# Remarks to programming exercise 5

Implement the Finite Element method for the two-dimensional Poisson problem

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

Follow the instructions given in the lecture and programming exercise 3 to obtain a Finite Element discretization of above problem. Use linear hat functions as a basis for $V_h$, given on the reference element with vertices $\hat{\mathbf{v}}_0 = (0,0)^T$, $\hat{\mathbf{v}}_1 = (1,0)^T$, $\hat{\mathbf{v}}_2 = (0,1)^T$ as

$$\varphi_0(\hat{\mathbf{x}}) = 1 - \hat{x} - \hat{y}, \qquad \varphi_1(\hat{\mathbf{x}}) = \hat{x}, \qquad \varphi_2(\hat{\mathbf{x}}) = \hat{y}.$$

In this ZIP-archive you will find some files that give you the overall structure of a Finite Element solver. You only have to implement a number of core functionalities in the file `implement_me.c`. You can add files, functions and data structures for your implementation but do not add anything to the other files besides `implement_me.c`.

A typical Finite Element solver has the following structure (see also lecture 7):

1. *Preprocessing*:
   - generate or load a triangulation of the domain;
   - allocate data structures for the linear system.
2. *Assembly*: for each element
   - compute local stiffness matrix,
   - compute local load vector,
   - assemble local matrix and vector into the global data structures.
3. *Apply boundary conditions*: modify the linear system to incorporate the boundary conditions.
4. *Solve* the linear system.

This algorithm is implemented in the function `fem()` in `fem.{c,h}`.

Your solver should be able to handle unstructured triangular meshes. A triangular mesh can be described by two simple data structures: the coordinates of the nodes and the list of nodes belonging to each triangle. These are combined in the data structure `mesh` in `mesh.h`. Its member array `coords` holds all node coordinates $\mathbf{x}_i = (x_i^{(1)}, x_i^{(2)})^T$, $i = 0, 1, ..., N_v - 1$ in an interleaved format:

$$\texttt{coords} = [x_0^{(1)}, x_0^{(2)}, x_1^{(1)}, x_1^{(2)}, \ldots]\,,$$

and the member array t2v lists for each triangle all three node indices *in counter-clockwise order, beginning with the lowest index*, e. g.

$$\texttt{coords} = [\underbrace{0,4,3}_{T_0},\underbrace{0,1,4}_{T_1},\ldots]\,.$$

The third array id_v marks each vertex to be either an interior node (value 0), or to be on one of the four boundaries (1 - south, 2 - east, 3 - north, 4 - west).

Start by implementing the mesh generation in the function get_mesh(). Use the grid from programming exercise 4 and split each of the quadrilaterals into an upper left and lower right triangle. Again, use a lexicographical ordering for all nodes. The $n \times n$ quadrilaterals $Q_{ij} = (x_{i,j}, x_{i+1,j}) \times (x_{i,j}, x_{i,j+1})$ $(i, j = 0, \ldots, n-1)$, are given by

$$x_{i+1,j} - x_{i,j} = \begin{cases} \frac{1}{n} + \frac{1}{n^2} & \text{if } i \text{ is even}\,, \\ \frac{1}{n} - \frac{1}{n^2} & \text{if } i \text{ is odd}\,, \end{cases}$$

and

$$x_{i,j+1} - x_{i,j} = \begin{cases} \frac{1}{n} + \frac{1}{n^2} & \text{if } j \text{ is even}\,, \\ \frac{1}{n} - \frac{1}{n^2} & \text{if } j \text{ is odd}\,. \end{cases}$$

The next step is the initialization of the CRS-matrix. The relevant data structures are combined in the struct crs_matrix in crs_matrix.h (for a short introduction to the CRS format see programming exercise 2). The connectivity information of the mesh is sufficient to determine the structure of the system matrix. Implement the function init_mesh(), which allocates the necessary data structures, initializes val to zero and fills rowPtr and colInd with the correct values (Hint: an ordered linked list might help here to first find the connected vertices for each node; use this information to initialize t2v).

After that the system matrix can be assembled. For each element a local stiffness matrix and load vector are computed (get_local_stiffness() and get_local_load(), respectively) and their entries are then put in the right places in the global system (assemble_local2global_{stiffness,load}()). Implement these functions.

Use an affine transformation from a reference triangle to the pysical triangle for the computation of the local stiffness matrices, as explained in lecture 7. Choose a quadrature rule to integrate the right-hand-side terms for the load vector.

The last step towards the final linear system is the application of the boundary conditions. Implement the function apply_dbc() which modifies the matrix and right hand side vector according to the Dirichlet boundary conditions.

To obtain the solution, implement a solver of your choice in the method solve().

In exercise5.c a number of test cases is implemented, which allow to test certain aspects of your implementation:

1. $u(x, y) = x + y$ (zero right hand side),

2. $u(x, y) = sin(\pi x)sin(\pi y)$ (zero boundaries),
3. $u(x, y) = cos(7x)cos(7y)$ (non-zero boundaries and right hand side).

The ZIP-archive also includes a Makefile to compile the code (simply write `make`). All generated files can be deleted again with `make clean`. A debugging version can be generated with `make debug`, which includes symbols in the binary and activates a number of debugging outputs (local and global matrices, mesh, etc.).

Some tips regarding the implementation:

- Start with small mesh sizes ($n = 1, 2$) and a simple case (e. g., equidistant nodes).
- Test each step of your implementation. Use the debugging output to verify your implementation.
- Your implementation should include correct error handling (`NULL`-pointers as function arguments, `malloc`-failure, etc.).
- Correct memory handling! All allocated resources must be free'd, no pointers to local objects may be returned, etc. Check your code with `valgrind`.
- The program prints the error for each mesh size. Verify the convergence rate.

Hand in your `implement_me.c` (and any additional files, if you needed them) via StudOn.