifold is no longer relevant.

```
CellIterator it = fisalis.iter_over ( tag::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) = x(P) / ( 1. + 300. * std::pow ( z(P) - 1.3, 3. ) );
      y(P) = y(P) / ( 1. + 300. * std::pow ( z(P) - 1.3, 3. ) );
      z(P) = z(P) * ( 1. + 10. * ( z(P) - 1.3 ) * ( z(P) - 1.3 ) );  }
   if ( z(P) > 0. ) z(P) *= 0.8;                                          }
```

We add a sequence of baricenter operations for smoothening the shape. Note that each baricenter operation is relative to a sector and it 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.

```
std::list<Mesh>::iterator it1;
for ( it1 = lm.begin(); it1 != lm.end(); it1++ )
{  Mesh sect = *it1;
   CellIterator it2 = sect.iter_over ( tag::cells_of_dim, 1 );
   for ( int i = 1; i < 20; i++ )
   for ( it2.reset(); it2.in_range(); it2++ )
   {  Cell seg = *it2;
      sect.baricenter ( seg.tip(), seg );
      seg = seg.reverse();
      sect.baricenter ( seg.tip(), seg );  }                    }
```

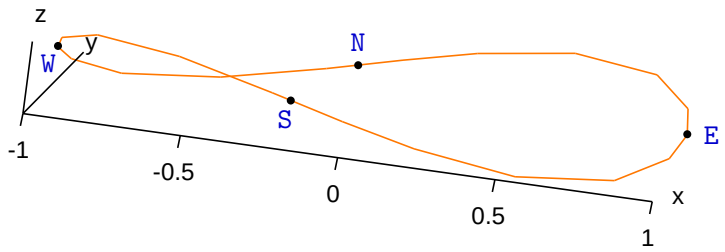
And we are happy with the final result :



## 2.11. A manifold defined by two equations

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

```
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 );
```



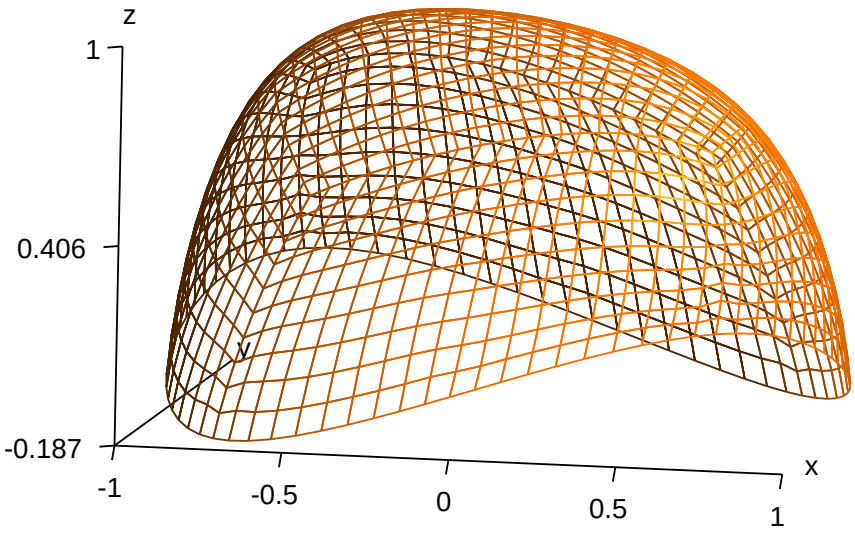

Paragraph 3.4 shows another way of meshing the same loop.

## 2.12. A submanifold of a submanifold

An implicit manifold has submanifolds. For instance, we can improve the look of the "bumpy hemisphere" in paragraph 2.6 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` :

```
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. );

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.;
// 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
Cell mSE ( tag::vertex );  x(mSE) =  1.;   y(mSE) = -1.;   z(mSW) = 0.;
base.project ( mSE );  // midway between S and E
// define similarly mNE and mNW
```

```
// now build eight segments, forming the base
Mesh W_mSW  ( tag::segment, W.reverse(), mSW, tag::divided_in, n );
Mesh S_mSW  ( tag::segment, S.reverse(), mSW, tag::divided_in, n );
// define similarly S_mSE, E_mSE, E_mNE, N_mNE, N_mNW, W_mNW

// 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. );
```

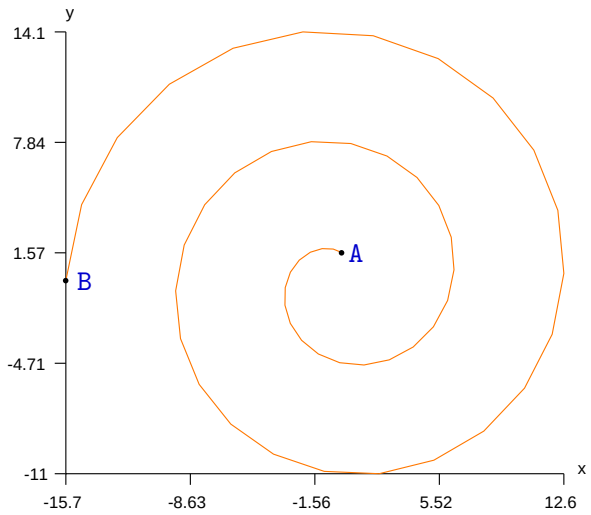
Paragraph 3.9 shows a way to mesh the same surface using fewer lines of code.

## 2.13. Parametric manifolds – a curve

Paragraphs 2.3 – 2.12 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.

```
// 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 on the 'spiral' manifold
spiral.set_coordinates ( x && y );

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




The operator `&&` joins two functions into one vector 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 $5\pi$ (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.

## 2.14. Closing a circle

In the approach of paragraph 2.13, it is possible but cumbersome to build a closed curve :

```
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 circle ( tag::segment, A.reverse(), B, tag::divided_in, 19 );

// now close the curve in a not very elegant manner
Cell BA ( tag::segment, B.reverse(), A );
BA.add_to ( circle );

// define new coordinates on circle_manif as expressions of t :
Function x = cos(t), y = sin(t);
circle_manif.set_coordinates ( x && y );

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

Paragraph 5.4 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.

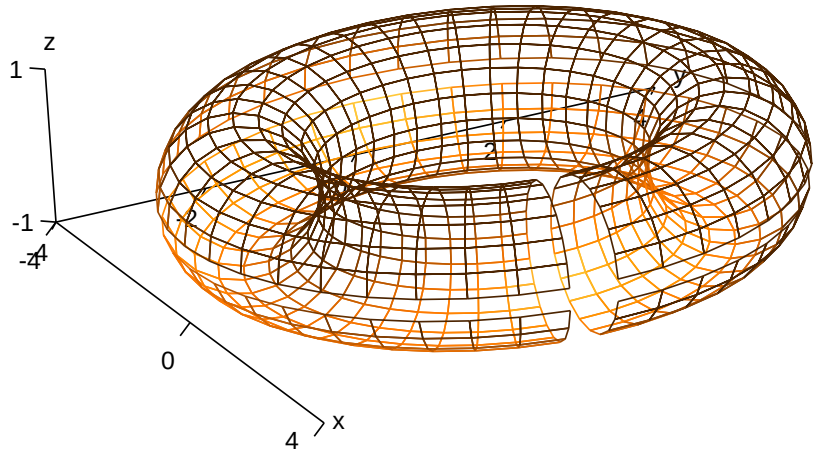
```
Manifold circle_manif ( tag::Euclid, tag::of_dim, 1 );
Function t = circle_manif.build_coordinate_system
   ( tag::Lagrange, tag::of_degree, 1 );

// 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 on circle_manif as expressions of t :
Function x = cos(t), y = sin(t);
circle_manif.set_coordinates ( x && y );
```

## 2.15. Parametric manifolds – a surface

Here is an example of a parametrized surface :

```
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);

// forget about alpha and beta :
torus.set_coordinates ( x && y && z );
// in future statements (e.g. for graphical representation)
// x, y and z will be used, not alpha nor beta :
ABCD.export_msh ("torus.msh");
```




Closing the torus in the cumbersome manner shown in paragraph 2.14 is possible but not practical. Paragraph 5.6 shows a more elegant solution.

If we only want a visual illusion of a closed surface, we may use the code below

```
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 );
// AB, BC, CD and DA look like closed circles
Mesh ABCD ( tag::rectangle, AB, BC, CD, DA );
// ABCD gives the illusion of a closed torus
```

## 2.16. 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; see paragraph ...

```
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.;       torus.project(A);
Cell B ( tag::vertex );  alpha(B) = 0.;       beta(B) = 1.9*pi;   torus.project(B);
Cell C ( tag::vertex );  alpha(C) = 1.95*pi;  beta(C) = 1.9*pi;   torus.project(C);
Cell D ( tag::vertex );  alpha(D) = 1.95*pi;  beta(D) = 0.;       torus.project(D);
// 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

// forget about alpha and beta :
torus.set_coordinates ( x && y && z );
// in future statements (e.g. for graphical representation)
// x, y and z will be used, not alpha nor beta :
ABCD.export_msh ("torus.msh");
```

The `Manifold::parametric` method is similar to `Manifold::implicit`, presented in paragraphs 2.3 – 2.12. The only difference is that by using `parametric` we declare an explicit dependence* 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.

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

# 3. Progressive mesh generation

Section 2 shows how to build meshes by joining rectangles and triangles together, like patches. The present section explains how to build a mesh starting from its boundary only; we call this approach "progressive mesh generation". It consists of starting with a given interface and add triangles, one by one, moving and deforming the interface, until it shrinks and disappears. In some 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.

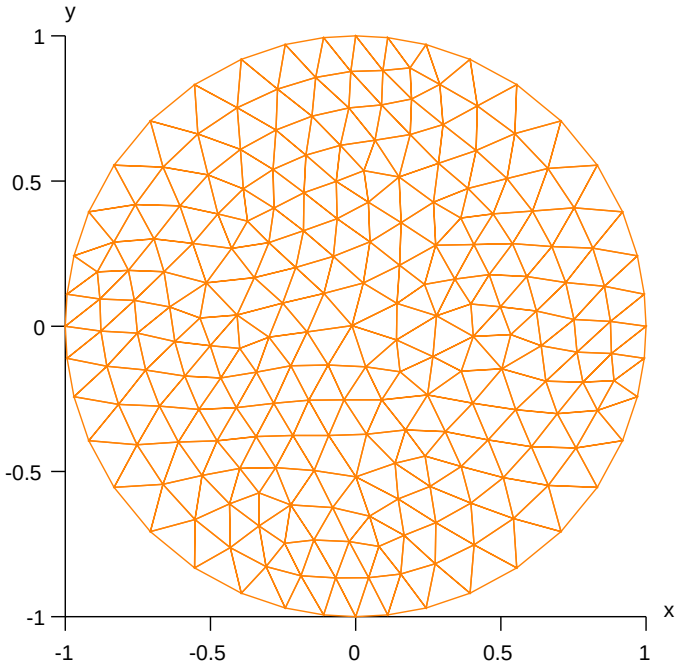
Progressive mesh generation works with segment cells (for one-dimensional meshes) and triangular cells (for two-dimensional meshes). Meshing of three-dimensional domains will be implemented in the future and will use tetrahedral cells.

Progressive mesh generation follows the shape of the current working manifold (the most recently declared `Manifold` object) by `projecting` each newly constructed vertex on that manifold. Recall that in this manual we use the term "manifold" to mean a manifold without boundary.

