LehrFEM - A 2D Finite Element Toolbox

Annegret Burtscher, Eivind Fonn, Patrick Meury

January 18, 2012

Contents

Introduction

sect:Intro

LehrFEM is a 2D finite element toolbox written in the programming language MATLAB for educational purpose.

The chapter 'Mesh Generation' was written by Patrick Meury in 2005. The rest of the basic framework was summerized in the chapters 'Local Shape Functions and its Gradients', 'Numerical Integration', 'Local Computations', 'Assembly', 'Boundary Conditions', 'Plotting the Solution' and 'Discretization Errors' by Annegret Burtscher in 2008.

Eivind Fonn contributed the 'Finite Volume Code for Solving Convection/Diffusion Equations' in 2007.

Reorganization and extension by the chapter 'Examples' by Christoph Wiesmeyr in 2009.

The chapters are organized in a way that they contain files of the same type and same folder of the LehrFEM. For an overview of both, the folder structure of the LehrFEM and the manual, see p. chap:overview 9.

Generally on the beginning of each chapter and section there is a summary of the task of the functions involved, as well as the input, output, call and main steps of the implementation. In the chapters explaining the implementation the focus is on the easiest finite elements, which involves linear or quadratic basis functions. However there is always a small explanation of the more evolved methods which can be ommited on the first reading. For further explanation it is recommeded to read the explanations in the chapter 'Examples', where more complicated FEM are explained in more detail.

Readers are expected to have a background in linear algebra, calculus and the numerical analysis of PDEs. The theoretical concepts behind the implementation are more or less omitted, but may be found in the lecture notes HS07 [?] and books about the FEM.

This manual may be found in the folder `/Lib/Documentation/MANUAL`. The keyfile is `manual.tex`.

MATLAB formulations are written in the `typewriter font`, as well as the `.m`-files (without `.m` in the end) and variables that appear in the functions. Folders start with an `/`, `*` is used as a wildcard character – mostly for a certain type of finite elements, e.g. in `shap_*` the `*` may be replaced by `LFE`.

Note: The purpose of the `startup`-function is to add the various directories to the search path. It must be run each time before working with the LehrFEM functions.

Overview

chap:overview

Folders and files in LehrFEM :

operations	folder	file names	chapter
mesh generation	/Lib/MeshGen	init_Mesh etc.	sect:MGEN 1.2
mesh refinements	/Lib/MeshGen	refine_REG etc.	sect:MR 1.4
load mesh	/Lib/MeshGen	load_Mesh	ssect:load_save_mesh 1.3.1
save mesh	/Lib/MeshGen	save_Mesh	ssect:load_save_mesh 1.3.1
plot mesh	/Lib/Plots	plot_Mesh_*	ssect:plot_mesh 1.3.2
shape functions	/Lib/Element	shap_*	chap:shap_fct 2
gradients of shape functions	/Lib/Element	grad_shap_*	chap:shap_fct 2
plot shape functions	/Examples/PlotShap resp. /Lib/Plots	main_Shap_* resp. plot_Shap_*	sect:shap_plot 2.2
quadrature rules	/Lib/QuadRules	P*O* etc.	chap:quad_rule 3
element stiffness matrices	/Lib/Element	STIMA_*	sect:stima 4.1
element mass matrices	/Lib/Element	MASS_*	sect:mass 4.2
element load vectors	/Lib/Element	LOAD_*	sect:load 4.3
assembly of stiffness/mass matrices	/Lib/Assembly	assemMat_*	sect:assem_mat 5.1
assembly of load vectors	/Lib/Assembly	assemLoad_*	sect:assem_load 5.2
incorporation of Dirichlet boundary conditions	/Lib/Assembly	assemDir_*	sect:assem_dir 6.1
incorporation of Neumann boundary conditions	/Lib/Assembly	assemNeu_*	sect:assem_neu 6.2
solvers	/Lib/Solvers	*_solve etc.	
plot of solution	/Lib/Plots	plot_*	sect:plot 7.1
plot section	/Lib/Plots	plotLine_*	sect:plot_line 7.2
plot contours	/Lib/Plots	contour_*	sect:plot_contour 7.3
H^1 discretization errors	/Lib/Errors	H1Err_*	sect:h1_err 8.1
H^1_s discretization errors	/Lib/Errors	H1SErr_*	sect:h1s_err 8.2
L^1 discretization errors	/Lib/Errors	L1Err_*	sect:l1_err 8.3
L^2 discretization errors	/Lib/Errors	L2Err_*	sect:l2_err 8.4
L^∞ discretization errors	/Lib/Errors	LInfErr_*	sect:linf_err 8.5
S^1 discretization errors	/Lib/Errors	HCurlSErr_*	
error estimates	/Lib/ErrEst	ErrEst_*	
error distributions	/Lib/ErrorDistr	H1ErrDistr_* and L2ErrDistr_*	
examples	/Examples	main_* etc.	

Chapter 1

Mesh Generation and Refinement

chap:mesh_gen

This chapter contains the documentation for the MATLAB library LehrFEM of the mesh data structures and the mesh generation/refinement routines. Currently the library supports creation of 2D structured triangular meshes for nearly arbitrarily shaped domains. Furthermore it is possible to create unstructured quadrilateral meshes by a kind of morphing procedure from triangular meshes. Structured meshes for both triangular and quadrilateral elements can be obtained throu uniform red refinements from coarse initial meshes.

1.1 Mesh Data Structure

sect:mesh_DS

All meshes in LehrFEM are represented as MATLAB structs. This makes it possible to encapsulate all data of a mesh inside only one variable in the workspace.

For the rest of this manual we denote by M the number of vertices, by N the number of elements and by P the number of edges for a given mesh.

The basic description of mesh contains the fields `Coordinates` of vertex coordinates, and a list of elements connecting them, corresponding to the field `Elements`. The details of this data structure can be seen on table 1.1.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 or N-by-4 matrix connecting vertices into elements

Table 1.1: Basic mesh data structure

tab:MSH_B

When using higher order finite elements we need to place global degrees of freedom on edges of the mesh. Even though a mesh is fully determined by the fields `Coordinates` and `Elements` it is necessary to extend the basic mesh data structure by edges and additional connectivity tables connecting them to elements and vertices. A call to the following routine

```
>> Mesh = add_Edges(Mesh);
```

will append the fields `Edges` and `Vert2Edges` to any basic mesh data structure containing the fields `Coordinates` and `Elements`. Detailed information on the additional fields can be found on table tab:MSH_E 1.2.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 or N-by-4 matrix connecting vertices into elements
`Edges`	P-by-2 matrix specifying all edges of the mesh
`Vert2Edge`	M-by-M sparse matrix specifying the edge number of the edge connecting vertices `i` and `j` (or zero if there is no edge)

Table 1.2: Mesh with additional edge data structure tab:MSH_E

To be able to incorporate boundary conditions into our variational formulations we need to seperate boundary edges from interior edges. Calling the function

```
>> Loc = get_BdEdges(Mesh);
```

provides us with a locator `Loc` of all boundary edges of the mesh in the field `Edges`. For all the boundary edges flags that specify the type of the boundary condition can be set. They are written in the field `BdFlags` and the convention is that every boundary edge gets a negative flag while the edges in the interior are flaged by 0.

When using discontinuous Galerkin finite element methods or edge-based adaptive estimators we need to compute jumps of solutions across edges. This makes it necessary to be able to determine the left and right hand side neighbouring elements of each edge. The function call

```
>> Mesh = add_Edge2Elem(Mesh);
```

adds the additional field `Edge2Elem`, which connects edges to their neighbouring elements for any mesh data structure containing the fields `Coordinates`, `Elements`, `Edges` and `Vert2Edge`. For details consider table tab:MSH_E2 1.3.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 or N-by-4 matrix connecting vertices into elements
`Edges`	P-by-2 matrix specifying all edges of the mesh
`Vert2Edge`	M-by-M sparse matrix specifying the edge number of the edge connecting vertices `i` and `j` (or zero if there is no edge)
`Edge2Elem`	P-by-2 matrix connecting edges to their neighbouring elements. The left hand side neighbour of edge `i` is given by `Edge2Elem(i,1)`, whereas `Edge2Elem(i,2)` specifies the right hand side neighbour, for boundary edges one entry is 0
`EdgeLoc`	P-by-3 or P-by-4 matrix connecting egdes to local edges of elements.

Table 1.3: Mesh with additional edge data structure and connectivity table tab:MSH_E2

In some finite element applications we need to compute all sets of elements sharing a specific vertex of the mesh. These sets of elements, usually called patches, can be appended to any mesh containing the fields `Coordinates` and

`Elements` by the following routine

```
>> Mesh = add_Patches(Mesh);
```

for details consider table tab:MSH_P 1.4.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 or N-by-4 matrix connecting vertices into elements
	M-by-Q matrix specifying all elements sharing a specific vertex of the mesh, $Q = \max($`AdjElements`$)$
`nAdjElements`	M-by-1 matrix specifying the exact number of neighbouring elements at each vertex of the mesh

Table 1.4: Mesh with additional patch data structure tab:MSH_P

For the Discontinuous Galerkin Method (DG) the normals and the edge orientation of every edge is needed. For a Mesh containing the fields `Coordinates`, `Elements`, `Edges`, `Vert2Edge`, `Edge2Elem` and `EdgeLoc` the missing fields are added by calling the routine

```
>> Mesh = add_DGData(Mesh);
```

1.2 Mesh Generation

sect:MGEN

Up to now LehrFEM supports two ways for generating triangular meshes. The first possibility is to manually build a MATLAB struct, which contains the fields `Coordinates` and `Elements`, that specify the vertices and elements of a given mesh.

The second more sophisticated method is to use the built-in unstructured mesh generator `init_Mesh`, which is a wrapper function for the mesh generator *DistMesh* from Per-Olof Persson and Gilbert Strang PER04 [?]. This code uses a *signed distance function* $d(x, y)$ to represent the geometry, which is negative inside the domain. It is is easy to create distance functions for simple geometries like circles or rectangles. Already contained in the current distribution are the functions `dist_circ`, which computes the (signed) distance from a point `x` to the the circle with center `c` and radius `r`, and `dist_rect`, which computes minimal distance to all the four boundary lines (each extended to infinity, and with the desired negative sign inside the rectangle) of the rectangle with lower left corner point `x0` and side lengths `a` and `b`. Note that this is *not* the correct distance to the four external regions whose nearest points are corners of the rectangle.

Some more complicated distance functions can be obtained by combining two geometries throu unions, intersections and set differences using the functions `dist_union`, `dist_isect` and `dist_diff`. They use the same simplifaction just mentioned for rectangles, a max or min that ignores "closest corners". We use seperate projections to the regions A and B, at distances $d_A(x, y)$ and $d_B(x, y)$:

$$\textbf{Union :} \quad d_{A\cup B}(x,y) = \min\{d_A(x,y), d_B(x,y)\} \tag{1.1}$$

$$\textbf{Difference :} \quad d_{A\setminus B}(x,y) = \max\{d_A(x,y), -d_B(x,y)\} \tag{1.2}$$

$$\textbf{Intersection} \quad d_{A\cap B}(x,y) = \max\{d_A(x,y), d_B(x,y)\} \tag{1.3}$$

The function `init_Mesh` must be called from the MATLAB command window using either one of the following argument signatures

```
>> Mesh = init_Mesh(BBox,h0,DHandle,HHandle,FixedPos,disp);
>> Mesh = init_Mesh(BBox,h0,DHandle,HHandle,FixedPos,disp,FParam);
```

for a detailed explanation on all the arguments, which need to be handled to the routine `init_mesh` consider table tab:ARG 1.5.

`BBox`	Enclosing bounding box of the domain
`h0`	Desired initial element size
`DHandle`	Signed distance function. Must be either a MATLAB function handle or an inline object
`HHandle`	Element size function. Must be either a MATLAB function handle or an inline object
`FixedPos`	Fixed boundary vertices of the mesh. Vertices located at corner points of the mesh must be fixed, otherwise the meshing method will not converge
`disp`	Display option flag. If set to zero, then no mesh will be displayed during the meshing process, else the mesh will be displayed and redrawn after every *delaunay* retriangulation
`FParam`	Optional variable length argument list, which will be directly handled to the signed distance and element size functions

Table 1.5: Argument list of the `init_Mesh` routine tab:ARG

Figure fig:MSH_ANN 1.1 shows an unstructured mesh of the right upper region of the annulus generated by the routine `init_Mesh`.

Figure 1.1: Mesh of upper right region of the annulus fig:MSH_ANN

sect:IO

1.3 Loading, Saving and Plotting Meshes

ssect:load_save_mesh

1.3.1 Loading and Saving Meshes

LehrFEM offers the possibility to load and save basic meshes, containing the fields `Coordinates` and `Elements`, from and to files in ASCII format. Loading and saving a mesh from or to the files `Coordinates.dat` and `Elements.dat` can be done by the the following two lines of code

```
>> Mesh = load_Mesh('Coordinates.dat','Elements.dat');
>> save_Mesh(Mesh,'Coordinates.dat','Elements.dat');
```

1.3.2 Plotting Routines

ssect:plot_mesh

In the current version there are three different types of mesh plotting routines implemented – `plot_Mesh`, `plot_Qual` and `plot_USR`. Besides, `plot_DomBd` plots the boundary of a mesh. All plotting routines are stored in the folder `/Lib/Plots` and explained in the following.

Plot of Elements

The first plotting routine prints out the elements of a mesh. It is called by one of the following lines:

```
>> H = plot_Mesh(Mesh);
```

For an example see figure fig:MSH_ANN 1.1. It is also possible to add specific element, edge and vertex labels to a plot by specifying an optional string argument:

```
>> H = plot_Mesh(Mesh,Opt);
```

Here `Opt` is a character string made from one element from any (or all) of the characters described in table tab:plot_mesh_opt 1.6.

`'f'`	does not create a new window for the mesh plot
`'p'`	adds vertex labels to the plot using `add_VertLabels`
`'t'`	adds element labels/flats to the plot using `add_ElemLabels`
`'e'`	adds edge labels/flags to the plot using `add_EdgeLabels`
`'a'`	displays axes on the plot
`'s'`	adds title and axes labels to the plot

Table 1.6: Optional characters `Opt`

tab:plot_mesh_opt

The following already mentioned subfunctions are used:

- `add_VertLabels` adds vertex labels to the current figure and is called by

  ```
  >> add_VertLabels(Coordinates);
  ```

- `add_ElemLabels` adds the element labels `Labels` to the current figure by

  ```
  >> add_ElemLabels(Coordinates,Elements,Labels);
  ```

- `add_EdgeLabels` adds the edge labels `Labels` to the current figure by

  ```
  >> add_EdgeLabels(Coordinates,Edges,Labels);
  ```

Element Quality Plot

The plotting routine of the second type generates a 2D plot of the element quality for all triangles T contained in the mesh according to the formula

$$q(T) = 2\frac{r_{\text{in}}}{r_{\text{out}}} \tag{1.4}$$

where r_{out} is the radius of the circumscribed circle and r_{in} of the inscribed circle. It is called by

```
>> H = plot_Qual(Mesh);
```

and returns the handle `H` to the figure. For an example see figure 1.2.

Figure 1.2: Element quality plot fig:ELEM_QUAL

Plot of Uniform Similarity Region

The plotting routine of the third kind generates plot of the uniform similarity region for triangles as described in the review article by D. A. Field [?]. The uniform similarity region seperates all elements into *obtuse* triangles with one angle larger than $\pi/2$, and *acute* triangles. The elements above the half circle are acute, whereas the elements below are obtuse. The function `plot_USR` is called by

```
>> H = plot_USR(Mesh);
```

For an example of this plot see figure 1.3.

Boundary Plot

Additionally, the function `plot_DomBd` generates a 2D boundary plot of the mesh boundary. The string characters `'a'` and `'s'` may be added in the argument `Opt`, see table 1.6. Therefore it is called by one of the following lines:

Figure 1.3: Plot of the uniform similarity region fig:USR

```
>> H = plot_DomBd(Mesh);
>> H = plot_DomBd(Mesh,Opt);
```

with e.g. `Opt = 'as'`.

1.4 Mesh Refinements

sect:MR

In order to obtain accurate numerical approximate solutions to partial differential equations it is necessary to refine meshes up to a very high degree. The current version of LehrFEM supports uniform red refinements and adaptive green refinenments of the mesh.

1.4.1 Uniform Mesh Refinements

sect:UMR

Uniform mesh refinement is based on the idea to subdivide every element of the mesh into four elements. For triangles this refinement strategy can be seen in figure fig:REG 1.4.

Figure 1.4: Red refinement for triangular elements fig:REG

In order to be able to perform regular red refinement steps, the mesh data structure needs to provide additional information about the edges forming part of the boundary and interior edges of the domain. This additional information is contained in the field `BdFlags` of the mesh. This array can be used to parametrize parts of the boundary of the domain. In the current version of LehrFEM we use the convention, that *edges forming part of the boundary have negative boundary flags*. For further details concerning the mesh data structure used for uniform red refinements consider table tab:MSH_R 1.7.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 or N-by-4 matrix connecting vertices into elements
`Edges`	P-by-2 matrix specifying all edges of the mesh
`BdFlags`	P-by-1 matrix specofying the the boundary flags each edge in the mesh. If edge number `i` belongs to the boundary, then `BdFlags(i)` must be smaller than zero. Boundary flags can be used to parametrize parts of the boundary
`Vert2Edge`	M-by-M sparse matrix specifying the edge number of the edge connecting vertices `i` and `j` (or zero if there is no edge)

Table 1.7: Mesh data structure for uniform red refinements tab:MSH_R

To perform one uniform red refinement step with a mesh suitable for red refinements just type the following line into the MATLAB command window

```
>> Mesh = refine_REG(Mesh);
```

refine_REG

During the refinement procedure it is also possible to project the new vertices created on the boundary edges of the mesh onto the boundary of the domain. This is necessary if the boundary can not be resembled exactly using triangles, e.g. circles. To do so just type the following command into the MATLAB command window

```
>> Mesh = refine_REG(Mesh,DHandle,DParam);
```

here `DHandle` denotes a MATLAB function handle or inline object to a signed distance function specifying the geometry of the domain, as described in section sect:MGEN 1.2. This distance function is used to move new vertices generated on boundary edges towards the boundary of the domain. `DParam` denotes an optional variable length argument list which is directly handled to the distance function.

