D-MATH HS 2015
Prof. Ch. Schwab

Numerical Methods for Elliptic and parabolic PDEs

Introduction to LehrFEM functions and data structures

Fabian Müller, 2015

Contents

1 Introduction **1**

2 Function handles in MATLAB **2**

3 Structs **4**

4 Meshes in LehrFEM **5**
4.1 The Mesh-struct 5
4.2 Flags for Elements and Edges 6
4.3 Mesh plotting 6
4.4 Mesh refinement 7
4.5 Constructing a mesh using signed distance functions 7

5 Quadrature rules **9**

6 Assembly of matrices **9**
6.1 The assembly-master file 9
6.2 The sparse command 11
6.3 Implemented local contributions 11

7 Assembly of the load vector **12**

8 Incorporation of boundary conditions **13**
8.1 Neumann conditions 13
8.2 Dirichlet conditions 15

9 Computation of errors **17**
9.1 In the L^2-norm 17
9.2 In the H^1-norm 18

10 Plots of solutions **19**

11 A final example on all sections **19**

1 Introduction

This tutorial will cover an introduction to LehrFEM functions and the data structures we use in all our codes during the lecture. Note that although differrent data structures are possible, these have proven to be easily understandable, comfortable to modify for small changes and efficient. The LehrFEM library has been created especially for NumPDE courses by different members of the Seminar for applied mathematics. Each function contains a comment about the author.
However, as the library is large and useful for purposes exceeding this introductory lecture, the official manual (see out website) is slightly confusing. After this tutorial, you should be able to

- define function handles in matlab and write functions that take such handles as input,
- use structs and understand the most important fields of the LehrFEM `Mesh` structs,
- create a triangular mesh on a bounded two-dimensional domain D,
- assemble the Galerkin matrix of operators of the kind $u \mapsto -\nabla \cdot (\kappa(x)\nabla u(x)) + \alpha(x)u(x)$ on a two-dimensional bounded domain $D \subseteq \mathbb{R}^2$ with piecewise linear, nodal, Lagrangian Finite Elements (from now on: "linear FEM"),
- assemble the load vector of the functional $v \mapsto \int_D f(x)v(x)\,dx$ for a given function f,
- incorporate Neumann and Dirichlet boundary conditions, or combinations of them according to LehrFEM,
- plot solutions to elliptic PDEs obtained by linear FEM.

This tutorial perhaps contains some matlab commands you have not known before. In order to find more about a command `example`, type `help example` or `doc example` in the matlab shell. This will output the internal help file.
All problems/meshes are thought to be in 2D, if not indicated otherwise. Hence, let $D \subseteq \mathbb{R}^2$ be a bounded domain with certain regularity of ∂D (must be as regular as required to be able to triangulate after all).

2 Function handles in MATLAB

Let $U \subseteq \mathbb{R}^d$ and we are given a function $g : U \to \mathbb{R}$. As an example, we take $d = 2$ and $U := \{\|x\| < R\}$, the disc of a fixed radius $R > 0$ in $\mathbb{R}^2$, and the functions $g_0(x) := x_1^2$, $g_1(x) := \sin(\arg(x))$, $g_2(x) := \begin{cases} x_1^2 & \text{if } \|x\| < \frac{R}{2} \\ 100 & \text{otherwise.} \end{cases}$
In order to implement their pointwise evaluation, we have three possibilities:

(i) To write three seperate `.m`-files `g1.m`, `g2.m`, `g3.m` and to store them in a folder which belongs to the matlab path.

(ii) To use the `inline` command:

```
1 g0 = inline('x(1)^2', 'x');
2 g1 = inline('sin(angle(x(1) + 1i*x(2)))', 'x');
3 g2 = inline('(norm(x)<R/2)*x(1)^2 + 100*(norm(x)>=R/2)');
```

where `(norm(x)<R/2)` is a *Boolean*, i.e. it returns 1 if the condition $\|x\| < \frac{R}{2}$ is satisfied and 0 otherwise. The syntax rule is as follows: You write the "function evaluation formula" in the first argument, and the input variables you need as the second variable. As an example, we could also write `g1 =inline('x^2', 'x', 'y')`, taking the coordinates as separable inputs.

(iii) To use a *function handle*. This is the most convenient form, and faster than the `inline` command. The rule is as follows: We write `g=@(LISTOFINPUTS SEPARATEDBYCOMMA) OUTPUT`. We must not put the single quotation marks here. Hence in our example,

```
1 g0 = @(x) x(1)^2; %or g0 = @(x,y) x^2;
2 g1 = @(x) sin(angle(x(1) + 1i*x(2)));
3 g2 = @(x)(norm(x)<R/2)*x(1)^2 + 100*(norm(x)>=R/2);
```

If g is a matlab function in the `inline` or handle-format, it can be passed to another function `fun.m` as an input. See the following example:

```
1 function f=TakeSqrtOfFunctions(x,g)
2   f=sqrt(g(x));
3 end
```

Now, if you want to evaluate f for g_i, $i = 0, 1, 2$, you just need to call `TakeSqrtOfFunctions.m` with three different inputs. This is very useful for us, since we do not want to write an extra load-vector assembly for every little change in the source function f!
Note that if you want to pass a function `subfunction()` which is defined in a separate `.m`-file `subfunction.m` as input to another function `function.m`, then you need to "tell" this to matlab via putting an `@` in front of the calling: `function(INPUT1, @subfunction, INPUT3,...)`.

3 Structs

In LehrFEM, there are several functions for solving PDEs on a given mesh or refining a given mesh. These important files save all mesh data in a so-called *struct*.
Basically, a struct is a collection of variables. A variable in MATLAB can be a string, a scalar, a vector, a matrix, or multi-dimensional array, as you always used them so far. But if you write large codes and are dealing with a lot of variables, it may be convenient to sort some of them in groups. As an example, set some variables:

```
var_1 = 2;
diff_var_1 = [1 2 3 4 5];
var_2 = [1 5 0 ;
         0 0 1 ;
         1 9 8];
diff_var_2 = {'This', 'is', 'a', 'bunch', 'of', 'strings'};
```

Imagine, `var_1` and `var_2` belong together and the other two as well. Then you can bundle them in *structs* that contain these variables in *fields*. Look at the following code:

```
struct_1.field1 = var_1;
struct_1.field2 = var_2;
struct_2.field1 = diff_var_1;
struct_2.field2 = diff_var_2;
```

this bundles the `var_`-variables in struct 1 and the `diff_var_`-variables in struct 2. If you type `struct_1` into MATLAB, you get:

```
>> struct_1
struct_1 =
    field1: 2
    field2: [3x3 double]
```

i.e. `struct_1` contains of two fields, one called `field1` that is a scalar 2, and one called `field2` that is a 3-times-4 matrix.
If you want to access the variables that you stored in the structs, then you simply write "STRUCT.FIELDNAME", and you can operate with that as with "usual" variables:

```
>> det(struct_1.field2)
ans =
    -4
>> struct_2.field2(4)
ans =
    'bunch'
```

If you need further information, type `doc struct` in MATLAB, or see e.g.
`http://www.cs.utah.edu/~germain/PPS/Topics/Matlab/structures.html`.