## 3.1. Filling a disk

Paragraph 2.8 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 progressively mesh the disk, starting from its boundary (a circle) and adding triangles, one by one, until the disk is completely covered :




```
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 );
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::progressive, 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 3.16 discusses this requirement).

Paragraph 9.2 gives more details about `tag`s.

## 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 :

```
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::progressive,
   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::progressive, tag::desired_length, 0.2 );

RR2.set_as_working_manifold();
Mesh disk ( tag::progressive, tag::boundary, circle, tag::desired_length, 0.2 );
disk.draw_ps ("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 progressive 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::progressive, tag::start_at, A, tag::desired_length, 0.2 );
```

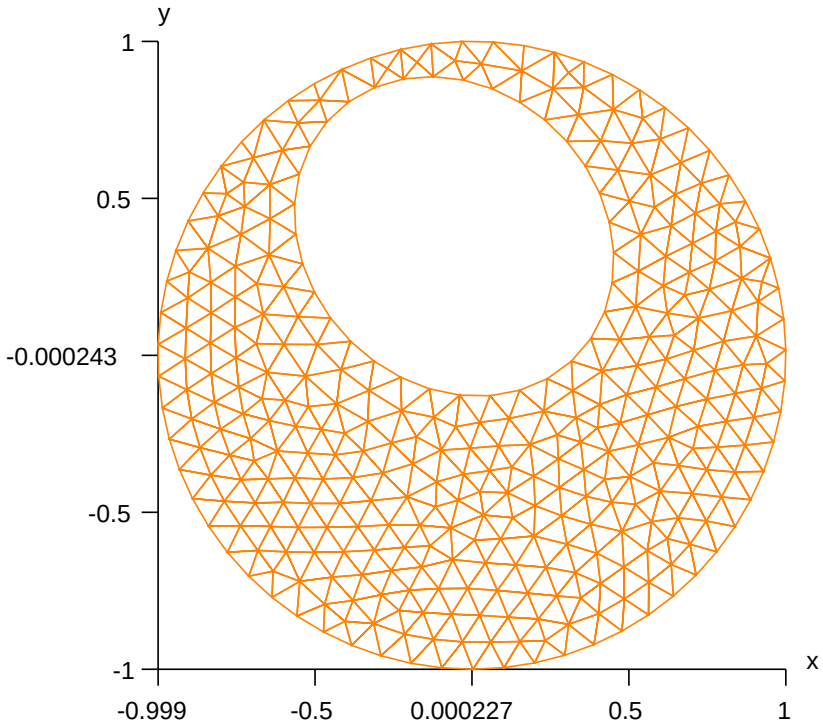
Also, maniFEM must infer the right orientation of the `circle` as explained in paragraphs 3.10 and 3.13. Choosing the other orientation would result in an endless

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 3.12.

## 3.3. Inner boundaries

Inner boundaries must have the reverse orientation :

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




Paragraphs 3.10 and 3.13 explain how *manifem* chooses the orientation of closed curves.

## 3.4. Meshing a three-dimensional loop

We may apply the same `progressive` algorithm for meshing the circle in $\mathbb{R}^3$ introduced in paragraph 2.11 :

```
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::progressive, tag::desired_length, 0.1, tag::random_orientation );
```

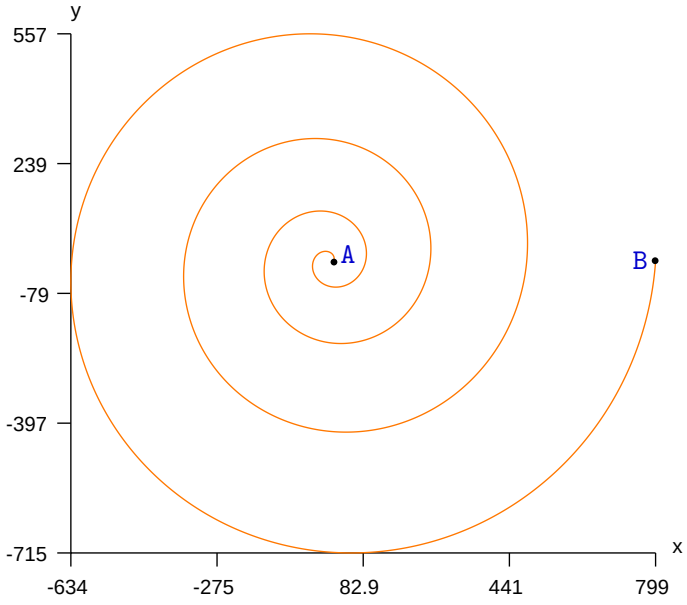
Unlike for the `circle` in paragraph 3.2, there is no way to choose between the two possible orientations of this `circle`. No one is more "correct" than the other. This is why *manifem* requires a supplementary argument `tag::random_orientation` for the `Mesh` constructor. This supplementary argument can be used even when it is not mandatory (like in the case of closed curves in $\mathbb{R}^2$), thus sparing the computer from the burden of finding the right orientation. Paragraphs 3.10 and 3.13 give more details.

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 3.12.

## 3.5. Starting and stopping points

In paragraphs 3.2 andd 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 2.13, let us define a spiral with a slightly different look and switch to progressive mesh generation.




```
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
```

```
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::progressive, tag::start_at, A,
              tag::stop_at, B, tag::desired_length, 1. );
```

Unlike in paragraph 2.13, 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 2.13 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 2.13 is that vertices are distributed along the spiral uniformly (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, so it performs 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::progressive, tag::start_at, A,
                     tag::stop_at, B, tag::desired_length, some_value );
```

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

Paragraph 3.12 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 – for this we should have given a name to the implicit manifold) 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.

```
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::progressive, 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 :

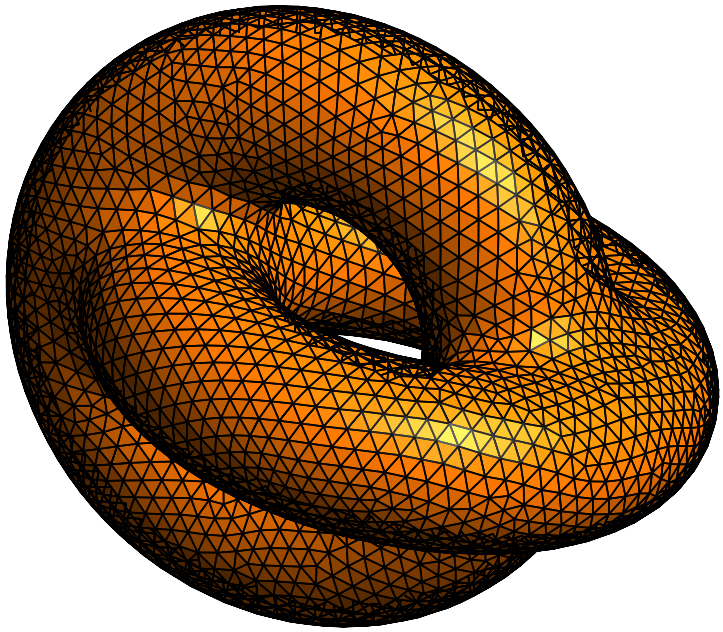
```
Cell A ( tag::vertex );  x(A) = 1.;  y(A) = 0.;  z(A) = 0.;
Mesh sphere ( tag::progressive, tag::start_at, A, tag::desired_length, 0.1 );
```

Also, some computational effort is made for choosing the right orientation of the sphere; paragraphs 3.10 and 3.13 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.

```
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 = x*x + y*y + 0.1;  // we add 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 = (x-0.4)*(x-0.4) + z*z + 0.1;  // we add 0.1 to avoid singularities
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 tori ( tag::progressive, 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.

If we want sharp edges, we must build the edges first as explained in paragraph 3.18.

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

This is a good place to mention that the progressive meshing algorithm has a bug which shows up very rarely, possibly related to the function `smooth_min`. We are doing our best to track the bug.

## 3.9. A bumpy hemisphere

We now look again at the surface in paragraph 2.12 and build it progressively, using fewer lines of code.

```
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::progressive, tag::desired_length, 0.1, tag::random_orientation );

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

In this example, the issue of the orientation can be really tricky. An orientation of the `circle` is chosen at random. Also, maniFEM will randomly choose to mesh the upper or the lower hemisphere. Paragraphs 3.10, 3.13 and 3.14 give more details.

## 3.10. How the orientation is chosen

In maniFEM, cells and meshes are oriented (paragraphs 1.2 and 8.7 provide details). For building a mesh progressively, 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 be propagated. Had we built the segments `NW`, `WS`, `SE` and `EN` in the opposite direction (that is, `WN`, `SW`, `ES` and `NE`) the program would have tried to mesh the exterior of the disk, never stopping.

Compact manifolds of co-dimension one (closed curves in $\mathbb{R}^2$, compact surfaces in $\mathbb{R}^3$) are an exception among implicit manifolds. There is a "privileged" orientation of such a manifold, which is compatible with the intrinsic orientation of the surrounding space. We call this orientation `inherent`; paragraph 3.13 discusses this topic.

Open curves (like the spiral in paragraps 3.5 and 3.12) have no privileged orientation. If we don't specify an orientation, these manifolds are considered not oriented. If

we don't specify a starting direction, maniFEM will perform a preliminary search in both directions and choose the one that first meets the stopping point.

Closed curves in $\mathbb{R}^3$ like the `circle` in paragraphs 3.4 and 3.9 have no privileged orientation either. 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::random_orientation` as last argument to the `Mesh` constructor. A statement like `Mesh circle ( tag::progressive, 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 bumpy` in paragraph 3.9 are discussed in paragraph 3.14.

## 3.12. 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::progressive, 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`.

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::progressive, 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.

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::progressive, 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, there the topologic considerations are more complex, so we delegate this discussion to paragraph 3.14.

## 3.13. 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 a `tag::intrinsinc_orientation` as argument to the `Mesh` constructor. However, this argument is not necessary; *manifem* will consider it by default. This happens for the `disk` in paragraphs 3.1 and 3.2, for the diamond in paragraph 3.17 and for the annulus in paragraphs 3.3, 3.22 and 3.24.

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". *manifem* will choose it in some situations, unless we provide a `tag::random_orientation` as argument to the `Mesh` constructor. We can enforce the requirement of finding the inherent orientation by providing a `tag::inherent_orientation` as argument to the `Mesh` constructor; however, *manifem* will consider this argument 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::inherent_orientation` as last argument 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. Then, again, 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 yourself, either by providing more information to the `Mesh` constructor as shown in paragraph 3.15 or by attaching this information to the manifold itself (this feature is not implemented yet).

If the orientation is not important for you, you can add to the `Mesh` constructor the last argument `tag::random_orientation` and *maniFEM* will not waste computing time to find the inherent orientation.

## 3.14. 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 a `tag::random_orientation` as a last argument to the `Mesh` constructor, or specify the orientation by providing a starting point and a starting direction as described in paragraph 3.12.

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. It 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 a `tag::inherent_orientation` as a last argument to the `Mesh` constructor of `bumpy`, then *maniFEM* will mesh (behind the curtains) both halves of the sphere; 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 `join`ed. But the reverse of any of them can be `join`ed with the other one, the result being a mesh on the entire bumpy sphere. Two choices are possible : either `msh1.reverse()` is `join`ed with `msh2` or `msh1` is `join`ed 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 still obtain 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 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::progressive, tag::desired_length, 0.1, tag::random_orientation );
nut.set_as_working_manifold();
Mesh bumpy ( tag::progressive, tag::boundary, circle,
             tag::desired_length, 0.1, tag::inherent_orientation );
```

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::progressive, tag::start_at, S, tag::towards, { 1., 0., 0. },
              tag::desired_length, 0.1                                        );
nut.set_as_working_manifold();
Mesh bumpy ( tag::progressive, tag::boundary, circle,
             tag::desired_length, 0.1, tag::inherent_orientation );
```

Code below will produce the lower hemisphere.

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

Code below will unpredictably mesh the upper or the lower hemisphere; in either case, the produced mesh will be oriented upwards because the orientation of `circle` points upwards.

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

Code below will unpredictably mesh the upper or the lower hemisphere; in either case, the produced mesh will be oriented downwards.

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

## 3.15. Specifying the direction

In paragraph 3.12 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 sphere" already considered in paragraphs 2.12, 3.9 and 3.14, oriented upwards.