Furthermore it is possible to extend a fine mesh, which has been obtained by one uniform red refinement from a coarse mesh, with multilevel data, that specifies the locations of vertices, elements and edges of the fine mesh on the coarse mesh. For details concerning these additional fields please consider table tab:MLEVEL 1.8.

In order to append these additional fields to a refined mesh just simply type in the following command into the MATLAB command window

```
>> Mesh = add_MLevel(Mesh);
```

No information about the coarse mesh is needed, since it is possible to deduce all vital information from the refinement algorithm.

`Father_Vert`	M-by-3 matrix specifying the ancestor vertex/edge/element of each vertex in the current mesh. If vertex `i` is located on a vertex of the coarse mesh, then `Father_Vert(i,1)` points to that father vertex in the coarse mesh. If vertex `i` is located on an edge of the coarse mesh, then `Father_Vert(i,2)` points to that edge in the coarse mesh. Last bu not least, if vertex `i` is located on an element of the coarse mesh, then `Father_Vert(i,3)` points to that element.
`Father_Elem`	N-by-1 matrix specifying the ancestor element of each element in the current mesh. `Father_Elem(i)` points to the father element of element `i` in the coarse mesh.
`Father_Edge`	P-by-2 matrix specifying the ancestor edge/element of each edge in the current mesh. If edge `i` is located inside an element of the coarse mesh, then `Father_Edge(i,2)` points to that element in the coarse mesh, else `Father_Edge(i,1)` points to the father edge of edge `i` in the coarse mesh.

Table 1.8: Additional multilevel data fields tab:MLEVEL

1.4.2 Adaptive Mesh Refinements

sect:AMR

The current version of LehrFEM also supports adaptive mesh refinement from an initial coarse mesh. Up to now LehrFEM only supports the largest edge bisection algorithm for triangular elements.

Adaptive mesh refinements are based on a-posteriori error estimates. Error estimates for every element can be computed and then the elements where the imposed error is large are marked for a refinement. The largest edge bisection algorithm is one way to subdivide triangles.

The implementation of the largest edge bisection is based on the data structure and algorithms presented in the manual of the adaptive hierarchical finite element toolbox ALBERT of A. Schmidt and K.G. Siebert ALB00 [?]. For every element one of its edges is marked as the refinement edge, and the element into two elements by cutting this edge at its midpoint (green refinement).

In our implementation we use the largest edge bisection in Mitchell's notation MI89 [?], which relies on the convention that the local enumeration of vertices in all elements is given in such a way that the longest edge is located between local vertex 1 and 2. Now for every element the longest edge is marked as the refinement edge and for all child elements the newly created vertex at the midpoint of this edge obtains the highest local vertex number. For details on the local element enumeration and the splitting process se figure fig:LEB 1.5.

Sometimes the refinement edge of a neighbour does not coincide with the refinement edge of the current element. Such a neighbour is not compatibly divisible and a green refinement on the current element would create a hanging node. Thus we have to perform a green refinement at the neighbours refinement edge first. The child of such a neighbour at the common edge is then compatibly divisible. Thus it can happen that elements not marked for refinement will be refined during the refinement procedure.

If an element is marked for green refinement we need to access the neighbouring element at the refinement edge and check wheter its local vertex enumeration is compatibly divisible with the current marked element. Thus for every element the data structure needs to provide us with a list of all the neighbouring ele-

Figure 1.5: largest edge bisection fig:LEB

ments, which is given by the field `Neigh`, and the local enumeration of vertices for all neighbouring elements, which is given by the list `Opp_Vert`. For details see table 1.9. This additional information can be created very easily by the following function call

```
>> Mesh = init_LEB(Mesh);
```

which initializes the struct `Mesh`, that contains the fields `Coordinates`, `Elements`, `Edges`, `BdFlags` and `Vert2Edge` for largest edge bisection. For details on the mesh data structure needed see table 1.7 again.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 matrix connecting the vertices into elements
`Neigh`	N-by-3 matrix specifying all elements sharing an edge with the current element. For element `i` `Neigh(i,j)` specifies the neighbouring element at the edge opposite of vertex `j`
`Opp_Vert`	N-by-3 matrix specifying the opposite local vertex of all neighbours of an element. For element`i` `Neigh(i,j)` specifies the local index of the opposite vertex of the neighbouring element at the edge opposite of vertex `j`

Table 1.9: Data structure used for largest edge bisection tab:MSH_LEB

By using a-posteriori estimates a list of elements to be refined can be created by storing the corresponding labels in `Marked_Elements`. With this information and the struct `Mesh` the following routine computes the refined mesh

```
>> Mesh = refine_LEB(Mesh,Marked_Elements);
```

1.5 Postprocessing

In the current version of LehrFEM a small amount of postprocessing routines are available.

1.5.1 Mesh Translation, Rotation and Stretching

In order to translate a mesh by a vector `x_0`, or rotate a mesh by the angle `phi` in counter-clockwise direction or stretch it by `x_dir` in x-direction and `y_dir` in y-direction just simply type the following commands into the MATLAB command window

```
>> Mesh = shift(Mesh,x_0);
>> Mesh = rotate(Mesh,phi);
>> Mesh = stretch(Mesh,x_dir,y_dir);
```

1.5.2 Laplacian Smoothing and Jiggle

The current version of LehrFEM supports unconstrained Laplacian smoothing for triangular and quadrilateral elements. In order to prevent the domain from shrinking it is possible to fix positions of certain vertices of the mesh. In order to use the unconstrained Laplacian smoother simply type the following command into the MATLAB command window

```
>> Mesh = smooth(Mesh,FixedPos);
```

here `FixedPos` is a M-by-1 matrix whose non-zero entries denote fixed vertices of the mesh.

To make a mesh less uniform there is a function that randomly moves inner vertices. This can for example be useful to investigate convergence rates on more random grids. To call the function type

```
>> Mesh = jiggle(Mesh,FixedPos);
```

Usually there is a jiggle parameter specified and based on which the Mesh is processed. The typical code segment can be found below.

```
switch(JIG)
   case 1
     New_Mesh = Mesh;
   case 2
     Loc = get_BdEdges(Mesh);
     Loc = unique([Mesh.Edges(Loc,1); Mesh.Edges(Loc,2)]);
     FixedPos = zeros(size(Mesh.Coordinates,1),1);
     FixedPos(Loc) = 1;
     New_Mesh = jiggle(Mesh,FixedPos);
   case 3
     Loc = get_BdEdges(Mesh);
     Loc = unique([Mesh.Edges(Loc,1); Mesh.Edges(Loc,2)]);
     FixedPos = zeros(size(Mesh.Coordinates,1),1);
     FixedPos(Loc) = 1;
     New_Mesh = smooth(Mesh,FixedPos);
end
```

Chapter 2

Local Shape Functions and its Gradients

chap:shap_fct

Assume we have already given or generated a mesh. The task is to find a basis of locally supported functions for the finite dimensional vector space V_N (of certain polynomials) s.t. the basis functions are associated to a single cell/edge/face/vertex of the mesh and that the supports are the closure of the cells associated to that cell/edge/face/vertex.

Once we have given the shape functions we can then continue to calculate the stiffness and mass matrices as well as the load vectors.

By restricting global shape functions to an element of the mesh we obtain the local shape functions. They are computed on one of the following standard reference elements: intervals $[0,1]$ or $[-1,1]$, triangles with vertices $(0,0)$, $(1,0)$ and $(0,1)$ or squares $[0,1]^2$. If not stated otherwise the dimension is two and the functions are real-valued.

Different methods are implemented in LehrFEM, stored in the folder `/Lib/Elements` and described in the following. For some shape functions you can run the scripts in the folder `/Examples/PlotShap` to see the plots.

2.1 Input and Output Arguments

Throughout `shap` was used for 'shape functions' and these programs compute the values of the shape functions for certain finite elements at the Q quadrature points `x` of the standard reference elements. The details for the input may be found in table tab:shap_in 2.1.

`x`	Q-by-1 (1D) or Q-by-2 (2D) matrix specifying all quadrature points
`p`	nonnegative integer which specifies the highest polynomial degree (only needed for Legendre and *hp* polynomials, cf. ssect:shap_leg 2.3.4 and ssect:shap_hp 2.3.5)

Table 2.1: Argument list for `shap`- and `grad_shap`-functions tab:shap_in

The file name `grad_shap` stands for 'gradient of shape functions' respectively. If not specified otherwise both functions are for example called by

```
>> shap = shap_LFE(x);
>> grad_shap = grad_shap_LFE(x);
```

The output in the real-valued case can be found in table 2.2. In the vectorial case every two subsequent columns form one 2-dimensional shape function, hence the matrix contains twice as many columns. The number of shape functions per element depends on the local degrees of freedom (l).

`shap`	Q-by-l matrix which contains the computed values of the shape functions at the quadrature points `x`.
`grad_shap`	Q-by-$2l$ matrix which contains the values of the gradients of the shape functions at `x`. Here the $(2i-1)$-th column represents the x_1-derivative resp. the $2i$-th column the x_2-derivative of the i-th shape function.

Table 2.2: Output of `shap`- and `grad_shap`-functions (real-valued) tab:shap_out

Before we start to discuss the different basis functions, the 3D plotting routines are desribed.

2.2 Plotting Shape Functions

sect:shap_plot

The function `plot_Shap` stored in `/Lib/Plots` generates a 3D plot with lighting effect of `U` on the domain represented by `Vertex`. It is called by

```
>> H = plot_Shap(Vertex,U);
```

and returs the handle `H` to the figure.

As already mentioned, most implemented shape functions may be plotted using the plotting routines in the folder `/Examples/PlotShap`. Their functions name is of the form `main_Shap_*` where `*` is replaced by the respective finite element. They make use of the function `plot_Shap` and the respective shape function `shap_*`. The 3 linear shape functions on the reference triangle are e.g. plotted by

```
>> main_Shap_LFE;
```

Furthermore, the function `plot_Shap` is applied to linear resp. quadratic basis functions in `plot_Shap_LFE` and `plot_Shap_QFE` (both stored in `/Lib/Plots`) which generate 3D plots of the shape functions in `shap_LFE` resp. `shap_QFE` on the domain represented by `Vertex`. They are e.g. called by

```
>> H = plot_Shap_LFE(NMesh,LNumber,Vertex);
```

The input arguments are described in table 2.3.

`NMesh`	number which determines how fine the mesh is
`LNumber`	integer from 1 to 3 (resp. 1 to 6) that determines which `LFE` (resp. `QFE`) shape function to take
`Vertex`	matrix which determines the set of point which represent the domain of plotting

Table 2.3: Input for `plot_Shap`-routines `tab:plot_shap_in`

Figure 2.1 shows the output for `plot_Shap_LFE` and `plot_Shap_QFE` on the unit triangle for first linear resp. quadratic shape function , cf. 2.3.1. `NMesh` was set to 100, `LNumber` is 1.

Figure 2.1: Output of `plot_Shap_LFE` and `plot_Shap_QFE` `fig:plot_Shap`

In the above functions `affine_map` is used to generate a mapping of all the vertices in the struct `Mesh` (resp. `Coordinates` in 1D) by the mapping from the reference element to the element which is formed by the given `Vertices` in row-wise orientation. It is called by one of the following lines.

```
>> Coordinates = affine_map(Coordinates,Vertices);
>> Mesh = affine_map(Mesh,Vertices);
```

2.2.1 Pyramid Plots

For linear and quadratic finite elements it's also possible to plot the global shape functions by combination of the local shape functions. These routines are named `plot_Pyramid_*` and stored in the folder `/Examples/PlotShap`. They make use of the above `plot_Shap`-function and are e.g. called by

```
>> plot_Pyramid_LFE;
```

See figure 2.2 for the output of `plot_Pyramid_LFE` and `plot_Pyramid_QFE` and compare them to figure 2.1.

Figure 2.2: Output of `plot_Pyramid_LFE` and `plot_Pyramid_QFE` fig:plot_pyramid

2.3 Different Shape Functions

sect:shap

2.3.1 Lagrangian Finite Elements

of order 1, H^1-conforming

sssect:shap_LFE

The function `shap_LFE` is used to compute the values of the three shape functions and `grad_shap_LFE` for the gradients for triangular Lagrangian finite elements.

These shape functions may be plotted using `main_Shap_LFE` in `/Examples/PlotShap` or `plot_Shap_LFE` in `/Lib/Plots`. The pyramid is generated by `plot_Pyramid_LFE` in `/Examples/PlotShap`. See the left figures in 2.1 and 2.2.

of order 1, vector-valued

`shap_LFE2` is also of order 1 but vector-valued, hence the functions are the same as in `shap_LFE` (in one coordinate, the other one is 0). In the output every two columns belong together and form one 2-dimensional shape function. They can be plotted on the reference triangle using the following command from the folder `/Examples/PlotShap`

```
>> plot_Shap_LFE2;
```

of order 2, conforming

sssect:shap_QFE

`shap_QFE` and `grad_shap_QFE` compute the values of the shape functions resp. gradients of the functions for triangular Langrangian finite elements. In `shap_QFE` the first three columns are the shape functions supported on vertices and the last three columns the ones supported on edges.

These shape functions may be plotted using `main_Shap_QFE` in `/Examples/PlotShap` or `plog_Shap_QFE` in `/Lib/Plots`. The pyramid is generated by `plot_Pyramid_QFE` in `/Examples/PlotShap`. See figures on the right hand side in 2.1 and 2.2.

`shap_EP2` and `grad_shap_EP2` compute the values of the functions resp. gradients of the functions for triangular Langragian finite elements connected to edges. As mentioned above these functions are also contained in `shap_QFE` and grad_shap_QFE, but in a different order.

These shape functions of order 1 and 2 are e.g. used for the Stokes problem.

Discontinuous Linear Lagrangian Finite Element in 1D ..

shap_DGLFE_1D and grad_shap_DGLFE_1D compute the values of the shape functions resp. its gradients for x$\in [-1, 1]$.

.. and 2D

shap_DGLFE and grad_shap_DGLFE are actually the same functions as shap_LFE and grad_shap_LFE.

They are used in the discontinuous Galerkin method.

Linear Finite Elements (1D)

sssect:shap_P1_1D

shap_P1_1D computes the values of the two linear shape functions in one dimension where x are points in the interval $[0, 1]$. The first column shap(:,1) is 1-x and shap(:,2) is x.

grad_shap_P1_1D computes the gradients -1 and 1 respectively.

Bilinear Finite Elements

sssect:shap_BFE

shap_BFE computes the values of the four bilinear shape functions at x$\in [0, 1]^2$, grad_shap_BFE the partial derivatives. The bilinear shape functions may be plotted using main_Shap_BFE in /Examples/PlotShap.

Crouzeix-Raviart Finite Elements

shap_CR and grad_shap_CR compute the shape functions resp. its gradients for the Crouzeix-Raviart finite element at the quadrature points x of a triangle. Unlike the conforming shape functions the Crouzeix-Raviart finite elements are 0 at the midpoints of two edges and 1 at the opposite midpoint.

The Crouzeix-Raviart shape functions may be plotted using main_Shap_CR in /Examples/PlotShap.

shap_DGCR and grad_shap_DGCR do the same for the discontinuous case. The Crouzeix-Raviart finite elements are e.g. used for the discontinuous Galerkin method and the Stokes problem.

2.3.2 MINI Element

shap_MINI computes the values of four shape functions for the triangular MINI element, grad_shap_MINI its gradients. The first three columns are the linear elements, the fourth one is the element shape function which is 0 on the edges and 1 in the center, also referred to as 'bubble function'.

ssect:shap_W1F

2.3.3 Whitney 1-Forms

Similar to `shap_LFE2` the Whitney 1-forms `shap_W1F` are vector-valued and two columns together form one function. Whitney forms are finite elements for differential forms. The 1-forms are edge elements and $H(curl)$-conforming. See e.g. [?] (MO03) for more details. For plotting the shape functions on the reference triangle use

```
>> plot_Shap_W1F;
```

which can be fount in the `/Examples/PlotShap` folder.

ssect:shap_Leg

2.3.4 Legendre Polynomials up to degree `p`

`shap_Leg_1D` computes the values of the Legendre polynomials up to degree `p`, used as shape functions for the 1D hpDG discretizations. They are called by

```
>> shap = shap_Leg_1D(x,p);
```

with `x`$\in [-1,1]$ and `p`≥ 0. The `(n+1)`-th column `shap(:,n+1)` in the output is the Legendre polynomial of degree `n` at `x`.

