Prof. Dr. Karsten Urban
Kristina Steih

Institute for Numerical Mathematics
Ulm University
Summer Term 2014

Numerical Finance – Sheet 10

(Exercise Class July 9th, 2014)

Programming Exercise 1: Uniform Triangulations 2D (21 points)

A triangulation is a division of a plane into triangles. In future exercises, we will use such a triangulation as discretization for the 2D FEM. As in 1D, the discretization has to be refined – often several times – in order to obtain better approximation results.

Such triangulations (or meshes) are usually given by

1. a list of 2D coordinates, containing all nodes (or vertices) that appear in the mesh,
2. a list of elements, where each element is given by the index numbers of its three vertices,

see tables (a) and (b).

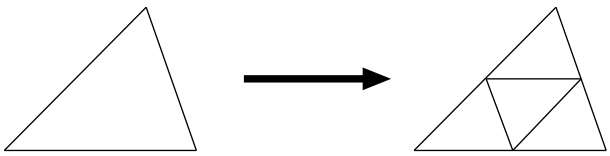
Index	x	y
1	0.0	0.0
2	1.0	0.0
3	0.0	1.0
4	1.0	1.0
5	0.5	0.5

(a) Coordinates

1	2	5
2	4	5
4	3	5
3	1	5

(b) Elements

A uniform refinement of a triangulation consists of a so-called *red refinement* of each element: new coordinates at the centers of all three edges are added to the coordinate list and the element is divided into 4 smaller triangles:



In the material on the homepage, you will find header files for the following classes:

- `Point2D`: A class that wraps a 2D coordinate, i.e. a vertex in the mesh. It provides a function to compute the center between two points.
- `Polygon`: A storage class for general polygons, which is used to handle the triangles that are the elements in the mesh.
- `Triangulation`: Contains the 2D discretization (i.e. the coordinate and element lists) of the mesh. Provides functions for the uniform refinement of the discretization as well as for reading an initial mesh, etc.

a) Implement the missing function implementations in `Point2D` and `Triangulation`.

b) Use the given file `test_triangulation.cpp` to verify that your refinement is correct. Plot the triangulations e.g. with `gnuplot ''outputfile'' w l`.

Hints:

- ***Edges:*** Each edge can be uniquely identified by (the indizes of) its two end points. Note that obviously, the edge from coordinate 1 to coordinate 2 is the same as the one from coordinate 2 to coordinate 1. In order to avoid confusion, we usually store edges with the smaller number first – thus, the above edge is given by the pair of indizes 1 2 (and not 2 1). As an auxiliary function, `sortedpair(const int a, const int b)` returns a pair of sorted indizes that can be used to represent an edge.
- ***Marking for refinement:*** Usually, edges belong to more than one triangle at the same time. In order to make sure that the new coordinates (the midpoints of the edges) are only added once during the refinement, it is sensible to store a list of all edges for which the midpoints have already been computed. This can be achieved by a structure

  ```
  std::map<std::pair<int,int>,int> edge2newpoint;
  ```

  that maps each edge (given by a `std::pair<int,int>`) to the index of its midpoint (i.e. the new coordinate).
- ***Refinement procedure:*** With the above structure, you can iterate over all elements and refine them individually, checking for each edge if the corresponding new coordinate exists already or has to be computed.
- You can make it easier for yourself if you arrange all triangles in the same way, i.e. listing the vertices clock- or counter-clockwise. The triangles of the given initial mesh are listed counter-clockwise.

Programming Exercise 2: Elliptic FEM 2D (24 points)

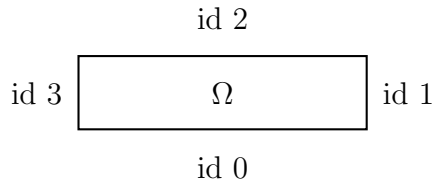
Solve the 2D Poisson problem

$$\begin{aligned} -\Delta u &= f \qquad && \text{on } \Omega = (0,1)^2, \\ u(x) &= g(x) \qquad && \text{on } \Gamma = \partial\Omega. \end{aligned}$$

using linear Lagrangian Finite Elements (the "hat basis"). We will use basically the same structure as for the 1D FEM, i.e., on the homepage, you find header files for the following classes:

- `EllipticFEM2D`: Contains again the finite element structures, i.e. the 2D mesh, the solution vector, and the reference to the PDE which we want to solve. Unlike in 1D, it now has a reference to a Dirichlet boundary object of type `DirichletBC`, see below.
- `PDE2D`: The (abstract) class for a PDE problem which we want to solve. The same interface as in 1D, only that the functions now take a two-dimensional x of type `Point2D` as argument.
- `Poisson2D`: As in 1D, one example of a concrete PDE problem. Again, it provides the assembly of the Laplace operator on a given finite element (in our 2D triangulation specified by the three vertex nodes $x_1^{(m)}$, $x_2^{(m)}$, $x_3^{(m)}$), now by a transformation onto the reference element.
- `DirichletBC`: An (abstract) class that serves as an interface for different implementations of boundary information. It specifies that all derived classes have to have the function `is_dirichlet_point(Point2D& x)`, that returns `true` or `false` depending whether `x` is on the Dirichlet boundary or not.

- `DirichletBC_Rectangle`: Our concrete Dirichlet boundary class. It assumes that the domain is a rectangle, thus having 4 boundary parts (the edges) labeled with boundary ids $0, \dots, 3$, and that each edge is either completely a Dirichlet boundary or not at all.



Moreover, you find a file `test_fem2d.cpp` that uses the above classes to set up a Poisson problem and solve this PDE.

Implement the missing function implementations and solve the problem for

a) $f(x) = 1$, $g(x) = 0$.

b) $f(x) = 10$, $g(x) = \|x\|_1 = |x_1| + |x_2|$.

Plot your solutions u.

Hints:

- ***Element-wise assembly:*** Again, when assembling, you should loop over all elements, call the function `assemble_on_element` on the PDE object and write the obtained information in the global stiffness matrix A_h and right-hand side b_h. Keep in mind that now between 0 and 3 of the three nodes in an element can be Dirichlet nodes (which you can check by calling `d_bc.is_dirichlet_point`). Recall that with the set of Dirichlet node indizes $\mathcal{D} := \{k : x_k \text{ is on Dirichlet boundary}\}$, we have

$$\begin{aligned} A_h &= \big(a(\psi_j, \psi_i)\big)_{i,j} && \text{if } i, j \notin \mathcal{D}, \\ b_h &= \Big((f, \psi_j)_0 - \sum_{k \in \mathcal{D}} a(\psi_k, \psi_j) g(x_k)\Big)_j && \text{if } j \notin \mathcal{D}, \end{aligned}$$

  i.e. that you have to look at the pairwise combinations of the three vertex nodes and that
  - you have a stiffness matrix contribution only if both nodes are *not* Dirichlet nodes,
  - you have a right hand side entry only for the nodes that are *not* on the boundary.
  - if one node is Dirichlet and the other not, you have to modify the corresponding right hand side entry by the boundary information.

- ***Assembly on element:*** Compute $A^{(m)}$ using the transformation onto the reference element as in the lecture. Note that the matrices $\hat{K}_1$, $\hat{K}_2$ and $\hat{K}_3$ are independent of any element information and can thus be initialized once in the constructor of `Poisson2D`. Both $|\det S|$ and the entries c_{11}, c_{12} and c_{22} of C, however, have to be computed for each element. Do this "by hand" using the respective formulas for the determinant and the inverse of a 2×2-matrix. (Do *not* compute a numerical matrix inversion!)

- ***Assembly of right-hand side:*** Using also the transformation onto the reference triangle and a trapezoidal numerical quadrature, the right-hand side contributions can be computed as

$$b^{(m)}_{x_i^{(m)}} \approx \frac{1}{6} \hat{f}(\hat{x}_i) |\det S| = \frac{1}{6} f(x_i^{(m)}) |\det S|, \qquad i = 1, \dots, 3.$$

- ***Plotting:*** The class `EllipticFEM2D` now has a function `write_solution_on_mesh` that outputs the solution in a format which gnuplot can understand. Use `splot ''u_2d.txt'' w lp`.