ManiFEM

This document describes `ManiFEM`, a software 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.

`ManiFEM` is written in C++ and uses the Standard Template Library (STL). It uses `Eigen` for storing matrices and solving systems of linear equations.

This document is divided in five sections. The first one describes `ManiFEM` from the viewpoint of the user who wants to use it as a black box, without bothering with too much detail. In particular, it should be accessible to readers who have some knowledge of `C++` but are not necessarily experts in `C++` programming. The second section gives more details about how cells and meshes are implemented in `ManiFEM`. It is useful for users who want finer control on the mesh, e.g. for implementing their own remeshing algorithms. The third section dwells on fields and functions, while the fourth describes how finite elements work in `ManiFEM`. The fifth section gives technical details, mainly for those interested in developing and extending `ManiFEM`.

TO DO :

• mesh a triangle, starting from its three sides

• mehs a tetrahedron, starting from its four faces

• build a mesh of cubes (done through cartesian product, but we would like to build just one cube based on its six faces and be able to join cubes afterwards)

• implement Dirichlet boundary conditions

• solve the system of linear equations

• visualize the solution of the PDE

• define a Gauss quadrature on segments

• define a Lagrange P1 finite element on triangles

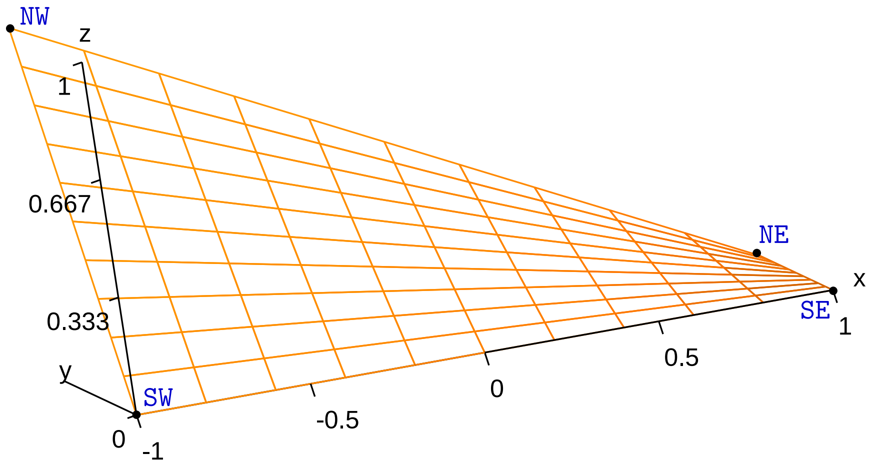
• eliminate memory leaks

1. General description

This section describes `ManiFEM` from the viewpoint of the user who is not interested in too much technical detail about its implementation. We show how to build meshes on (rather simple) domains in $\mathbb{R}^2$ and $\mathbb{R}^3$ and how to solve (rather simple) partial differential equations.

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. See paragraph 1.3 for a purely two-dimensional example. See paragraph 1.8 for a more complex example of solving a partial differential equation.



```
#include <iostream>
#include "Mesh.h"
using namespace std;

int main () {

   // we choose our (geometric) space dimension :
   auto & environment = Manifold::euclid (3);
   // xyz is a map defined on our future mesh with values in 'environment' :
   auto & xyz = environment.coordinate_system ("Lagrange degree one");

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

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

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

```
  // Now build the four sides of the rectangle :
  auto & south = Mesh::segment ( SW, SE, 10 );
  auto & east  = Mesh::segment ( SE, NE, 10 );
  auto & north = Mesh::segment ( NE, NW, 10 );
  auto & west  = Mesh::segment ( NW, SW, 10 );

  // And now the rectangle :
  auto & rect_mesh = Mesh::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
  auto & f = x*x+1/(5+y);

  // and compute its integral on the rectangle,
  // using Gauss quadrature with 9 points :
  auto & integ = Integrator::gauss ("Q9");
  cout << "integral = " << f.integrate ( rect_mesh, integ ) << endl;

}
```

In the present section, we often use the keyword `auto` in order to avoid declaring explicitly the type. This makes things easier for the final user. However, in the other examples presented in this section, we also include the type of the object as a comment, for those users who want to understand more of the underlying details. In subsequent sections of this manual we do not use `auto` at all. See paragraph 2.1 for an explanation about the ampersand following `auto`.

The string argument of `Cell::point` is an optional name.

See paragraph 1.3 for an explanation about why we build the rectangle based on its four sides rather than on its four vertices.

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.

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 5.2), 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 wich 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.

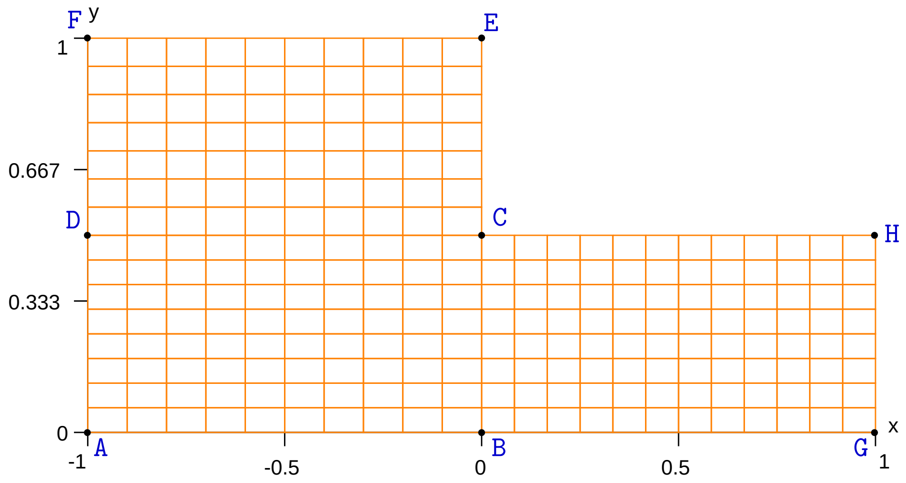
Paragraph 2.6 gives more details about orientation of cells.

Cells are topological entities; they carry no geometry. In particular, points do not have coordinates. Coordinates are stored externally, see paragraph 1.6.

1.3. Building meshes

The example in paragraph 1.1 could have been shortened had we used the overloaded version of `Mesh::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 :