`grad_shap_Leg_1D` computes the gradients analogously.

ssect:shap_hp

2.3.5 Hierarchical Shape Functions up to polynomial degree `p`

`shap_hp` computes the values and gradients of the hierarchical shape functions on the reference element up to polynomial degree `p` at the points `x`. It is called by

```
>> shap = shap_hp(x,p);
```

Vertex, edge and element shape functions are computed. If `i` is the number of the vertex/edge, then the associated polynomials of degree `p` and its gradients are called by

```
>> shap.vshap{i}.{p};
>> shap.vshap{i}.grad_shap{p};
>> shap.eshap{i}.shap{p};
>> shap.eshap{i}.grad_shap{p};
>> shap.cshap.shap{p};
>> shap.cshap.grad_shap{p};
```

where `vshap` stands for vertex shape function, `eshap` for edge and `cshap` for element shape function respectively.

This program is part of the hpFEM. They are called hierarchical shape functions because when enriching from order `p` to `p+1` the existing shape functions don't change, but new ones are added. See [?] (AI03) for more information.

The hierarchical shape functions may be plotted using `main_Shap_hp` in `/Examples/PlotShap`.

Chapter 3

Numerical Integration

chap:quad_rule

Numerical integration is needed for the computation of the load vector in sect:assem_load 5.2, the incorporation of the Neumann boundary conditions in sect:assem_neu 6.2 and in some cases – e.g. for the *hp*FEM – also for the computation of the element stiffness matrices (if the differential operator doesn't permit analytic integration) and the incorporation of the Dirichlet boundary conditions.

The quadrature rules in LehrFEM are stored in the folder `/Lib/QuadRules`. There are 1D and 2D quadrature rules implemented and listed below. For some basic formulas of numerical integration see for example AS64 [?], chapter 25.

3.1 Data Structure of Quadrature Rules

A Q-point quadrature rule `QuadRule` computes a quadrature rule on a standard reference element. The MATLAB struct contains the fields weights `w` and abscissae `x`. They are specified in table tab:quad_rule 3.1.

`w`	Q-by-1 matrix specifying the weights of the quadrature rule
`x`	Q-by-1 (1D) or Q-by-2 (2D) matrix specifying the abscissae of the quadrature rule

Table 3.1: Quadrature rule structure tab:quad_rule

In the following sections `QuadRule_1D` and `QuadRule_2D` are used instead of `QuadRule` to highlight their dimension.

The barycentric coordinates `xbar` of the quadrature points `x` may be recovered by

```
>> xbar = [Quadrule.x, 1-sum(QuadRule.x,2)];
```

3.2 1D Quadrature Rules

sect:quad_rule_1d

The 1D quadrature rules are used for the incorporation of the boundary conditions and generally for 1D problems. The 1D Gauss-Legendre quadrature rule `gauleg` and the 1D Legendre-Gauss-Lobatto quadrature rule `gaulob` are implemented in LehrFEM.

They are called by

```
>> QuadRule_1D = gauleg(a,b,n,tol);
>> QuadRule_1D = gaulob(a,b,n,tol);
```

and compute the respective `n`-point quadrature rules on the interval `[a,b]`. The prescribed tolerance `tol` determines the accuracy of the computed integration points. If no tolerance is prescribed the machine precision `eps` is used.

All orders of the quadrature rules `gauleg` are of order 2`n`−1, the ones of `gaulob` are of order 2`n`−3. The abscissas for quadrature order `n` are given by the roots of the Legendre polynomials $P_{\mathtt{n}}(x)$. In the Lobatte quadrature the two endpoints of the interval are included as well.

The 1D quadrature rules may be transformed to rules on squares resp. triangles by `TProd` resp. `TProd` and `Duffy`. See ssect:quad_trans 3.3.1 below.

3.3 2D Quadrature Rules

sect:quad_rule_2d

In two dimension two reference elements can be distinguished – the unit square $[0,1]^2$ and the triangle with the vertices $(0,0)$, $(1,0)$, and $(0,1)$. There are also two different ways to build quadrature rules – from 1D quadrature rules or from scratch. Both approaches are used in the LehrFEM and are examined in the following.

3.3.1 Transformed 1D Quadrature Rules

ssect:quad_trans

The quadrature formula for the unit square are Gaussian quadrature rules which are the tensorized version of 1-dimensional formulas. To this end the tensor product `TProd` is applied by

```
>> QuadRule_2D_square = TProd(QuadRule_1D);
```

where `QuadRule_1D` is a 1-dimensional quadrature rule of sect:quad_rule_1d 3.2, i.e. `gauleg` or `gaulob`. This type of integration is used for finite elements defined on sqares such as $[0,1]^2$, e.g. bilinear finite elements.

Furthermore, by the use of the Duffy transformation `Duffy` of the integration points and the weights one obtains a 2-dimensional quadrature rule for triangular elements:

```
>> QuadRule_2D_triangle = Duffy(TProd(QuadRule_1D));
```

The i-th integration point then has the coordinates $x_{i,1}$ and $x_{i,2}(1-x_{i,1})$, where $x_{i,1}$ and $x_{i,2}$ are the two coordinates of the non transformed point. Furtheremore the corresponding weights are transformed according to $w_i(1-x_{i,1})$.

3.3.2 2D Gaussian Quadrature Rules

ssect:quad_po

Several Gaussian quadrature rules on the above mentioned reference triangle are implemented. Their file names are of the form `PnOo`, which stands for '`n`-point quadrature rule of order `O`', e.g. `P4O3`.

These quadrature rules do not need any input since the number of points and the integration domain, i.e. the unit triangle are specified.

The quadrature rules are called by e.g.

```
>> QuadRule_2D_triangle = P4O3();
```

So far the following quadrature rules are implemented in LehrFEM: `P1O2`, `P3O2`, `P3O3`, `P4O3`, `P6O4`, `P7O4`, `P7O6`, `P10O4` and `P10O5`.

3.3.3 2D Newton-Cotes Quadrature Rules

The Newton-Coates quadrature rule for the reference triangle is implemented in `/Lib/QuadRules/private`. Here `ncc_triangle_rule` is called by

```
>> nc = ncc_triangle_rule(o,n);
```

where `o` is the order and `n` the number of points. Because the output of this function doesn't have the right format, the program `NCC` tranforms it. After all

```
>> QuadRule_2D_triangle = NCC(o);
```

provides the right data structure as specified in table tab:quad_rule 3.1 for `n`= 10.

Chapter 4

Local Computations

chap:local_comp

The interpretation of a partial differential equation in the weak sense yields the variational formulation of the boundary value problem. A linear variational problem is of the form

$$u \in V : \quad a(u, v) = f(v) \quad \forall v \in V \tag{4.1}$$

eq:int_eq

where V is the test space, a a (symmetric) bilinear form, f a linear form and u the solution. The terms a and f depend on the differential operator resp. the right hand side of the equation. Due to discretization, V is replaced by the discrete test space V_N, a discrete variational problem has to be solved. By choosing a basis $B_N = \left\{b_N^1, \ldots, b_N^N\right\}$ for V_N the integral equation (4.1) eq:int_eq is transformed to an algebraic equation

$$\mathtt{A}\mu = \mathtt{L} \tag{4.2}$$

with stiffness matrix $\mathtt{A} = (a(b_N^k, b_N^j))_{j,k=1}^N$, load vector $\mathtt{L} = (f(b_N^j))_{j=1}^N$ and coeffient vector μ from which the solution $\mathtt{u} = \sum_{k=1}^N \mu_k b_N^k$ may be recovered.

All the basis functions occuring in in the definition of `A` and `L` are composed of element shape functions. For reasons concerning computational time the stiffness matrix is not assembled entry by entry, which would need two loops over all basis functions. It is better to loop over all elements and compute the contribution to the stiffness matrix. To do so one has to consider all the shape functions b_K^l on the triangular element K. The corresponding local stiffness matrix is then given by $\mathtt{Aloc}_{i,j} = a(b_K^i, b_K^j)$ and the load vector by $\mathtt{Lloc}_i = f(b_K^i)$. Furthermore there the computation of the mass matrix given by the L^2 inner product $\mathtt{Mloc}_{i,j} = (b_K^i, b_K^j)$ is implemented The aim of this section is to introduce the necessary MATLAB functions which compute these matrices and vectors.

The local computations are then summed up to the global matrices `A` and `M` and the global load vector `L` by the assembly routines described in chapter 5. chap:assem

Since local computations vary depending on the equation they are used for there is no point in listing and describing all those MATLAB functions of the LehrFEM. Still, the central theme is treated in the following. For more information please read the well-documented code of the functions stored in `/Lib/Element`. The file names are abbreviations for

`STIMA_**_*`	element stiffness matrix for the operator `**` and finite elements `*`
`MASS_*`	element mass matrix for finite elements `*`
`LOAD_**_*`	element load vector for the operator `**` and finite elements `*`

Table 4.1: File names for element functions `tab:element`

4.1 Element Stiffness Matrices

`sect:stima`

4.1.1 Input Arguments

The main input arguments for the computations in 2D are listed in table 4.2. In the 1-dimensional case `Vertices` is obviously a 2-by-1 matrix.

`Vertices`	3-by-2 or 4-by-2 matrix specifying the vertices of the current element in a row wise orientation
`ElemInfo`	integer parameter which is used to specify additional element information
`QuadRule`	struct, which specifies the Gauss qaudrature that is used to do the integration (see 3.3, p. 32)
`EHandle`	function handle for the differential operator
`EParam`	variable length argument list for `EHandle`

Table 4.2: Input arguments for `STIMA` (2D) `tab:stima_in`

Besides the mesh and the operator, the shape functions resp. its gradients are needed for the computation of the local contributions. The functions `shap_*` and `grad_shap_*` are called within the program.

For some operators, e.g. the Laplacian, no quadrature rules, operator nor shape functions are required, cause they are already included in the program resp. the matrix entries are computed using barycentric coordinates.

The functions for the boundary terms require data of the `Edges` instead of `Vertices` etc.

4.1.2 Output

In all `STIMA`-functions the outputs are l-by-l matrices `Aloc` where l are the local degrees of freedom. The element in the k-th row and j-th column is the contribution $a(b_N^k, b_N^j)$ of the k-th and j-th shape functions on the current element.

The element stiffness matrices are assembled using the assembly routines `assemMat` in `/Lib/Assembly`, cf. section 5.1, p. 47.

4.1.3 Principles of Computation

First, an affine (linear) transformation of the finite element to a standard reference element is done, i.e. the square $[0,1]^2$ or the triangle with the vertices $(0,0)$, $(0,1)$ and $(1,0)$ in the 2-dimensional case. Then the matrix entries `Aloc(k,j)` are computed using the given quadrature rule `QuadRule` and the respective shape functions. In very easy cases occuring integrals can be computed analytically. Then the computation of the element stiffness matrix is done directly and neither a transformation nor quadrature rules are used.

If the bilinear form a is symmetric, then `Aloc` is a symmetric matrix and only the upper triangle needs to be computed.

4.1.4 Examples

Laplacian

The element matrix for the Laplacian using linear finite elements is computed by the routine

`>> Aloc = STIMA_Lapl_LFE(Vertices);`

The corresponding matlab code for computing the 3-by-3 matrix is

```
function Aloc = STIMA_Lapl_LFE(Vertices,varargin)
 % Preallocate memory

 Aloc = zeros(3,3);

 % Analytic computation of matrix entries using
 % barycentric coordinates

 a = norm(Vertices(3,:)-Vertices(2,:));
 b = norm(Vertices(3,:)-Vertices(1,:));
 c = norm(Vertices(2,:)-Vertices(1,:));
 s = (a+b+c)/2;
 r = sqrt((s-a)*(s-b)*(s-c)/s);
 cot_1 = cot(2*atan(r/(s-a)));
 cot_2 = cot(2*atan(r/(s-b)));
 cot_3 = cot(2*atan(r/(s-c)));

 Aloc(1,1) = 1/2*(cot_3+cot_2);
 Aloc(1,2) = 1/2*(-cot_3);
 Aloc(1,3) = 1/2*(-cot_2);
 Aloc(2,2) = 1/2*(cot_3+cot_1);
 Aloc(2,3) = 1/2*(-cot_1);
 Aloc(3,3) = 1/2*(cot_2+cot_1);

 % Update lower triangular part

 Aloc(2,1) = Aloc(1,2);
 Aloc(3,1) = Aloc(1,3);
 Aloc(3,2) = Aloc(2,3);

return
```

The element matrices for the Laplacian and Crouzeix-Raviart elements e.g. by

`>> Aloc = STIMA_Lapl_CR(Vertices);`

Heat Equation

The element matrices for the heat equation and bilinear finite elements are e.g. generated by

```
>> Aloc = STIMA_Heat_BFE(Vertices,ElemInfo,QuadRule, ...
EHandle,EParam);
```

This routine is for quadrilateral meshes, the specification of the quadrature rule could for example be

```
QuadRule = TProd(gauleg(0,1,2));
```

Helmholtz Equation

The following computes the entries of the element stiffness matrix for a discontinuous plane wave discretization of the Helmholtz equation on Neumann boundary edges:

```
>> Aloc = STIMA_Helm_Neu_Bnd_PWDG(Edge,Normal,Params,Data, ...
omega,QuadRule,varargin);
```

Here, `Params` and `Data` (contains the left and right hand side element data) are structs with the fields specified in tables 4.3 and 4.4.

`b`	scalar coefficient for a term containing jumps of the normal derivative in the numerical flux for the gradient
`nDofs`	total number of degrees of freedom on elements adjacent to current edge
`L2`	L^2 inner product matrix on the current edge

Table 4.3: Basic `Params` data structure tab:params

`Element`	integer specifying the neighbouring element
`ElemData`	structure contains the fields `NDofs` (number of degrees of freedom on the current element) and `Dir` (P-by-2 matrix containing the propagation directions of the plane wave basis functions in its rows)
`Vertices`	3-by-2 or 4-by-2 matrix specifying the vertices of the neighbouring element
`EdgeLoc`	integer specifying the local edge number on the neighbouring element
`Match`	integer specifying the relative orientation of the edge w.r.t. the orientation of the neighbouring element

Table 4.4: Basic `Data` structure tab:data

`Edge` is a 2-by-2 matrix whose rows contain the start and end node of the current edge, `Normal` is a 1-by-2 matrix which contains the interior unit normal vector w.r.t. the current edge `Edge`.

DG finite elements

ssec:dg

In the case of a DG discretization in addition to the volume terms one gets additional contributions that correspond to the discontinuities along the edges. The local bilinear form on an element T for the symmetric interior penalty method for two basis functions b_i and b_j is given by

$$a(b_i, b_j) = \int_T \nabla b_i \cdot \nabla b_j - \int_{\partial T} (\{\nabla b_i\} \cdot [b_j] + \{\nabla b_j\} \cdot [b_i]) + \int_{\partial T} a[b_i][b_j]. \quad (4.3)$$

eq:ipm

On the edges we define $\{\nabla b\} = (\nabla b^+ + \nabla b^-)/2$ and $[b_i] = b_i^+ \mathbf{n}^+ + b_i^- \mathbf{n}^-$. For more details see [].

The computation of the stiffness matrix is divided into five steps and in each of the steps a local matrix has to be computed.

For any element of the mesh the first integral gives rise to a l-by-l matrix, where l denotes the degrees of freedom on every element. For the Laplace operator in combination with Lagrangian finite elements of degree p this is implemented. The code for assembling the matrix, which is a part of the function STIMA_Lapl_Vol_PDG can be found below.

```
for j1 = 1:nDofs % loop over columns
  loc_1 = 2*(j1-1) + [1 2];
  for j2 = j1:nDofs % loop over lines
    loc_2 = 2*(j2-1) + [1 2];
    Aloc(j1,j2) = sum(QuadRule.w.*sum(grad_N(:,loc_1).*...
     (grad_N(:,loc_2)*TK),2)); % numerical integration
  end
end

% Fill in lower trinagular part (symmetry)

tri = triu(Aloc);
Aloc = tril(transpose(tri),-1)+tri;
```

Along every interior edge one gets a local $2l$-by-$2l$ matrix for both, the second and the third integral in (4.3). Following the example from above with the Laplacian as differential operator and Lagrangian finite elements this can be computed using the functions STIMA_Lapl_Inn_PDG and STIMA_InnPen_PDG.

Furthermore along boundary edges the second and third integral in (4.3) both yield a l-by-l matrix. For the example from before these can be computed using STIMA_Lapl_Bnd_PDG and STIMA_BndPen_PDG.

hp-finite elements

The local stiffness matrix for the Laplacian operator using *hp*-finite elements can be computed using

```
>> Aloc = STIMA_Lapl_hp(Vertices,ElemInfo,EDofs,EDir,CDofs,QuadRule,Shap);
```

The integers EDofs ans CDofs specify the degrees of freedom on the element and in the interior respectively. Details can be found in Table 4.5.

`EDofs`	1-by-3 matrix specifying the maximum polynomial degree of the edge shape functions on every edge
`EDir`	1-by-3 matrix specifying the orientation of the edges of the element
`CDofs`	integer specifying the maximum polynomial degree inside the element

Table 4.5: Additional input for `MASS_hp` tab:mass_hp

4.2 Element Mass Matrices

sect:mass

The main input argument for the computation of the element mass matrices are the `Vertices` of the current element, in most cases it is the only one. On the other hand, `QuadRule` is needed and shape functions are called within the function to compute the element map. See table 4.2 for details. Further arguments are defined when they appear. No operator is to be specified in a function handle, but e.g. weight functions which appear in the `MASS_Weight`-functions.

If not specified otherwise the functions are e.g. called by

```
>> Mloc = MASS_LFE(Vertices);
```

The computation of the element mass matrices `Mloc` is quite similar the one for the element stiffness matrices `Aloc` described in the previous section.