```
Cell S ( tag::vertex );    x(S) =  0.;  y(S) = -1.;  z(S) =  0.;
std::vector < double > tau { 1., 0., 0. };
Mesh circle ( tag::progressive, 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::progressive, 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.

## 3.16. Geometric limitations

Progressive 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.

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 3.22, 3.23 or 3.24.

Sharp edges and singularities 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 2.3 and 2.4 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.

This is a good place to mention that the progressive meshing algorithm has a bug which shows up very rarely. It has been observed in examples similar to the one in paragraph 3.7, which means it could be related not to the meshing algorithm itself but to the function `smooth_min`. We are doing our best to track the bug. Fingers crossed.

## 3.17. Sharp angles

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

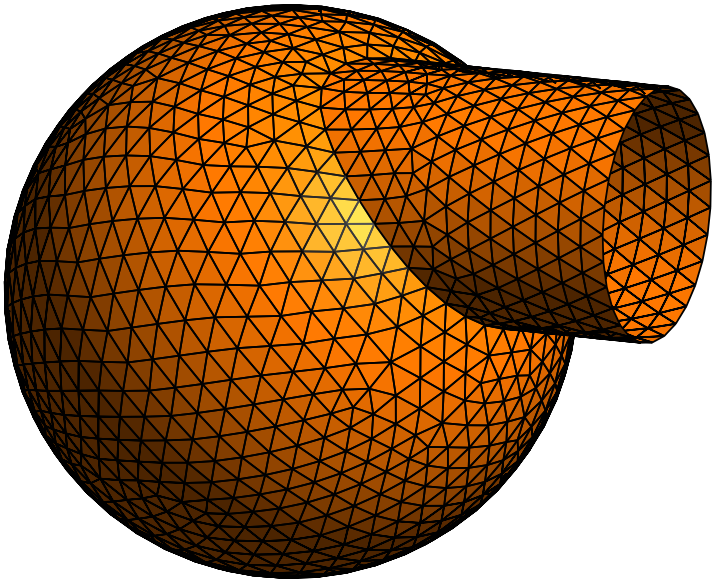
Here is how to mesh the diamond-shaped domain in paragraph 2.9.

```
// 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::progressive like in
RR2.implicit ( x*y - x - y ==  1. );
Mesh WS ( tag::progressive, 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::progressive, tag::boundary, bdry, tag::desired_length, 0.1 );
```

## 3.18. Sharp edges

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

```
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];

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 );

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;
Mesh circle_1 ( tag::progressive, 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 intersection = cylinder.implicit ( x*x + y*y + z*z == rs*rs );
Cell start_2 ( tag::vertex );
x ( start_2 ) = 1.;  y ( start_2 ) = 0.;  z ( start_2 ) = 0.5 - rc;
intersection.project ( start_2 );
Mesh circle_2 ( tag::progressive, 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::progressive, tag::boundary, two_circles,
                    tag::start_at, start_1, tag::towards, { -1., 0., 0. },
                    tag::desired_length, seg_size                          );

RR3.implicit ( x*x + y*y + z*z == rs*rs );
Mesh piece_of_sph ( tag::progressive, 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. 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`. The common boundary 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.

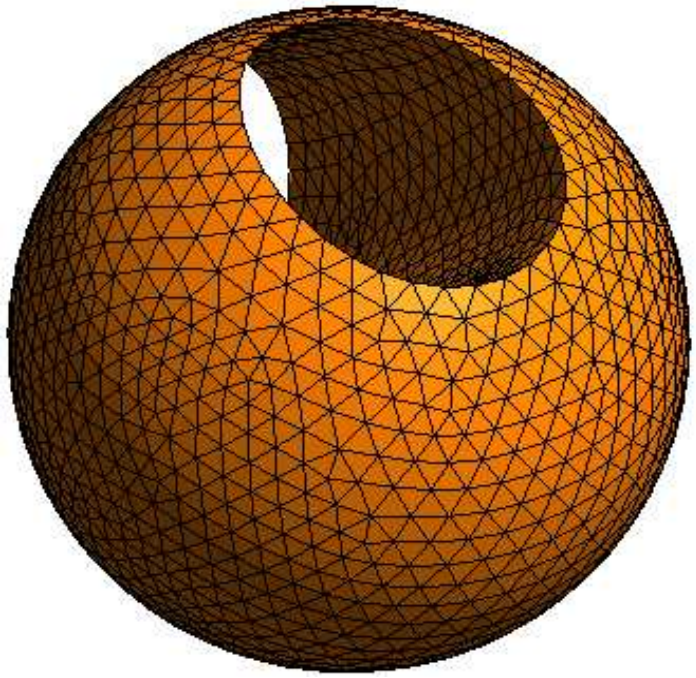
If we want to close the extremity of the cylinder, is suffices to add a few lines of code :

```
RR3.implicit ( x == 1.5 );
Mesh disk ( tag::progressive, 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 3.21.

## 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.



```
Manifold cylinder = RR3.implicit ( y*y + (z-0.5)*(z-0.5) == rc*rc );
Manifold intersection = cylinder.implicit ( x*x + y*y + z*z == rs*rs );
Cell start_1 ( tag::vertex );
x ( start_1 ) = 1.;  y ( start_1 ) = 0.;  z ( start_1 ) = 0.5 - rc;
intersection.project ( start_1 );
```

```
Mesh circle_1 ( tag::progressive, tag::start_at, start_1,
                tag::towards, { 0., 1., 0. }, tag::desired_length, seg_size );
Cell start_2 ( tag::vertex );
x ( start_2 ) = -1.;  y ( start_2 ) = 0.;  z ( start_2 ) = 0.5 - rc;
intersection.project ( start_2 );
Mesh circle_2 ( tag::progressive, tag::start_at, start_2,
                tag::towards, { 0., 1., 0. }, tag::desired_length, seg_size );
Mesh two_circles ( tag::join, circle_1.reverse(), circle_2 );
cylinder.set_as_working_manifold();
Mesh piece_of_cyl ( tag::progressive, tag::boundary, two_circles,
                    tag::start:at, start_1, tag::towards, { -1., 0., 0. },
                    tag::desired_length, seg_size                          );
Mesh two_circles_rev ( tag::join, circle_1, circle_2.reverse() );
RR3.implicit ( x*x + y*y + z*z == rs*rs );
Mesh piece_of_sph ( tag::progressive, tag::boundary, two_circles_rev,
                    tag::start:at, start_1, tag::towards, { 0., 0., -1. },
                    tag::desired_length, seg_size                          );
Mesh perf_sphere ( tag::join, piece_os_sph, piece_of_cyl );
```

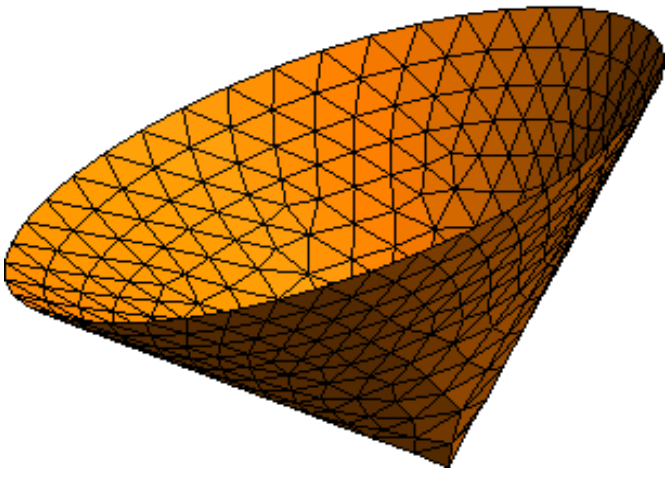
*Manifen* deals well with a disconnected manifold like `intersection`. Each of the constructors `Mesh circle_1` and `Mesh circle_2` meshes only a connected component of `intersection`, depending on the starting point we provide.

Had we chosen a cylinder of diameter 1, a singular point would have appeared; the two "circles" would touch at point $(0, 0, 1)$. This is a situation which *manifen* cannot handle yet; it is discussed in paragraph 3.21.

## 3.20. 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 *manifen* knowledge about these singularities. A typical example is the vertex of a cone.



```
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::progressive, 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::progressive, tag::boundary, circle,
            tag::start_at, A, tag::towards, { -1., 0., -1. },
            tag::singular, V, tag::desired_length, 0.1        );
```

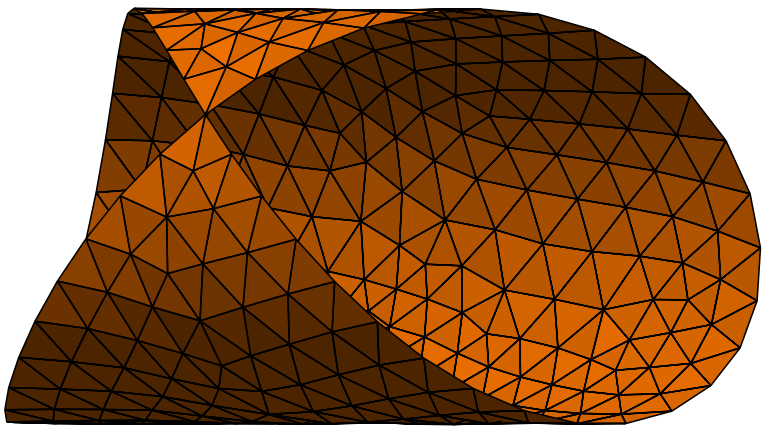
## 3.21. Singularities, again

**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 3.20, 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 `join`ed with a sphere with two holes).

```
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::progressive,
                tag:: start_at, V, tag::towards, { 1., 1., 0. },
                tag::singular, V, tag::desired_length, 0.1       );
Mesh circle_2 ( tag::progressive,
                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::progressive, tag::boundary, circle_1, circle_2,
                    tag::start_at, W, tag::towards, { -1., 1., 0. },
                    tag::singular, V, tag::desired_length, 0.1           );
```



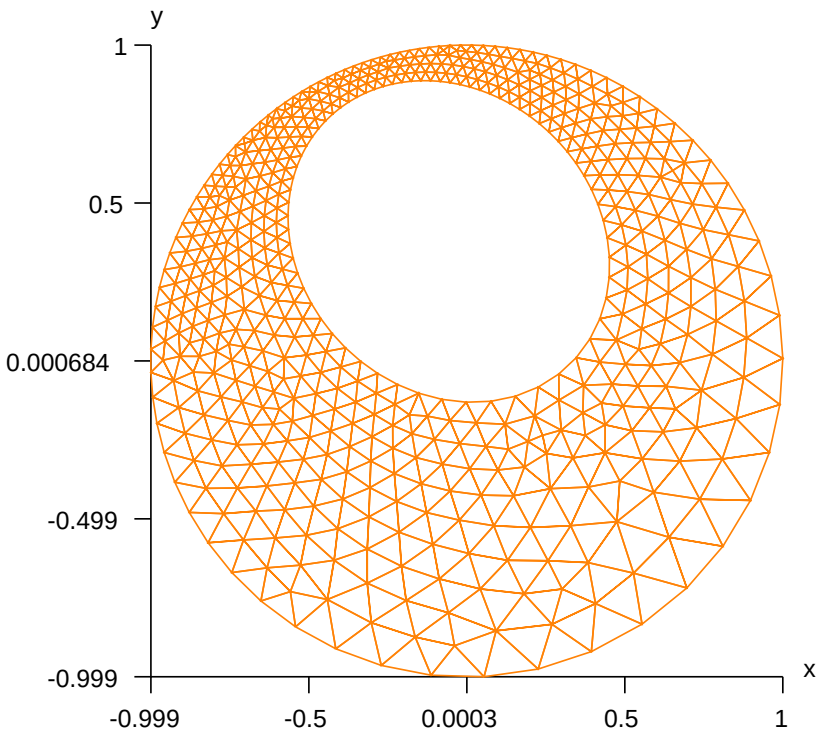
ManiFEM accepts as `Manifold` a self-intersecting set like `infinity`. However, in the `Mesh` constructor we must specify `V` as singular point.