```
auto & environment = Manifold::euclid (2);  // Manifold & environment
auto & xy = environment.coordinate_system ("Lagrange degree one");
// FunctionOnMesh::Function & xy
auto & x = xy[0], & y = xy[1];  // FuncionOnMesh::Function & x, & y
auto & A = Cell::point ("A");  x == -1;  y == 0;   // Cell & A
auto & B = Cell::point ("B");  x ==  0;  y == 0;
auto & C = Cell::point ("C");  x ==  0;  y == 0.5;
auto & D = Cell::point ("D");  x == -1;  y == 0.5;
auto & E = Cell::point ("E");  x ==  0;  y == 1;
auto & F = Cell::point ("F");  x == -1;  y == 1;
auto & G = Cell::point ("G");  x ==  1;  y == 0;
auto & H = Cell::point ("H");  x ==  1;  y == 0.5;
auto & AB = Mesh::segment ( A, B, 10 );   // Mesh & AB
auto & BC = Mesh::segment ( B, C, 8 );
auto & CD = Mesh::segment ( C, D, 10 );
auto & DA = Mesh::segment ( D, A, 8 );
auto & CE = Mesh::segment ( C, E, 7 );
auto & EF = Mesh::segment ( E, F, 10 );
auto & FD = Mesh::segment ( F, D, 7 );
auto & BG = Mesh::segment ( B, G, 12 );
auto & GH = Mesh::segment ( G, H, 8 );
auto & HC = Mesh::segment ( H, C, 12 );
auto & ABCD = Mesh::rectangle ( AB, BC, CD, DA );  // Mesh & ABCD
auto & CEFD = Mesh::rectangle ( CE, EF, FD, CD.reverse() );
auto & BGHC = Mesh::rectangle ( BG, GH, HC, BC.reverse() );
auto & L_shaped = Mesh::join ( ABCD, CEFD, BGHC );  // Mesh & L_shaped
```



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::join method does not work properly. For instance, the two rectangles defined by

```
auto & ABCD = Mesh::rectangle ( A, B, C, D, 10, 8 );
auto & CEFD = Mesh::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).

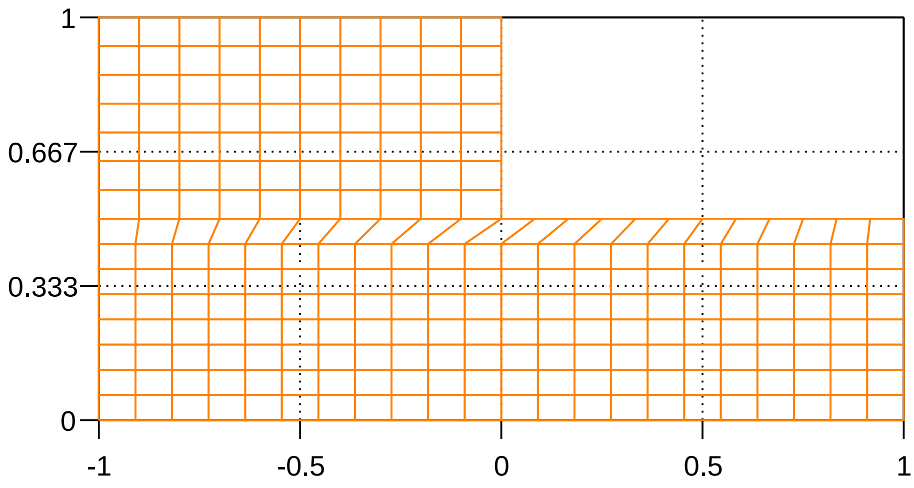
1.4. Another way of joining rectangles

Here is another way of building the same L-shaped mesh as in paragraph 1.3 :

```
auto & environment = Manifold::euclid (2);  // Manifold & environment
auto & xy = environment.coordinate_system ("Lagrange degree one");
// FunctionOnMesh::Function & xy
auto & x = xy[0], & y = xy[1];  // FuncionOnMesh::Function & x, & y
auto & A = Cell::point ("A"); x == -1; y == 0;   // Cell & A
auto & C = Cell::point ("C"); x == 0; y == 0.5;
auto & D = Cell::point ("D"); x == -1; y == 0.5;
auto & E = Cell::point ("E"); x == 0; y == 1;
auto & F = Cell::point ("F"); x == -1; y == 1;
auto & G = Cell::point ("G"); x == 1; y == 0;
auto & H = Cell::point ("H"); x == 1; y == 0.5;
auto & AG = Mesh::segment ( A, G, 22 );  // Mesh & AG
```