The element mass matrices `Mloc` are l-by-l matrices where l are the local degrees of freedom. Like the element stiffness matrices they are assembled to the mass matrix `M` by the `assemMat`-functions, p. 47.

4.2.1 Constant Finite Elements

To call `MASS_P0_1D` and `MASS_P0` only the coordinates of the vertices are needed:

```
>> Mloc = MASS_P0_1D(Vertices);
>> Mloc = MASS_P0(Vertices);
```

The weighted versions `MASS_Weight_P0_1D` and `MASS_Weight_P0` are called by

```
>> Mloc = MASS_Weight_P0_1D(Vertices,QuadRule,FHandle,FParam);
>> Mloc = MASS_Weight_P0(Vertices,ElemInfo,QuadRule, ...
FHandle,FParam);
```

4.2.2 Linear Finite Elements

.. in 1D

`MASS_P1_1D` computes the 2-by-2 element mass matrix using linear finite elemnts. The weighted version `MASS_Weight_P1_1D` uses `shap_P1_1D` and is called by

```
>> Mloc = MASS_Weight_P1_1D(Vertices,QuadRule,FHandle,FParam);
```

.. in 2D

Similarily, MASS_LFE and MASS_LFE2 compute the element mass matrix in 2D. Here, Mloc is a 3-by-3 and – in the vectorial case – a 6-by-6 matrix.

The function MASS_Weight_LFE computes the element mass matrix with a given weight. It makes use of the shape functions shap_LFE and is called by

```
>> Mloc = MASS_Weight_LFE(Vertices,ElemInfo,QuadRule, ...
FHandle,FParam);
```

where FHandle denotes the function handle to the weight function and FParam its variable length argument list.

4.2.3 Bilinear Finite Elements

The 4-by-4 element mass matrix using bilinear Lagrangian elements is computed by

```
>> Mloc = MASS_BFE(Vertices,ElemInfo,QuadRule);
```

shap_BFE and grad_shap_BFE are used for the computation of the element mapping.

4.2.4 Crouzeix-Raviart Finite Elements

MASS_CR computes the 3-by-3 element mass matrix. MASS_Vol_DGCR computes the element mass matrix using discontinuous Crouzeix-Raviart finite elements.

4.2.5 Quadratic Finite Elements

The 6-by-6 element mass matrix is given by MASS_QFE.

4.2.6 Whitney 1-Forms

MASS_W1F computes the element mass matrix with weight MU_Handle for edge elements. It is specified in table tab:mu_handle 4.6.

MU_Handle	handle to a functions expecting a matrix whose rows represent position arguments. Return value must be a vector (variable arguments MU_Param will be passed to this function).

Table 4.6: Weight MU_Handle tab:mu_handle

The function is called by

```
>> Mloc = MASS_W1F(Vertices,ElemInfo,MU_Handle,QuadRule,MU_Param);
```

4.2.7 *hp* Finite Elements

.. in 1D

The (p+1)-by-(p+1) element mass matrix MASS_hpDG_1D in the 1-dimensional case is computed by

```
>> Mloc = MASS_hpDG_1D(Coordinates,p,QuadRule,Shap);
```

MASS_Vol_hpDG_1D is called by

```
>> Mloc = MASS_Vol_hpDG_1D(Vertices,p,QuadRule,Shap);
```

.. in 2D

More input arguments are needed for the 2-dimensional MASS_hp, e.g. the different polynomial degrees of the shape functions as well as the orientation of the edges. These new input arguments are listed in Table tab:mass_hp 4.5.

Furthermore, the pre-computed shape functions Shap (cf. ssect:shap_hp 2.3.5, p. 28) and quadrature rules QuadRule (cf. chap:quad_rule 3, p. 31ff) are required. The mass matrix is called by

```
>> Mloc = MASS_hp(Vertices,ElemInfo,EDofs,EDir,CDofs, ...
QuadRule,Shap,varargin);
```

and e.g. used for the routine main_hp in the folder /Examples/DiffConv_exp. Actually the input arguments ElemInfo and varargin are not needed for MASS_hp but for assemMat_hp.

The mass matrix Mloc for the *hp*FEM is of dimension $(3+\sum \texttt{EDof}+\texttt{CDof})$-by-$(3+\sum \texttt{EDof}+\texttt{CDof})$.

4.2.8 Mixed Finite Elements

MASS_P1P0 computes the element mass matrix using linear and constant Lagrangian finite elements. It is called by

```
>> Mloc = MASS_P1P0(Vertices);
```

The element mass matrix with weight is computed by MASS_Weight_P1P0.

4.3 Element Load Vectors

sect:load

For the standard continuous galerkin discretization, i.e. continuous solution in the node points there is no element load vector computed. The computation of the whole vector is done in the assembly file as described in Section sect:assem_load 5.2.

For the discontinuous galerkin method additional boundary terms occur on the dirichlet boundary. The test space is in this case not only contains functions that vanish on the this part of the boundary. For example for the Laplacian operator and lagrangian finite elements of order p the computations of the volume part are done in the function `@LOAD_Vol_PDG` and the boundary contributions can be obtained by using `@LOAD_Lapl_Bnd_PDG`.

Note that generally – i.e. for the easy cases described in section 5.2, p. 52 – the computation of the element load vectors is even done *within* the `assemLoad`-functions. The following element load vectors are just needed for the discontinuous Galerkin method.

4.3.1 Boundary Contributions

The boundary contributions for the discontinuous Galerkin method come from the Dirichlet boundary data. The test functions are in general not zero on the boundary, therefore an additional integral over all boundary Dirichlet edges occurs. For the DG method implemented in LehrFEM only pure Dirichlet boundary data is allowed.

Input Arguments

sssect:load_bnd_in

The input arguments `Edge`, `Normal` and `BdFlag` are described in table 4.7.

`Edge`	2-by-2 matrix whose rows contain the start and end node of the current edge
`Normal`	1-by-2 matrix which contains the interior unit normal with respect to the current edge `Edge`
`BdFlag`	integer which denotes the boundary flag of the current edge

Table 4.7: Mesh input arguments for the computation of the element load vector tab:load_in_mesh

`BdFlag` is only needed for some functions handles. On the other hand some parametrization of the edge may be supplied by `Params` (see documentation for more details). The struct `Data` contains the left and right hand side element data, see table 4.8.

`Element`	integer specifying the neighbouring element
`ElemData`	structure contains the fields `nDofs` (number of degrees of freedom on the current element) and `Dir` (P-by-2 matrix containing the propagation directions of the plane wave basis functions in its rows)
`Vertices`	3-by-2 or 4-by-2 matrix specifying the vertices of the neighbouring element
`EdgeLoc`	integer specifying the local edge number on the neighbouring element
`Match`	integer specifying the relative orientation of the edge w.r.t. the orientation of the neighbouring element

Table 4.8: Required `Data` structure tab:load_in_data

The right hand side is given as function handle, e.g. `FHandle`. It should take at least the argument `x`. Furthermore, a quadrature rule `QuadRule` is needed for the calculation of the integrals. Some additional constants and parameters may be required depending on the right hand side and equation they are used for.

Output

The outputs are l-by-1 vectors `Lloc_bnd` where l again are the local degrees of freedom on the current element. The j-th element corresponds to the j-th contribution $f(b_N^j)$ of the current element.

The local boundary terms are assembled by `assemLoad_Bnd`-functions stored in `/Lib/Assembly`.

Examples

- `LOAD_Bnd_DGLFE` computes the entries of the element load vector for the boundary load data on discontinuous linear finite elements. It is called by

  ```
  >> Lloc_bnd = LOAD_Bnd_DGLFE(Edge,Normal,BdFlag,Data, ...
  QuadRule,s,SHandle,FHandle,varargin);
  ```

  where the integer `s` specifies wheter the diffusive fluxes are discretized in a symmetric or anti-symmetric way ($+1$ anti-symmetric, -1 symmetric), `SHandle` is a function pointer to the edge weight function and `FHandle` a function pointer to the load data.

- `LOAD_Dir_Bnd_PWDG` computes the entries of the element load vector corresponding to Dirichlet boundary conditions for discontinuous plane waves. It is called by

  ```
  >> Lloc_bnd = LOAD_Dir_Bnd_PWDG(Edge,Normal,Params,Data, ...
  QuadRule,omega,GHandle,GParam);
  ```

  `GHandle` is a function handle for the impedence boundary conditions, `omega` is the wave number of the Helholtz equation.

- `LOAD_Lapl_Bnd_PDG` computes the entries of the element load vector for the boundary load data `FHandle` using the shape functions given by the function handle `Shap` and `grad_Shap` by

  ```
  >> Lloc_bnd = LOAD_Lapl_Bnd_PDG(Edge,Normal,BdFlag,Data, ...
  QuadRule,Shap,grad_Shap,SHandle,FHandle,FParam);
  ```

4.3.2 Volume Contributions

Input Arguments

In contrast to the boundary contributions `Vertices` and `ElemData` are also needed for the computation, but not stored in the struct `Data` but stand-alone. As usual, `Vertices` is 3-by-2 or 4-by-2 matrix. The struct `ElemData` is described in table tab:elem_data 4.9.

`nDof`	number of degrees of freedom on the current element
`Dir`	P-by-2 matrix containing the propagation directions of the plane wave basis functions in its rows

Table 4.9: Basic `ElemData` data structure tab:elem_data

The rest of the input arguments are pretty much the same as mentioned in sssect:load_bnd_in 4.3.1.

Output

The outputs are also l-by-1 vectors `Lloc_vol`. The j-th element corresponds to the j-th contribution $f(b_N^j)$ on the current element.

The local volume terms are assembled by`assemLoad_Vol`-functions stored in `/Lib/Assembly`.

Examples

- `LOAD_EnergyProj_Vol_PWDG` computes the volume contributions to the element load vector for discontinuous plane waves with right hand side given by an energy-norm scalar product. It is called by

  ```
  >> Lloc_vol = LOAD_EnergyProj_Vol_PWDG(Vertices,ElemData, ...
  QuadRule,omega,UHandle,DUHandle,varargin);
  ```

  `UHandle` and `DUHandle` are function handles for the load data and its gradient, `omega` is the wave number of the plane waves. The functions calls `shap_BFE` and `grad_shap_BFE` as well as the quadrature rules `P3O3` and `P7O6`.

- `LOAD_Vol_DGLFE` computes the entries of the element load vector for the volume load data for discontinuous linear elements. The function `shap_DGLFE` is called for the calculaction of the values of the shape functions. The 3-by-1 vector `Lloc_vol` is created by

  ```
  >> Lloc_vol = LOAD_Vol_DGLFE(Vertices,ElemFlag,QuadRule, ...
  FHandle,FParam);
  ```

Chapter 5

Assembling

chap:assem

On the following pages the main assembling routines for the assembly of the global matrices and load vectors are summerized. Their job is to assembly the local element matrices and load vectors for various operators resp. right hand side given by function handles `EHandle` and `FHandle`. In finite elements it is common to consider the matrices and load vectors without including boundary conditions first. The boundary conditions are incorporated later. The general input arguments (e.g. the required `Mesh` data structure) and central ideas of the computations are condensed at the beginning of both sections.

The following finite elements are considered in this manual:

- constant finite elements (1D and 2D)
- linear finite elements (1D and 2D, also vector-valued)
- bilinear finite elements (2D)
- Crouzeix-Raviart finite elements (2D)
- quadratic finite elements (2D)
- Whitney 1-forms (2D, vector-valued)
- *hp* finite elements (1D and 2D)

5.1 Assembling the Global Matrix

sect:assem_mat

The assembling of the stiffness and mass matrices in LehrFEM is element-based. The local stiffness matrix of the element K `Aloc` is an l-by-l matrix (l = local degrees of freedom). By definition $\texttt{Aloc}_{ij} = a(b_K^i, b_K^j)$, where b_K^i denotes the local shape functions. The shape function b_K^i is 1 on the local node i and 0 in all the other nodes of the corresponding element, therefore it equals the global basis function b^I restricted to the element K, where I denotes the global label of the local node i. The bilinear form is defined by an integral over the computational domain which allows to compute $a(b^I, b^J)$ as sum of integrals over the elements. The value $a(b_K^i, b_K^j)$ is therefore one contributing summand to $a(b^I, b^J)$, where

I and J are the global labels corresponding to the local nodes i and j. The same computation strategy can be used to compute the mass matrix.

The struct `Mesh` contains information about its `Coordinates`, `Elements` and eventually contains additional element information in `ElemFlag` (e.g. weights). tab:assem_mesh The necessary details of this data structure are summed up in table 5.1.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 or N-by-4 matrix connecting vertices into elements
`ElemFlag`	N-by-1 matrix specifying additional element information

Table 5.1: Mesh data structure for the assembling process (2D) tab:assem_mesh

The code of the function `assemMat_LFE` for linear finite elements can be found below.

```
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); % contains line number
    J(loc) = set_Cols(idx,3); % contains column number
    A(loc) = Aloc(:); % contains matrix entries
    loc = loc+9;
```

```
40
  end

  % Assign output arguments

45  if(nargout > 1)
    varargout{1} = I;
    varargout{2} = J;
    varargout{3} = A;
  else
50  varargout{1} = sparse(I,J,A);
  end

return
```

The input argument `varargin` is directly passed on to the function handle `EHandle` for the computation of the local contributions. Depending on the operator this input is needed or not. For example for the Laplacian with linear finite elements it is sufficient to call

```
>> A = assemMat_LFE(Mesh,@STIMA_Lapl_LFE);
```

In general the possibilities to call the function can be found below.

```
>> A = assemMat_LFE(Mesh,EHandle);
>> A = assemMat_LFE(Mesh,EHandle,EParam);
>> [I,J,A] = assemMat_LFE(Mesh,EHandle);
```

Here `A`, `I` and `J` are E-by-1 matrices where $E = l^2 \cdot N$. In the first two examples the matrix `A` is returned in a sparse representation, in the latter case in an array representation. The programs `set_Rows` and `set_Cols` (lines 36,37) generate the row resp. column index set `I` resp. `J` used for the transformation of an element matrix into an array.

The main steps in the computation are:

- prealocating memory and defining constants (lines 4-22)
- loop over all elements in `Mesh` (lines 23-41)
  - computing element contributions `Aloc` using `EHandle` (mostly seperate files for various operators in `/Lib/Elements`, see section 4.1, p. 36) (line 32)
  - adding these contributions to the global matrix `A` (lines 36-38)

For the computation of the local contributions see chapter 4, p. 35ff.

5.1.1 Constant Finite Elements

.. in 1D

In this simplest case the output is a diagonal matrix created by `assemMat_P0_1D`.

.. in 2D

For each element there is only one contribution. They are assembled in `assemMat_P0`.

5.1.2 Linear Finite Elements

.. in 1D

assemMat_P1_1D assembles the linear element contributions in 1D. The band matrix is updated in each run of the loop. It is called by

```
>> A = assemMat_P1_1D(Coordinates,EHandle,EParam);
```

.. in 2D

On a linear finite element 9 contributions have to be computed, e.g. by the operator /Lib/Elements/STIMA_Lapl_LFE in the Laplacian case as discussed above. Their local stiffness matrices are calculated by using barycentric coordinates.

In assemMat_LFE the element contributions are assembled. Additional information in Mesh.ElemFlag may be taken into account.

In the vector-valued case there are 6 shape functions per element, hence 36 nodal contributions are assembled in assemMat_LFE2.

5.1.3 Bilinear Finite Elements

assemMat_BFE assembles the element contributions by extracting the elements and its vertices from the mesh, computing the local matrices for the given operator EHandle and merging them together. Each element consists of four vertices (the reference element is $[0, 1]^2$) and four bilinear shape functions are hereon defined, hence 16 contributions per element are calculated.

5.1.4 Crouzeix-Raviart Finite Elements

assemMat_CR computes 9 contributions per triangular element and adds them to the global stiffness matrix. Since the Crouzeix-Raviart elements are connected to the midpoints of the edges, the additional information Mesh.Vert2Edge is necessary in order to assign the edge number to two connected vertices of the triangle.

5.1.5 Quadratic Finite Elements

There are 6 quadratic shape functions on one triangular element (3 connected to vertices and 3 to edges), hence there are 36 local element contributions which are assembled in assemMat_QFE.

5.1.6 Whitney 1-Forms

Whitney 1-forms are 3 vector-valued shape functions, thus 9 contributions are computed locally. Additionally the edge orientations are determined to scale the element matrices.

The program assemMat_W1F assembles then the global element matrix. Additional element information may be stored in the field Mesh.ElemFlag.

Similarily, `assemMat_WReg_W1F` does the assembly for the weak regularization W1F finite element solver. There `EHandle` is called with the 2D quadrature rule `P706` (7 point Gauss quadrature rule of order 6, cf. ssect:quad_po 33).

5.1.7 *hp* Finite Elements

ssec:asseMathp

`assemMat_hp` assembles the *hp*FEM contributions. It is e.g. called by

```
>> A = assemMat_hp(Mesh,Elem2Dof,EHandle,EParam);
```

elem2dof

where the struct `Elem2Dof` describes the element to dof (degrees of freedom) mapping obtained from the routine `build_DofMaps`. The degrees of freedom are placed on elements according to the distance to the corner points, where singularities are expected. The distance of an element T to the corner point c denoted by $d(T, c)$ is measured by the minimal number edges needed to get a path from c to any of the vertices of T. In this implementation the polynomial degree for the element T is given by

$$p_T = \min_{c \in C}(\max(3, d(T, c))), \tag{5.1}$$

where C denotes the set of all corner nodes. The degrees of freedom on the edges are computed as the minimum of the degrees of freedom on the neighboring elements.