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 `Meshes`. This is why we provide the two pieces of boundary separately.

## 3.22. Non-uniform meshing

The `desired_length` may be a non-constant function.

```
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::progressive, 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::progressive, tag::desired_length, d );
Mesh bdry ( tag::join, outer, inner.reverse() );
RR2.set_as_working_manifold();
Mesh disk ( tag::progressive, tag::boundary, bdry, tag::desired_length, d );
```




Compare with the mesh in paragraph 3.3.

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

## 3.23. Changing the Riemann metric

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

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 3.22.

```
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 ( 1. / d );

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

These two approaches (the one described in paragraph 3.22 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.

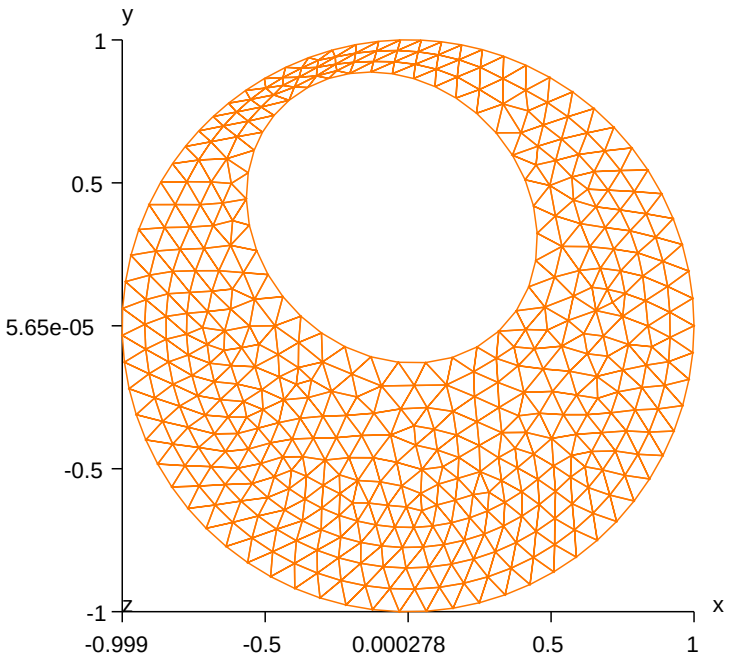
## 3.24. Anisotropic metric

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

The technique described in paragraph 3.23 can be generalized to an anisotropic Riemann metric. We define the metric by means of a square matrix $M$. $M$ must be symmetric positive definite. The arguments of `set_metric` are $m_{11}$, $m_{12}$ and $m_{22}$ for two dimensions, $m_{11}$, $m_{12}$, $m_{13}$, $m_{22}$, $m_{23}$ and $m_{33}$ for three dimensions.

```
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 + (x+0.3)*(x+0.3) + (y-0.9)*(y-0.9);
RR2.set_metric ( 1. + 1./d, -3./d, 1. + 9./d );
// or, equivalently :
// RR2.set_metric ( tag::principal_part, 1.,
//                  tag::deviatoric_part, 1./d, -3./d, 9./d );

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




Compare with the mesh in paragraph 3.3.

This result cannot be achieved using the approach of paragraph 3.22 (by setting a non-constant `desired_length`).

## 3.25. Future work

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

# 4. Meshing of three-dimensional domains

Work in progress.

# 5. Quotient manifolds

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

The roots of *manifem* go back to a PhD thesis in 2002 where finite elements on a torus were implemented in `FORTRAN`.* 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 "donut" built in paragraph 2.15. 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 $\mathbb{R}^2$ but also with other groups of transformations.

## 5.1. A one-dimensional circle

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

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

```
// 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 RR2 (a translation)
Manifold::Action g;  g(x) = 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::jump, g );
```

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

## 5.2. A flat torus

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

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

```
// begin with the usual two-dimensional space
Manifold RR2 ( tag::Euclid, tag::of_dim, 2 );
```

* See C. Barbarosie, Shape optimization of periodic structures, Computers & Structures 30, 2003

```
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 g1, g2;
g1(x,y) = (x+1) && y;
g2(x,y) = x && (y+1);
Manifold torus_manif = RR2.quotient ( g1, g2 );

// 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::jump, g1 );
Mesh seg_vert  ( tag::segment, A.reverse(), A,
                 tag::divided_in, 10, tag::jump, g2 );
// and a rectangle
Mesh torus ( tag::rectangle, seg_horiz, seg_vert,
             seg_horiz.reverse(), seg_vert.reverse() );

// it would be meaningless to export 'square' as a msh file
// we can however export an unfolded mesh :
torus.unfold(-0.5,-0.2,0.5,0.2).export_msh ("unfolded-torus.msh");
```

We have added a shadow representing the periodicity cell $[0,1]^2$. This gives a hint about the segments being repeated by the unfolding.

## 5.3. A skew flat torus

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

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

```
Manifold::Action g1, g2;
g1(x,y) = (x+1)   && (y+0.1);
g2(x,y) = (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.

## 5.4. A curved circle

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

We are now in a position to resume the example in paragraph 2.14, this time in a less cumbersome manner.

```
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;  g(theta) = theta + 2*pi;
Manifold circle = RR.quotient ( g );

Cell A ( tag::vertex );  theta(A) = 0.;
```

```
Mesh seg ( tag::segment, A.reverse(), A, tag::divided_in, 10, tag::jump, 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
circle.set_coordinates ( x && y );
seg.export_msh ("circle.msh");
```

## 5.5. A cylinder

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

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

```
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;  g( theta, z ) = (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, 10, tag::jump,  g );
Mesh AB ( tag::segment, A.reverse(), B, tag::divided_in, 10 );  // no jump
Mesh BB ( tag::segment, B.reverse(), B, tag::divided_in, 10, tag::jump, -g );
Mesh cylinder ( tag::rectangle, AA, AB, BB, AB.reverse() );

// 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
cylinder_manif.set_coordinates ( x && y & z );
cylinder.export_msh ("circle.msh");
```

## 5.6. A curved torus

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

We are now in a position to resume the example in paragraph 2.15, this time by using the quotient manifold $\mathbb{R}^2/\mathbb{Z}^2$ introduced in paragraph 5.2.

```
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, g2;
g1(alpha,beta) = (alpha+2.*pi) && beta;
g2(alpha,beta) = 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, 10, tag::jump, g1 );
Mesh seg_vert  ( tag::segment, A.reverse(), A,
                tag::divided_in, 10, tag::jump, g2 );
Mesh torus ( tag::rectangle, seg_horiz, seg_vert,
             seg_horiz.reverse(), seg_vert.reverse() );

// parametrize the donut
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 :
torus.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_msh ("torus.msh");
```

# 6. Fields, functions and variational formulations

## 6.1. Fields and functions

As the reader may have already noticed, 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 9.2 for more details about tags.

In the above, `xy` is a `Function`, a vector field actually, with two components, `x` and `y`. The declaration of `xy` starts a complex process; under the curtains *manifem* declares a `Field` object associated to `xy`. 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 `x` component of the `Field` associated to `xy` at `A`.

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` at the middle of segment `AB` by using the syntax `y(AB) = 0.75`.

`Function` objects allow for arithmetic expressions like in

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

The `deriv` method performs symbolic differentiation :

```
Function norm_x = norm.deriv ( x );
Function norm_y = norm.deriv ( y );
```

Functions can also be integrated, see section 7.

## 6.2. Fields and functions [outdated]

**The code described in this paragraph is outdated and does not work.**

Consider the mesh built by the code below.

```
Mesh::intended_dimension = 2; // topological dimension
Mesh::initialize();
auto & xy = NumericField::multi_dim ("size", 2, "lives on", "points");
auto & x = xy[0], & y = xy[1];
auto & u = NumericField::one_dim ("lives on", "points");
auto SW = Cell::point ("SW"); x(SW) = -1.1; y(SW) = 0.3;
auto SE = Cell::point ("SE"); x(SE) = 1; y(SE) = 0;
auto NE = Cell::point ("NE"); x(NE) = 1; y(NE) = 1;
auto NW = Cell::point ("NW"); x(NW) = -1; y(NW) = 1;
auto south ( tag::segment, SW.reverse(), SE, tag::divided_in, 4, xy );
auto east  ( tag::segment, SE.reverse(), NE, tag::divided_in, 2, xy );
auto north ( tag::segment, NE.reverse(), NW, tag::divided_in, 4, xy );
```

```
auto west  ( tag::segment, NW.reverse(), SW, tag::divided_in, 2, xy );
auto rect_mesh ( tag::rectangle, south, east, north, west, xy );
```

Note that, besides the `NumericField` objects `x` and `y` which we have already encountered in paragraph 1.1, we introduce another `NumericField`, `u`, which is meant to hold the values of the solution of some PDE.

We now declare two functions defined on the mesh, linked to the two fields `x` and `y`. We can declare each function individually, as we did in paragraph 1.1, or we can declare the pair and then extract each component :

```
auto & xxyy = FunctionOnMesh::from_field ( xy, "Lagrange degree one" );
auto & xx = xxyy[0], & yy = xxyy[1];
```

These functions are described as `"Lagrange degree one"`, which means that they vary linearly along segments and also inside triangles. On quadrilaterals, they are polynomials of degree one, meaning they have a linear part plus a bi-linear one.

We can also declare an unknown function and a test function, to be used in a future variational problem :

```
auto & uu = FunctionOnMesh::unknown ( u, "Lagrange degree one");
auto & w = FunctionOnMesh::test ( uu );
```

The unknown `uu` is related to the `NumericField` previously declared, `u`, which provides space to hold, at each vertex of the mesh, a real value. Hopefully, at the end of the day these will be the values of the solution of our PDE. The test function does not need to hold any values. The test function is an abstract object whose only use is to express the variational formulation.

`FunctionOnMesh` objects obey to usual arithmetic rules. For instance, `(xx+yy*w)/uu` is a valid expression in *manifem*. They can also be differentiated, like in `(xx*yy).deriv(xx)`. Note that this expression will be evaluated right away, producing the result `yy`, while expressions involving an unknown function or a test function will produce a delayed derivative object, to be evaluated later. Thus, `w.deriv(xx)` will be evaluated only after replacing the test function `w` by some function in a base of a discretized Hilbert space.

`FunctionOnMesh` objects can also be integrated through their method `integrate`, which produces a delayed integral expression. The integral is not evaluated right away, but only later, with the use of an `Integrator` object (which could be a Gauss quadrature).

Integral expressions are stored as objects belonging to the class `FunctionOnMesh::combinIntegrals`. These objects can be equated by using their `operator==`, which returns a VariationalProblem object. Thus, the operator `FunctionOnMesh::combinIntegrals::operator==` acts as a factory function for `VariationalProblem` objects. For instance, `x.integrate(rect_mesh) == y.integrate(south)` is a syntactically valid expression which would produce a `VariationalProblem` but actually gives a run-time error because the `operator==` checks that the right hand side of the variational problem actually contains a function declared as uknown and one declared as test, and that the left hand side contains a test function but no unknown. Here is a more meaningful example :

```
auto & var_pb =
    ( uu.deriv(xx)*w.deriv(xx) + uu.deriv(yy)*w.deriv(yy) ) .integrate(rect_mesh)
```

```
   == w.integrate(rect_mesh) + w.integrate(south);
```

Note how we can mix integrals on different domains; recall that `south` is a side of `rect_mesh`.

Dirichlet-type boundary conditions are implemented through the directive `prescribe_on` followed by one or more equalities.

