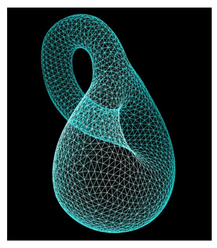
Meshes and finite elements on manifolds with ManiFEM

AUTHORS: Cristian Barbarosie, Anca-Maria Toader, Sérgio Lopes
R&D UNIT: CMAF-CIO CONTACT: cabarbarosie@fc.ul.pt

What is ManiFEM ?



ManiFEM is 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 dealing 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, it has been written with the goal of being conceptually clear and easy to read. We hope it will prove particularly useful for users who want fine control on the mesh, e.g. for implementing their own meshing or remeshing algorithms. ManiFEM is free software; it is copyrighted by Cristian Barbarosie under the GNU Lesser General Public Licence.
The home page of is `http://manifem.rd.ciencias.ulisboa.pt` (where the manual can be found).
The source code can be found at `https://github.com/cristian-barbarosie/manifem`.

The manual is divided in 13 sections describing ManiFEM with increasing degree of technical detail :

1 General description A quick overview of ManiFEM's capabilities
2 Meshes and manifolds; patchwork Shows how to build meshes by joining simple shapes, like patches, some of them on manifolds.
3 Progressive mesh generation; knitting Shows how to build meshes starting from their boundary alone, some of them on manifolds.
4 Meshing three-dimensional domains Work in progress.
5 Fields, functions and variational formulations Some details about functions, a lot of work to do.
6 Finite elements and integrators Shows two examples of finite element computations, still rudimentary.
7 Quotient manifolds and periodic boundary conditions Describes meshes on quotient manifolds.
8 More on manifolds Describes manifolds, particularly the projection defined manifolds.
9 Cells, meshes, iterators Gives details about neighbourhood relations between cells in a mesh, orientation and iterators.
10 Remeshing Some simple examples showing how to manipulate a mesh.
11 Technical details Various implementation detais, programming style, compilation options, frequent errors.
12 Internal details Mainly drawings intended to support the developer in understanding the source code.
13 Frequently asked questions

Sections 1 to 8 should be accessible to readers who have some knowledge of C++ but are not necessarily experts in C++ programming. Sections 9 and 10 should be useful for users who want finer control on the mesh, e.g. for implementing their own remeshing algorithms Sections 11 and 12 give technical details, mainly for those interested in developing and extending ManiFEM.

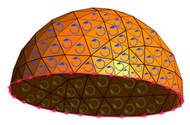
Cells and meshes

In ManiFEM, all basic constituents of meshes are called "cells". Points are zero-dimensional cells, segments are one-dimensional cells, triangles are two-dimensional cells, and so on. Roughly speaking, a mesh is a collection of cells of the same dimension. 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). 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. We can think of an oriented segment as an arrow pointing from its negative extremity (base) towards its positive extremity (tip). We can think of an oriented polygon as having an arrow attached to each of it sides, or we can imagine a small oriented circle inside the polygon.

Oriented segment and oriented pentagon

A one-dimensional oriented mesh can be thought of as a chain of arrows, each one pointing to the next segment's base.

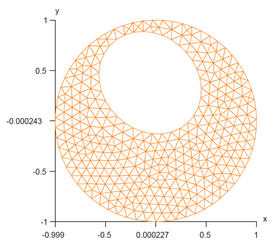
A two-dimensional oriented triangular mesh can be thought of as a web of triangles, each triangle having a small oriented circle inside. The orientations of neighbour cells must be compatible : each segment must be seen in opposite orientations from the point of view of its two neighbor triangles.



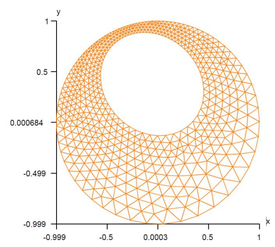
Note also that an orientation of a mesh defines an orientation of its boundary. This convention is used by Stokes' theorem.
Outer and inner boundaries of a domain are first meshed and should have compatible orientation, that is the inner boundary should be reversed, in order to permit a consistent meshing of the domain. In the case of a disk with a hole, the ManiFEM code looks like the one on the right:

```
// builds a circle with an eccentric hole

#include "maniFEM.h"
using namespace maniFEM;

int main ( )

{   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];

    double d = 0.065;

    Manifold circle = RR2.implicit ( x*x + y*y == 1. );
    Mesh outer ( tag::progressive, tag::entire_manifold, circle, tag::desired_length, d );

    double y0 = 0.36;
    Manifold ellipse = RR2.implicit ( x*x + (y-y0)*(y-y0) + 0.3*x*y == 0.25 );
    Mesh inner ( tag::progressive, tag::entire_manifold, ellipse, tag::desired_length, d );

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

    RR2.set_as_working_manifold();
    Mesh disk ( tag::progressive, tag::boundary, circles, tag::desired_length, d );

    disk.export_msh ("disk.msh");
    disk.draw_ps ("disk.eps");

    std::cout << "produced files disk.msh and disk.eps" << std::endl;
}
```