The data strut `Elem2Dof` stores information about the local polynomial orders and the local edge orientations and contains the fields `Elem2Dof`, `EDofs`, `CDofs`, `tot_EDofs` and `tot_CDofs` which are explained in detail in Table tab:elem2dof 5.2.

`EDofs` consists of 3 cells, which are related to the three local edges of the element; cell number $i = 1, \dots, 3$ has the fields

- `Dofs`: N cells, which specify the labels of the degrees of freedom placed on the local edge i of every element
- `nDofs`: N-by-1 matrix containing the number of degrees of freedom for the local edge i of every element
- `Dir`: N-by-1 matrix with boolean entries determining the orientation of the local edges for every element

`CDofs` consists of the fields

- `Dofs`: N cells, which specify the labels of the degrees of freedom placed on each element
- `nDofs`: N-by-1 matrix containing the number of degrees of freedom for each element

`tot_EDofs` integer number specifying the total number of degrees of freedom placed on the edges

`tot_CDofs` integer number specifying the total number of degrees of freedom placed on the elements

Table 5.2: Data structure for storing the degrees of freedom

tab:elem2dof

`EParam` e.g. contains informatiom about the quadrature rule and shape functions which is needed for the computation of the computation of the element stiffness/mass matrices.

ssec:ddg

5.1.8 DG finite elements

As already mentioned on page 39 the computation of the stiffness matrix is divided into five steps. Firstly the volume contributions are assembled using the `assemMat_Vol_DG` or the `assemMat_Vol_PDG` function. Along inner edges the local matrices coming from the additional boundary integrals are assembled using `assemMat_Inn_DG` or `assemMat_Inn_PDG`. The penalty term for occuring discontinuities can be assembled likewise. The last assembly routine is either `assemMat_Bnd_DG` or `assemMat_Bnd_PDG`. These are used for incorporating the contributions of the boundary integrals and the penalty terms along boundary edges.

5.1.9 Mixed Finite Elements

Linear and Constant Finite Elements in 1D

`assemMat_P1P0_1D` assembles constant and linear element contributions in 1D. The output is a band matrix with bandwidth 2.

.. and in 2D

In `assemMat_P1P0` 3 element contributions are assembled.

sect:assem_load

5.2 Assembling the Load Vector

The right-hand side is treated in a similar manner, i.e. the assembling of the load vector is also element-based. The input arguments are `Mesh`, `QuadRule` and `FHandle`. The struct `Mesh` must contain the fields `Coordinates`, `Elements` and `ElemFlag` which are described in the table 5.1, p. 48. The Gauss quadrature for the integration is specified in `QuadRule`, where the field `w` stands for 'weights' and `x` for 'abscissae', cf. table 3.1, p. 31. `FHandle` is a function handle for the right hand side of the differential equation.

The `assemLoad`-functions are stored in `/Lib/Assembly` and are e.g. called by

```
>> L = assemLoad_LFE(Mesh,QuadRule,FHandle);
>> L = assemLoad_LFE(Mesh,QuadRule,FHandle,FParam);
```

The code for the linear finite elements can be found below.

```
function L = assemLoad_LFE(Mesh,QuadRule,FHandle,varargin)
%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

% Initialize constants
```

```
9  nPts = size(QuadRule.w,1);
10 nCoordinates = size(Mesh.Coordinates,1);
11 nElements = size(Mesh.Elements,1);
12 
13 % Preallocate memory
14 
15 L = zeros(nCoordinates,1);
16 
17 % Precompute shape functions
18 
19 N = shap_LFE(QuadRule.x);
20 
21 % Assemble element contributions
22 
23 for i = 1:nElements
24 
25   % Extract vertices
26 
27   vidx = Mesh.Elements(i,:);
28 
29   % Compute element mapping
30 
31   bK = Mesh.Coordinates(vidx(1),:);
32   BK = [Mesh.Coordinates(vidx(2),:)-...
33      bK; Mesh.Coordinates(vidx(3),:)-bK];
34   det_BK = abs(det(BK));
35 
36   x = QuadRule.x*BK + ones(nPts,1)*bK;
37 
38   % Compute load data
39 
40   FVal = FHandle(x,Mesh.ElemFlag(i),varargin{:});
41 
42   % Add contributions to global load vector
43 
44   L(vidx(1))=L(vidx(1))+sum(QuadRule.w.*FVal.*N(:,1))*det_BK;
45   L(vidx(2))=L(vidx(2))+sum(QuadRule.w.*FVal.*N(:,2))*det_BK;
46   L(vidx(3))=L(vidx(3))+sum(QuadRule.w.*FVal.*N(:,3))*det_BK;
47 
48 end
49 
50 return
```

The load vector `L` is provided as M-by-1 matrix, where the i-th entry implies the contribution of the i-th finite element.

The computation follows the steps below:

- initialization of constants and allocation of memory (lines 9-15)
- computation of the shape functions in the quadrature points on the reference triangle (line 19)
- loop over all elements (lines 23-48)

- compute element mapping and determine the local degrees of freedom (lines 27-36)
- compute right hand side function value in transformed quadrature points (line 40)
- add local load contribution to the global vector; integration over the reference triangle (lines 44-46)

Generally the corresponding shape function values are precomputed in the program at the given quadrature points `QuadRule.x`, i.e. `shap_LFE` is called in this case. The respective shape functions may be found in `/Lib/Elements`, cf. section 2.3, p. 26.

In contrast to the assembling of the matrices, the element transformations (to the standard reference element) are computed within the `assemLoad`-files. The load data for the `i`-th element is then computed at the standard points `x` by

```
>> FVal = FHandle(x,Mesh.ElemFlag(i),FParam);
```

Usually the integrals can't be evaluated exactly and therefore have to be approximated by a quadrature rule. Different implemented quadrature rules may be found in `/Lib/QuadRules`, cf. chapter 3, p. 31.

5.2.1 Constant Finite Elements

`assemLoad_P0` assembles the constant finite element contributions. In this case there is obviously no need for the computation of the shape functions.

5.2.2 Linear Finite Elements

.. in 1D

In `assemLoad_P1_1D` the function `shap_P1_1D` is called to compute the values of the shape functions at the quadrature points. The evaluation of the transformation matrix is exceptionally easy. After computing the element load data with the given right-hand side `FHandle` they are added up.

.. in 2D

For the computation of the values of the linear shape functions at the quadrature points `shap_LFE` is used. The transformation and assembly is done in `assemLoad_LFE` as discussed above.

The vector-valued function `assemLoad_LFE2` works analogously using the shape functions in `shap_LFE2`.

5.2.3 Bilinear Finite Elements

The program `assemLoad_BFE` assembles the bilinear finite element contributions. For each element, first the vertex coordinates are extracted and renumbered and then the 4 shape function `shap_BFE` and its gradient `grad_shap_BFE` are evaluated at the quadrature points `QuadRule.x`. Finally the transformation to the reference element $[0,1]^2$ and the local load data `FVal` are computed and the contributions are added to the global load vector.

5.2.4 Crouzeix-Raviart Finite Elements

`assemLoad_CR` does the assemly for the Crouzeix-Raviart finite elements. For every element the values of the 3 shape functions in `shap_CR` are precomputed. Then vertices and connecting edges are extracted and renumbered and the element mapping is computed. The local load data is evaluated and added to the global load vector.

5.2.5 Quadratic Finite Elements

In order to assemble the contributions of the quadratic finite elements in `assemLoad_QFE`, the struct `Mesh` must additionally contain the fields `Edges` and `Vert2Edge` cause 3 out of the 6 shape functions are edge supported. The function `shap_QFE` is used to compute the values of the shape functions, cf. sssect:shap_QFE 2.3.1.

5.2.6 Whitney 1-Forms

In `assemLoad_W1F` the shape functions `shap_W1F` are evaluated at the quadrature points. The vertices and edges are extracted from the mesh and the transformation is computed. Because the Whitney 1-forms are connected to edges, the fields `Vert2Edge` and `Edges` of `Mesh` are also required.

5.2.7 DG finite elements

As mentioned in Section sect:load 4.3 for DG finite elements the computation of the load vector is divided into two steps. Firstly the volume contributions, which are assembled using the `assemLoad_Vol_DG` or the `assemLoad_Vol_PDG` routine depending on the used finite elements. In the second step the boundary contributions are assembled. The boundary contributions to the load vector come from the Dirichlet boundary data.

5.2.8 *hp* Finite Elements

.. in 1D

The 1D assembly `assemLoad_hpDG_1D` of the *hp*FEM contributions is called by

```
>> L = assemLoad_hpDG_1D(Coordinates,p,QuadRule,Shap, ...
FHandle,FParam);
```

The global load vector consists of $\texttt{sum(p)} + \sharp$ Coordinates -1 elements. The transformation depends only on `h` and hence is easy to compute. The local load data is calculated and the global load vector then computed using the given shape function `Shap` (e.g. Legendre polynomials, cf. ssect:shap_Leg 2.3.4, p. ssect:shap_Leg 28) and the quadrature rule `QuadRule`.

.. in 2D

The function `assemLoad_hp` assembles the global load vector for the *hp*FEM contributions and is called by

```
>> L = assemLoad_hpDG(Mesh,Elem2Dof,QuadRule,Shap,FHandle,FParam);
```

As usual, in the beginning the vertices of the local element are extracted, and in this case also the polynomial degree and edge orientations stored in `Elem2Dof`. The element map, function values and the element load vectors for vertex/edge/element shape functions are computed and added to the global load vector.

Chapter 6

Boundary Conditions

chap:bound

The next step after the discretization and assembling of the stiffness matrix and load vector is the incorporation of the boundary data. There are two different types of boundary conditions, Dirichlet and Neumann boundary conditions. They both affect the values at the nodes the dirichlet conditions explicitely reduces the dimension of the linear system that has to be solved.

In the the two following sections, there is a short introduction about the data required and the main computational steps. Afterwards the MATLAB programs for various finite elements of LehrFEM are listed and discussed.

6.1 Dirichlet Boundary Conditions

sect:assem_dir

For the incorporation of the Dirichlet boundary conditions in LehrFEM, the functions `assemDir` in `/Lib/Assembly` are used. They specify the values a solution needs to take on the boundary of the domain, hence reduce the degrees of freedom.

Prescribing Dirichlet data means defining the value of the solution in some of the nodes. Therefore the dimension of the resulting linear system can be reduced by the number of nodes with prescribed value. If the value $\mathtt{U}_i$ is known do

1. adjust the right hand side by computing $\mathtt{L} - \mathtt{A}_{ji}\mathtt{U}_i$
2. delete the i-th line

Besides the usual data stored in `Mesh` – like `Coordinates` and `Edges` – and the boundary data function `FHandle`, it must be obvious which edges belong to the boundary. This additional information is stored in the field `BdFlags` of the struct `Mesh`. The boundary condition is then only enforced at the vertices of the edges whose boundary flag is equal to the integers `BdFlag` specified in the input. Be aware of the difference between `BdFlags` and `BdFlag` in the following. In the MATLAB code both are called `BdFlags`, but for the sake of clarity the manual differs from it.

For example it is possible to put the flag of each edge, where a Neumann boundary condition should hold to -1 and the flags of those with Dirichlet data to −2. However the choice is totally up to the user and there is no a-priori connection between the flags and the boundary conditions.

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Edges`	P-by-2 matrix specifying all edges of the mesh
`BdFlags`	P-by-1 matrix specifying the boundary type of each boundary edge in the mesh. If edge number `i` belongs to the boundary, then `BdFlags(i)` must be smaller than zero. Boundary flags can be used to parametrize parts of the boundary.

Table 6.1: Necessary mesh structure for boundary conditions (2D) tab:bound_mesh

If not stated otherwise the assembly programs for the Dirichlet boundary conditions are e.g. called by

```
>> [U,FreeDofs] = assemDir_LFE(Mesh,BdFlag,FHandle);
>> [U,FreeDofs] = assemDir_LFE(Mesh,BdFlag,FHandle,FParam);
```

where `FParam` handles the variable argument list for the Dirichlet boundary conditions with the data given by the function handle `FHandle`. `BdFlag` specifies the boundary flag(s) associated to the dirichlet boundary with the function `FHandle`.

The code for linear finite elements can be found below.

```
function [U,FreeDofs]=assemDir_LFE(Mesh,BdFlags,FHandle,varargin)
%   Copyright 2005-2005 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

% 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
30
   % Compute set of free dofs

   FreeDofs = setdiff(1:nCoordinates,tmp);

35 return
```

The main steps of the routine above are:

- initialization of the constants (lines 7-11)
- loop over all Dirichlet boundary flags specified in `BdFlags` (lines 13-29)
  - determine Dirichlet nodes (lines 15-19)
  - set the value of the finite element solution `U` in the dirichlet nodes (line 23)
- compute the nodes that are not part of the Dirichlet boundary

The Dirichlet boundary conditions are then incorporated in the finite element solution `U` of the boundary terms and `FreeDofs`:

`U`	sparse M-by-1 matrix which contains the coefficient vector of the finite element solution with incorporated Dirichlet boundary conditions. The entries are 0 on non-boundary terms.
`FreeDofs`	Q-by-1 matrix specifying the degrees of freedom with no prescribed Dirichlet boundary data ($Q = M - \sharp$ Dirichlet nodes)

Table 6.2: Output of the `assemDir`-functions tab:bound_out

All `assemDir`-functions follow more or less the mentioned structure (for all Dirichlet `BdFlag`):

For edge supported elements like Crouzeix-Raviart, quadratic finite elements and Whitney 1-forms additionally the `Midpoints` of the Dirichlet edges `DEdges` need to be computed.

6.1.1 Linear Finite Elements

.. in 1D

In the 1-dimensional case the input already contains the information about the 1 to 2 boundary point(s) `DNodes` which shortens the function `assemDir_P1_1D` a little. It is called by

```
>> [U,FreeDofs] = assemDir_P1_1D(Coordinates,DNodes,FHandle, ...
FParam);
```

.. in 2D

The function `assemDir_LFE` handles the 2-dimensional case and `assemDir_LFE2` the vector-valued linear elements. The code is quite similar and the output in the vector-valued case are $2M$- resp. $2N$-vectors, where the second coordinates are attached after all first coordinates.

6.1.2 Bilinear Finite Elements

The Dirichlet boundary conditions are incorporated in `assemDir_BFE` as described above.

6.1.3 Crouzeix-Raviart Finite Elements

Since the Crouzeix-Raviart elements are connected to midpoints of edges, they are additionaly computed and stored as `MidPoints` in `assemDir_CR`. Them `U(DEdges)` is the boundary function evaluated at these midpoints.

6.1.4 Quadratic Finite Elements

As already mentioned in 2.3.1, the quadratic finite elements are connected to vertices and edges. Hence the Dirichlet boundary conditions `assemDir_QFE` are computed on both, `DNodes` and `Midpoints`, as well as the degrees of freedom. The nodes' contributions are stored as usual and the edges' contributions are attached afterwards (with position $+M$).

6.1.5 Whitney 1-Forms

The midpoints for the elements are computed using `QuadRule_1D` (e.g. the Gauss-Legendre quadrature rule `gauleg`, p. 32), which is an additional argument. Hence the according function `assemDir_W1F` is called by

```
>> [U,FreeDofs] = assemDir_W1F(Mesh,BdFlag,FHandle, ...
QuadRule_1D,FParam);
```

The transformation formula is used for the computation of `U(DEdges)`.

6.1.6 *hp* Finite Elements

.. in 1D

The function `assemDir_hpDG_1D` is called by

```
>> Aloc = assemDir_hpDG_1D(Coordinates,p,Ghandle,shap, ...
grad_shap,s,alpha,GParam);
```

ssect:hp_dir **.. in 2D**

The *hp*FEM works a little bit different, the additional inputs that are needed for `assemDir` are the `Mesh` fields specified in table 6.3, the struct `Elem2Dof` which describes the element to dof (= degrees of freedom) mapping, a quadrature rule `QuadRule` (cf. 3.1, p. 31) and the values of the shape functions `Shap` at the quadrature points `QuadRule.x` (e.g. the hierarchical shape functions, cf. 2.3.5, p. 28).

The information stored in `EdgeLoc` is needed for the struct `Elem2Dof`. It contains the fields `tot_EDofs` and `tot_CDofs` which specify the total amount of degrees of freedom for edges resp. elements. The degrees of freedom depend on the chosen polynomial degree `p`. `Elem2Dof` is generated by the function

`Edge2Elem`	P-by-2 matrix connecting edges to their neighbouring elements. The first column specifies the left hand side element where the second column specifies the right hand side element.
`EdgeLoc`	P-by-2 matrix connecting egdes to local edges of elements. The first column specifies the local number for the left hand side element, the second column the local number for the right hand side element.

Table 6.3: Additional mesh data structure tab:dir_hp

`build_DofMaps` (stored in `/Lib/Assembly`) in the following way

```
>> Elem2Dof = build_DofMaps(Mesh,EDofs,CDofs);
```

The Dirichlet boundary conditions are incorporated by

```
>> [U,FreeDofs] = assemDir_hp(Mesh,Elem2Dof,BdFlag,QuadRule, ...
Shap,FHandle,FParam);
```

where `U` is a (M+dof)-by-1 matrix. In the first part of the program the contributions of the vertices are computed, the contributions of the linear vertex shape functions are substracted from the function values and then the contributions of the edge shape functions are extracted using the quadrature rule.