4 Meshes in LehrFEM

4.1 The Mesh-struct

As an example, we consider the mesh depicted in Figure 1 A triangular mesh on D is uniquely determined by

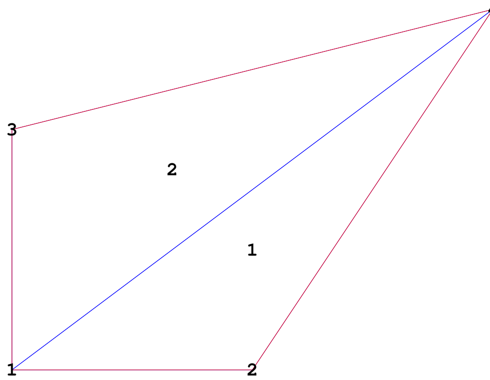

Figure 1: The example Mesh. Numbers inside elements give the element number, number at vertices the vertex number.

- A list of coordinates of the vertices:

```
Coords=[0   0;
        1   0;
        0   1;
        2 1.5];
```

- A list of elements which contains in the k-th row the vertices of the k-th element *in positive orientation*[1]:

```
Elements=[1 2 4;
          4 3 1];
```

This theoretically determines our triangulation uniquely. However, it is sometimes useful to save more information. LehrFEM contains everything which can be said about a mesh in one *struct* `Mesh`. Since this struct is used in all LehrFEM functions, it is important to make some conventions. The table below contains an explanation of the most important fields of `Mesh`, where M is the number of vertices of the triangulation, N the number of elements, P the number of edges. You will find more fields in the LehrFEM manual.

[1]This is a very frequent bug.

`Coordinates`	A M-by-2 matrix that contains in the k-th row the Coordinates of the k-th Point.
`Elements`	A N-by-3 matrix that contains in the k-th row the indices of the local vertices of the k-th element, in consistent orientation.
`Edges`	A P-by-2 matrix containing in the k-th row the starting point of the k-th edge in the first entry, and the ending point of the edge in the second entry. (This field can be added to the struct by the file `add_Edges.m`)
`Vert2Edge`	M-by-M sparse matrix that contains in the (i, j)-th entry the number of the edge that connects point i with point j. (This field can be added to the struct by the file `add_Edges.m`)
`BdFlags`	A P - length vector that contains in the k-th entry 0 iff the k-th edge is not at the boundary and a negative entry if it is at the boundary. In this field, you have to distinguish between Dirichlet and Neumann (or other types of) boundary conditions: E.g. you set the entries corresponding to all Dirichlet nodes to -1, the entries corresponding to Neumann boundary conditions to -2. (In order to add this field, you must write a code by yourself, since this generally depends on your problem.)

Although the triangulation is theoretically determined by the fields `Coordinates` and `Elements`, we often want more information, e.g. about the edges or the boundary. This is important if we want to incorporate boundary conditions on the boundary edges. In order to add additional information to your struct, you can make use of LehrFEM functions.
What you will mostly use is the following case: If the field `Edges` is not yet contained in your struct, the function `add_Edges.m` will add the fields `Edges` and `Vert2Edge`.
There are also functions which extract information without adding it to the struct, e.g. the function `get_BdEdges.m`. Without the field `BdFlags`, it is impossible to you to directly determine whether an edge is a boundary edge or not. This function does the job for you: `loc=get_BdEdges.m` returns a vector `loc` which contains the indices of all boundary *edges*. In order to find the said edges, type `Mesh.Edges(loc,:)`.
Note that we could have a convention that these matrices are saved as the transpose of what stands here. In the case of the element list, this would actually make more sense, since matlab saves arrays in a column major. However, by convention, we do save it that way, hence `Mesh.Coordinates` must be a M-by-2 array, `Mesh.Elements` a N-by-3 or N-by-4 array (depending on if you work with triangular or quadrilateral FEM), etc.

4.2 Flags for Elements and Edges

LehrFEM allows you to mark elements and edges with a flag. This can be useful in some cases. Imagine, you have a source function $f(x)$ that is $= 1$ on some small part of the domain and $= 0$ everywhere else. Then, it would make sense to mark those elements on which f is $= 1$ with some integer, say, 1, and the other elements with 0. This is done by the additional field `Mesh.ElemFlag`, which must be of size N_{Elements}-by-1, i.e. a column vector.
Another, more frequent problem encountered is a mixed Neumann/Dirichlet problem. Suppose $\partial_D = \bar{\Gamma}_1 \sup \bar{\Gamma}_2$, where $\Gamma_1 \cap \Gamma_2 = \emptyset$. On Γ_1, you impose Dirichlet coniditions while on Γ_2, you impose Neumann conditions. Then it would make sense to mark the Dirichlet Edges by some flag, the Neumann edges by some other flag, and the interiour edges by flag 0.This is done in the additional field `Mesh.BdFlags`, which is another colum vector of size P, i.e. number of edges. This will needed in the implementation of boundary conditions!
The convention is the following: An interiour edge has flag 0, a boundary edge a *negative* flag. In this case, the Neumann edges could have flag -1 and the Dirichlet edges flag -2. See the chapter about implementation of boundary conditions.

4.3 Mesh plotting

If you want to see whether you have implemented your mesh well, you can plot it with the routine `plot_Mesh.m`. The syntax is quite easy. Just type `plot_Mesh(Mesh)` and you will get a plot. Like this, the routine opens a new figure where the Mesh is plotted. If you want to include the plot into another figuer (e.g. by `subplot`, use

the option `'f'`, see below.
Several options are available: To call the routine with options, type `plot_Mesh(Mesh, 'options')`, where `options` is a combination of the following letters:

`t`	Add element numbers to the plot
`p`	Add vertex numbers to the plot
`e`	Add edge numbers to the plot
`a`	Display axes in the plot
`f`	Do not open a new window for the plot
`s`	Add title and axis labels to the plot

In combination, do not use new quotation marks for every option. As an example, `plot_Mesh(Mesh, 'tpf')` plots the Mesh, displays the vertex and element numbers and does not open a new figure.

4.4 Mesh refinement

Suppose that you are given a mesh, but you would like to refine it. There are several ways to refine a mesh and in general, this is a very difficult question! In this lecture, adaptive FEM and local mesh refinements will not be treated. Therefore, it is sufficient to have a routine which refines a mesh by creating for "children" for each element, see Figure 2. This is done by the routine `refine_REG.m`. There are no options to this file, just type `refine_REG(Mesh)` to refine your mesh.