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

```
auto & GH = Mesh::segment ( G, H, 8 );
auto & HC = Mesh::segment ( H, C, 12 );
auto & CD = Mesh::segment ( C, D, 10 );
auto & HD = Mesh::join ( HC, CD );       // Mesh & HD
auto & DA = Mesh::segment ( D, A, 8 );
auto & CE = Mesh::segment ( C, E, 7 );
auto & EF = Mesh::segment ( E, F, 10 );
auto & FD = Mesh::segment ( F, D, 7 );
auto & AGHD = Mesh::rectangle ( AG, GH, HD, DA );   // Mesh & ABCD
auto & CEFD = Mesh::rectangle ( CE, EF, FD, CD.reverse() );
auto & L_shaped = Mesh::join ( AGHD, CEFD );   // Mesh & L_shaped
```



The only difference between the two meshes is a slight distortion in the central zone, due to the non-uniform distribution of the vertices along `HD`.

Incidentally, note that the `Mesh::rectangle` method 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 `A`, `B`, `C` and `D`. This can be done even in more than two (geometric) dimensions, see paragraph 1.1.

See paragraph 2.2 for a more complex use of `Mesh::join`.

1.5. Triangular meshes

On a rectangular domain, we can build a mesh of triangles by adding an option to the `Mesh::rectangle` method. For instance, in the example in paragraph 1.3, if we re-write the definition of `BGHC` as

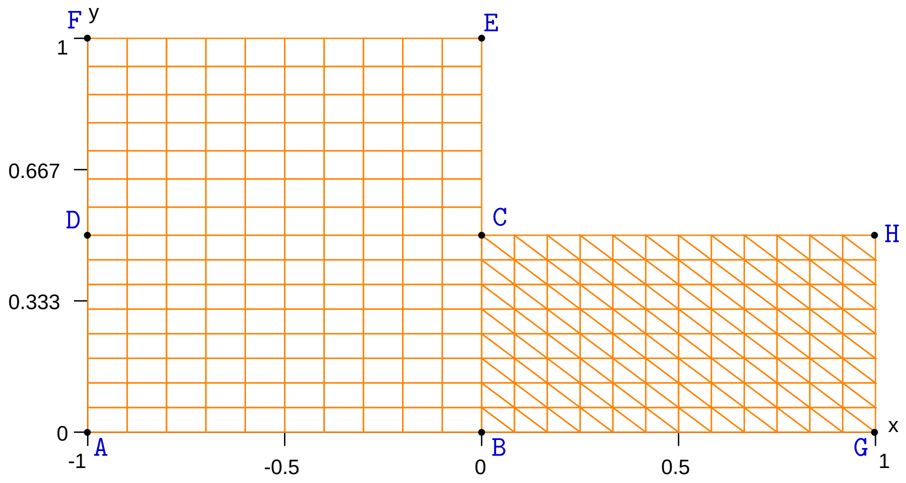
```
auto & BGHC = Mesh::rectangle ( BG, GH, HC, BC.reverse(), tag::with_triangles );
```

we get the mesh shown below.

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

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

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



We can also build meshes on triangular domains and then join them as we wish :
[. . .]

1.6. Fields (outdated)

`ManiFEM` uses `NumericField` objects to store values associated to points (or to other cells). For instance, coordinates of points can be stored in a `NumericField` (see the example in paragraph 1.1). Other values, like the solution of a PDE, can also be stored in a `NumericField`.

`NumericField` objects can live not only on points. For instance, if we work with curved finite elements we may want to store, for each segment, information about the curvature of the segment. We may keep the coordinates of the middle point (a pair of values in 2D, a triplet of values in 3D). Such a `NumericField` should be declared (before the creation of any segment) like this :

```
int d = 2; // or 3
auto middle_coord = NumericField::multi_dim ("size", d, "lives on", "segments");
```

After this declaration, every segment will be created with a heap space for `d` real numbers. These values can then be accessed as `middle_coord(seg)`.

Another example is when we have a coefficient which depends on the space variable but is piecewise constant. We may model such a function by declaring a `NumericField` which lives on two-dimensional cells (if the topological dimension is 2) :

```
auto coef = NumericField::one_dim ("lives on", "2d cells");
```

or on three-dimensional cells (if the topological dimension is 3) :

```
auto coef = NumericField::one_dim ("lives on", "3d cells");
```

In contrast, a function of the space variable can be modelled by a `FunctionOnMesh` object, see paragraph 1.7.

We should be careful to declare all `NumericField` objects before building the meshes. This is because, when creating a cell, `ManiFEM` reserves a heap space for keeping values, accordingly to the existing `NumericField` objects. If we create a point, say, `P`, and afterwards declare a NumericField object, say, x, which lives on points, then any reference like x(P) will produce an error.

As mentioned above, the operator() is used to access the value associated to a particular cell. In contrast, the operator[] is used to access a particular coordinate of a multi-dimensional field. The opposite operation, concatenation of two or more fields, can be achieved through the && operator. For instance, the two pieces of code below are equivalent :

```
auto xy = NumericField::multi_dim ("size", 2, "lives on", "points");
auto x = xy[0], y = xy[1];
```

and

```
auto x = NumericField::one_dimension ("lives on", "points");
auto y = NumericField::one_dimension ("lives on", "points");
auto xy = x && y;
```

1.7. Functions, derivation and integration (outdated)

Unlike NumericFields, FunctionOnMesh objects are used to model functions which vary in a continuum (a cell or a whole mesh). They might be just variables like x and y, or arithmetic expressions of such variables.