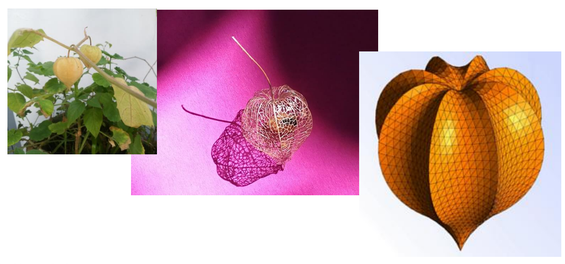
ManiFEM generates the regular triangular mesh



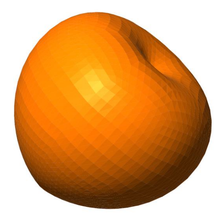

```
Function d = 0.03 + 0.04 * ( ( x + 0.3 ) * ( x + 0.3 ) + ( y - 0.9 ) * ( y - 0.9 ) );

Manifold circle = RR2.implicit ( x*x + y*y == 1. );
Mesh outer ( tag::progressive, tag::entire_manifold, circle, tag::desired_length, d );

double y0 = 0.37;
Manifold ellipse = RR2.implicit ( x*x + (y-y0)*(y-y0) + 0.3*x*y == 0.25 );
Mesh inner ( tag::progressive, tag::entire_manifold, ellipse, tag::desired_length, d );

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

RR2.set_as_working_manifold();
Mesh disk ( tag::progressive, tag::boundary, circles, tag::desired_length, d );
```

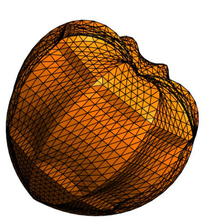


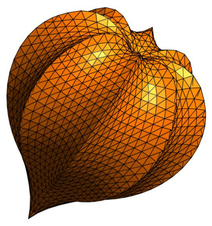
Organic shapes like
a Physalis (Chinese lantern)
can be generated with ManiFEM.



```
// a physalis-like surface

#include "maniFEM.h"

using namespace maniFEM;


int main ( )

{   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 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 );

    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 C1D ( tag::segment, C1.reverse(), D, tag::divided_in, 10 );
    Cell B2 ( tag::vertex );  x(B2) =  0.707;  y(B2) =  0.707;  z(B2) =  1.;
    Cell C2 ( tag::vertex );  x(C2) =  0.707;  y(C2) =  0.707;  z(C2) =  0.;
    // and many other vertices and segments ...

    RR3.set_as_working_manifold();
    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 parches ...

    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 );

    std::list<Mesh>::iterator it1;
    for ( it1 = lm.begin(); it1 != lm.end(); it1++ )
    {   Mesh sect = *it1;
        CellIterator it2 = sect.iterator ( tag::over_vertices );
        for ( int i = 1; i < 20; i++ )
        for ( it2.reset(); it2.in_range(); it2++ )
        {   Cell ver = *it2;
            if ( ver.is_inner_to ( sect ) ) sect.baricenter ( ver );  }  }

    fisalis.export_msh ("physalis.msh");

    std::cout << "produced file physalis.msh" << std::endl;
}
```





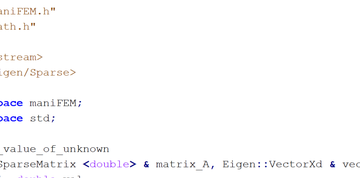


The meshing process is done in three steps as illustrated on the right side.

Finite Elements in ManiFEM

We present an example about the Laplace operator with non-homogeneous Dirichlet boundary conditions. The domain is an annulus (a disk with a hole). The current approach is rather low-level. We are working hard to make ManiFEM understand statements describing variational formulations given as C++ objects. When this part of the code is done, the programming style will become much more elegant and compact.
The notion of a finite element is quite complex. The purpose of a `FiniteElement` is to build a list of functions, say, φ, 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 φ , the test function by another φ and, by evaluating the integrals, obtain the coefficients of a system of linear equations. Some external solver (`Eigen`) 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). 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 occupying the square $[-1, 1] \times [-1, 1]$ 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`.