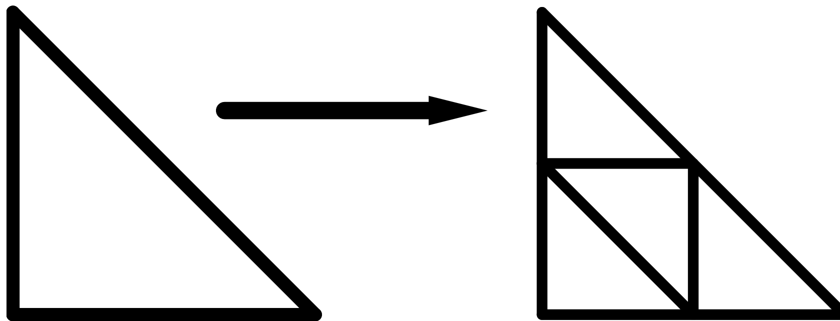

Figure 2: Regular mesh refinement. Note: This refinement rule does not change the shape regularity constant!

4.5 Constructing a mesh using signed distance functions

If you are given D and you would like to construct a mesh on D, there are several possibilities, as given in the lecture. The method implemented in LehrFEM is the `distmesh` package developped at the MIT.
In order to give the geometry, we need a *signed distance function* to the boundary:

$$\mathrm{sdiff}(x) := \begin{cases} \mathrm{dist}(x, \partial D) & \text{if } x \in D, \\ -\mathrm{dist}(x, \partial D) & \text{if } x \in \mathbb{R}^2 \setminus D \; . \end{cases}$$

As an example, the signed distance function of a circle of radius R with midpoint x_0 is given by $\mathrm{sdiff}(x) = R - \|x - x_0\|$. It turns out to be very useful to know sdiff if, given D_1 and D_2, $D = D_1 \cup D_2$, or $D = D_1 \cap D_2$, or $D = D_1 \setminus D_2$ (certainly, only if the resulting D is a domain, i.e. still open, bounded, connected, nonempty). Let sdiff_i, $i = 1, 2$ be the signed distance functions of the domains D_i. In these cases, the signed distance functions for D can be expressed in terms of sdiff_i as:

$$\begin{aligned} D = D_1 \cup D_2: &\quad \mathrm{sdiff}_D(x) = \min(\mathrm{sdiff}_1(x), \mathrm{sdiff}_2(x)) \\ D = D_1 \cap D_2: &\quad \mathrm{sdiff}_D(x) = \max(\mathrm{sdiff}_1(x), \mathrm{sdiff}_2(x)) \\ D_2 \subset D_1 \text{ and } D = D_1 \setminus D_2: &\quad \mathrm{sdiff}_D(x) = \max(\mathrm{sdiff}_1(x), -\mathrm{sdiff}_2(x)) \; . \end{aligned}$$

Let us turn to the mesh generation. The package `distmesh` is implemented in LehrFEM in the file `init_Mesh.m`, following the syntax

```
Mesh=init_Mesh(BBox, h0, DHandle, HHandle, FixedPos, Disp, varargin)
```

We explain step by step.

- `BBox`, or the bounding box, defines a rhomboid where your mesh must lie in. It contains limits of x-axis and y-axis in the first and second column. For example, if you want your mesh to lie in a cube $[a, b] \times [c, d]$, you will define `BBox=[a d; b d];`.
- `h0` is the desired initial meshwidth.
- `DHandle` is the function handle of the signed distance function we have just explained.
- `HHandle` is a function which takes x as input and returns a scaling of the meshsize near x. If you want to generate a uniform mesh, write `HHandle=@(x) ones(size(x, 1), 1)`, e.g. it returns a vector of 1, with the same length as x.
- `FixedPos` defines a set of points you wish to have always as nodes. If your D is polygonal (e.g. a square), it makes sense to put the coordinates of the vertices into this list. Otherwise, you can set it as an empty array by `FixedPos=[];`.
- `Disp` is a Boolean (i.e. takes values 1 or 0) that determines whether the whole mesh generating process is plotted or not. If you just wish to work with the mesh, and are not interested in the generating process, set it to 0.
- The last argument `varargin` is a dummy argument for options to the functions `DHandle` and `HHandle` which will be redirected to these functions. Hence, if you write a signed distance function for a circle which takes the midpoint and the radius as an argument, i.e. `d=distCircle(x, Centre, Radius)`, you will have to write `Centre, Radius` in the place of `varargin`. Since all the inputs given in `varargin` are directed to both `DHandle` and `HHandle`, you will have to implement `HHandle` in a way such that it takes `Centre, Radius` as arguments, too. Certainly, it may be independent of these inputs!

5 Quadrature rules

In LehrFEM, quadrature rules are saved as structs, say, `QuadRule`, with fields

`x` Contains the coordinates of the l-th quadrature node in the l-th row,
`w` A vector, containing the l-th weight in the l-th entry.

Here, we have to distinguish between 1D and 2D quadrature rules. When integrating over a boundary, 1D quadratures are needed. In LehrFEM, the Gauss-Legendre rules and the Gauss-Lobatto rules have been implemented in the functions `gauleg.m` and `gaulob.m`. The syntax for both of is `QuadRule=gauleg(a,b, NGauss)`, where a, b are the bounds of the interval the nodes lie in, and `NGauss` is the number of nodes/weights. The same holds for `gaulob.m`
In 2D, several quadratures have been implemented. The nodes always lie in the reference triangle $\hat{K}$ bounded by the vertices $(0,0),(0,1)$ and $(1,0)$. You will need to integrate over different triangles than $\hat{K}$, but in this case, you can just map the nodes `QuadRule.x` to your triangle of interest using the affine element map. The 2D triangular quadrature rules implemented in LehrFEM are called `PXOY.m`, where `X` is the number of nodes and `Y` is the order of the respective quadrature rule. So far, we have only used `QuadRule=P7O6()` which is enough for linear FEM (see the chapter about variational crimes).
For bilinear, (quadrilateral) FEM, you will need to have a quadrature with nodes in the reference rectangle, rather than the reference triangle. This is way easier than the triangle-case, since the quadrature nodes in a rectangle can be constructed upon products of 1D-quadrature nodes: Let the 1D-Quadrature nodes $\{(\xi_x^i, \omega_x^i), i = 1, \dots, N_{\text{Quad},x}, \xi_x^i \in [a,b]\}$ and $\{(\xi_y^j, \omega_y^j), j = 1, \dots, N_{\text{Quad},y}, \xi_y^j \in [c,d]\}$ be given, then the products $\{((\xi_x^i, \xi_y^j), \omega_i \omega_j), i = 1, \dots, N_{\text{Quad},x}, j = 1, \dots, N_{\text{Quad},y}\}$ defines the product quadrature on the rectangle $[a,b] \times [c,d]$.

6 Assembly of matrices

6.1 The assembly-master file

The assembly is the most important part of FEM implementation. You have seen the algorithm in a fairly concrete case of piecewise polynomial (linear), nodal FEM, but the principle of splitting integrals over D into a sum of integrals over subdomains can be formulated more generally.
There are different kinds of Galerkin methods, but the principle of assembly stays the same. In principle, we only need three main ingredients:

- a partition into subdomains (the Mesh): This allows us to run a for-loop over the subdomains (the elements).
- a rule of "which integrals to compute at the local nodes" in each subdomain: The element contribution. So far, we have seen the element mass and stiffness matrices in functions `STIMA_Lapl_LFE.m` and `MASS_LFE.m`.
- a rule of "where to insert these computed values into the global matrix": In this lecture, this will always be contained in the `Mesh.Elements` field.