They can be differentiated; for instance, the expression (xx*yy-1/xx).deriv(yy) will produce the result xx.

They can also be integrated with the use of an Integrator object, like in the example in paragraph 1.1.

1.8. Partial differential equations (outdated)

Let's consider again the rectangular mesh from paragraph 1.1, this time with only two geometric dimensions.

```
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 xxyy = FunctionOnMesh::from_field ( xy, "Lagrange degree one" );
auto xx = xxyy[0], yy = xxyy[1];
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 = Mesh::segment ( SW, SE, 4, xy );
auto east  = Mesh::segment ( SE, NE, 2, xy );
auto north = Mesh::segment ( NE, NW, 4, xy );
auto west  = Mesh::segment ( NW, SW, 2, xy );
auto rect_mesh = Mesh::rectangle ( south, east, north, west, xy );
```

There is a new NumericField, u, which is meant to hold the values (at vertices) of the final solution of the partial differential equation.

In order to introduce the PDE, by means of a variational formulation, we declare an unknown function `uu`, related to the field `u`, and a test function `w` :

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

We are now in a position to define the variational formulation of our problem, by means of methods `.deriv` and `.integrate`, using a syntax mimicking the mathematical notation of variational calculus :

```
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);
```

Dirichlet-like boundary conditions are declared as

```
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.;
```

We then declare a finite element (Lagrange Q1), a method of integrating functions (Gauss quadrature wih 3 nodes on segments and 9 nodes on rectangles)

```
auto fe = FiniteElement::Lagrange ("Q1");
fe.set_integrator ("on cells of dimension", 1, "gauss", 3, "nodes");
fe.set_integrator ("on cells of dimension", 2, "gauss", 9, "nodes");
```

The following line of code triggers a complex process of discretizing the variational formulation `var_pb` and solving the associated linear system :

```
fe.discretize ( var_pb );
```

2. Cells and meshes

This section gives more details about how cells and meshes are implemented in `ManiFEM`.

2.1. Building cells and meshes

As with any `C++` object, cells can be created by means of their constructor. For instance, the instruction `Cell P(0)` declares a cell of dimension 0 (a vertex), while `Cell * Q = new Cell(0)` declares a pointer to a vertex. We can even de-reference the pointer and declare `Cell & R = * ( new Cell(0) )`. In the subsequent code, `P` and `R` can be used in the same manner, the only meaningful difference being that `P` will be deleted at the end of the block while the cells created by `new` will continue to exist. However, both the cell pointed to by `Q` and the one referenced by `R` will become inaccessible ouside the current syntactic block, unless we keep track of them in some other way (for instance, integrate them in a mesh and keep access to the mesh).

In `ManiFEM`, cells and meshes are implemented as persistent obejcts, built using `new` and thus having no syntactic scope. They are never destroyed; rather, if one is no longer used (e.g. during a remeshing process), it can be explicitly released by the user and is placed in a pool. Such unused cells and meshes will be later re-used. This is achieved through specific methods which build cells, meshes and other objects. We call these methods "factory functions".

For cells, there are factory functions like `Cell::point`, `Cell::segment`, `Cell::triangle` and others. Except for the first one, the ordinary user will not need to use these factory functions, relying instead on factory functions for meshes (see below). A point may be declared as `Cell & R = Cell::point()`. We can even use the `auto` keyword which leaves to the `C++` compiler the task of guessing the type : `auto & R = Cell::point()`. However, we should not forget the ampersand; if we declare `auto R = Mesh::point()`, a compilation error will emerge; see paragraph ? for technical details. The `auto` keyword is extensively used in section 1.

For meshes, there are factory functions like `Mesh::segment`, `Mesh::loop`, `Mesh::rectangle` and others. Internally, these build new cells by using factory functions for cells. Note that `Mesh::segment` is quite different from `Cell::segment`. The latter builds a one-dimensional cell, the former produces a mesh which is a chain of one-dimensional cells.

The intention of the authors is that the final user employ (reference to) objects of types `Cell` and `Mesh` rather than pointers. However, in specific cases the use of pointers cannot be avoided, see e.g. paragraphs 2.2 and 2.8.

Recall that, in `ManiFEM`, cells and meshes are oriented. When a cell or mesh is declared, it is built as positive and has no reverse. Its reverse is a negative cell or mesh and will be built only if necessary. Cells and meshes 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 or mesh on-the-fly and returns it. Paragraph 2.6 gives a more detailed explanation about orientation of cells and meshes. See also paragraph 5.2.

The ordinary user will not need the `reverse` method for cells. However, we may see it occasionaly for meshes. In the example in paragraph 1.3, only meshes `CD` and `BC` have reverse (because only these are needed).

Factory functions are intensively used in `ManiFEM`. Besides cells and meshes, they are used e.g. for building iterators (see paragraph 2.5), functions, variational formulations, finite elements and integrators.

As a general rule, we use lower-case letters for factory functions and capital initial(s) for class names; see also paragraph 5.5.