6.2 Neumann Boundary Conditions

sect:assem_neu

The structure for the incorporation of the Neumann boundary conditions into the right hand side load vector `L` is similar to the Dirichlet case described in the previous section. The functions are called `assemNeu` and are also stored in the folder `/Lib/Assembly`. The Neumann boundary conditions specify the values that the normal derivative of a solution is to take on the boundary of the domain.

Generally the assembly functions for the incorporation of the Neumann boundary conditions into the right hand side load vector `L` are e.g. called by

```
>> L = assemNeu_LFE(Mesh,BdFlag,L,QuadRule,FHandle,FParam);
```

The struct `Mesh` must contain the fields `Coordinates`, `Elements` and `BdFlags` as well as `EdgeLoc` and `Edge2Elem` as specified in table tab:bound_mesh 6.1 resp. table tab:dir_hp 6.3. Again, the integers `BdFlag` specify the edges on which the boundary conditions are enforced. `L` is the right hand side load vector as computed by the `assemLoad`-functions, cf. section sect:assem_load 5.2, p. 52. The 1D struct `QuadRule` is used to do the numerical integration along the edges, see table tab:quad_rule 3.1, p. 31, for its data structure. Finally, `FHandle` is the function handle which describes the Neumann boundary conditions, and `FParam` its variable length parameter list.

The code in the case of linear finite elements can be found below.

```
function L = assemNeu_LFE(Mesh,BdFlags,L,QuadRule,FHandle,varargin)
%   Copyright 2005-2005 Patrick Meury & Kah-Ling Sia
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

% Initialize constants

nCoordinates = size(Mesh.Coordinates,1);
nGauss = size(QuadRule.w,1);

% Precompute shape functions

N = shap_LFE([QuadRule.x zeros(nGauss,1)]);

Lloc = zeros(2,1);
for j1 = BdFlags

  % Extract Neumann edges

  Loc = get_BdEdges(Mesh);
  Loc = Loc(Mesh.BdFlags(Loc) == j1);

  for j2 = Loc'

    % Compute element map

    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);
      id_e = Mesh.Elements(Elem,rem(EdgeLoc+1,3)+1);

    else

      % Match orientation to right hand side element

      Elem = Mesh.Edge2Elem(j2,2);
      EdgeLoc = Mesh.EdgeLoc(j2,2);
      id_s = Mesh.Elements(Elem,rem(EdgeLoc,3)+1);
      id_e = Mesh.Elements(Elem,rem(EdgeLoc+1,3)+1);

    end
    Q0 = Mesh.Coordinates(id_s,:);
    Q1 = Mesh.Coordinates(id_e,:);
    x = ones(nGauss,1)*Q0+QuadRule.x*(Q1-Q0);
    dS = norm(Q1-Q0);

    % Evaluate Neumannn boundary data

    FVal = FHandle(x,j1,varargin{:});
```

```
55
56     % Numerical integration along an edge
57
58     Lloc(1) = sum(QuadRule.w.*FVal.*N(:,1))*dS;
59     Lloc(2) = sum(QuadRule.w.*FVal.*N(:,2))*dS;
60
61     % Add contributions of Neumann data to load vector
62
63     L(id_s) = L(id_s)+Lloc(1);
64     L(id_e) = L(id_e)+Lloc(2);
65
66   end
67 end
68
69 return
```

The incorporation of the Neumann data always follows roughly along the same lines.

- initialize constants and compute shape function values in quadrature points (lines 7-16)
- loop over all Neumann boundary edges (lines 17-67)
  - determine corresponding boundary element and adjust orientation to element (lines 28-46)
  - compute element mapping and value of the normal derivative that is to prescribe on the edge (lines 47-54)
  - compute local load vector (line 56-59)
  - add local load vector to the global load vector (line 61-64)

The output `L` is the right hand side load vector which inherits the Neumann boundary conditions:

`L`	M-by-1 matrix which contains the right hand side load vector with incorporated Neumann boundary conditions. The entries remain unchanged on non-boundary terms, otherwise the contributions of the Neumann boundary data is added.

Table 6.4: Output of the `assemNeu`-functions `tab:neu_out`

6.2.1 Linear Finite Elements

.. in 1D

Again, in the 1-dimensional case, no shape functions nor boundary flags are required. The `assemNeu_P1_1D`-function is called by

```
>> L = assemNeu_P1_1D(Coordinates,NNodes,L,FHandle,FParam);
```

and adds the Neumann boundary contributions given by `FHandle` to the vertices `NNodes`.

.. in 2D

In `assemNeu_LFE` the shape functions from `shap_LFE` are called.

6.2.2 Bilinear Finite Elements

The shape functions `shap_BFE` are used in `assemNeu_BFE`.

6.2.3 Quadratic Finite Elements

In `assem_QFE` three modifications to the original load vector are to be made, two w.r.t. vertex shape functions and one w.r.t. the respective edge shape function from `shap_QFE`.

6.2.4 *hp* Finite Elements

The routine `assemNeu_hp` for the *hp*FEM requires the additional inputs `Elem2Dof` and `Shap` (cf. 6.1.6). The program is called by

```
>> L = assemNeu_hp(Mesh,Elem2Dof,L,BdFlag,QuadRule,Shap, ...
FHandle,FParam);
```

There is not a big difference to the operations described before. Just the values of the shape functions have to be computed beforehand and not in the program `assemNeu_hp` itself. The struct `Elem2Dof` is needed in order to extract the dof numbers of the edges.

Chapter 7

Plotting the Solution

In the following plotting routines stored in `/Lib/Plots` are explained. They are mostly used to plot the solutions.

7.1 Ordinary Plot

sect:plot

Once the coefficient vector `U` of the finite element solution is computed via the \-operator in MATLAB, an illustration is possible using the `plot`-routines saved in the folder `/Lib/Plots`.

The required input arguments are listed in table tab:plot_in 7.1.

`U`	M-by-1 matrix specifying the values of the coeffiecient vetor at the nodes of the finite element solution
`Mesh`	struct that at least contains the fields `Coordinates` and `Elements` (cf. table tab:MSH_B 1.1, p. tab:MSH_B 11)

Table 7.1: Input for `plot`-routines tab:plot_in

The `plot`-functions are e.g. called by

```
>> H = plot_LFE(U,Mesh);
```

Figure fig:plot_sol 7.1 shows the finite element solution of the Laplacian obtained using linear finite elements in `main_LFE` in `/Examples/QFE_LFE`. It is plotted using `plot_LFE`. The colorbar was added separately using the built-in `colorbar`-command in MATLAB.

The function returns a handle to the figure `H`. In case the solution is complex-valued both, the real and imaginary term, are plotted separately.

The following steps are implemented:

1. computation of axes limits
2. plotting of real (and eventually also imaginary) finite element solution using the MATLAB commands `figure` and `patch`

Figure 7.1: Plot of LFE solution in main_LFE fig:plot_sol

7.1.1 Constant Finite Elements

plot_P0 generates the plot for the solution on constant finite elements.

7.1.2 Linear Finite Elements

.. in 1D

plot_1D is called by

```
>> H = plot_1D(U,Coordinates);
```

where Coordinates is the M-by-1 matrix specifying the nodes of the mesh.

.. in 2D

plot_LFE and plot_LFEfancy generate plots of the finite element solution, plot_DGLFE plots the finite element solution for discontinuous linear finite elements.

The solution for vector-valued linear finite elements is plotted by plot_LFE2.

7.1.3 Bilinear Finite Elements

plot_BFE plots the finite element solution computed using bilinear finite elements.

7.1.4 Crouzeix-Raviart Finite Elements

For the plot of the solution computed by Crouzeix-Raviart finite elements, an auxiliary mesh is generated and the solution updated there. The function plot_CR plots the solution, plot_DGCR plots the solution for discontinuous Crouzeix-Raviart elements.

7.1.5 Quadratic Finite Elements

In `plot_QFE` the mesh is refined using `refine_REG` before plotting the finite element solution. See p. 18f for more details on this function.

7.1.6 Whitney 1-Forms

The function `plot_W1F` generates a plot of the velocity field represented by the W1F solution `U` on the struct `Mesh` and returns a handle to the figure. The functions `get_MeshWidth` and `shape_W1F` are called within the program. The transformation on the reference element is done and there the shape functions and velocity field are computed at its barycenter. Finally, the MATLAB command `quiver` plots the velocity vectors as arrows.

Note that as usual for the Whitney 1-forms the `Mesh`-fields `Edges` and `Vert2Edge` are needed.

7.1.7 *hp* Finite Elements

.. in 1D

The function `plot_hp_1D` generates a plot of the *hp*DG solution `U` on the mesh specified by the M-by-1 matrix `Coordinates` using the polynomial degrees specified by `p` and the shape functions provided by the function handle `Shap` (cf. `shap_Leg_1D`, p. 28) on each element. It is called by

```
>> H = plot_hpDG_1D(Coordinates,p,u,Shap);
```

The built-in MATLAB command `plot` is used for the illustration.

.. in 2D

By `plot_hp` a plot of the piecewise polynomial function of maximum polynomial degree `p` given by `U` on the struct `Mesh` is generated.

In order to split the reference element according to the maximum polynomial degree the function `split_RefElem` is called. It splits the reference triangle into smaller elements according to the resolution `res` by

```
>> [RefCoord,RefElem] = split_RefElem(res);
```

In `plot_hp` the choosen resolution `res` is `1/(2*p)`. The MATLAB command `delaunay` is used to generate the set of triangles such that no data points are contained in any triangle's circumcircle. For details on the output `[RefCoord,RefElem]` of `split_RefElem` see table 7.2 (A and B depend on the choosen `res`).

`RefCoord`	A-by-2 matrix specifying all vertex coordinates
`RefElem`	B-by-3 matrix connecting vertices into triangular elements

Table 7.2: Output of the function `split_RefElem` tab:res_out

On the refined element, the ouput of the shape functions `shap_hp` is computed at the vertices `RefCoord`. Finally, the function values on the elements are computed and then plotted.

The `plot`-function for the *hp*FEM is called by

```
>> H = plot_hp(U,Mesh,Elem2Dof,p);
```

The field `Elem2Dof` is explained on p. 51.

7.2 Plot Line

sect:plot_line

The `plotLine`-functions in `/Lib/Plots` generate a plot of the finite element solution `U` on a certain line of the mesh. Again, the solution `U` and the `Mesh` are needed, see table 7.1. To specify the line the additional input arguments `x_start` and `x_end` are required, see table 7.3.

`x_start`	1-by-2 matrix specifying the starting point of the section
`x_end`	1-by-2 matrix specifying the end point of the section

Table 7.3: Input for `plotLine`-routines

tab:plotline_in

So far these section plots are implemented for linear and quadratic finite element solutions. The functions are e.g. called by

```
>> L = plotLine_LFE(U,Mesh,x_start,x_end);
```

Figure 7.2 shows the output for the Laplacian solved with linear finite elements in `main_LFE` in `/Examples/QFE_LFE`, cf. figure 7.1. The starting point of the section is $(0, 0)$, the end point is $(1, 1)$.

Figure 7.2: Plot section of `LFE` solution in `main_LFE`

fig:plot_line_sol

The element contributions are computed using the specific shape functions, cf. section 2.3. The figures are generated by the MATLAB command `plot`.

7.2.1 Linear Finite Elements

The function `plotLine_LFE` plots the finite element solution on the desired section using `shap_LFE` to compute the element contributions.

7.2.2 Quadratic Finite Elements

Similarily, `plotLine_QFE` calls `shap_QFE` to compute the element contributions on the section specified.

7.3 Plot Contours

sect:plot_contour

The `contour`-functions in the folder `/Lib/Plots` generate a contour-plot of the finite element solution `U` on the `Mesh`, see table 7.1. The functions are called by

```
>> H = contour_LFE(U,Mesh);
```

where `H` is a handle to the generated figure. Further variable input arguments are listed in table 7.4.

`levels`	vector that specifies at which values contour lines should be drawn
`'c'`	specifies the color of the level curves. See MATLAB help for colorspec for predefined colors. The default color is `'b'` (blue).
`'colorbar'`	appends a colorbar to the current axes in the right location

Table 7.4: Additional input for `contour`-routines

tab:plot_add

Hence the differing calls are e.g.

```
>> H = contour_LFE(U,Mesh,levels);
>> H = contour_LFE(U,Mesh,levels,'c');
>> H = contour_LFE(U,Mesh,levels,'colorbar');
```

Figure 7.3 shows the output of the function `main_ContourExample` stored in the folder `/Examples/QFE_LFE`. It's the solution of the Laplacian using linear finite elements, cf. figures 7.1 and 7.2.

7.3.1 Linear Finite Elements

By `get_MeshWidth` the mesh width for a given triangular mesh is computed. For each element, the coordinates of the vertices are extracted, the element mapping is computed and the element transformed to the reference element using the shape functions in `shap_LFE`. The solution is then plotted using the MATLAB command `contour`. All this is done by the function `contour_LFE` which may be called as explained above.

An application of `contour_LFE` is included in `main_ContourExample` which is stored in the folder `/Examples/QFE_LFE`.

Figure 7.3: Contour plot of `LFE` solution in `main_ContourExample` fig:plot_cont_sol

7.3.2 Crouzeix-Raviart Finite Elements

As above the solution in `contour_DGCR` is computed using transformed grid points by `shap_DGCR`.

The function is used to plot the approximate solution for the circular advection problem `main_CA_2` stored in `/Examples/FVDG`.

Chapter 8

Discretization Errors

chap:err

The following (semi-)norms monitor the difference between the computed finite element solution (using the \-operator in MATLAB to compute the coefficient vector `u`) and the exact solution given by the function handle `FHandle`. The respective error functions are stored in the folder `/Lib/Errors`. There the abbreviations stand for

`H1Err_*`	discretization error in the H^1-norm for finite element *
`H1SErr_*`	discretization error in the H^1-semi-norm for finite element *
`L1Err_*`	discretization error in the L^1-norm for finite element *
`L2Err_*`	discretization error in the L^2-norm for finite element *
`LInfErr_*`	discretization error in the L^∞-norm for finite element *
`HCurlSErr_*`	discretization error in the S^1-norm for (vector-valued) finite elements *

Table 8.1: File names for the computation of discretization errors tab:err_norms

As already mentioned, the main input arguments are of course the exact and the finite element solution. They must be provided as stated in table tab:err_sol 8.2. Here `u` corresponds to the vector μ on p. chap:local_comp 35.

`u`	values of the finite element solution (in fact its coefficient vector) at the vertices of the mesh
`FHandle`	function handle to the exact solution
`FParam`	variable length argument list for `FHandle`

Table 8.2: Exact and FE solution tab:err_sol

The other input arguments are the `Mesh` and a quadrature rule `QuadRule` (see section chap:quad_rule 3, p. chap:quad_rule 31ff) with a sufficient order.

The struct `Mesh` must at least contain the fields specified in table tab:err_mesh1 8.3. In case the shape functions are connected to edges also the ones in table tab:err_mesh2 8.4 are necessary.

The `L2Err`-functions are e.g. called by

`Coordinates`	M-by-2 matrix specifying all vertex coordinates
`Elements`	N-by-3 or N-by-4 matrix connecting vertices into elements

Table 8.3: Basic mesh data structure (2D) tab:err_mesh1

`Edges`	P-by-2 matrix specifying all edges of the mesh
`Vert2Edge`	M-by-M sparse matrix which specifies whether the two vertices `i` and `j` are connected by an edge with number `Vert2Edge(i,j)`

Table 8.4: Additional mesh data structure (2D) tab:err_mesh2

```
>> err = L2Err_LFE(Mesh,u,QuadRule,FHandle,FParam);
```

where, as usual, `FParam` handles the variable length argument list to the exact solution `FHandle`. The output `err` is the error in the L^2-norm summed up over all elements.

The MATLAB-code follows the line:

1. pre-computation of the shape functions `shap_*` (stored in `/Lib/Element`) at the given quadrature points `QuadRule.x`, e.g.

   ```
   >> N = shap_LFE(QuadRule.x);
   ```

2. for each element extraction of the vertices and computation of the transformation map to the standard reference element

3. evaluation of the exact and finite element solutions `u_EX` resp. `u_FE` on that element `i` (e.g. in the case of triangular finite elements) by

   ```
   >> u_EX = FHandle(x,FParam);
   >> u_FE = u(Mesh.Elements(i,1))*N(:,1)+ ...
   u(Mesh.Elements(i,2))*N(:,2)+u(Mesh.Elements(i,3))*N(:,3);
   ```

   where `x` are the transformed quadrature points.

4. computation of the error using the respective norm and `QuadRule`

5. summation of all element errors

For the Sobolev-(semi-)norms it's also necessary to compute the gradients `grad_u_EX` resp `grad_u_FE` of the solutions. See section sect:h1_err 8.1 for further details.

All norms should be contained in any functional analysis book.

8.1 H^1-Norm

sect:h1_err

The discretization error between the exact and the finite element solution on the given mesh w.r.t. the H^1-norm is implemented in the `H1Err`-functions stored in

/Lib/Errors. The H^1-norm for suffieciently smooth functions $f : \mathbb{R}^n \supseteq \Omega \to \mathbb{R}$ is of the form

$$\|f\|_{H^1(\Omega)} = \left(\int_\Omega |f(x)|^2 + |Df(x)|^2 dx\right)^{\frac{1}{2}} \tag{8.1}$$

eq:H1_norm

where $|Df(x)|^2 = |\partial_1 f(x)|^2 + \ldots + |\partial_n f(x)|^2$.