Hence, we can boil the FEM assembly down to the following pseudo-algorithm:

```
function A = generalAssembly_Pseudo(Mesh, ElementContribution, localToGlobal)
 A = zeros(Dim_V_h, Dim_V_h);        % Preallocation of A
 for el=1:N_Elements
       LocalIndices = [1; 2; 3];    % or sth. different for another kind of FEM
       GlobalIndices =localToGlobal(LocalIndices);
       LocalVertices = Mesh.Coordinates(GlobalIndices, :);
       LocalMatrix = ElementContribution(LocalVertices);
       A(GlobalIndices, GlobalIndices) =  A(GlobalIndices, GlobalIndices) + LocalMatrix;
 end
end
```

This general kind of assembly-call function for linear FEM is implemented in LehrFEM under the name assemMat_LFE.m. Look at the following listing which is a copy of the LehrFEM function assemMat_LFE.m:

```
function varargout = assemMat_LFE(Mesh,EHandle,varargin)
% Initialize constants
nElements = size(Mesh.Elements,1);

% Preallocate memory
I = zeros(9*nElements,1);
J = zeros(9*nElements,1);
A = zeros(9*nElements,1);

% Check for element flags
if(isfield(Mesh,'ElemFlag')),
    flags = Mesh.ElemFlag;
else
    flags = zeros(nElements,1);
end

% Assemble element contributions
loc = 1:9;
for i = 1:nElements
    % Extract vertices of current element
    idx = Mesh.Elements(i,:);
    Vertices = Mesh.Coordinates(idx,:);

    % Compute element contributions
    Aloc = EHandle(Vertices,flags(i),varargin{:});

    % Add contributions to stiffness matrix
    I(loc) = set_Rows(idx,3);
    J(loc) = set_Cols(idx,3);
    A(loc) = Aloc(:);
    loc = loc+9;
end

% Assign output arguments
if(nargout > 1)
    varargout{1} = I;
    varargout{2} = J;
    varargout{3} = A;
else
    varargout{1} = sparse(I,J,A);
end

return
```

This file does exactly what the pseudocode from above does. The local-to-global map is already given in line 21 since this file only handles the piecewise linear case.
Let us look at the inputs: The first input Mesh is clear. The element contribution EHandle, e.g. @STIMA_Lapl_LFE or @MASS_LFE is handed to the code as second input. The third varargin dummy input is passed inside the function as optional input to the element contribution EHandle: We have seen an example, where we wanted to assemble $M_{ij} := \int_D \alpha(x) b_i(x) b_j(x)\, dx$ for a given α. Hence, we write our element contribution function in a way such that it takes a function handle alpha as an input.
As an example, if we want to compute M as defined before, we need to define a function Compute_M(Vertices, alpha)

which returns the local contribution of the element determined by `Vertices` and using the function handle `alpha`. The call of the global assembly would be `M =assemMat_LFE(Mesh, @Compute_M, alpha)`, where `Compute_M.m` is a separate `.m`-file, but `alpha` is a function handle.

6.2 The sparse command

Let us consider the output. We see another dummy output `varargout`, and in lines 35-41, the output is assigned with a distinction by the size of the output. This means, that if you call `M =assemMat_LFE(...)`, the code jumps to line 40 and returns a full matrix. However, if you write `[I, J, M]=assemMat_LFE(...)`, it returns `I`, `J`, and `M`. Now, what do `I` and `J` mean, and what goes on in lines 28-31?
This has to be explained with `sparse` matrices, i.e. matrices whose zero entries are not saved. Matlab can handle them quite effectively. Have a look at the pseudocode listing in the beginning of this section. The update-step of the global matrix is done via `A(Ind, Ind) = A(Ind, Ind) + LocalMatrix`. This will slower down your code, since that way, matlab needs to copy `A` to a dummy variable, update the desired nine entries, and put them back where they belong. An easier way to set up sparse matrices is the `sparse` command. This command has several faces. For example, if you write `A=sparse(n,m)`, this will generate a n-by-m zero matrix, but saved in the sparse format. Every manipulation you do with `A` will conserve the format, unless you write `A=full(A)`, which is the proper command to convert from sparse to the full format.
The interesting part for us is the syntax `A = sparse(I,J,A_Entries)`. In this case, `I` and `J` are vectors of same lengthes, containing row- and column indices, and `A_Entries` contains the corresponding entries. Matlab then generates a sparse matrix `A`, such that `A(I(k), J(k)) = A_Entries(k)`, for $k = 1, \dots, \#I$. In our case, `A_Entries` is called `A`, too, and `I` and `J` are set up by the `.m` files `set_Rows.m` and `set_Cols.m`.

6.3 Implemented local contributions

In LehrFEM, several local contributions have been implemented. Some of you might have solved the programming exercises and thus have implemented the contributions for the stiffness and mass matrices by themselves. The most frequent files you will need are

`STIMA_Lapl_LFE.m`	Takes `Vertices` as input and returns the local stiffness matrix for $-\Delta$, discretized by linear FEM (LFE).
`MASS_LFE.m`	Takes `Vertices` as input and returns the local mass matrix by LFE.
`STIMA_Lapl_QFE.m`	Same as `STIMA_Lapl_LFE`, but with quadratic Finite Elements (QFE)
`MASS_QFE.m`	Same as `MASS_LFE`, but with QFE.
`STIMA_Lapl_BFE.m`	Same as `STIMA_Lapl_LFE`, but with bilinear FEM on quadrilateral meshes (BFE).
`MASS_BFE.m`	Same as `MASS_LFE`, but with BFE.

In case you forgot, let $\{b_j(x)\}$ be the basis functions of the Galerkin subspace V_N. The stiffness matrix is the matrix defined by $A_{ij} := \int_D \nabla b_j \cdot \nabla b_i$ and the mass matrix is the matrix defined by $M_{ij} := \int_D b_i\, b_j$.
If you want to compute a matrix with entries $\int_D \alpha\, b_i\, b_j$ or $\int_D \kappa \nabla b_i \cdot \nabla b_j$, you have to manipulate the LehrFEM code. See the corresponding exercise.