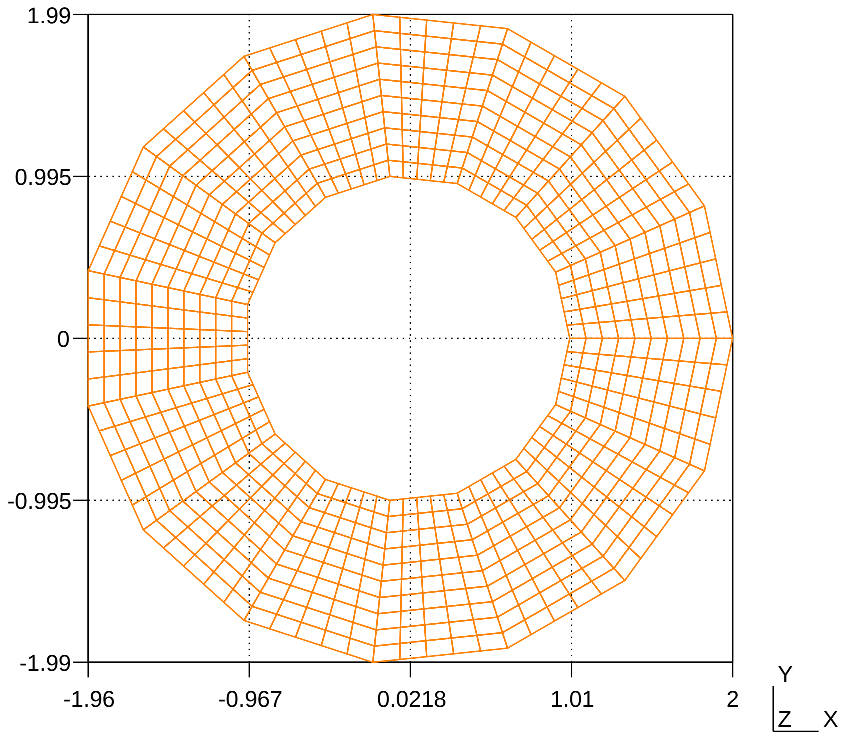
2.2. A ring-shaped mesh

If we want to create many meshes within a cycle, we must use pointers to `Meshes`. Since the factory functions return references to `Mesh` objects, we must keep the address of these objects in a pointer.

Note how we use a version of `Mesh::join` taking as argument a list of pointers to `Meshes`.

```
Manifold & environment = Manifold::euclid (2);
FunctionOnMesh::Function & xy =
  environment.coordinate_system ("Lagrange degree one");
FunctionOnMesh::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 & A = Cell::point (); x == 1.; y == 0.;
Cell & B = Cell::point (); x == 2.; y == 0.;
Mesh & ini_seg = Mesh::segment ( A, B, radial_divisions );
// we will need to vary ini_seg, A and B, so we use pointers :
Mesh * prev_seg_p = & ini_seg;
Cell * A_p = &A, * B_p = &B;
list <Mesh*> l;

for ( short int i = 1; i < n_sectors; i++ )
{ double theta = i * step_theta;
  // we build two new points
  Cell & C = Cell::point (); x == cos(theta);    y == sin(theta);
  Cell & D = Cell::point (); x == 2.*cos(theta); y == 2.*sin(theta);
  // and three new segments
  Mesh & BD = Mesh::segment ( *B_p, D, rot_divisions );
  Mesh & DC = Mesh::segment ( D, C, radial_divisions );
  Mesh & CA = Mesh::segment ( C, *A_p, rot_divisions );
  Mesh & rect = Mesh::rectangle ( *prev_seg_p, BD, DC, CA );
  l.push_back ( &rect );
  prev_seg_p = & ( DC.reverse() );
  A_p = &C; B_p = &D;                                       }

// we now build the last sector, and close the ring
// prev_seg_p, A_p and B_p have rotated during the construction process
// but ini_seg, A and B are the same, initial, ones
```

```
Mesh & outer = Mesh::segment ( *B_p, B, rot_divisions );
Mesh & inner = Mesh::segment ( A, *A_p, rot_divisions );
Mesh & rect = Mesh::rectangle ( outer, ini_seg.reverse(), inner, *prev_seg_p );
l.push_back ( &rect );

Mesh & ring = Mesh::join ( l );
ring.export_msh ("ring.msh");
```



2.3. 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.cells[0]` points to a list of pointers to cells holding all vertices of that mesh, `msh.cells[1]` points to a list of pointers to cells holding all segments and `msh.cells[2]` points to a list of pointers to cells holding all triangles.

Thus, if we want to iterate over all segments of the mesh, we can use a loop like below.

```
list<Cell*>::iterator it;
for ( it = msh.cells[1]->begin(); it != msh.cells[1]->end(); it++ )
{ Cell * seg = *it;  do_something_to (*seg);  }
```

Note that the above only works for a positive mesh. See paragraph 2.5 for 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.

```
list<Cell*> * li = hex.boundary().cells[0];
list<Cell*>::iterator it;
for ( it = li->begin(); it != li->end(); it++ )
{ Cell * P = *it;  do_something_to (*P);  }
```

Note that in the above we have no guarantee about the order in which the vertices will show up in the loop. See paragraph 2.5 for other ways of iterating over cells, some of which allow us to control the order.

2.4. Cells and segments

As explained in paragraph 1.2, the class `Cell` is used to hold points, segments, triangles and other types of cells.

However, in the particular case of segments, it would be nice if we had specific methods to get the base and the tip of that segment.