To this end, the gradients grad_u_EX and grad_u_FE of the solutions are to be computed. For the exact solution this information simple needs to be included in the evaluation of the function handle, i.e.

```
>> [u_EX,grad_u_EX] = FHandle(x,FParam);
```

For the computation of grad_u_FE, the gradients grad_N of the respective shape functions are needed, i.e. in H1Err_LFE for the i-th element

```
>> grad_u_FE = (u(Mesh.Elements(i,1))*grad_N(:,1:2)+ ...
u(Mesh.Elements(i,2))*grad_N(:,3:4)+ ...
u(Mesh.Elements(i,3))*grad_N(:,5:6))*transpose(inv(BK));
```

where BK is the transformation matrix and

```
>> grad_N = grad_shap_LFE(QuadRule.x);
```

For an extensive discussion of the input parameters and the general procedure of computation see p. 71f. chap:err

8.1.1 Linear Finite Elements

.. in 1D

The functionH1Err_P1_1D requires shap_P1_1D and grad_shap_P1_1D. It is called by

```
>> err = H1Err_P1_1D(Coordinates,u,QuadRule,FHandle,FParam);
```

.. in 2D

Using shap_LFE and grad_shap_LFE the function H1Err_LFE computes the discretization error between the exact solution given and the finite element solution.

8.1.2 Bilinear Finite Elements

Similarily, H1Err_BFE makes use of shap_BFE and grad_shap_BFE.

8.1.3 Quadratic Finite Elements

ssect:h1err_qfe

In this case there are nodes on the edges, hence the field Vert2Edge (see table 8.4, p. 72) is needed for the extraction of the edge numbers. tab:err_mesh2 The computation in H1Err_QFE is the same as described above. The functions shap_QFE and grad_shap_QFE are called.

H1Err_PBD computes the discretization error w.r.t. the H^1-norm for quadratic finite elements with parabolic boundary approximation. If the boundary correction term stored in the Mesh field Delta (see table 8.5) is greater than eps then the computation is done for a curved element, otherwise for a straight element.

Delta	P-by-1 matrix specifying the boundary correction term on every edge

Table 8.5: Field Delta of struct Mesh tab:mesh.delta

8.1.4 *hp* Finite Elements

ssect:h1err_hp

The computation of the H^1 discretization error in H1Err_hp is more complex, more precisely the computation of the finite element solution u_FE and its gradient grad_u_FE. Additional input arguments are the shape functions Shap (contains the values of the shape functions and its gradients computed by shap_hp, p. 28) and Elem2Dof (extracts the degrees of freedom). The error is computed by

```
>> err = H1Err_hp(Mesh,u,Elem2Dof,QuadRule,Shap,FHandle,FParam);
```

8.2 H^1-Semi-Norm

sect:h1s_err

The H1SErr-functions in /Lib/Errors compute the discretization error in the H^1-semi-norm, i.e.

$$\|f\|_{H^1_s(\Omega)} = \left(\int_\Omega |Df(x)|^2 dx \right)^{\frac{1}{2}} \tag{8.2}$$

with $|Df(x)|^2 = |\partial_1 f(x)|^2 + \ldots + |\partial_n f(x)|^2$. The first 0-order term is missing (cf. $\|f\|_{H^1}$ in (8.1), p. 73) which reduces it to a semi-norm.

Here, the function handle FHandle specifies the gradient of the exact solution which is then given by

```
>> grad_u_EX = FHandle(x,FParam);
```

The gradient grad_u_FE is gained as in the H1Err-functions in the previous section. Again, the grad_shap-functions are used for the computation. Further details are summerized on p. 71f.

8.2.1 Linear Finite Elements

.. in 1D

For the 1-dimensional discretization error the M-by-1 matrix Coordinates is needed. The routine H1SErr_P1_1D is called by

```
>> err = H1SErr_P1_1D(Coordinates,u,QuadRule,FHandle,FParam);
```

The values of the gradients are computed by `grad_shap_P1_1D`.

.. in 2D

`H1SErr_LFE` computes the discretization error in the H^1-semi-norm using `grad_shap_LFE`.

`H1SErr_DGLFE` measures the error for discontinuous linear Lagrangian finite elements. The gradients of the shape functions are computed by `grad_shap_DGLFE`. In case additional element information is stored in `ElemFlag`, it is taken into account for the computation of the gradient of the exact error `grad_u_EX`. For the `i`-th element this is

```
>> grad_u_EX = FHandle(x,ElemFlag(i),FParam);
```

8.2.2 Bilinear Finite Elements

`H1SErr_BFE` computes the discretization error in the H^1-semi-norm. The routines `shap_BFE` and `grad_shap_BFE` are both needed, the first just for the transformation of the quadrature points.

8.2.3 Crouzeix-Raviart Finite Elements

With the help of `grad_shap_CR` the function `H1SErr_CR` computes the discretization error for Crouzeix-Raviart elements and `H1SErr_DGCR` the discretization error for the discontinuous Crouzeix-Raviart elements. Optional element information stored in the `Mesh` field `ElemFlag` may be taken into account in the latter. The field `Vert2Edge` is obligatory.

8.2.4 Quadratic Finite Elements

The discretization error in the H^1-semi-norm for quadratic finite element is computed by `H1SErr_QFE`. The `Mesh` field `Vert2Edge` is required.

Similar to `H1Err_PBD` (see 8.1.3) the function `H1SErr_PBD` provides the discretization error for quadratic finite elements with parabolic boundary approximation.

8.2.5 *hp* Finite Elements

The discretization error in the H^1-semi-norm for *hp* finite elements is evaluated by `H1SErr_hp`. It is called by

```
>> err = H1Err_hp(Mesh,u,Elem2Dof,QuadRule,Shap,FHandle,FParam);
```

For further information on the input parameters see 8.1.4, p. 74.

8.3 L^1-Norm

The discretizatin error in the L^1-norm is computed by the `L1Err`-functions stored in `/Lib/Errors`. The norm is defined by

$$\|f\|_2 = \int_\Omega |f(x)|dx \quad . \tag{8.3}$$

The required input arguments are listed on p. 71f.

8.3.1 Crouzeix-Raviart Finite Elements

The finite element solution `u_FE` is computed using the shape functions `shap_DGCR`, `L1Err_DGCR` determines the discretization error in the L^1-norm. As usual it is called by

```
>> err = L1Err_DGCR(Mesh,u,QuadRule,FHandle,FParam);
```

8.3.2 *hp* Finite Elements

The finite element discretization error is computed in `L1Err_hpDG_1D`. It is called by

```
>> err = L1Err_hpDG_1D(Coordinates,p,u,QuadRule,Shap, ...
FHandle,FParam);
```

Obviously the input arguments for the *hp*FEM differ from the previous mentioned input at the beginning of the chapter. For the 1-dimensional function `L1Err_hpDG_1D` the required data is listed in table 8.6.

`Coordinates`	$n+1$ mesh points
`p`	polynomial approxiamtion of each of the n cells
`u`	solution for the last step
`QuadRule`	1D quadrature rule (cf. 3.2, p. 32)
`Shap`	basis functions evaluated at quadrature points (e.g. `shap_Leg_1D`, see 2.3.4, p. 28)
`FHandle`	function handle of the exact solution
`FParam`	variable length argument list to `FHandle`

Table 8.6: Input arguments for `L1Err_hpDG_1D` and `L2Err_hpDG_1D`

8.3.3 Linear Finite Volumes

`L1Err_LFV` computes the discretization error in L^1-norm for linear finite volumes, `shap_LFE` is called within the program. See section 10.1, p. 101, for the context.

8.4 L^2-Norm

sect:l2_err

The L2Err-functions stored in /Lib/Errors compute the discretization error of the finite element solution u_FE to the exact solution u_EX in the L^2-norm, which is defined by

$$\|f\|_2 = \left(\int_\Omega |f(x)|^2 dx \right)^{\frac{1}{2}} \quad . \tag{8.4}$$

For the computation of the integral, the shape functions shap_* are used. The quadrature rule QuadRule is to be specified in the input. General information about the input arguments and concept of computation may be found on p. chap:err 71f.

8.4.1 Constant Finite Elements

L2Err_PC computes the discretization error in the L^2-norm for piecewise constant finite elements.

8.4.2 Linear Finite Elements

.. in 1D

L2Err_P1_1D computes the discretization error in the L^2-norm for 1D linear finite elements using the shape functions shap_P1_1D. Mainly the Gauss-Legendre quadrature rule gauleg is applied, e.g. in the routine main_1D in /Examples/1D_FEM.

.. in 2D

The function L2Err_LFE computes the discretization error for linear finite elements using shap_LFE and L2Err_LFE2 the one for the vector-valued shape functions shap_LFE2.

For the discontinous Galerkin method L2Err_DGLFE is used. It calls the shape functions shap_DGLFE and is currently only called by main_4 in /Examples/DGFEM with quadrature rule P303.

8.4.3 Bilinear Finite Elements

L2Err_BFE makes use of shap_BFE and grad_shap_BFE to compute the discretization error in the L^2-norm. A quadrature rule on the unit square $[0,1]^2$ is e.g. TProd(gauleg(0,1,2), see ssect:quad_trans sect:quad_trans 3.3.1, p. 32.

8.4.4 Crouzeix-Raviart Finite Elements

Crouzeix-Raviart finite elements correspond to edges, hence the field Vert2Edge of table tab:err_mesh2 8.4 is required. Furthermore, L2Err_CR calls the shape functions shap_CR and the discontinous version L2Err_DGCR the shape functions shap_DGCR.

8.4.5 Quadratic Finite Elements

Similarily, 3 out of 6 quadratic shape functions correspond to edges. The discretization error `L2Err_QFE` calls `shap_QFE` and uses `Vert2Edge` to extract the edge numbers of the elements.

`L2Err_PBD` computes the discretization error with respect to the L^2-norm for quadratic finite elements with parabolic boundary approximation.

8.4.6 Whitney 1-Forms

In `L2Err_W1F` the field `Vert2Edge` is needed too. Furthermore `Edges` is used to determine the orientation.

8.4.7 *hp* Finite Elements

.. in 1D

The function `L2Err_hpDG_1D` for the computation of the L^2 discretization error is called by

```
>> err = L2Err_hpDG_1D(Coordinates,p,u,QuadRule,Shap, ...
FHandle,FParam);
```

For a explanation of the input arguments see table tab:err_hp_1d 8.6.

.. in 2D

The discretization error for the *hp*FEM is computed in a iterative procedure, more precisely the finite element solution `u_FE` is gained this way. The struct `Elem2Dof` extracts the degrees of freedom from the elements. As in the 1-dimensional case the shape functions are not called within the program, but provided as an input. The hierarchical shape functions `shap_hp` are used, cf. ssect:shap_hp 2.3.5, p. ssect:shap_hp 28. `L2Err_hp` is called by

```
>> err = L2Err_hp(Mesh,u,Elem2Dof,QuadRule,Shap,FHandle,FParam);
```

8.4.8 Further Functions

`L2Err_LFV` computes the discretization error in L^2-norm for linear finite volumes (see also p. L2Err_LFV 107).

`L2Err_PWDG` computes the L^2-norm discretization error for discontinuous plane waves.

8.5 L^∞-Norm

sect:linf_err

The computation of the discretization error in the L^∞-norm

$$\|f\|_\infty = \sup_{x\in\Omega} |f(x)| \tag{8.5}$$

is much easier then the computation of the previous norms. Basically no shape functions nor quadrature rules are needed, except for the *hp*FEM.

The `LInfErr`-functions are e.g. called by

```
>> err = LInfErr_LFE(Mesh,u,FHandle,FParam);
```

an compute the discretization error between the finite element solution `u` and the exact solution given by the function handle `FHandle`. Mostly the code just consists of the line

```
>> err = max(abs(u-FHandle(Mesh.Coordinates,FParam)));
```

8.5.1 Linear Finite Elements

.. in 1D

`LInfErr_1D` is called by

```
>> err = LInfErr_1D(Coordinates,u,FHandle,FParam);
```

.. in 2D

`LInfErr_LFE` computes the discretization error in the 2-dimensional case.

The function `LInfErr_PBD` calculates the error on linear finite elements with parabolic boundary approximation. The boundary edges of the mesh are extracted by the function `get_BdEdges`. Furthermore the midpoints of the straight and curved edges have to be taken into account which makes the program much more lengthy.

8.5.2 Bilinear Finite Elements

`LInfErr_BFE` computes the discretization error on square elements.

8.5.3 Quadratic Finite Elements

The discretization error `LInfErr_QFE` on the quadratic finite elements is a maximum of those at the edges and midpoints.

8.5.4 *hp* Finite Elements

In `LinfErr_hpDG_1D` the discretization error is computed via a loop over all elements. The function is called by

```
>> err = LinfErr_hpDG_1D(Coordinates,p,u,QuadRule,Shap, ...
FHandle,FParam);
```

The input arguments are specified in table 8.6, p. 76.

8.5.5 Linear Finite Volumes

LInfErr_LFV computes the discretization error in the L^{∞}-norm for linear finite volumes, cf. p. 108.

Chapter 9

Examples

In the previous chapters a lot of different finite elements for different problems were introduced. The goal of this chapter will be to discuss some of the examples which can be found in the folders contained in `LehrFEM/Examples`. All of these folders contain files caled `main_...`, which are routines that contain executeable code for different operators and problems.

In the discussion of the example problems some of the original code is inserted. A discussion of the code shown can always be found below the boxes.

9.1 Linear and Quadratic finite elements

sec:lqfe

Probably the easiest example for a 2-dimensional finite element method is to use piecewise linear basis functions. This can be found in the folder `LehrFEM/Examples/QFE_LFE`. The driver routine is called `main_LFE`, the code can be found below.

```
% Run script for piecewise linear finite element solver.

%   Copyright 2005-2005 Patrick Meury & Kah Ling Sia
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

 % Initialize constants

 NREFS = 5;                 % Number of red refinement steps
 F_HANDLE = @f_LShap;       % Right hand side source term
 GD_HANDLE = @g_D_LShap;    % Dirichlet boundary data
 GN_HANDLE = @g_N_LShap;    % Neumann boundary data

 % Initialize mesh

 Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
 Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
 Mesh = add_Edges(Mesh);
 Loc = get_BdEdges(Mesh);
 Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
 Mesh.BdFlags(Loc) = [-1 -2 -7 -7 -3 -4];
 for i = 1:NREFS
```

```
    Mesh = refine_REG(Mesh);
25  end
    Mesh = add_Edge2Elem(Mesh);

    % Assemble stiffness matrix and load vector

30  A = assemMat_LFE(Mesh,@STIMA_Lapl_LFE);
    L = assemLoad_LFE(Mesh,P706(),F_HANDLE,0,1,2);

    % Incorporate Neumann boundary data

35  L = assemNeu_LFE(Mesh,-1:-1:-4,L,gauleg(0,1,4),GN_HANDLE);

    % Incorporate Dirichlet boundary data

    [U,FreeDofs] = assemDir_LFE(Mesh,-7,GD_HANDLE);
40  L = L - A*U;

    % Solve the linear system

    U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);
45
    % Plot out solution

    plot_LFE(U,Mesh);colorbar;
    plotLine_LFE(U,Mesh,[0 0] ,[1 1]);
50
    clear all;
```

This routine solves the Laplace equation on an L-shaped domain with mixed boundary conditions (Neumann and Dirichlet) using linear finite elements.

$$
\begin{aligned}
-\Delta u(x) &= f(x), && x \in \Omega \\
u(x) &= g_D(x), && x \in \partial\Omega
\end{aligned}
$$

The stiffness matrix and the load vector are computed as discussed in Chapter 4 and 5, the routines for linear finite elements are shown there as the reference example. The resulting linear system is solved in line 44. A plot of the solution on the computational domain is done in line 48. The solution along the line $(0,0),(1,1)$ is plotted in line 49.

The structure of the driver routine for using quadratic finite elements instead of linear finite elements looks very similar. Only the assembly routines and the function handles for computing local stiffness matrices have to be replaced. The name of the routine is `main_QFE`.