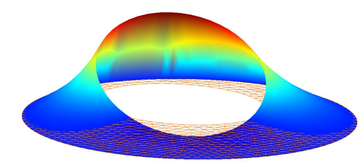
```
#include "maniFEM.h"
#include "math.h"

#include <fstream>
#include <Eigen/Sparse>

using namespace maniFEM;
using namespace std;

void impose_value_of_unknown
(   Eigen::SparseMatrix <double> & matrix_A, Eigen::VectorXd & vector_b,
    size_t i, double val                                                 )

// in a system of linear equations, destroy equation 'i' and impose u(i) = val
// change also column 'i' of the matrix, just to preserve symmetry

// used for imposing Dirichlet boundary conditions

{   size_t size_matrix = matrix_A.innerSize();
    vector_b(i) = val;
    for ( size_t j = 0; j < size_matrix; j++ )
        matrix_A.coeffRef ( i, j ) = 0.;
    matrix_A.coeffRef ( i, i ) = 1.;
    for ( size_t j = 0; j < size_matrix; j++ )
    {   if ( i == j ) continue;
        vector_b(j) -= matrix_A.coeffRef ( i, j ) * val;
int main ()

{   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];

    double d = 0.065;

    Manifold circle = RR2.implicit ( x*x + y*y == 1. );
    Mesh outer ( tag::progressive, tag::entire_manifold, circle, tag::desired_length, d );

    double y0 = 0.36;
    Manifold ellipse = RR2.implicit ( x*x + (y-y0)*(y-y0) + 0.3*x*y == 0.25 );
    Mesh inner ( tag::progressive, tag::entire_manifold, ellipse, tag::desired_length, d );

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

    RR2.set_as_working_manifold();
    Mesh disk_with_hole ( tag::progressive, tag::boundary, circles, tag::desired_length, d );

    // declare the type of finite element
    FiniteElement fe ( tag::with_master, tag::triangle, tag::Lagrange, tag::of_degree, 1 );
    Integrator integ = fe.set_integrator ( tag::Gauss, tag::tri_6 );

    // there will be a more elegant and efficient way of producing the numbering
    std::map < Cell::Core *, size_t > numbering;
    { // just a block of code for hiding 'it' and 'counter'
    CellIterator it = disk_with_hole.iterator ( tag::over_vertices );
    size_t counter = 0;
```



```
for ( it.reset() ; it.in_range() ; it++ )
{   Cell p = *it;  ++counter;  numbering [p.core] = counter;  }
} // just a block of code

size_t size_matrix = disk_with_hole.number_of ( tag::vertices );
assert ( size_matrix == numbering.size() );
std::cout << "global matrix " << size_matrix << "x" << size_matrix << std::endl;
Eigen::SparseMatrix <double> matrix_A ( size_matrix, size_matrix );
Eigen::VectorXd vector_b ( size_matrix ), vector_sol ( size_matrix );
vector_b.setZero();

// run over all triangle cells composing disk_with_hole
{ // just a block of code for hiding 'it'
int counter = 0;
CellIterator it = disk_with_hole.iterator ( tag::over_cells_of_max_dim );
for ( it.reset(); it.in_range(); it++ )
{   Cell small_tri = *it;
    fe.dock_on ( small_tri );
    // run twice over the four vertices of 'small_tri'
    CellIterator it1 = small_tri.boundary().iterator ( tag::over_vertices );
    CellIterator it2 = small_tri.boundary().iterator ( tag::over_vertices );
    for ( it1.reset(); it1.in_range(); it1++ )
    for ( it2.reset(); it2.in_range(); it2++ )
    {   Cell V = *it1, W = *it2;
        Function 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_tri' 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 );
    }  }
} // just a block of code

// impose Dirichlet boundary conditions  u = xy
{ // just a block of code for hiding 'it'
CellIterator it = outer.iterator ( tag::over_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, 0. );  }
} { // just a block of code for hiding 'it'
CellIterator it = inner.iterator ( tag::over_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) );  }

// solve the system of linear equations
Eigen::ConjugateGradient < Eigen::SparseMatrix<double>,
                           Eigen::Lower|Eigen::Upper     > cg;
cg.compute ( matrix_A );
vector_sol = cg.solve ( vector_b );

disk_with_hole.export_msh ("coroa-Dirichlet.msh", numbering );
{ // just a block of code for hiding variables
ofstream solution_file ("coroa-Dirichlet.msh", fstream::app );
solution_file << "$NodeData" << endl;
solution_file << "1" << endl;   // one string follows
solution_file << "\"temperature\"" << endl;
solution_file << "1" << endl;   //  one real follows
solution_file << "0.0" << endl;  // time [??]
solution_file << "3" << endl;   // three integers follow
solution_file << "0" << endl;   // time step [??]
solution_file << "1" << endl;  // scalar values of u
solution_file << disk_with_hole.number_of ( tag::vertices ) << endl;
// number of values listed below
CellIterator it = disk_with_hole.iterator ( tag::over_vertices );
for ( it.reset(); it.in_range(); it++ )
{   Cell P = *it;
    size_t i = numbering[P.core];
    solution_file << i << " " << vector_sol[i-1] << std::endl;   }
} // just a block of code

std::cout << "produced file coroa-Dirichlet.msh" << std::endl;

return 0;
}
```