Recall that in `ManiFEM` a segment is roughly a pair of points, one with a positive sign attached and the other one with a negative sign attached. We call "tip" the positive one and "base" the negative one (this convention is related to integration of a function along that segment).

If we wanted to get the tip of a segment `seg` without a dedicated method, we would have to search through the list `seg.boundary().cells[0]` and pick the positive one. This would be cumbersome and time-consuming, so `ManiFEM` implements segments as a subclass `Segment` derived from `Cell`. The class `Segment` holds internally pointers to the base and the tip of the segment and provides two methods, `base` and `tip`, which return the respective vertex. Note that `base` returns always a negative zero-dimensional cell while `tip` returns a positive one.

Actually, the `base` and `tip` methods exist for any `Cell` object but, when in debug mode, they produce a run-time error if invoked for a cell of dimension other than 1.

2.5. Iterators over cells

As explained in paragraph 1.2, a mesh is essentially a list 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 should use a code like

```
list<Cell*>::iterator it;
for ( it = msh.cells[2]->begin(); it != msh.cells[2]->end(); it++ )
{  Cell * cll = *it;  do_something_to (*cll);  }
```

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

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

See paragraph 5.5 for more details about tags.

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

```
CellIterator it = msh.iter_over ( tag::cells, tag::of_dim, 0, tag::not_oriented );
for ( it.reset(); it.in_range(); it++ )
{  Cell & P = *it;  do_something_to (P);  }
```

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

```
CellIterator it = msh.iter_over ( tag::cells, tag::of_dim, 1, tag::not_oriented );
for ( it.reset(); it.in_range(); it++ )
{  Cell & seg = *it;  do_something_to (seg);  }
```

There is also an iterator which goes through oriented cells :

```
CellIterator it = msh.iter_over ( tag::cells, tag::of_max_dim, tag::oriented );
```

See paragraph 2.6 for a discussion about the orientation of cells inside a mesh.

If we only want to know how many cells there are in a certain mesh, instead of using `msh.cells[d]->size()` (`d` being the desired dimension of the cells) we should use the method `number_of` :

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

or, for cells of dimension equal to the dimension of the mesh :

```
size_t n = msh.number_of ( tag::cells, tag::of_max_dim );
```

One-dimensional meshes have more specific iterators. Unlike the iterators above, which sweep the mesh in a rather impredictible order, the iterators below follow the natural order of the cells (either vertices or segments) given by the topology of the mesh.

A one dimensional mesh can be either an open chain of segments or a closed one (a loop). We can iterate over all vertices along an open chain, beginning at one end and finishing at the other end :

```
CellIterator it1 = msh.iter_over ( tag::vertices, tag::along );
for ( it1.reset(); it1.in_range(); it1++ )
{  Cell & P = *it1;  do_something_to (P);  }
```

Or, we can iterate over all segments :

```
CellIterator it2 = msh.iter_over ( tag::segments, tag::along );
```

Note that `it1` produces positive points, while `it2` produces oriented segments (positive or negative).

There are also reversed versions of these iterators (they go backwards) :

```
CellIterator it = msh.iter_over ( tag::vertices, tag::along, tag::reverse );
CellIterator it = msh.iter_over ( tag::segments, tag::along, tag::reverse );
```

Similar iterators exist for a closed chain of segments (a loop) :

```
CellIterator it = msh.iter_over ( tag::vertices, tag::around );
CellIterator it = msh.iter_over ( tag::segments, tag::around );
CellIterator it = msh.iter_over ( tag::vertices, tag::around, tag::reverse );
CellIterator it = msh.iter_over ( tag::segments, tag::around, tag::reverse );
```

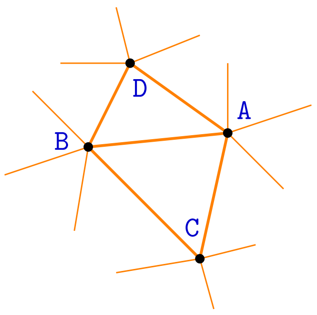
Iterators over closed chains (loops) will start the iteration process at some vertex or segment (the first one in the internal list). If we want to start at a specific location, we can make a `reset` call with one argument. For iterators over vertices, this argument should be a vertex, while for iterators 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 also be used for an open chain, but beware : the vertices or segments previous to the provided argument will not show up in the iteration process.

See paragraph 5.4 for some technical details.

2.6. 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, `tri_mesh` will have no reverse mesh and the triangles composing it 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 2.5. 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 `ManiFEM` only provides an iterator over oriented cells of maximum dimension.

We also understand that there is no difference between these two iterators :

```
CellIterator it1 =
  tri_msh.iter_over ( tag::cells, tag::of_max_dim, tag::oriented );
CellIterator it2 =
  tri_msh.iter_over ( tag::cells, tag::of_dim, 2, tag::not_oriented );
```

because all triangles composing `tri_mesh` should be positive. However, if some of the triangles are negative (which may happen only if `tri_mesh` is built in some fancy, unusual manner) `it1` will behave differently from `it2`.

On the other hand, the two iterators below will probably have different behaviours, depending on which segments happen to be positive :

```
CellIterator it3 =
  ABC.boundary().iter_over ( tag::cells, tag::of_max_dim, tag::oriented );
CellIterator it4 =
  ABC.boundary().iter_over ( tag::cells, tag::of_dim, 1, tag::not_oriented );
```

Also, there is no difference between the two iterators below (both produce positive points).