7 Assembly of the load vector

Given a functional $v \mapsto l(v)$ and a set of basis functions as before, the load vector is the vector with entries $l_i := l(b_i)$.In our case, $l(v) := (f, v)_{L^2(D)}$ for a given function f, hence we have to integrate fv over the whole domain. This is done by assembly again: We go through each element K, approximate $\int_K fv$ by a quadrature rule and transform back. This quadrature rule could be defined on a triangle or a rectangle, as given by the FE-Method we are using.
Let us discuss how to pass f as an input to an assembly function. In the section about the struct `Mesh`, we have seen that elements can be flagged. If your f has been implemented in a way such that it depends in the flag, this has to be taken into account by the assembly function. Therefore, the syntax of f *must be* `f(x, flag, ADDITIONAL)`, where some additional options (e.g. time dependence) can be passed in the argument `ADDITIONAL`. This input is not compulsory, but `flag` is. Therefore, before calling the vector assembly, make sure that your `f` takes `flag` as an input, and that `Mesh.ElemFlag` is defined. If not needed, set `Mesh.ElemFlag=zeros(size(Mesh.Elements,1), 1)`.
Here is the listing of the assembly function `assemLoad_LFE.m` As before in the assembly of matrices, several assemblies have been implemented to LehrFEM, the most important ones for this course being `assemLoad_QFE.m` or `assemLoad_BFE.m`.

```
function L = assemLoad_LFE(Mesh,QuadRule,FHandle,varargin)

% Initialize constants
nPts = size(QuadRule.w,1);
nCoordinates = size(Mesh.Coordinates,1);
nElements = size(Mesh.Elements,1);

% Preallocate memory
L = zeros(nCoordinates,1);

% Precompute shape functions
N = shap_LFE(QuadRule.x);

% Assemble element contributions
for i = 1:nElements
    % Extract vertices
    vidx = Mesh.Elements(i,:);

    % Compute element mapping
    bK = Mesh.Coordinates(vidx(1),:);
    BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
    det_BK = abs(det(BK));

    %Transform quadrature nodes to local element
    x = QuadRule.x*BK + ones(nPts,1)*bK;

    % Compute load data
    FVal = FHandle(x,Mesh.ElemFlag(i),varargin{:});

    % Add contributions to global load vector
    L(vidx(1)) = L(vidx(1)) + sum(QuadRule.w.*FVal.*N(:,1))*det_BK;
    L(vidx(2)) = L(vidx(2)) + sum(QuadRule.w.*FVal.*N(:,2))*det_BK;
    L(vidx(3)) = L(vidx(3)) + sum(QuadRule.w.*FVal.*N(:,3))*det_BK;

end

return
```

Again, potentially existing optinal inputs for `FHandle` are passed to `assemLoad_LFE.m` through the dummy input argument `varargin`.

8 Incorporation of boundary conditions

Let us consider the PDE $-\Delta u = f$ on D, with homogeneous Dirichlet conditions on Γ_1 and Neumann conditions $\nabla u \cdot \boldsymbol{\nu} \equiv g$ on Γ_2. We have seen that this yields the variational formulation
Find $u \in V$ s.t.

$$\int_D \nabla u \nabla v = \int_D fv + \int_{\Gamma_2} gv \quad \forall v \in V \ .$$

But how to incorporate these conditions into the equation?
We have already seen that in such a case, the boundary flags have to be properly set in the field `Mesh.BdFlags`, by putting the values to 0 for interiour edges, to `DirFlag` for Dirichlet edges, i.e. edges in Γ_1, and `NeuFlag` for Neumann edges. It is important that `NeuFlag` and `DirFlag` are negative numbers!

8.1 Neumann conditions

When dealing with variational formulations, the Neumann conditions are easier to deal with in implementation. As you can see above, if $g = 0$, the extra term is equal to zero, hence it is cancelled.
The implementation works as follows: First, we assemble the load vector L without boundary conditions using `assemLoad_LFE.m`. Then, we update the Neumann entries again by assembly in a function `assemNeu_LFE.m`, where the `LFE` could be replaced by `QFE` or `BFE` for quadratic or bilinear FEM.
The strategy is as follows: We loop over all Neumann edges (i.e. loop over all Edges `e` for which `Mesh.BdFlags(e)==NeuFlag`) and integrate gb_i over this edge. Hence, we will need a 1D-Quadrature rule whose nodes lie in the interval $[0, 1]$, e.g. `QuadRule=gauleg(0,1,NGauss)`.
As in the load vector assembly, the implementation of g *must* take the boundary flag as second input, hence, written as a function handle, `g=@(x, flag, ADDITIONAL) (...)` , where in `...` you may write what ever your g does. See the following listing:

```
function L = assemNeu_LFE(Mesh,BdFlags,L,QuadRule,FHandle,varargin)
% Initialize constants
nGauss = size(QuadRule.w,1);

% Precompute shape functions
N = shap_LFE([QuadRule.x zeros(nGauss,1)]);

% Preallocate
Lloc = zeros(2,1);

%Loop over all Boundary Flags which mark Neumann Edges
for j1 = BdFlags

    % Extract Neumann edges corresponding to flag j1
    Loc = get_BdEdges(Mesh);
    Loc = Loc(Mesh.BdFlags(Loc) == j1);

    % Loop over all respective edges
    for j2 = Loc'
        % Compute element map: Need to know orientation of edge!
        if(Mesh.Edge2Elem(j2,1))
            % Match orientation to left hand side element
            Elem = Mesh.Edge2Elem(j2,1);
            EdgeLoc = Mesh.EdgeLoc(j2,1);
            id_s = Mesh.Elements(Elem,rem(EdgeLoc,3)+1);
```

```
26              id_e = Mesh.Elements(Elem,rem(EdgeLoc+1,3)+1);
27          else
28              % Match orientation to right hand side element
29              Elem = Mesh.Edge2Elem(j2,2);
30              EdgeLoc = Mesh.EdgeLoc(j2,2);
31              id_s = Mesh.Elements(Elem,rem(EdgeLoc,3)+1);
32              id_e = Mesh.Elements(Elem,rem(EdgeLoc+1,3)+1);
33          end
34
35          %Define start and end points
36          Q0 = Mesh.Coordinates(id_s,:);
37          Q1 = Mesh.Coordinates(id_e,:);
38
39          %Transform 1DÂăQuadrature to edge
40          x = ones(nGauss,1)*Q0+QuadRule.x*(Q1-Q0);
41          dS = norm(Q1-Q0);
42
43          % Evaluate Neumannn boundary data
44          FVal = FHandle(x,j1,varargin{:});
45
46          % Numerical integration along an edge
47          Lloc(1) = sum(QuadRule.w.*FVal.*N(:,1))*dS;
48          Lloc(2) = sum(QuadRule.w.*FVal.*N(:,2))*dS;
49
50          % Add contributions of Neumann data to load vector
51          L(id_s) = L(id_s)+Lloc(1);
52          L(id_e) = L(id_e)+Lloc(2);
53      end
54  end
55
56  return
```

Note that this code works also if you have several `NeuFlags` for different types of Neumann edges.
Example: The solution of $-\Delta u = 0$ with an incorporation of inhomogeneous Neumann conditions $\nabla u(x)\cdot\boldsymbol{\nu}(x) = h(x) := x_1^2 x_2^2$ for all $x \in \partial D$ would happen the following way:

```
1   %%%%ÂăDEFINEÂăFUNCTIONS %%%%
2   f = @(x, flag) zeros(size(x,1), 1);   %Right hand side of equation is zero
3   g = @(x, flag) x(:,1).^2 .* x(:,2).^2;  %Dirichlet conditions
4
5   %%%% DEFINEÂăFLAGS %%%%
6   NeuFlag=-2;
7   IntFlag=0;
8
9   %%%% ASSEMBLE MATRIX AND VECTORÂă%%%%
10  A = assemMat_LFE(Mesh, @STIMA_Lapl_LFE);
11  l = assemLoad_LFE(Mesh, P7O6(), f);
12
13  %%%% INCORPORATEÂăNEUMANNÂăCONDITIONSÂăBEFOREÂăSOLVING %%%%
14  l = assemNeu_LFE(Mesh, NeuFlag, l, P7O6(), g);
15
16  %%%% SOLVEÂăEQUATION %%%%
17  u = A\l;
```

Note that `Mesh` needs to contain the fields `Mesh.ElemFlag` and `Mesh.BdFlags`.