```
var_pb.prescribe_on (north);  uu == 0.;          w == 0.;
var_pb.prescribe_on (east);   uu == xx*(1.-yy);  w == 0.;
var_pb.prescribe_on (west);   uu == 0.;          w == 0.;
```

# 7. Finite elements and integrators

Finite elments are still incipient in *manifest*. Paragraph 7.1 discusses the concept of finite element and paragraph 7.2 gives an example of rudimentary use. There is a lot of ongoing work on this subject.

## 7.1. Finite elements

The notion of a finite element is quite complex. The purpose of a `FiniteElement` is to build a list of functions, say, $\psi$, defined on our mesh. The linear span of these functions will be a discretized Hilbert space. It is the `FiniteElement`'s job to replace, in the variational formulation, the unknown function by one $\psi$, the test function by another $\psi$ 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.

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. When a Gauss quadrature is used, the separation between a `FiniteElement`'s job and the `Itegrator`'s job is not very sharp because often the Gauss quadrature is perfomed not on the physical cell but rather on a master element which is built and handled by the `FiniteElement`. The authors of *manifest* have tried to separate these two concepts as much as possible, especially because some users may want to use a `FiniteElement` with no master element, or an `Integrator` acting directly on the physical cell.

Thus, there is a base class `FiniteElement` and a derived class `FiniteElement::withMaster` which keeps, as an extra attribute, the map transforming the master element to 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.

For instance, the class `FiniteElement::Lagrange_Q1` is a class derived from `FiniteElement::withMaster`. It will only `dock_on` quadrilaterals (two-dimensional `Cells` with four sides). When docking on a cell, the `FiniteElement::Lagrange_Q1` object will build four "shape functions" and a transformation map (a diffeomorphism between a master element occuppying the square $[-1,1]^2$ and the current cell). It will also build the jacobian of this transformation map. The four shape functions can be accessed through the method `basis_function`, as shown in paragraph 7.2.

## 7.2. A rudimentary example

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

It should be stressed that the approach presented in this paragraph is rather low-level. We are working hard to make *manifest* 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.

```
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];

// 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 );

// 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 );
```

Vertices are not numbered. In the future, a more elegant solution for automatic numbering of vertices will be implemented; for now, we build a map `numbering` (see the source code in file `main-7.2.cpp` in the distribution tree), to be used as shown below. The matrix of the linear system and the vector holding the free coefficients are declared as objects of the `Eigen` library.

```
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();
```

We now run over all rectangular cells of `ABCD`, dock the finite element, 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:

```
// run over all square cells composing ABCD
CellIterator it = ABCD.iter_over ( tag::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'
   CellIterator it1 = small_square.boundary().iter_over ( tag::vertices );
   CellIterator it2 = small_square.boundary().iter_over ( tag::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.core]-1, numbering[W.core]-1 ) +=
   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.

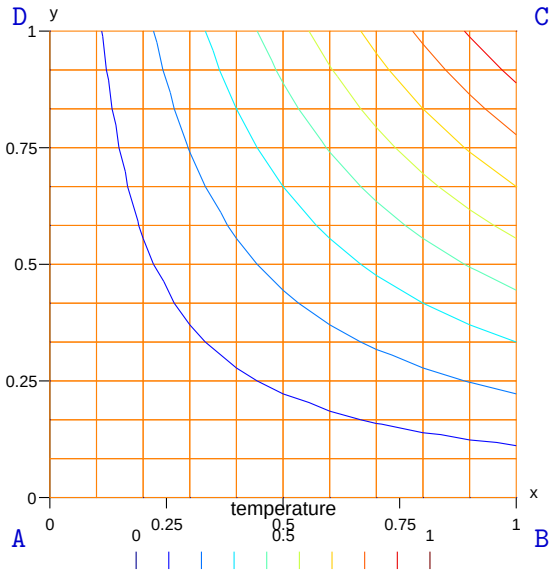
We impose Dirichlet boundary conditions $u(x, y) = xy$ (this way, we know beforehand the exact solution will be $u(x, y) = xy$). We use a function `impose_value_of_unknown` which changes the `matrix_A` and the `vector_b` in order to impose the Dirichlet condition `u(i) = some_value`. See the source code in file `main-7.2.cpp` in the distribution tree for the definition of the `impose_value_of_unknown` function.

```
CellIterator it = BC.iter_over ( tag::vertices );
for ( it.reset(); it.in_range(); it++ )
{   Cell P = *it;
    size_t i = numbering[P.core]-1;
    impose_value_of_unknown ( matrix_A, vector_b, i, y(P) );  }
```

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

```
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 :




We stress again that this example shows a rather rudimentary way of using finite elements in *manifem*. We are working hard to reach a more elegant, compact and high-level style. For instance, the numbering of vertices should be automated. Also, the user should have the possibility to declare the partial differential equation through a compact declaration of the corresponding variational formulation.

Another limitation of maniFEM is that, at present, finite element computations are rather slow. This is so because each docking operation implies a lot of symbolic calculations; the same happens when we differentiate and then integrate functions. There are two ways of circunventing 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 around is a deep optimization of the maniFEM library, object of on-going work. A significant part of the symbolic calculations should be done just once, when the finite element is declared. Only a small chunk of symbolic computation will be performed at docking and later, when we build the elementary matrix.

# 8. A closer look at cells and meshes

This section gives details about cells and meshes.

## 8.1. Building cells and meshes

As we have already seen in examples in previous sections, 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 9.2 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 but the core remains alive. If a cell or mesh is no longer needed, the user must explicitly dispose of it, as explained in paragraph 9.3.

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 8.2. 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 8.10 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 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 8.7 gives a more detailed explanation about orientation of cells and meshes.

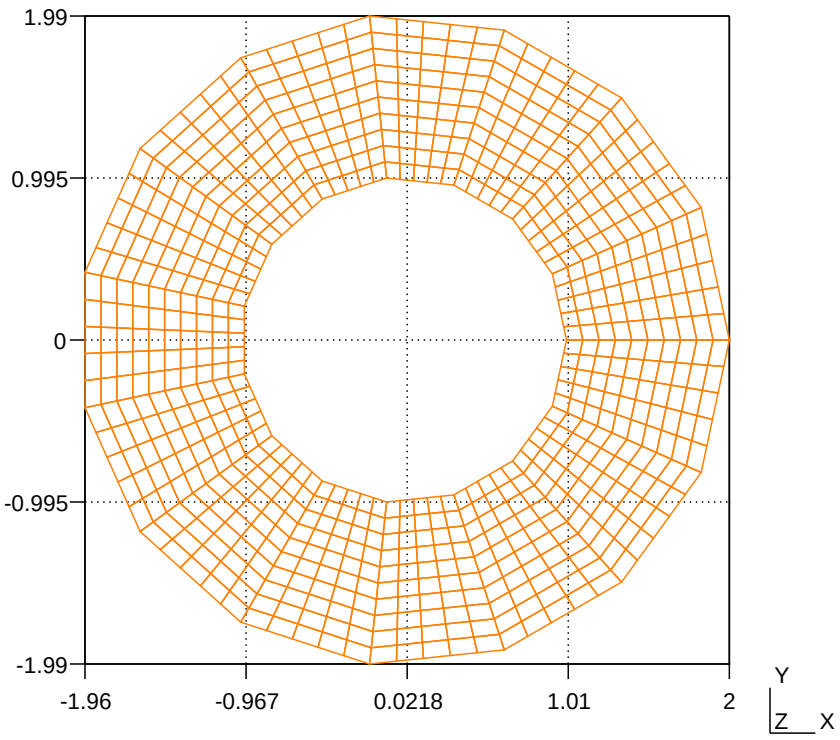
`Mesh`es have also a `reverse` method. Note that reverse meshes exist always (negative meshes are temporary objects built on-the-fly).

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 1.3, reverses of meshes `CD` and `BC` are used.

## 8.2. 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 `Meshes`. They are cheap to store and to copy. We call these customized pointers "wrappers"; their behaviour is described in some detail in paragraph 9.3.

```
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];

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;
list < 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) = cos(theta);    y(C) = sin(theta);
   Cell D ( tag::vertex ); x(D) = 2.*cos(theta); y(D) = 2.*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_msh ("ring.msh");
```

Note how we use a version of the `Mesh` constructor with `tag::join` taking as argument a list of `Mesh`es; we have already seen it in paragraph 2.6.

We might have set curved boundaries by using a submanifold of $\mathbb{R}^2$, like in paragraph 2.8.

See also paragraph 9.7.

## 8.3. Lists of cells inside a mesh

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 pointers to cells holding all vertices of that mesh, `msh.core->cells[1]` is a list of pointers to cells holding all segments and `msh.core->cells[2]` is a list of pointers to cells holding all triangles.

Thus, we could use a loop like the one below for iterating over all segments of the mesh.

```
for ( auto it = msh.core->cells[1].begin();
           it != msh.core->cells[1].end();  it++ )
{ auto seg = *it;  do_something_to (*seg);  }
```

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).

The `auto` keyword tells the `C++` compiler to guess the type of a variable according to the expression used to initialize it.

Note that the above code only works for a positive mesh. Paragraphs 8.5 and 8.6 describe nicer ways to iterate over cells of a mesh.

On the other hand, a cell is roughly defined by its boundary which in turn is a mesh of lower dimension. Thus, if `hex` is a `Cell` object modelling a hexagon, then `hex.boundary()` is a `Mesh` object modelling a one-dimensional mesh (a closed chain of six segments). So, if we want to iterate, say, over all vertices of that hexagon, we can use a loop like below.