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

However, the two iterators below are quite different.

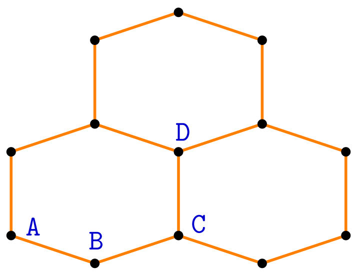
```
CellIterator it7 =
  ABC.boundary().iter_over ( tag::segments, tag::around, tag::reverse );
CellIterator it8 =
  ABC.boundary().reverse().iter_over ( tag::segments, tag::around );
```

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 the mesh `ABC.boundary()` has no reverse initially; however, a reverse mesh is built on-the-fly when we call the `reverse` method in the declaration of `it6`.

2.7. 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 the given mesh. Together with methods `base` and `tip` (see paragraph 2.4), 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, tag::of_max_dim );
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, tag::around );
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 2.5, are implemented internally).

We can navigate towards a neighbour hexagon :

```
Cell & hex_2 = hex_msh.cell_in_front_of (CD);
```

Of course, since we 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 2.8 for a way to check if 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`.

2.8. Navigating at the boundary of a mesh

Consider the example in paragraph 2.7. Suppose you try to get a neighbour hexagon which does not exist :

```
  Cell & no_such_hex = hex_msh.cell_in_front_of (AB);
```

You will get an execution error of the type

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

You may check previously the existence of the neighbour cell this by using a pointer to a `Cell` and a tag `may_not_exist` :

```
  Cell * possible_hex = hex_msh.cell_in_front_of ( AB, tag::may_not_exist );
  if ( possible_hex == NULL ) cout << "no neighbour !" << endl;
  else do_something_to ( *possible_hex );
```

thus avoiding errors at execution time.

3. Fields, functions and variational formulations

3.1. Fields and functions (outdated)

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 = Mesh::segment ( SW, SE, 4, xy );
auto east  = Mesh::segment ( SE, NE, 2, xy );
auto north = Mesh::segment ( NE, NW, 4, xy );
auto west  = Mesh::segment ( NW, SW, 2, xy );
auto rect_mesh = Mesh::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.;
```

4. Finite elements and integrators

Let's take over the example in paragraphs 1.8 and 3.1.

The last few lines of code look rather simple :

```
auto & fe = FiniteElement::Lagrange ("Q1");
fe.set_integrator ("on cells of dimension", 1, "gauss", 3, "nodes");
fe.set_integrator ("on cells of dimension", 2, "gauss", 9, "nodes");
fe.discretize ( var_pb );
```

This simplicity hides from the final user a complicated process : the discretization of the variational formulation by a finite element.

4.2. Finite elements

The notion of a finite element is quite complex. The purpose of a `FiniteElement` is to build a list of functions, say, ψ, defined on our mesh. The linear span of these functions will be a discretized Hilbert space. The `FiniteElement` should replace, in the variational formulation, the unknown function by one ψ, the test function by another ψ and, by evaluating the integrals, obtain the coefficients of a system of linear equations. Some external solver will then solve the system, and it is the job of the finite element to transform back the vector produced by the solver into a function defined on our mesh.

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 `ManiFEM` 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). Recall that in `ManiFEM` the vertices do not have necessarily any number attached. When we create a `FiniteElement::Lagrange_Q1` object, it will enumerate vertices of the mesh if a mesh exists already, and it will prepare the ground for vertices created in the future to be enumerated, too. This enumeration is useful to identify a vector of real numbers with the values (at each vertex) of a function defined on the mesh. When docking on a cell, the `FiniteElement::Lagrange_Q1` object will build four "shape functions", a transformation map (from a master element occuppying the square $[-1,1]^2$ to the current cell) and the jacobian of this transformation.

5. Technical details

The present section is meant for those intereseted in developing and extending `ManiFEM`. Of course the ultimate documentation is the source code; this section can be used as a guide through the source code.

5.1. Hidden, private, public

In `ManiFEM`, there are no `private` class members of 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 other methods. Often, the latter have names beginning with `hidden_` or `hidden::`. Also, in the source code there are comments like

```
// do not use directly, let [some other method] do the job
```

As a rule of thumb, there is no reason for you to use in your programs any name beginning with `hidden`, nor any class members or methods not mentioned in this manual. On the other hand, you are encouraged to read them, understand them and hopefully improve them.

5.2. Orientation of cells and meshes

Both classes `Cell` and `Mesh` have an attribute `hidden_reverse` which is `NULL` if the cell or mesh has no reverse, otherwise points to the reverse object. Note that negative cells and meshes must have a reverse, which is a positive cell or mesh. Positive cells and meshes may have no reverse. In the code below, `cll.reverse` simply returns `cll.hidden_reverse` :

```
Cell & cll = ...; // some factory function
... more code ...
Cell * r_cll_p = cll.reverse ( tag::may_not_exist );
```

Note that this is quite different from the code below where a reverse cell is built on-the-fly if needed (also, a reference is returned rather than a pointer) :