8.2 Dirichlet conditions

This is slightly more difficult. You might remember that the choice of variational space V becomes slightly more challenging for nonhomogeneous Dirichlet conditions and even the variational problem should be manipulated. We refer to the lecture and the exercises.

Assume that $\Gamma_2 = \emptyset$ and the basis functions $\{b_j\}$ are ordered in a way such that b_j corresponds to interiour nodes for $j \leq N_0$ and $b_{N_0}, \dots, b_N$ belong to boundary nodes. The (inhomogeneous) Dirichlet condition is $u|_{\Gamma_1} \equiv h$. Then, the linear system arising form discretization by linear FEM can be written as

$$\begin{bmatrix} A_0 & A_{0\partial} \\ A_{0\partial}^T & A_{\partial\partial} \end{bmatrix} \begin{bmatrix} u_0 \\ u_\partial \end{bmatrix} = \begin{bmatrix} l_0 \\ l_\partial \end{bmatrix} ,$$

where u_∂ is of course given by the values of $h(x)$ at the respective nodes. The first block-line of this equation yields $A_0 u_0 = l_0 - A_{0\partial} u_\partial$. It is an exercise for you to see what happens if $\Gamma_2 \neq \emptyset$ and the basis functions are not ordered in that way.

The function `assemDir_LFE.m` takes the following inputs: The mesh `Mesh`, the vector of flags which correspond to Dirichlet edges (in this case, a scalar `DirFlags`) and the function handle for h, let us call it `FHandle`, where the implementation of `FHandle` is with `flag` as second input: `FHandle =@(x, flag, ADDITIONAL) (...)`.

More important are the outputs of this function: It returns two outputs. A vector `u` which contains the values of h at the Dirichlet degrees of freedom, and a vector `FreeDofs` which returns the indices of vertices which do not lie on the boundary. Obviously `u(FreeDofs)=0`, since we still have to solve the equation at the free degrees of freedom. The following listing contains the source code of `assemDir_LFE.m`:

```
function [U,FreeDofs] = assemDir_LFE(Mesh,BdFlags,FHandle,varargin)
% Intialize constants
nCoordinates = size(Mesh.Coordinates,1);
tmp = [];
U = zeros(nCoordinates,1);

for j = BdFlags
    % Extract Dirichlet nodes
    Loc = get_BdEdges(Mesh);
    DEdges = Loc(Mesh.BdFlags(Loc) == j);
    DNodes = unique([Mesh.Edges(DEdges,1); Mesh.Edges(DEdges,2)]);

    % Compute Dirichlet boundary conditions
    U(DNodes) = FHandle(Mesh.Coordinates(DNodes,:),j,varargin{:});

    % Collect Dirichlet nodes in temporary container
    tmp = [tmp; DNodes];

end

% Compute set of free dofs
FreeDofs = setdiff(1:nCoordinates,tmp);

return
```

The solution of $-\Delta u = 0$ with an incorporation of inhomogeneous Dirichlet conditions $u(x) = h(x) := x_1^2 x_2^2$ for all $x \in \partial D$ would happen the following way:

```
%%%%ÂăDEFINEÂăFUNCTIONS %%%%
f = @(x, flag) zeros(size(x,1), 1);   %Right hand side of equation is zero
h =  @(x, flag) x(:,1).^2 .* x(:,2).^2;  %Dirichlet conditions

%%%% DEFINEÂăFLAGS %%%%
DirFlag=-1;
```

```
IntFlag=0;

%%%% ASSEMBLE MATRIX AND VECTORÂă%%%%
A = assemMat_LFE(Mesh, @STIMA_Lapl_LFE);
l = assemLoad_LFE(Mesh, P706(), f);

%%%% INCORPORATEÂăDIRICHLETÂăCONDITIONSÂăBEFOREÂăSOLVING %%%%
[u, FreeDofs] = assemDir(Mesh, DirFlag, h);
l = l - A*u;     %Update l according to the eqn. derived before

%%%% SOLVEÂăEQUATION %%%%
u(FreeDofs) = A(FreeDofs,FreeDofs)\l(FreeDofs);
```

Note that `Mesh` needs to contain the fields `Mesh.ElemFlag` and `Mesh.BdFlags`.

Exercise

This should be easy right now: Take $D := \{x \in \mathbb{R}^2 \,:\, \|x\| < 1\}$, create a triangular mesh on it using `distmesh`. On this mesh, solve the boundary value problem

$$-\Delta u(x) = f(x) \quad x \in D, \qquad u|_{\Gamma_1}(x) = \sin(\arg(x)), \qquad (\nabla u \cdot \boldsymbol{\nu})\,|_{\Gamma_2}(x) = 10\,.$$

for $f(x) := \chi_{\{x \in D \,:\, \|x\| < \frac{1}{4}\}}$, where for $D' \subseteq D$ we define $\chi_{D'}$ to be the characteristic function taking value 1 at D' and 0 on $D \setminus D'$.

9 Computation of errors

9.1 In the L^2-norm

In many exercises, or to test a code you are going to apply later, it is convenient to test whether the convergence rates which result from your code coincide with the rates predicted by theory. In theory, we have got estimates for $\epsilon(x) := u(x) - u_h(x)$ in $\|\cdot\|_{L^2(D)}$, $\|\cdot\|_{H^1(D)}$, $|\cdot|_{H^1(D)}$ (the H^1-seminorm which can be defined by $|v|^2_{H^1(D)} := \|v\|^2_{H^1(D)} - \|u\|^2_{L^2(D)}$).
To this end, one usually is given the exact solution $u(x)$, which is implemented by the function handle `u`. Our FEM code yields a numerical approximation $u_h(x)$, implemented by the vector of its basis coefficients `uh`.
First of all, we are interested in $\|\epsilon\|_{L^2(D)}$. The LehrFEM routine `L2Err_LFE.m` proceeds the following way:

```
Define some quadrature nodes in the reference element
Set err=0

for el=1:N_Elements
    Compute quadrature nodes in local element

    Evaluate u in local quadrature nodes
    Evaluate uh in local quadrature nodes (to this end, evaluate local shape functions)

    Integrate numerically |u-uh|^2 on the local element, call this loc_int

    err=err+loc_int
end

err=sqrt(err)
```