```
auto & li = hex.boundary().cells[0];
for ( auto it = li.begin(); it != li.end(); it++ )
{ auto P = *it;  do_something_to (*P);  }
```

Note that the above only works if `hex` is a positive cell. Also, we have no guarantee about the order in which the vertices will show up in the loop. Paragraph 8.6 describes other ways of iterating over one-dimensional meshes, which follow the natural order of the vertices or segments.

## 8.5. Iterators over cells

This paragraph assumes that the reader is familiar to the notion of iterator in `C++`. If this is not the case, you should read an introductory book on the `C++` Standard Template Library (STL) before proceeding.

As explained in paragraphs 1.2 and 8.3, a mesh is essentially a collection of cells. In many situations, we may want to iterate over all cells of a mesh.

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 could use a code like

```
for ( auto it = msh.core->cells[2].begin();
           it != msh.core->cells[2].end(); it++ )
{  auto cll = *it;  do_something_to (*cll);  }
```

This style is slightly cumbersome and doesn't work for negative meshes, so we provide specific iterators. The code above is equivalent to

```
CellIterator it = msh.iter_over ( tag::cells_of_dim, 2 );
for ( it.reset(); it.in_range(); it++ )
{  Cell cll = *it;  do_something_to (cll);  }
```

Paragraph 9.2 gives some details about tags.

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 `CellIterator`; this operation returns a reference to a `Cell` object. Of

course, dereferencing a `CellIterator` does not produce a new cell, just provides access to a previously built cell (see also paragraph 8.10).

If we want to iterate over all vertices of the mesh, we can use

```
CellIterator it = msh.iter_over ( tag::cells_of_dim, 0 );
for ( it.reset(); it.in_range(); it++ )
{  Cell P = *it;  do_something_to (P);  }
// or, equivalently :
CellIterator it = msh.iter_over ( tag::vertices );
```

If we want to iterate over all segments of the mesh, we can use

```
CellIterator it = msh.iter_over ( tag::cells_of_dim, 1 );
for ( it.reset(); it.in_range(); it++ )
{  Cell seg = *it;  do_something_to (seg);  }
// or, equivalently :
CellIterator it = msh.iter_over ( tag::segments );
```

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 8.7 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

```
CellIterator it = msh.iter_over ( tag::cells_of_dim, 2, tag::force_positive );
```

Note that it is not safe to modify a mesh while iterating over its cells. After modifying a mesh, you may re-use a previously declared iterator by `reset`ting it. An exception to the above rule happens for one-dimensional meshes, described in paragraph 8.6.

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` :

```
short int d = 2;
size_t n = msh.number_of ( tag::cells_of_dim, d );
```

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 8.6 describes iterators specific to one-dimensional meshes.

## 8.6. Iterators over chains of segments

One-dimensional meshes have a specific structure (technical details provided in paragraph 9.14) so we provide specialized iterators. Unlike the iterators described in paragraph 8.5, which sweep the mesh in a rather unpredictible order, the iterators below follow the natural order of the cells (either vertices or segments) given by the topology of the mesh.

The syntax is the same as for higher-dimensional meshes :

```
Mesh chain ( tag::segment, A.reverse(), B, tag::divided_in, n );
// A and B are (positive) vertices, n is an integer

CellIterator it1 = chain.iter_over ( tag::cells_of_dim, 1 );
for ( it1.reset(); it1.in_range(); it1++ )
{  Cell seg = *it1;  do_something_to (seg);  }

CellIterator it0 = chain.iter_over ( tag::cells_of_dim, 0 );
for ( it0.reset(); it0.in_range(); it0++ )
{  Cell P = *it0;  do_something_to (P);  }

// or, equivalently,
CellIterator it0 = chain.iter_over ( tag::vertices );
CellIterator it1 = chain.iter_over ( tag::segments );
```

Unlike iterators over cells of meshes of dimension 2 or higher, presented in paragraph 8.5, iterators over one-dimensional meshes require the mesh to be connected.

A connected one dimensional mesh can be either an open chain of segments or a closed one (a loop). For an open chain, `it0` will begin at the first vertex and end at the last vertex, `it1` will begin at the first segment and end at the last segment. For a loop, they will begin at some arbitrary vertex or segment in the chain and produce all the vertices or segments following the natural order given by the topology of the mesh.

If we want to start at a specific location, we can make a `reset` call with one argument. For iterators over vertices, this argument should be a vertex, while for iterators over segments, this argument should be a segment. If such an argument is given, then 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 8.7.

There are also reversed versions of these iterators (they go backwards), obtained by adding the `tag::reverse` :

```
CellIterator it1r = chain.iter_over ( tag::segments, tag::reverse );
CellIterator it0r = chain.iter_over ( tag::vertices, tag::reverse );
```

Method `number_of` works just as for higher-dimensional iterators (see paragraph 8.5).

One-dimensional meshes have also 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 both `Mesh::first_vertex` and `Mesh::last_vertex` return positive vertices, unlike the `Cell::base` method (described in paragraph 1.2) which returns a negative vertex. [this should change]

Recall that it is not safe to modify a mesh while iterating over its cells. After modifying a mesh, you may re-use a previously declared iterator by `reset`ting it. However, if you modify a one-dimensional mesh and change its topology (cut a loop, thus making it an open chain, or contrarywise, close a chain, thus making it a loop), then you cannot re-use a previously declared iterator on that mesh (you must declare a new

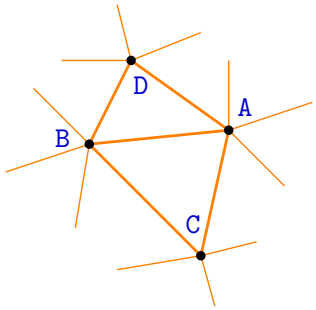
iterator). A change in the code is planned; when we `reset` an iterator, it will adapt itself to the new shape of the mesh.

As explained in paragraph 8.10, dereferencing a `CellIterator` does not produce a new cell, just provides access to a previously built cell.

## 8.7. Orientation of cells inside 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. 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).

However, the segments 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 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`, which has `B` as base and `A` as tip. `AB` and `BA` are different `Cell` objects; each is the reverse of the other. 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` 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, but other segments like `CA` see `A` as positive.

Let's look again at iterators described in paragraph 8.5. 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 there is no difference between these two iterators :

```
CellIterator it1 = tri_msh.iter_over ( tag::cells_of_dim, 2 );
CellIterator it2 =
   tri_msh.iter_over ( tag::cells_of_dim, 2, 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 8.6. The two iterators below will probably have different behaviours, depending on which segments happen to be positive :

```
CellIterator it3 = ABC.boundary().iter_over ( tag::segments );
CellIterator it4 =
   ABC.boundary().iter_over ( tag::segments, tag::force_positive );
```

There is no difference between the two iterators below (both produce positive points).

```
CellIterator it5 = ABC.boundary().iter_over ( tag::vertices, tag::reverse );
CellIterator it6 = ABC.boundary().reverse().iter_over( tag::vertices );
```

However, the two iterators below are quite different.

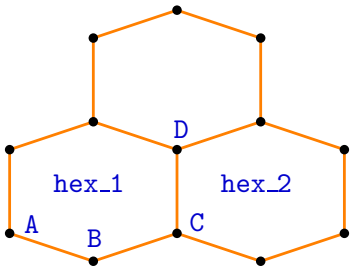
```
CellIterator it7 = ABC.boundary().iter_over ( tag::segments, tag::reverse );
CellIterator it8 = ABC.boundary().reverse().iter_over ( tag::segments );
```

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).

## 8.8. 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.




Consider the mesh `hex_msh` shown above, made of three hexagons. Pick one of them, at random :

```
CellIterator it1 = hex_msh.iter_over ( tag::cells_of_dim, 2 );
it1.reset();
Cell hex_1 = *it1;
```

Now choose a random segment on the boundary of `hex_1` :

```
CellIterator it2 = hex_1.boundary().iter_over ( tag::segments );
it2.reset();
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 one-dimensional meshes, described in paragraph 8.6, are implemented internally).

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 the picture above. 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 8.9 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`.

## 8.9. Navigating at the boundary of a mesh

Consider the example in paragraph 8.8. 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 at the beginning of paragraph 9.12.

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 8.10 gives a complete list of operations which return an existing cell or build a new one.

## 8.10. Declaring cells and meshes

As explained in paragraph 9.3, the `Cell` class is just a thin wrapper around a `Cell::Core`, and similarly for `Meshes`.

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 CellIterator
Mesh msh_copy = msh;  // msh is a Mesh
Mesh bd = cll.boundary();  // cll is a Cell of dimension at least 2
```

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 at the beginning of paragraph 9.12.

```
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 8.9.

```
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 );
```

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 ...
```

Paragraph 9.6 explains similar operations on `Cell::Cores`.

# 9. Technical details

Sections 9 and 10 are meant for those intereseted in developing and extending maniFEM. Of course the ultimate documentation is the source code; these sections can be used as a guide through the source code.

## 9.1. 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::CellIterator` instead of `CellIterator`, 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 9.2) 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` : `Cell`, `Mesh`, `CellIterator`, `Function`, `Manifold`, `VariationalFormulation`, `Integrator`, `FiniteElement`, the rarely needed `Field` and the even more rarely needed `MeshIterator`. 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`, `CellIterator::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
```

## 9.2. Tags

We use extensively `tags`. These are 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 `CellIterators` are defined (by means of the same method `Mesh::iter_over`) which behave very differently (see paragraphs 8.5 and 8.6).

```
// 'chain' is a one-dimensional Mesh
CellIterator it1 = chain.iter_over ( tag::vertices );
CellIterator it2 = chain.iter_over ( tag::vertices, tag::reverse );
CellIterator it3 = chain.iter_over ( tag::segments );
CellIterator it4 = chain.iter_over ( tag::segments, tag::reverse );
CellIterator it5 = chain.iter_over ( tag::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.

In maniFEM, namespaces and class names begin with capital letter (see paragraph 9.1) 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::reverse_of`, the reader's eye would catch `Tag` much before `reverse_of`; thus, `tag::reverse_of` 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`.

## 9.3. Wrappers and cores

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 it).

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.

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`, all derived from `Cell::Core::Positive`. And we have `Cell::Negative::Vertex`, `Cell::Negative::Segment` and `Cell::Negative`, all derived from `Cell::Core::Negative`. Both `Cell::Core::Positive` and `Cell::Core::Negative` are derived from `Cell::Core`.

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. 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 8.10 gives more details). You can think of `Cell`s and `Mesh`es as customized pointers towards `Cell::Core`s and `Mesh::Core`s, respectively.

Negative meshes contain no useful information (they only appear as boundaries of negative cells) so 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 a specific structure (they are chains of segments) and will receive in the future a specific treatment (a specialized class).

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.

## 9.5. 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 at the beginning of paragraph 9.12.

## 9.6. Declaring cell cores

As explained in paragraph 9.3, the `Cell` class is just a thin wrapper around a `Cell::Core`. While it is possible and useful to copy `Cell`s (think of them as customized pointers to `Cell::Core`s), objects in the `Cell::Core` class cannot be copied. Statements below will produce a compilation errror.

```
Cell::Core A ( B );  //  B is a Cell::Core
C = D;  //  C and D are Cell::Core objects
```

Objects in the Cell::Core class have an attribute reverse_p which is nullptr if the cell has no reverse (yet), otherwise points to the reverse core cell. Note that negative cells must have a reverse, which is a positive cell. Positive cells may have no reverse.

Cell::Cores have a method method reverse, which requires a tag::build_if_not_exists as argument. It behaves similarly to the method reverse in class Cell, described in paragraph 8.10. That is, a reverse Cell::Core object is built on-the-fly if needed :

```
Cell::Core * rev_cll_p = cll_p->reverse ( tag::build_if_not_exists );
assert ( rev_cll_p );
```

Recall that reverse meshes exist always (negative meshes are temporary objects built on-the-fly).

Methods cell_behind and cell_in_front_of in class Mesh also accept a pointer to a Cell::Core as an argument. When the tag::may_not_exist is provided, they return a pointer towards the respective neighbour cell, if it exists, otherwise they return nullptr.

```
Mesh msh ( ... );  // some constructor
Cell::Core * f_p = ... ;  //  f_p points to a face within msh
Cell::Core * cll_p = msh.cell_in_front_of ( f_p, tag::may_not_exist );
if ( cll_p ) do_something_to ( *cll_p );
else cout << "no neighbour !" << endl;
```

When the tag::surely_exists (or no tag at all) is provided as an argument, they return a pointer to an existing cell :

```
Cell::Core * cll_p = msh.cell_in_front_of ( f_p, tag::surely_exists );
// or, equivalently :
Cell::Core * cll_p = msh.cell_in_front_of ( f_p );

// first_vertex, last_vertex, first_segment, last_segment
```

This version of cell_behind and cell_in_front_of should be slightly faster than the one with tag::may_not_exist. However, you should only use it when you are confident that the neighbour cell already exists. 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 at the beginning of paragraph 9.12.

Paragraph 8.10 explains similar operations on wrapper classes Cell and Mesh.

## 9.7. Disposing of meshes

In paragraph 1.3 and others we have shown how to join meshes. The question arises, do we need to keep the intermediate meshes or just the final, big, one ? That's up to the user to decide. If we prefer to keep only the final mesh, we can "get rid" of the meshes we don't need anymore by means of the method dispose.