For Homogenization Theory purposes, ManiFEM is prepared for solving periodic problems. For a given micro geometry, the cellular problem is solved numerically using a mesh on the quotient torus. Given an effective strain, the following code computes the cellular solutions in terms of which the homogenized elastic tensor is expressed.

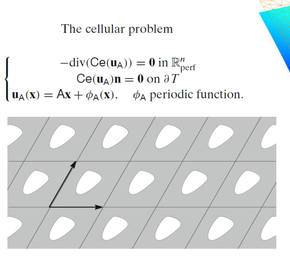
```
// solve a celullar problem
// square periodicity, triangular elements, circular hole

#include "maniFEM.h"
#include <fstream>
#include <Eigen/Sparse>
#include <Eigen/OrderingMethods>
using namespace maniFEM;

int main ( )

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

    size_t n = 20;
    double d = 2.6 / double(n);

    Cell A ( tag::vertex );  x(A) = -1.3;  y(A) = -1.3;
    Cell B ( tag::vertex );  x(B) =  1.3;  y(B) = -1.3;
    Cell C ( tag::vertex );  x(C) =  1.3;  y(C) =  1.3;
    Cell D ( tag::vertex );  x(D) = -1.3;  y(D) =  1.3;

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

    Manifold circle = RR2.implicit ( x*x + y*y == 0.7 );
    Mesh inner ( tag::progressive, tag::entire manifold, circle, tag::desired length, d );

    Mesh bdry ( tag::join, AB, BC, CD, DA, inner.reverse() );

    RR2.set_as_working_manifold();
    Mesh square ( tag::progressive, tag::boundary, bdry, tag::desired_length, d );

    Mesh torus = square.fold ( tag::identify, AB, tag::with, CD.reverse(),
                               tag::identify, BC, tag::with, DA.reverse(),
                               tag::use_existing_vertices                 );

    // declare the type of finite element
    FiniteElement fe ( tag::with_master, tag::triangle, tag::Lagrange, tag::of_degree, 1 );
    Integrator integ = fe.set_integrator ( tag::Gauss, tag::tri_4 );

    // we number all nodes in 'square', not only those belonging to 'torus'
    std::map < Cell, size_t > numbering;
    { // just a block of code for hiding 'it' and 'counter'
    CellIterator it = square.iterator ( tag::over_vertices );
    size_t counter = 0;
    for ( it.reset() ; it.in_range(); it++ )
    {   Cell V = *it;  numbering[V] = counter;  ++counter;  }
    assert ( counter == numbering.size() );
    } // just a block of code

    size_t size_matrix = numbering.size();
    std::cout << "global matrix " << size_matrix + 1 << "x" << size_matrix << std::endl;
    Eigen::SparseMatrix < double > matrix_A ( size_matrix + 1, size_matrix );

    matrix_A.reserve ( Eigen::VectorXi::Constant ( size_matrix, 8 ) );
    // since we will be working with a mesh of triangles,
    // there will be, in average, eight non-zero elements per column
    // the diagonal entry plus six neighbour vertices plus the last equation

    // we fill the main diagonal with ones
    // then we put zero for vertices belonging to 'torus'
    { // just a block of code for hiding 'it'
    for ( size_t i = 0; i < size_matrix; i++ ) matrix_A.coeffRef ( i, i ) = 1.;
    CellIterator it = torus.iterator ( tag::over_vertices );
    for ( it.reset(); it.in_range(); it++ )
    {   Cell V = *it;
        matrix_A.coeffRef ( numbering[V], numbering[V] ) = 0.;  }
    } // just a block of code for hiding 'it'

    Eigen::VectorXd vector_b ( size_matrix + 1 ), vector_sol ( size_matrix );
    vector_b.setZero();

    xy = Manifold::working.coordinates();
    x = xy[0];  y = xy[1];
```



```
// macroscopic temperature gradient
std::vector < double > macro_grad { 1., 0.5 };
Function::Jump jump_of_solution = macro_grad[0] * x.jump() + macro_grad[1] * y.jump();