The code is contained in the following listing:

```
function err = L2Err_LFE(Mesh,u,QuadRule,UHandle,varargin)
% Intialize constants
nPts = size(QuadRule.w,1);
nElements = size(Mesh.Elements,1);

% Precompute shape functions
N = shap_LFE(QuadRule.x);

% Compute discretization error
err = 0;
for i = 1:nElements
    % Extract vertex numbers
    vidx = Mesh.Elements(i,:);

    % Compute element mapping
    bK = Mesh.Coordinates(vidx(1),:);
    BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
    det_BK = abs(det(BK));

    % Transform quadrature points
    x = QuadRule.x*BK+ones(nPts,1)*bK;

    % Evaluate solutions
    u_EX = UHandle(x,varargin{:});
    u_FE = u(vidx(1))*N(:,1) + u(vidx(2))*N(:,2) + u(vidx(3))*N(:,3);

```

```
27      % Compute error on current element
28      err = err+sum(QuadRule.w.*abs(u_EX-u_FE).^2)*det_BK;
29  end
30  err = sqrt(err);
31
32  return
```

Let us explain the input values: `Mesh` and `QuadRule` do not need further explanation. `u` is what we call u_h, the vector of basis coefficients which is denoted by `uh` in the pseudocode above. The function handle `UHANDLE` evaluates the exact solution. The output is a number `err`, containing (the numerical approximation of) $\|\epsilon\|_{L^2(D)}$.

9.2 In the H^1-norm

In order to get the H^1-error, we need to add an evaluation of the gradients of u_h and u in each element and add up the two integrals. The evaluation of the gradients of the shape functions is done automatically by the function `shap_grad.m`, while (see before) the evaluation of the shape functions is done by the function `shap.m`. By the way, we follow a widely accepted convention that gradients are always considered to be row vectors, while all other vectors are columns.
Let us look at the source of the H^1-Error computation function `H1Err_LFE.m`:

```
1   function err = H1Err_LFE(Mesh, u, QuadRule, UHandle, varargin)
2   % Inititialize constants
3   nPts = size(QuadRule.w,1);
4   nElements = size(Mesh.Elements,1);
5
6   % Precompute gradients and values of shape functions
7   N = shap_LFE(QuadRule.x);
8   grad_N = grad_shap_LFE(QuadRule.x);
9
10  % Compute discretization error
11  err = 0;
12  for i= 1:nElements
13      % Extract vertex numbers
14      vidx = Mesh.Elements(i,:);
15
16      % Compute element mapping
17      bK = Mesh.Coordinates(vidx(1),:);
18      BK = [Mesh.Coordinates(vidx(2),:)-bK; Mesh.Coordinates(vidx(3),:)-bK];
19      inv_BK = inv(BK);
20      det_BK = abs(det(BK));
21
22      % Transform quadrature points
23      x = QuadRule.x*BK+ones(nPts,1)*bK;
24
25      % Evaluate solutions
26      [u_EX grad_u_EX] = UHandle(x, varargin{:});
27      u_FE = u(vidx(1))*N(:,1)+u(vidx(2))*N(:,2)+u(vidx(3))*N(:,3);
28      grad_u_FE = (u(vidx(1))*grad_N(:,1:2)+ ...
29          u(vidx(2))*grad_N(:,3:4)+ ...
30          u(vidx(3))*grad_N(:,5:6))*transpose(inv_BK);
31
32      % Compute error on the current element
33      err = err+sum(QuadRule.w.*(abs(u_EX-u_FE).^2+...
34          sum(abs(grad_u_FE-grad_u_EX).^2,2)))*det_BK;
35  end
```

```
36 err = sqrt(err);
37 return
```

The only thing which changes here in the input variables is that `UHandle(x)` returns *two outputs*: The first output is the evaluation of the exact solution u as before, but the second output is the evaluation of the gradient of u. If `x` is a list of coordinates of size $N \times 2$, and we call `[u grad_u]=UHandle(x)`, then `u` is a column vector of length N while `grad_u` is a $N \times 2$ array, containing $\nabla u($`x(i,:)`$)$ in the i-th column. See line 26 in the listing. Again, if e.g. u is time-dependent or you wish to add some other extra input variables to `UHandle`, in both cases discussed here, `varargin` redirects extra inputs of `L2Err_LFE` and `H1Err_LFE`, respectively, to `UHandle`.

10 Plots of solutions

LehrFEM provides you with functions which are able to plot the obtained solution on your mesh. We take as examples the functions `plot_BFE.m` for rectangular bilinear FEM and `plot_LFE.m` for triangular linear FEM. The syntax is always the same: `plot_LFE(U,Mesh)` will plot your numerical solution given by the vector `U` on the mesh `Mesh`. Analogously `plot_BFE(U,Mesh)`.
We quickly would like to show you how to plot a solution on a triangular mesh only using matlab-tools. You might know the commands `surf` and `mesh`. The key tool for triangular meshes is `trimesh` or `trisurf` (see the help for explanation of the differences between these functions).
This works the following (assume `U` is the solution vector and `Mesh` is the mesh struct):

```
1 figure()    %open new figure
2 trisurf(Mesh.Elements, Mesh.Coordinates(:,1), Mesh.Coordinates(:, 2), U);
3 OPTIONS
```

where `OPTIONS` could be `view(2)` for a view from above, or `shading interp` if you do not want the element borders to be displayed. If you use `view(2)`, the plot will be 2D, and D will be coloured according to the values of u_h. You might be interested in knowing which color belongs to which value; in this case you should add `colorbar` to your `OPTIONS`.

11 A final example on all sections

Let $D := [0,1]^2$, $\Gamma_1 := \{x \in D : x_1 \in \{0,1\}\}$, $\Gamma_2 := \{x \in D : x_2 \in \{0,1\}\}$, create a uniform triangular mesh on D using `distmesh`.
On this mesh, solve the boundary value problem

$$-\Delta u(x) = f(x) \quad x \in D, \qquad u|_{\Gamma_1}(x) = h(x), \qquad (\nabla u \cdot \boldsymbol{\nu})\,|_{\Gamma_2}(x) = g(x)\ .$$

for an exact solution $u(x) := \sin(f_1 x_1)\sin(f_2 x_2)$, with given frequencies $f_i \in \mathbb{R}$.
To this for an initial meshwidth $h_0 = \frac{1}{2}$, refine the initial mesh 6 times and compute for each refinement the L^2 and H^1-error. Find the fitted convergence rate and plot it with its comparison lines.