Note that it is not enough for the respective C++ object to go out of scope. In *manifem*, cells and meshes are created on the free store and survive outside their syntactic scope (see paragraph 9.3). The user must release them explicitly.

For instance, in paragraph 1.3, after building L_shaped, we may add the following lines of code :

```
BC.dispose();  // BC.reverse() will be discarded, too
```

```
CD.dispose();  // CD.reverse() will be discarded, too
ABCD.dispose();  CEFD.dispose();  BGHC.dispose();
```

The `dispose` operation will only discard component cells which belong to no other mesh. The `shread` method can be used to forcefully discard all cells in the mesh, but the need for such should be rare.

## 9.8. About `init_cell` (outdated)

The `Cell` class has static attributes `init_cell`, `init_cell_r`, `data_for_init`, `data_for_init_r`. The attribute `init_cell` is a list of pointers to functions to be called by the constructor of a positive cell, while `data_for_init` is a void pointer which can be used to pass supplementary information to `init_cell`. Attributes `init_cell_r` and `data_for_init_r` fulfill a similar task when building negative cells.

For instance, `Manifold::coordinate_system` inserts into the list `Cell::init_cell[0]` a call to `Mesh::prescribe_on` which in turn calls `FunctionOnMesh::prescribe_on` in order to prepare the ground for instructions like `x == 1.0` to produce the desired effect after the creation of each point.

Another example is the constructor `FiniteElement::Lagrange::Q1` which adds a call to `FiniteElement::Lagrange::Q1::enumerate_new_vertex` to the list `Cell::init_cell[0]`. This way, future vertices will receive automatically a `size_t` label, to be used by Lagrange finite elements.

## 9.10. Programming style

I (Cristian) have chosen some program-writing conventions which may seem unusual for other 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 that 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 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.

On the other hand, I am very fussy about indentation. I guess my mind has been formatted by Python.

`CellIterator`s obey to syntactic rules divergent from the conventions for iterators in the Standard Template Library. See paragraph 8.5.

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 ?

## 9.11. Frequent errors at compile time

```
variable [name] set but not used [-Wunused-but-set-variables]
[name] 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.

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.

```
'class ManiFEM::Cell::Core' has no member named 'name'
-- or --
'class ManiFEM::Cell::Core' has no member named 'get_name'
```

It seems you are trying to compile your code in `NDEBUG` mode (see the beginning of paragraph 9.12). In `NDEBUG` mode, cells and meshes do not have names.

```
cannot declare variable 'cll' to be of abstract type 'ManiFEM::Cell::Core'
-- or similar for PositiveBaseCell or NegativeBaseCell or CoreMesh --
```

Classes `Cell::Core`, `PositiveBaseCell` and `NegativeBaseCell` are abstract and cannot be instantiated. You must be more specific; `PositiveVertex`, `NegativeVertex`, `PositiveSegment`, `NegativeSegment`, `PositiveCell` and `NegativeCell` can be instantiated.

```
use of deleted function 'ManiFEM::PositiveVertex& ManiFEM::
PositiveVertex::PositiveVertex(const ManiFEM::PositiveVertex&)'
-- or similar for NegativeVertex or PositiveSegment or NegativeSegment --
--      or PositiveCell or NegativeCell or PositiveMesh --
-- or --
use of deleted function 'ManiFEM::PositiveVertex& ManiFEM::
PositiveVertex::operator=(const ManiFEM::PositiveVertex&)'
-- or similar for NegativeVertex or PositiveSegment or NegativeSegment --
--      or PositiveCell or NegativeCell or PositiveMesh or PosOneDimMesh --
```

Core cells and meshes cannot be copied. You cannot ask for things like `cell_2 = cell_1` or `mesh_2 = mesh_1`. See paragraph 9.6 for more details.

## 9.12. Frequent errors at run time

Some errors give explicit messages. For instance :

```
only one-dimensional meshes have first vertex
-- or --
ManiFEM::Cell& ManiFEM::Cell::tip(): Assertion 'dim == 1' failed.
```

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 want to add other optimization options like `-O2`. Remember to `make clean` before re-building your application.

If your program produces unpredictible, random errors at run-time, e.g. throws `segmentation fault`, try running 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.

Errors described below are produced by `assert`ions and thus will only show up in `DEBUG` mode.

```
ManiFEM::Cell& ManiFEM::Cell::base(): Assertion 'dim == 1' failed.
-- or --
ManiFEM::Cell& ManiFEM::Cell::tip(): Assertion 'dim == 1' failed.
```

It seems you are trying to get the base or the tip of a cell of dimension different from 1. This does not make sense.

```
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. For any cell, you may use the `positive` attribute which is a pointer equal to `this` if the cell is positive or points to the reverse if the cell is negative (so the reverse is positive). Or you may check if a certain cell is positive or negative by using the method `is_positive` (which simply returns the result of the comparison `this == this->positive`). Paragraph 8.7 gives more details about orientation of cells and meshes. See also paragraphs 8.10 and 9.6.

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 8.5 and 8.6.

```
ManiFEM::Cell& ManiFEM::Mesh::cell_in_front_of(ManiFEM::Cell&,
const ManiFEM::tag::SurelyExists&): Assertion 'cll != NULL' failed.
```

```
-- or --
ManiFEM::Cell& ManiFEM::Mesh::cell_behind(ManiFEM::Cell&,
const ManiFEM::tag::SurelyExists&): Assertion 'cll != NULL' failed.
```

You are navigating dangerously close to the boundary of a mesh. See paragraph 8.9.

```
static size_t ManiFEM::Mesh::diff(size_t, size_t): Assertion 'a >= b' failed.
-- or --
virtual void ManiFEM::PositiveCell::add_to(ManiFEM::CoreMesh*):
Assertion 'this->meshes.size() > 0' failed.
```

You are trying to build meshes of dimension higher than those maniFEM expects. Did you re-define 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 9.5.

If `gmsh` shows an empty drawing, go to `Tools` → `Options` → `Mesh`. For viewing one-dimensional meshes, you need to select `1D Elements`.

## 9.14. Chains of segments

One-dimensional meshes are special. They are mere chains of segments, connected or disconnected. If they are connected, they may be open chains or closed ones (loops).

We want iterators over cells of one-dimensional meshes to behave orderly. If the chain is closed, we want the iterator to follow the natural order of the segments. If the chain is open, we want more, we want the iterator to begin at one end and to go through until it meets the other end. These requirements are not compatible with the idea of a disconnected mesh.

So, we have chosen the following solution. One-dimensional meshes are implemented in class `Mesh::OneDim::Positive` and have two attributes (which other meshes have not) `first_ver` and `last_ver`. Every time we change a one-dimensional mesh through `add_to`, `remove_from`, `glue_on_bdry_of` or `cut_from_bdry_of`, the attribute `first_ver` of the mesh will be set to `nullptr`. This means the mesh is in an unordered state. It may even be disconnected. This is also the state of a new, empty, mesh.

In the `reset` method of a `CellIterator` over a one-dimensional mesh, we first check the attribute `first_ver` of the mesh. If it is `nullptr`, this means the mesh is unordered, it may even be disconnected. We then go through all segments and check the structure of the mesh, `assert`ing that it is connected, determining whether it is open or closed and setting accordingly the members `first_ver` and `last_ver`. [implement this in methods `get_first` and `get_last`]

If `first_ver` is equal to `Cell::ghost`, this means that the chain is closed (it is a loop). Any other value of `first_ver` means that the chain is open and that its ends are stored as `first_ver` and `last_ver` and can be used for initializing the iterator.

The `Cell::ghost` should, of course, not be used for any other purpose. It is the only negative cell whose `reverse_p` is `nullptr`. Its `heaps` have size zero.

A different kind of iterators is under construction. It will use a `tag::unordered` in its declaration and will behave like an interator over cells of higher-dimensional meshes,

that is, it will not take into account the chain structure of a one-dimensional mesh. These `unordered` iterators will accept to run over a disconnected mesh. Disconnected one-dimensional meshes are used, for instance, in progressive meshing.

In the future, one-dimensional meshes will keep only a list of segments, without storing the vertices. Or perhaps not even that, perhaps they will only keep `first_ver` and `last_ver`.

## 9.15. The cloud

During the mesh generation process described in paragraph 10.5, we have to check frequently the distance between some vertex on the evolving interface and all other vertices of the 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 used 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 of *manifold*. It is templated over the type of `Point`s (the metric space) and over a callable object returning the square of the distance between any two `Point`s. However, it still needs the touch of someone experienced in the subtleties of `C++`. Help is welcome.

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 importantly, method `find_close_neighbours_of` receives a `Point P` and a distance threshold and returns a list of `Node`s 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 progressive mesh generation).

---

* `MetricTree` is also available separately at `https://github.com/cristian-barbarosie/MetricTree`

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 9.3).

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.

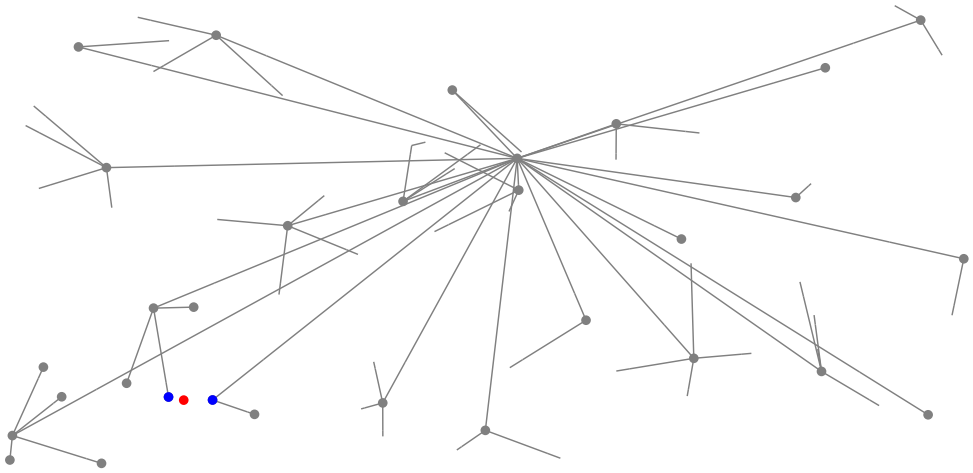
To each rank there is a distance `dist` associated. Children of a node `N` are no farther than `dist[rank[N]]` from `N`. Note that a point `P` being at distance less than `dist[rank[N]]` from `N` does not imply that `P` must be a child (not even an indirect descendant) of `N`. In other words : zones overlap.

`dist[k]` is a geometric sequence with ratio `ratio`. `ratio` must be greater than two; we recommend some value between 5 and 10. 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.



The above drawing 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; if it belongs, it will be returned as element of the list.

The drawing above 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`, `main-9.15.cpp` and `metric-tree-verbose.h`) from `https://github.com/cristian-barbarosie/manifem/tree/master/src`, then `make run-9.15`.

## 9.16. The cloud in progressive mesh generation

The use of `MetricTree` for progressive mesh generation (described in paragraph 10.5, with examples in section 3) is tricky when we are meshing a submanifold of $\mathbb{R}^n$ (a curve in $\mathbb{R}^2$ or $\mathbb{R}^3$, a surface in $\mathbb{R}^3$) because there are two distances we must deal with. There is the global, Euclidian, distance in the surrounding space and there is the local distance on the tangent space of the manifold. They are equal locally (at short distances) unless we attach a specific Riemann metric to the manifold as shown in paragraphs 3.23 and 3.24.

Even if we don't play with a specific Riemann metric, the Euclidian metric is different, at large distances, from the metric on the manifold defined by means of geodesics. And anyway, computing the distance by means of geodesics is not affordable (it is too heavy computationally).