```
Cell & cll = ...; // some factory function
... more code ...
Cell & r_cll = cll.reverse();
```

Although the most natural choice for implementing the orientation may seem to be a `sign` attribute of classes `Cell` and `Mesh`, we have chosen a different internal implementation. There is an attribute `positive` in both classes which points to the object itself if the cell or mesh is positive, and points to the reverse object if the cell or mesh is negative. In the former case, `positive == this`; in the latter, `positive == hidden_reverse`. The method `is_positive` hides these details from the user (it simply returns the result of `this == positive`).

5.3. About init_cell

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.

5.4. About hook

Both Cell and Mesh classes have an attribute called hook which allows the user to attach supplementary data to a cell without having to declare a derived class. The hook attribute is a map from strings to void pointers. The user may insert (pointers to) data of any type into this map by converting them to void* through explicit casting. In order to be used later, this data must be converted back to the original type. The authors of ManiFEM are aware that this approach is prone to errors. The progammer must be very careful when casting back and forth between types, and must keep record of keys of the hook map which are already in use (recall that these keys are strings).

Below is a list of keys already used in maps Cell::hook and Mesh::hook. Note that they exist in positive cells and meshes, not in negative ones.

The factory function Mesh::segment inserts pointers towards the first and last nodes as hook["first point"] and hook["last point"]. The method Mesh::join honors this convention by setting these pointers as appropriate, if applied to Meshes having these attributes. These attributes are used in the reset method of iterators over cells of open chains msh.iter_over (tag::vertices, tag::along), msh.iter_over (tag::segments, tag::along), msh.iter_over (tag::vertices, tag::along, tag::reverse), msh.iter_over (tag::segments, tag::along, tag::reverse) (described in paragraph 2.5).

If a one-dimensional mesh is built by some means other than Mesh::segment and Mesh::join, the hook map may have no keys "first point" and "last point". A safe way of getting the first and last vertices is to use an iterator like in

```
CellIterator it1 = msh.iter_over ( tag::vertices, tag::along );
it1.reset(); Cell & first_ver = *it1;
CellIterator it2 = msh.iter_over ( tag::vertices, tag::along, tag::reverse );
it2.reset(); Cell & last_ver = *it2;
```

This will always provide the correct vertices because the reset method looks for the key "first point" or "last point" and, if it doesn't find it, performs a direct search to find the first or last vertex. Beware that, if the mesh is not an open chain (if it's a loop), this search may last forever. The above method has the advantage of working correctly even for a negative mesh.

The factory function `Mesh::rectangle` inserts pointers to two maps `hook["sides"]` and `hook["corners"]`. The first one of these maps holds pointers to the four sides of the rectangle under the keys `"south"`, `"east"`, `"north"` and `"west"`. The second map holds pointers to the four corners of the rectangle under the keys `"SW"`, `"SE"`, `"NE"` and `"NW"`.

5.5. Namespaces and class names

All names in `ManiFEM` are wrapped into the namespace `ManiFEM`. We recomend to be `using namespace ManiFEM` in your code, otherwise the text will become cumbersome. For instance, you will have to write `ManiFEM::Mesh::rectangle` instead of `Mesh::rectangle`, and so on. We are `using namespace ManiFEM` in the source code of `ManiFEM` itself, as well as in this manual.

As a general rule, namespaces and class names are written with capital initial letter : `Cell`, `Mesh`, `Integrator`, `FiniteElement`, `VariationalFormulation`. In contrast, factory functions (see paragraph 2.1) are named with lower case letters, e.g. `Mesh::rectangle` or `msh.iter_over`.

The namespace `tag` (or `ManiFEM::tag` if you are not `using namespace ManiFEM`) is an exception to the above rule. This namespace contains classes (structures, actually) which contain no data; most of them have only one instance. Only their type is useful, at compile time, for overloading functions (mainly factory functions, see e.g. paragraph 2.5). 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`.

5.6. Frequent errors at compile time

```
use of deleted function 'ManiFEM::Cell::Cell(const ManiFEM::Cell&)
-- or --
use of deleted function 'ManiFEM::Mesh::Mesh(const ManiFEM::Mesh&)
```

Cells and meshes cannot be copied. You cannot ask for things like `cell_2 = cell_1` or `mesh_2 = mesh_1`. Also, when initializing a cell or a mesh through a factory function, you shoud use a reference; `Cell tri = Cell::triangle (...)` is illegal, as well as `auto tri = Cell::triangle (...)`; use `Cell & tri = Cell::triangle (...)` or `auto & tri = Cell::triangle (...)`.

5.7. Frequent errors at run time

If your program produces impredictible, random errors at run-time, e.g. throws `Segmentation fault`, try to run 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 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 == positive`). See paragraphs 2.6 and 5.2 for more details about orientation of cells and meshes.

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. Some iterators have versions ending in `tag::not_oriented` which produce positive cells. Iterators with `tag::vertices_along` or `tag::vertices_around` produce positive points, while iterators with `tag::segments_along` or `tag::segments_around` produce oriented segments (which may be positive or negative), see paragraph 2.5.

```
ManiFEM::Cell::Cell(short int, bool): Assertion 'is_segment or ( d != 1 )' failed.
```

Did you try to build a one-dimensional cell by using directly the `Cell` constructor ? Don't. Always use the factory function `Cell::segment`. When you build a cell whose dimension `d` is unknown at compile time, use an `if` statement to give special treatment to the case `d == 1`. `Segments` belong to a separate class, derived from `Cell` (see paragraph 2.4). The fact that `ManiFEM` stores them as `Cell` objects, rather than as `Segment` objects, can be considered poor design. We endorse this criticism with no further comments.

```
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 2.8.

Index