// run over all square cells composing 'torus'
{ // just a block of code for hiding 'it'
CellIterator it = torus.iterator ( tag::over_cells_of_max_dim );
for ( it.reset(); it.in_range(); it++ )
{   Cell small_tri = *it;
    fe.dock_on ( small_tri, tag::spin );
    // run twice over the four vertices of 'small_tri'
    CellIterator it_V = small_tri.boundary().iterator ( tag::over_vertices );
    for ( it_V.reset(); it_V.in_range(); it_V++ )
    {   Cell V = *it_V;
        // perhaps implement an interator returning a vertex and a segment
        Cell seg = small_tri .boundary(). cell_in_front_of ( V );
        Cell W = V;
        double jump_V_W = 0.;
        while ( true )
        {   assert ( W == seg.base().reverse() );
            // V may be the same as W, no problem about that
            Function psi_V = fe .basis_function ( V ),
                     psi_W = fe .basis_function ( W ),
                     d_psi_V_dx = psi_V .deriv ( x ),
                     d_psi_V_dy = psi_V .deriv ( y ),
                     d_psi_W_dx = psi_W .deriv ( x ),
                     d_psi_W_dy = psi_W .deriv ( y );
            // 'fe' is already docked on 'small_tri' so this
            // will be the domain of integration
            double integral = fe.integrate ( d_psi_V_dx * d_psi_W_dx
            + d_psi_V_dy * d_psi_W_dy );
            matrix_A.coeffRef ( numbering[V], numbering[W] ) += integral;
            vector_b ( numbering[V] ) -= jump_V_W * integral;
            jump_V_W += jump_of_solution ( seg.spin() );
            W = seg.tip();
            if ( V == W ) break;
            seg = small_tri .boundary() .cell_in_front_of ( seg.tip() );
        // here  jump_V_W  should be zero again
        // but we do not assert that, rounding errors may mess up things
        }  }
} // just a block of code for hiding 'it'

// we add, as last equation, the condition of zero average
// actually, a more rudimentary condition : a line of ones
for ( size_t i = 0; i < size_matrix; i++ ) matrix_A.coeffRef ( size_matrix, i ) = 1.;

matrix_A .makeCompressed();

Eigen::SparseQR < Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int> > solver;

solver.compute ( matrix_A );
if ( solver.info() != Eigen::Success )
{   std::cout << "Eigen solver.compute failed" << std::endl;
    exit ( 0 );                                              }

vector_sol = solver.solve ( vector_b );
if ( solver.info() != Eigen::Success )
{   std::cout << "Eigen solver.solve failed" << std::endl;
    exit ( 0 );                                            }
RR2 .set_as_working_manifold();
xy = Manifold::working.coordinates();
x = xy[0];  y = xy[1];
square.export_msh ("cell.msh", numbering );
```

The cellular problem

$$\begin{cases} -\mathrm{div}(\mathrm{Ce}(\mathbf{u_A})) = \mathbf{0} \text{ in } \mathbb{R}^n_{\mathrm{perf}} \\ \mathrm{Ce}(\mathbf{u_A})\mathbf{n} = \mathbf{0} \text{ on } \partial T \\ \mathbf{u_A}(\mathbf{x}) = \mathbf{A}\mathbf{x} + \phi_{\mathbf{A}}(\mathbf{x}), \quad \phi_{\mathbf{A}} \text{ periodic function.} \end{cases}$$



Periodically perforated plane $\mathbb{R}^2_{\mathrm{perf}}$

AKNOWLEDGEMENTS

Development of ManiFEM is supported by National Funding from FCT – Fundação para a Ciência e a Tecnologia (Portugal), through Faculdade de Ciências da Universidade de Lisboa and Centro de Matemática, Aplicações Fundamentais e Investigação Operacional, project UID/MAT/04561/2020.
Anabela Silva contributed with drawings and source text coloring.