We have chosen the following work-out. We use a `MetricTree` with (the square of) the usual Euclidian distance (which is computationally cheap). This means that a call to `MetricTree::find_close_neighbours_of` will produce a list which is too large in the sense that it may contain points which are farther than the distance given as argument. The calling program must then filter this list by measuring distances with the local metric. The difference shouldn't be important if we are careful to choose the `desired_distance` (be it a constant or a function) small when compared with the curvature of the manifold (see paragraph 3.16).

If we define a non-uniform Riemann metric (as in paragraph 3.23), we must call `find_close_neighbours_of` with a modified value of the distance. If we define an anisotropic Riemann metric (as in paragraph 3.24), we need a lower bound on the Rayleigh coefficient of the matrix $M$ (i.e. a lower bound on its eigenvalues, or, equivalently, an upper bound of the spectral radius of the inverse matrix) for computing the modified value to provide to `find_close_neighbours_of`. Providing separately the `principal_part` and the `deviatoric_part` helps. The `principal_part` is a positive scalar, while `deviatoric_part` is a matrix responsible for the anisotropy; this matrix should be semi-definite positive. Desirably, the `deviatoric_part` should be a singular matrix (having a null eigenvalue); this way, the lower bound on the Rayleigh coefficient is simply the `principal_part`. If we provide only the sum $M$, *manifem* will compute, at each step, the lower bound on the Rayleigh coefficient, which may be a heavy computational burden. When the metric is highly anisotropic, the list returned by method `find_close_neighbours_of` will be singnificantly larger than the correct, filtered, list.

# 10. 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*.

Some components of *manifem* (namely, those described in paragraphs 10.2, 10.3 and 10.4) are written in two "flavors", a pretty version which is easier to read and an ugly version which should be slightly faster. They are functionally equivalent. The first one is recognizable by a `tag::pretty`. Reading both, side to side, should help the user understand the "lower level" programming style in *manifem*, so that later he or she may write new functions for extending and generalizing *manifem*.

## 10.2. Building a chain of segments

One of the simplest meshes is an open chain of segments. Constructor `Mesh ( tag::segment,...)`, declared in `mesh.h` and defined in `global.cpp`, 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 constructor `Cell ( 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 `space` (an object belonging to the class `Manifold`). For an 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 (paragraphs 2.3 – 2.9 show such examples). 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 (paragraphs ... show such a situation).

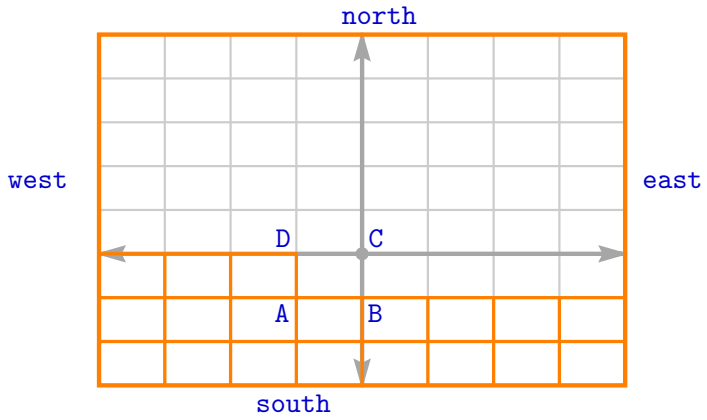
## 10.3. Building a rectangular mesh

Constructor `Mesh::Mesh ( tag::rectangle,...)`, declared in `mesh.h` and defined in `global.cpp`, 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 have already their internal divisions. Of course, opposite sides should have the same number of (segment) cells.

Paragraphs 1.3, 1.4 and many others (e.g. in section 2) show the use of this constructor.

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 or 2.6 – 2.9. Tags `rectangle`, `quadrangle` and `quadrilateral` can be used interchangeably.

Providing a `tag::with_triangles` makes the constructor cut each rectangle in halves; results are shown in paragraphs 2.2 and 2.7.

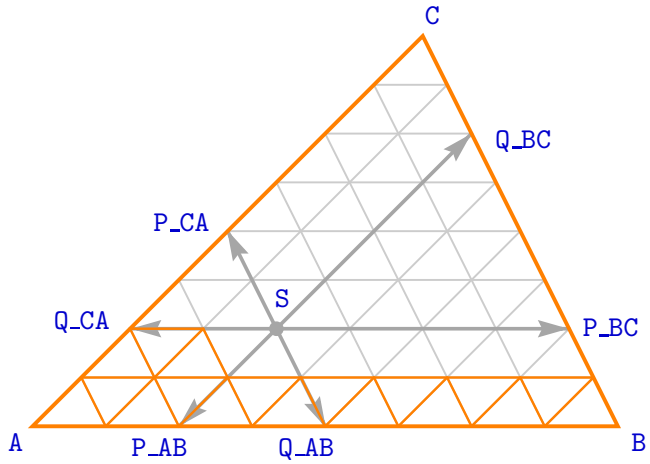



The coordinates of each new vertex are defined by interpolating the coordinates of four vertices on the four given sides, as shown in the figure above. The complicated formula for the coefficients (involving $\alpha^3$ and $\beta^3$) is there to ensure a smooth transition for the distribution of inner vertices in the case of non-uniform distribution along one or several sides. Paragraphs 2.1, 2.8 and 2.10 show examples where this is useful.

The remarks made in paragraph 10.2 about the interpolation operation hold here. Paragraph 2.6 shows an example where the interpolation operation consists of a convex combination followed by a projection. Paragraph . . . shows an example where the convex combination is done on an external parameter rahter than on the coordinates themselves.

## 10.4. Building a triangular mesh

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 10.3); the fact that we interpolate from six vertices compensates that.

Examples are presented in paragraphs 1.4, 1.5 and 2.5

## 10.5. Progressive mesh generation

Constructor `Mesh::Mesh ( tag::progressive, ... )`, declared in `mesh.h` and defined in `progressive.cpp`, builds a mesh on a given manifold starting with almost nothing. More precisely, it starts with the boundary of the future mesh, then it 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. Section 3 gives several examples.

This is a complex process, much more complex than building a regular mesh of rectangles (as described in paragraph 10.3) or of triangles (as described in paragraph 10.4). The constructor delegates the job to the function `progressive_construct`.

The mesh must follow the shape of the manifold; this is achieved by `projecting` newly created vertices on the working manifold.

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 10.8 discusses this event).

Detecting such touching points requires evaluating the distance between many pairs of points. This may become extremely time consuming, unless special care is taken to organize points belonging to the interface in a hierarchy allowing one to eliminate many pairs of points from the evaluation process. Paragraph 9.15 describes this hierarchy.

The next few paragraphs describe specific parts of this process of progressive mesh generation.

## 10.6. The normals

The meshing process described in paragraph 10.5 uses a set of normal vectors. Each segment 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 that two-dimensional manifold). These normal vectors provide the sense in which we want the mesh to grow (to the left or to the right of the interface), that is, they define the orientation of the manifold and of the mesh under construction. They are used when we create a new vertex in order to decide its placement.

Actually, 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 10.8), *manifem* propagates the normal vectors to the new connected component.

Each time a new segment is added to the interface (this happens in situations described in paragraphs 10.7 and 10.8), the normal associated to the new segment must be computed (propagated from neighbour segments).

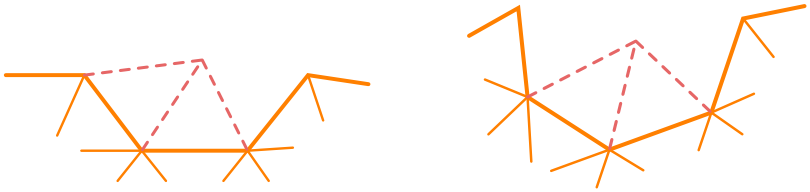
## 10.7. Filling triangles

Perhaps the simplest part of the meshing process described in paragraph 10.5 is just walking along the interface and adding new triangles.

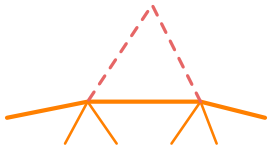
A trivial case is when an angle is encountered which is close to 60° (see figure below left). Then we only have to fill the space with a new triangle. Of course, before creating this new triangle a new segment `AB` must be created (no new vertex is needed). Two old segments must be removed from the interface and the new one must be added. Its normal must be computed (see paragraph 10.6).


A more complicated situation is when an angle is encountered which is close to 120° (see figure above right). A new vertex `P` is created; its position is defined based on the two normals of the adjacent segments. Two new segments `AP` and `BP` are created, then two new triangles are created and added to the mesh under construction. Two old segments are removed from the interface then `AP` and `BP` are added to the interface (their normals must be computed).

Special situations must be dealt with. For instance, if one or both neighbour angles are also close to 120°, we must take more vertices into account when we place the newly created vertex `P`; figure below shows such situations.



Finally, if all angles of the current connected component of the interface are wide, we may want to create a triangle "out of the blue", like in figure below.

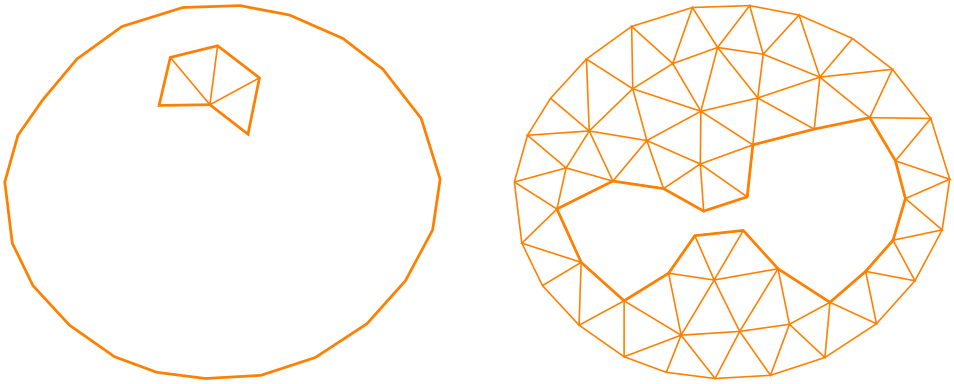


Every time a new vertex is created, we must check whether it came close to another zone of the interface and take action as described in paragraph 10.8.

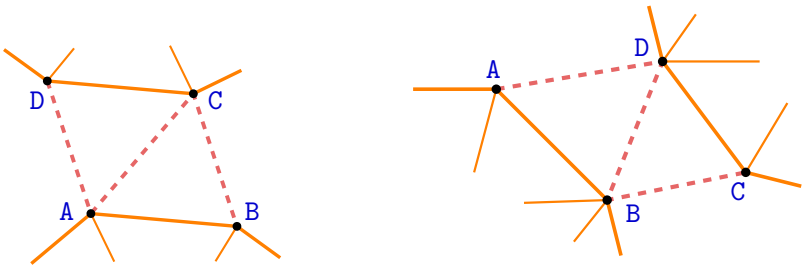
## 10.8. Touching the interface

During the meshing process described in paragraph 10.5, distinct zones of the moving interface may come close to each other and the algorithm will eventually put them in contact. Figure below shows the situation just prior to the contact. The two zones may belong to different connected components of the interface, as in the configuration shown below on the left. In this case, the two components will merge,

producing a new chain. The normal vectors will be propagated to the segments which did not have an associated normal vector (see paragraph 10.6). But it may also happen that they are part of the same connected component, see the drawing below on the right. In this case, the current chain will split in two.



Methods `glue_two_segs_S` and `glue_two_segs_Z` deal with two possible ways in which two zones of the interface may touch, independently of whether the two zones belong to the same connected component or not. In the drawing below on the right hand side an S-shaped connection is created, while in the other drawing a Z-shaped connection is created.



# Index