Interdisziplinäres Zentrum
für Wissenschaftliches Rechnen (IWR)
Ruprecht–Karls–Universität Heidelberg

Prof. Dr. Vincent Heuveline
Dr. Philipp Gerstner

Programming Sheet 1

06.11.2020 – 27.11.2020

Finite Elements (Winter Term 2020/2021)

Important Notes:
The programming exercises have to be solved within the provided C++ framework. Please add detailled comments to your code. We only evaluate codes, which compile without any bugs and do not exhibit run time errors. *We do not debug your code!* You can compile your code by opening a terminal within the Code directory and typing `make`.
After you have put your solution into the provided framework, create an zip or tar archive and upload it to moodle. In addition to your code, this archive should contain a PDF file, in which the terminal output (and plots) created by your program are stated.
We test your solution on Ubuntu 20.04 or Linux Mint XFCE. If you want to ensure that your solution compiles successfully on our computers, you should use one of these OS (possibly installing it on a virtual machine, e.g. VirtualBox). Further, you should install the package `build-essential` (`sudo apt install build-essential`) for compiling C++ codes on Debian systems (includes Ubuntu and Linux Mint).

Exercise 1 (Programming). Grid refinement (15 Points)
In this exercise, we consider the class `GRID`, which represents an unstructured, two-dimensional, triangular grid. The class contains `std::vector<Coord> coords_` for storing the grid points and `std::vector<Triangle> conn_` for storing the connections between that points, that form a triangle. The index of a point or triangle in `GRID` corresponds to its position in the respective `std::vector`. Moreover, `GRID` provides routines for adding and accessing points and triangles. You can obtain a terminal output of all grid data by the member function `print()`.

Each vertex in the triangular grid is represented by the data structure `Coord`; this structure contains a `double`-array of length 2 for the vertex coordinates (x, y). You may access these coordinates by means of the operator `[]`.

These vertices are connected to form triangles by means of the data structure `Triangle`. This structure contains a `int`-array of length 3 whose entries correspond to the `GRID`-indices of its vertices. For instance, the triangle with entries `10 3 25` is formed by vertices 10, 3 and 25, where the numbering is done counter-clockwise. Again, the operator `[]` provides access to the indices of the vertices.

One of the most important operations concerning grids is refinement. Based on a coarse grid, a new, refined grid is created (in order to increase the approximation quality of the underlying finite element space). To this end, each triangle is decomposed into 4 smaller triangles, according to Figure 1.

Applied to an entire grid, one thus obtains a refined grid, see Figure 2.

a. Look at the provided code framework and make yourself familiar with the data structures `GRID`, `Coord`

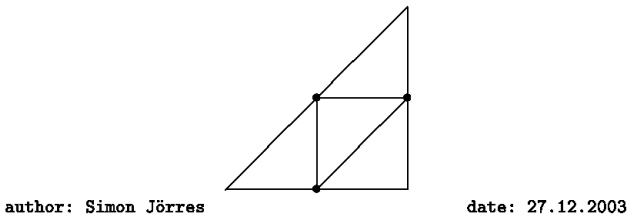

Abbildung 1: Refinement of a triangle into 4 smaller triangles.

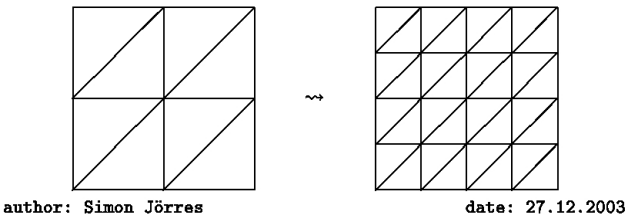

Abbildung 2: Refinement of an entire grid.

and `Triangle` and the already implemented routines. (You do not have to write anything for this subtask)

b. The `main`-function in the file `test.cpp` executes the main steps of the programm. These steps are

- reading the grid
- refining the grid `grids-1` times
- writing the output files in pvd/vtu-format. These files are stored in the directory `data` and can be visualised by ParaView (www.paraview.org, `sudo apt install paraview`).

Briefly state where these steps are implemented in the code.

c. Implement the previously described strategy for grid refinement, given by the function

```
void GRID::refine_ip(
   const FE_VEC in[],
   int num_vec,
   GRID &newgrid,
   FE_VEC out[]
)
```

in the file `grid.cc`. This function should refine the (coarse) `GRID`, from which it is called, once and store the resulting (refined) grid in `newgrid`. In doing so, the existing (coarse) `GRID` must not be modified (i.e. no vertices and triangles should be added to the coarse grid). You should keep track of those triangular edges, that have already been refined and of the corresponding, created vertices. Use the auxiliary data structure `std::tr1::unordered_map<int, int> refinement_info_` in class `GRID` for

this task. Thereby, the idea is to compute a unique number/ ID for each edge in the coarse grid and store this ID together with the index of the created vertex as value pair in `unordered_map`.

You can ignore the function arguments `const FE_VEC in[]`, `int num_vec` and `FE_VEC out[]`, they will be needed for upcoming exercises.

If you correctly implemented the refinement algorithm, then your program, applied to the initial grid shown by Figure 3, creates the following values:

Refinement level	1	2	3	4	5	6	7
Number of vertices	9	25	81	289	1089	4225	16641
Number of triangles	8	32	128	512	2048	8192	32768

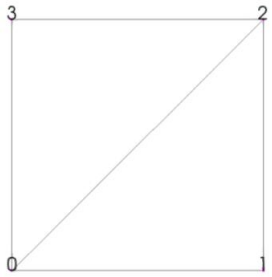

Abbildung 3: Initial grid which is read in.