The following code solves this little exercise for $f_1 = \pi, f_2 = 2\pi$. See figure 3-5 for the plots of the meshes, the solutions and the errors.

```
1  clear all, close all
2
3  %%% SET INPUTS %%%
4  % mesh inputs
5  BBox=[-1.2 -1.2; 1.2 1.2];
6  h0=2.^(-1);
7  DHandle=@(x)-min(min(min(x(:,2),1-x(:,2)), x(:,1)), 1-x(:,1)); %Rectangle
8  HHandle=@(x)ones(size(x,1),1);
9  FixedPos=[0 0; 1 0; 1 1; 0 1;.5 .5];
10 Disp=0;
11 nRef=6;
```

```
% Equation inputs
f1=pi;
f2=pi;
u=@(x)sin(f1*x(:, 1)).*sin(f2*x(:, 2));
GradHandle=@(x) [f1*cos(f1*x(:,1)).*sin(f2*x(:, 2))...
    f2*cos(f2*x(:,2)).*sin(f1*x(:, 1))];
f=@(x,flag) (f1^2+f2^2)*u(x);
dirHandle=@(x, flag)u(x);
neuHandle=@(x, flag) (x(:, 1)==0).*(-f1*cos(f1*x(:,1)).*sin(f2*x(:,2))) +...
    (x(:, 1)==1).*(f1*cos(f1*x(:,1)).*sin(f2*x(:,2))) +...
    (x(:, 2)==0).*(-f2*cos(f2*x(:, 2)).*sin(f1*x(:,1))) +...
    (x(:, 2)==1).*(f2*cos(f2*x(:, 2)).*sin(f1*x(:,1))) ;

%Define Flags
NeuFlag=-1;
DirFlag=-2;

%Preallocate
errL2=zeros(nRef,1);
errH1=errL2;
h=errL2;

%%% GENERATE INITIALÂăMESH %%%
Mesh=init_Mesh(BBox, h0, DHandle, HHandle, FixedPos, Disp);
Mesh.ElemFlag=ones(size(Mesh.Elements, 1), 1);
Mesh=add_Edges(Mesh);
Mesh=add_Edge2Elem(Mesh);

%%% Add BdFlags %%%
%Extract Bdry Edge midpoints
loc=get_BdEdges(Mesh);
EdgeMidPts=.5 * (Mesh.Coordinates(Mesh.Edges(loc, 1),:) + ...
    Mesh.Coordinates(Mesh.Edges(loc, 2),:));
%Define Flags according to coordinates of Edge midpoints
Mesh.BdFlags=zeros(size(Mesh.Edges, 1), 1);
Mesh.BdFlags(loc(EdgeMidPts(:,1)==0))=DirFlag;
Mesh.BdFlags(loc(EdgeMidPts(:,1)==1))=DirFlag;
Mesh.BdFlags(loc(EdgeMidPts(:,2)==0))=NeuFlag;
Mesh.BdFlags(loc(EdgeMidPts(:,2)==1))=NeuFlag;

%%% REFINEMENTÂăLOOP %%%
for idxRef=1:nRef
    disp(['Refinement step ', num2str(idxRef), ' out of ', num2str(nRef),
        '...'])

    %%% Plot the MEsh %%%
    figure(1)
    subplot(2,3,idxRef)
    plot_Mesh(Mesh, 'f')

    %%% ASSEMBLEÂăMATRIX ANDÂăVECTOR %%%
    A=assemMat_LFE(Mesh, @STIMA_Lapl_LFE);
```

```
    l=assemLoad_LFE(Mesh, P706(), f);

    %%% Manipulate B.C. %%%
    l=assemNeu_LFE(Mesh, NeuFlag, l, gauleg(0,1,10), neuHandle);
    [Uh, FreeDofs]=assemDir_LFE(Mesh, DirFlag, dirHandle);
    l=l-A*Uh;

    %%% Solve %%%
    Uh(FreeDofs)=A(FreeDofs, FreeDofs)\l(FreeDofs);

    %%% Error computation %%%
    errL2(idxRef)=L2Err_LFE(Mesh, Uh, P706(), u);
    semierr=H1SErr_LFE(Mesh,Uh,P706(),GradHandle);
    errH1(idxRef)=sqrt(errL2(idxRef)^2+semierr^2);

    %%% Save Meshwidth %%%
    h(idxRef)=get_MeshWidth(Mesh);

    %%% Solution plot %%%
    figure(2)
    subplot(2, 3, idxRef)
    trisurf(Mesh.Elements, Mesh.Coordinates(:, 1), Mesh.Coordinates(:, 2),
       Uh);
    view(2)
    shading interp
    colorbar

    %%% REFINEÂăMESH %%%
    Mesh=refine_REG(Mesh);
end
%%% FITTEDÂăCONVERGENCEÂăRATEÂă%%%
rateL2=polyfit(log(h), log(errL2), 1);
rateL2=rateL2(1);
rateH1=polyfit(log(h), log(errH1), 1);
rateH1=rateH1(1);

disp('---')
disp('Fitted convergence rates:')
disp(['                         L2-Norm: ', num2str(rateL2)]);
disp(['                         H1-Norm: ', num2str(rateH1)]);

%%% Loglog Plot %%%
figure(3)
loglog(h, errL2, 'ko-', h, errH1, 'ks-',...
    h, h.^(1), 'k-.', h, h.^2, 'k:')
title('Experimental H^1 and L^2 Errors')
legend('L^2-Error', 'H^1-Error', 'Rate 1', 'Rate 2', ...
    'Location', 'SouthEast')
%%%
```

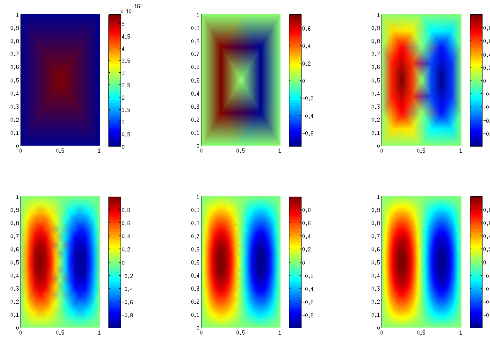

Figure 3: Experimental solutions of the example

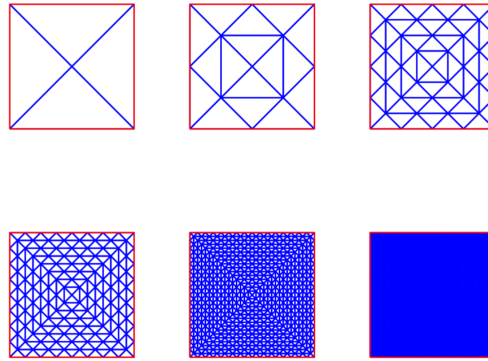

Figure 4: Meshes in the example

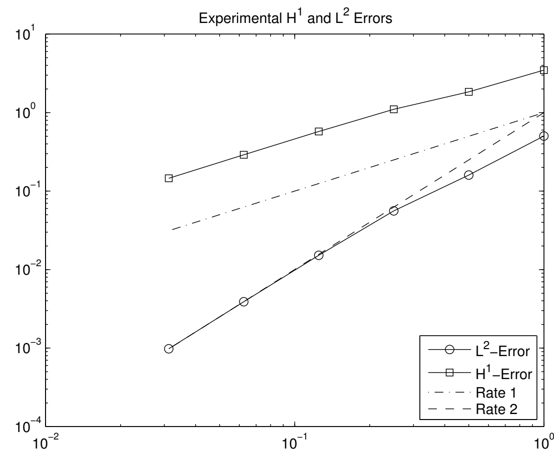

Figure 5: Convergence plots from the example