The files `DiscrErrors_...` in the same folder investigate the convergence rate of the used method. In these files the finite element solution is computed for problems, where the exact solution is known. This is done for several refinement steps, i.e. mesh widths. A part of the code for computing the convergence rate of linear and quadratic finite elements can be found below.

```
%%%%%%%%%%%%%%%%%
% Initialize constants...
%%%%%%%%%%%%%%%%%
```

```
% Initialize mesh

Mesh = load_Mesh('Coord_Sqr.dat','Elem_Sqr.dat');
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
Mesh = add_Edges(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
Mesh.BdFlags(Loc) = -1;
for i = 1:NREFS

  % Do red mesh refinement

  Mesh = refine_REG(Mesh);
  Mesh = add_Edge2Elem(Mesh);


  % Assemble Stiffness matrix, load vector and incorporate BC

  A_QFE = assemMat_QFE(NewMesh,@STIMA_Lapl_QFE);
  L_QFE = assemLoad_QFE(NewMesh,P706(),F_HANDLE);

  A_LFE = assemMat_LFE(NewMesh,@STIMA_Lapl_LFE);
  L_LFE = assemLoad_LFE(NewMesh,P706(),F_HANDLE);

  % Incorporate Dirichlet and Neumann boundary data

  [U_LFE,FreeDofs_LFE] = assemDir_LFE(NewMesh,-1,GD_HANDLE);
  L_LFE = L_LFE - A_LFE*U_LFE;

  [U_QFE,FreeDofs_QFE] = assemDir_QFE(NewMesh,-1,GD_HANDLE);
  L_QFE = L_QFE - A_QFE*U_QFE;

  % Solve the linear system

  U_LFE(FreeDofs_LFE) = A_LFE(FreeDofs_LFE,FreeDofs_LFE)\...
              L_LFE(FreeDofs_LFE);
  U_QFE(FreeDofs_QFE) = A_QFE(FreeDofs_QFE,FreeDofs_QFE)\...
              L_QFE(FreeDofs_QFE);

  % Compute discretization error

  LInf_Error_LFE(i) = LInfErr_LFE(NewMesh,U_LFE,U_EX_1);
  L2_Error_LFE(i) = L2Err_LFE(NewMesh,U_LFE,P706(),U_EX_1);
  H1S_Error_LFE(i) = H1SErr_LFE(NewMesh,U_LFE,P706(),U_EX_2);
  N_LFE(i) = size(NewMesh.Coordinates,1);

  LInf_Error_QFE(i) = LInfErr_QFE(NewMesh,U_QFE,U_EX_1);
  L2_Error_QFE(i) = L2Err_QFE(NewMesh,U_QFE,P706(),U_EX_1);
  H1S_Error_QFE(i) = H1SErr_QFE(NewMesh,U_QFE,P706(),U_EX_2);
  N_QFE(i) = size(NewMesh.Coordinates,1) + ...
        size(NewMesh.Edges,1);

  h(i) = get_MeshWidth(Mesh);
```

```
end
60
%%%%%%%%%%%%%%%%%
% plot computed errors...
%%%%%%%%%%%%%%%%%
```

The variable `NREFS` specifies the number of refinement steps. In the loop starting in line 13 firstly the mesh is refined, then the matrices and the load vector assembled. With these it is possible to determine the finite element solution by solving a linear system. This is done for linear and quadratic finite elements. Since the exact solution is known the desired errors, i.e. L^∞, L^2, H^1-seminorm can be computed (lines 44-55). After the loop one can plot the errors against the mesh width or the number of degrees of freedom using the data stored in the corresponding variables. Such a plot can be found in Figure fig:lfeqfe 9.1 for the solution of the Poisson equation on the square for the L^2 and the H^1-semi norm.

Figure 9.1: Convergence rates for linear and quadratic finite elements fig:lfeqfe

9.2 DG finite elements

The driver routines for this method can be found in the folder `LehrFEM/Examples/DGFEM`. In the discontinuous galerkin method it is allowed to have discontinuities along the edges, therefore the underlying space of basis functions consists of the shape functions transformed to an arbitrary triangle using the uniquely defined linear affine transform. For example in the case of linear shape functions there are 3 degrees of freedom on every triangular element with vertices (a_1, a_2, a_3), which are the uniquely determined linear functions satisfying $b_i(a_j) = \delta_{ij}$. This approach also affects the bilinear form, as described on p. 39.

The numbering of the degrees of freedom in this case is done elementwise, which means that the corresponding degrees of freedom to the element i are given by $l(i-1) + (1, \dots, l)$, where l denotes the number of degrees of freedom on every element. The driver routine `main_1` implements discontinuous Galerkin discretization using Crouzeix-Raviart elements. It solves the problem

$$\begin{aligned} -\Delta u(x) &= f(x),\ x \in \Omega \\ u(x) &= g_D(x),\ x \in \partial\Omega \end{aligned}$$

The code can be found below.

```
% Run script for discontinuous Galerkin finite element solver.

%   Copyright 2006-2006 Patrick Meury
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

% Initialize constants

% Number of red refinement steps
NREFS = 4;
% Right hand side load data
G = @(x,varargin)-4*ones(size(x,1),1);
% Dirichlet boundary data
UD = @(x,varargin)x(:,1).^2+x(:,2).^2;
% Symmetric (+1) or antisymmetric (-1) discretization
S = 1;
% Edge weight function
SIGMA = @(P0,P1,varargin)10/norm(P1-P0);

% Initialize mesh

Mesh.Coordinates = [-1 -1; 1 -1; 1 1; -1 1];
Mesh.Elements = [1 2 3; 1 3 4];
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
Mesh = add_Edges(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
Mesh.BdFlags(Loc) = -1;
for i = 1:NREFS
  Mesh = refine_REG(Mesh);
end
```

```
33 Mesh = add_Edge2Elem(Mesh);
34 Mesh = add_DGData(Mesh);
35
36 % Assemble matrices and load vectors
37 % (discontinuous Raviart elements)
38
39 QuadRule_1D = gauleg(0,1,2);
40 QuadRule_2D = P3O3();
41
42 [I1,J1,Avol] = assemMat_Vol_DG(Mesh,@STIMA_Lapl_Vol_DGCR);
43
44 [I2,J2,Jinn] = assemMat_Inn_DG(Mesh,@STIMA_InnPen_DGCR,SIGMA);
45 [I2,J2,Ainn] = assemMat_Inn_DG(Mesh,@STIMA_Inn_DGCR,S);
46
47 [I3,J3,Jbnd] = assemMat_Bnd_DG(Mesh,@STIMA_BndPen_DGCR,SIGMA);
48 [I3,J3,Abnd] = assemMat_Bnd_DG(Mesh,@STIMA_Bnd_DGCR,S);
49
50 Lvol = assemLoad_Vol_DG(Mesh,@LOAD_Vol_DGCR,QuadRule_2D,G);
51 Lbnd = assemLoad_Bnd_DG(Mesh,@LOAD_Bnd_DGCR,...
52           QuadRule_1D,S,SIGMA,UD);
53
54 % Create system matrix
55
56 A = sparse([J1; J2; J3; J2; J3], ...
57            [I1; I2; I3; I2; I3], ...
58            [Avol; Ainn; Abnd; Jinn; Jbnd]);
59 L = Lvol + Lbnd;
60
61 % Solve the linear system
62
63 U = A\L;
64 plot_DGCR(U,Mesh);
65 colorbar;
66
67 % Clear memory
68
69 clear all;
```

As described on pages 39 and 52 the computation of the stiffness matrix is done in five steps (lines 42-49). The assembly of the load vector is done in the lines 50-52.

The driver routine `main_2` determines the convergence rate for this method. The structure of the code is very similar to the convergence rate computation in the case of linear and quadratic finite elements which can be found in Section 9.1. Again one specifies the number of refinement steps and then the errors with respect to the exact solution are computed for different mesh widths.

Now we shall add a small description of the other driver routines contained in this folder.

- `main_0`: plots the mesh with the edge normals
- `main_1`: explained above
- `main_2`: explained above

- `main_3`: solves the poisson equation on a square using linear finite elements
- `main_4`: convergence rates on a square using linear finite elements
- `main_5`: solves the poisson equation on a square using quadratic finite elements (number of degrees of freedom per element can be specified)
- `main_6`: convergence rates on a square using quadratic finite elements (number of degrees of freedom per element can be specified)
- `main_7`: as `main_6` but with another exact solution
- `main_8`: compares the convergence rates on a square for symmetric discretization and anti-symmetric discretization

9.3 Whitney-1-forms

The implemented examples using Whitney-1-forms can be found in the folder `Examples/W1F`. As already described in the chapters on local computations and assembly of the matrices the basis functions used are vector valued. Here the degrees of freedom are given by edge integrals. The driver routine called `main_W1F` solves the problem

$$\nabla \times \mu_2 \nabla \times u + \mu_1 u = f \tag{9.1}$$

eq:ccurl

for given functions $\mu_{1,2}$ and f. On the boundary solely Dirichlet conditions are enforced. The code can be found below.

```
% Run script for W1F finite element solver.
%   Copyright 2005-2005 Patrick Meury & Mengyu Wang
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

% Initialize constant

NREFS = 4;
MU1_HANDLE=@(x,varargin)1;
MU2_Handle = @(x,varargin)ones(size(x,1),1);
F_Handle = @(x,varargin)(pi^2+1)*[sin(pi*x(:,2)) ...
          sin(pi*x(:,1))];
GD_Handle = @(x,varargin)[sin(pi*x(:,2)) sin(pi*x(:,1))];

% Initialize mesh

Mesh.Coordinates = [-1 -1;1 -1;1 1;-1 1];
Mesh.Elements = [1 2 4;2 3 4];
Mesh = add_Edges(Mesh);
Mesh = add_Edge2Elem(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
Mesh.BdFlags(Loc) = -1;
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);

for i=1:NREFS
```

```
   Mesh = refine_REG(Mesh);
end

% Assemble Curl-curl matrix, MASS matrix and load vector
t = cputime;
[IC,JC,C] = assemMat_W1F(Mesh,@STIMA_Curl_W1F,MU2_Handle,P706());
[IM,JM,M] = assemMat_W1F(Mesh,@MASS_W1F,MU1_HANDLE, P303());
A = sparse([IC;IM],[JC;JM],[C;M]);
L = assemLoad_W1F(Mesh,P706(),F_Handle);

% Incorporate Dirichlet boundary data

[U,FreeDofs] = assemDir_W1F(Mesh,-1,GD_Handle,gauleg(0,1,1));
L = L - A*U;

% Solve the system

U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);
fprintf('Runtime of direct solver [s]  :  %f\n',cputime-t);

% Plot the solution

plot_W1F(U,Mesh);

clear all
```

The routine `main_W1F2` solves the problem from (9.1) with an additional convective term, therefore the problem is given by

$$\nabla \times \mu_2 \nabla \times u + v \times u + \mu_1 u = f. \tag{9.2}$$

The function v is some given velocity field.

The convergence rates for the above mentioned problems can be computed using the `Script_W1F_...` and `Script_W1F2_...` routines depending on the computational domain one is interested in.

9.4 *hp*-FEM

All the driver routines for the *hp*-FEM implementation can be found in the folder `Examples/hpFEM`. For further information on the theory behind the implementation see [?]. The basic structure is the same as in driver routines for other FEM. The code of the routine `main_2`, which solves the Poisson equation on an L-shaped domain can be found below.

```
% Runs script for hp-FEM.

% Copyright 2006-2006 Patrick Meury
% SAM - Seminar for Applied Mathematics
% ETH-Zentrum
% CH-8092 Zurich, Switzerland

% Initialize constants
```

```
NREFS = 10;
F = @(x,varargin)zeros(size(x,1),1);
GD = @gD_LShap;

% Initialize mesh

Mesh = load_Mesh('Coord_LShap.dat','Elem_LShap.dat');
Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
Mesh = add_Edges(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
Mesh.BdFlags(Loc) = -1;

CNodes = transpose(1:size(Mesh.Coordinates,1));

% Prepare mesh for longest edge bisection

Mesh = init_LEB(Mesh);
for i = 1:NREFS
  Mesh = refine_hp(Mesh,CNodes);
end

% Generate mesh data structure for hp-FEM

Mesh_hp.Coordinates = Mesh.Coordinates;
Mesh_hp.Elements = Mesh.Elements;
Mesh_hp.ElemFlag = zeros(size(Mesh_hp.Elements,1),1);
Mesh_hp = add_Edges(Mesh_hp);
Loc = get_BdEdges(Mesh_hp);
Mesh_hp.BdFlags = zeros(size(Mesh_hp.Edges,1),1);
Mesh_hp.BdFlags(Loc) = -1;
Mesh_hp = add_Edge2Elem(Mesh_hp);

% Assign polynomial degrees and build dof maps

[EDofs,CDofs,ElemDeg] = assign_pdeg(Mesh_hp,CNodes,NREFS);
Elem2Dof = build_DofMaps(Mesh_hp,EDofs,CDofs);
pmax = max(ElemDeg);

% Build shape functions and quadrature rules

QuadRule_1D = gauleg(0,1,2*pmax);
Shap_1D = shap_hp([QuadRule_1D.x zeros(size(QuadRule_1D.x))],pmax);
QuadRule_2D = Duffy(TProd(QuadRule_1D));
Shap_2D = shap_hp(QuadRule_2D.x,pmax);

% Assemble global load vector and mass matrix

A = assemMat_hp(Mesh_hp,Elem2Dof,...
     @STIMA_Lapl_hp,QuadRule_2D,Shap_2D);
L = assemLoad_hp(Mesh_hp,Elem2Dof,QuadRule_2D,Shap_2D,F);

% Incoporate Dirichlet boundary conditions
```

```
64 [U,FreeDofs] = assemDir_hp(Mesh_hp,Elem2Dof,...
65                 -1,QuadRule_1D,Shap_1D,GD);
66 L = L - A*U;
67
68 % Solve the linear system
69
70 U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);
71
72 % Plot hp-FEM solution
73
74 plot_hp(U,Mesh_hp,Elem2Dof,pmax);
75 colorbar;
76
77 % Clear memory
78
79 clear all;
```

Here we shall give a description of the *hp*-FEM specific code segments.

- In line 23 the corner nodes `CNodes` are specified. The refinement strategy implemented in the function `refine_hp` (line 29) performs the largest edge bisection algorithm on all the elements sharing one of the nodes specified in `CNodes`.

- In the lines 43-47 the degrees of freedom are assigned to the elements and numbered. Firstly the function `assign_pdeg` assigns the maximum polynomial degree to every element, which is then stored in the vector of length `nElements` called `ElemDeg`. Furthermore the vectors `CDofs` and `EDofs` are assigned, which have length `nElements` and `nEdges` respectively and contain the number of degrees of freedom on the edges and in the interior. The polynomial degree stored in `ElemDeg` increases linearly away from the corner nodes. Based on that the number of degrees of freedom on every edge and the internal degrees of freedom are computed. Then the function `build_DofMaps` sets up the numbering of the degrees of freedom, which can then be found in the variable `Elem2Dof`. For details on the structure of this variable see below.

- In lines 49-54 the quadrature rules are set, for both 1D and 2D integrals. The accuracy of the quadrature rules is chosen in accordance to the maximal polynomial degree used in defining the finite element space. The shape functions in the quadrature points are pre-computed and then handled to all the functions using numerical integration. The shape functions are hierarchical (cf. Subsection ssect:shap_hp 2.3.5), therefore they can be pre-computed even though different polynomial degrees are used.

- The assembly of the stiffness matrix and the load vector (lines 56-60) are done in the same spirit as for other finite elements. The assembly routine `assemMat_hp` calls the function `STIMA_Lapl_hp` to compute the local element contributions and assembles the stiffness matrix making use of the numbering of the degrees of freedom stored in the variable `Elem2Dof`.

- A plot of the finite element solution can be done using the function `plot_hp` (lines 74-75).

The variable `Elem2Dof` that is assigned in line 46 is a struct consisting of the following fields:

- `EDofs`: is a cell array of length 3 containing information on the degrees of freedom on all 3 local edges; all of the cells are structs containing the fields
  - `Dofs`: cell containing `nEldges` arrays, which contains the numbers of the degrees of freedom on the corresponding edge
  - `nDofs`: vector of length `nEldges` with the number of degrees of freedom for every edge
  - `Dir`: vector of length `nEldges` containing information about the direction of the local edges; -1 if the local edge has the same orientation as the global one and 1 otherwise;
- `CDofs`: is a struct containing the fields
  - `Dofs`: cell containing `nElements` arrays, which contains the numbers of the inner degrees of freedom on the corresponding element
  - `nDofs`: vector of length `nElements` with the number of inner degrees of freedom for every element
- `tot_EDofs`: contains the total number of degrees of freedom on all edges of the mesh
- `tot_CDofs`: contains the total number of inner degrees of freedom on all elements of the mesh

To make this complicated structure more clear we shall give some examples. The input

`>> Elem2Dof.EDofs{1}.Dofs{6}`

returns the numbers of degrees of freedom on the local edge with number 1 on the element with number 6. The inner degrees of freedom are called similarly. The call

`>> Elem2Dof.CDofs.Dofs{6}`

returns the numbers of the inner degrees of freedom on the element number 6. The numbers of degrees of freedom can be obtained by replacing `Dofs{ 6}` in the above lines by `nDofs(6)`.

Similarly to the routines for *hp* finite elements one can also do just *p* refinement, which means rising the polynomial degrees without refining the mesh. In this case the degrees of freedom are assembled differently. The code piece below replaces the lines 43-47 of the code above.

```
EDofs = (pmax-1)*ones(size(Mesh.Edges,1),1);
if(pmax > 2)
  CDofs = (pmax-1)*(pmax-2)/2*ones(size(Mesh.Elements,1),1);
else
  CDofs = zeros(size(Mesh.Elements,1),1);
end

Elem2Dof = build_DofMaps(Mesh,EDofs,CDofs);
```

The number of degrees of freedom on every element is chosen such that the corresponding shape functions form a basis of the space of polynomials of total degree $p_{\max}$ on the element (cf. SO4 [?]). The variable `Elem2Dof` which contains all the information on the relation between the degrees of freedom and the elements is again a result of a call to the function `build_DofMaps`.

There are also driver routines for visualizing the convergence rates of the *hp*-FEM. Since the theory tells us that for these finite elements we have

$$\|u - u_N\| \leq C \ \exp(-b\sqrt[3]{N}), \tag{9.3}$$

where N denotes the degrees of freedom and u_N denotes the corresponding finite element solution. Therefore we use linear scaling on the x-axis to draw $\sqrt[3]{N}$ and logarithmic scaling on the y-axis that shows the corresponding error. The line then has to be a line with slope $-b$. The result of the driver routine `main_3`, which computes the discretization error on an L-shaped domain can be found in Figure fig:hp 9.2.

Figure 9.2: Convergence rates for *hp*-FEM fig:hp

Now we shall give a short explanation of the other driver routines contained in this folder.

- `main_1`: Produces a plot that shows the distribution of the polynomial degrees on the mesh. The polynomial degree increases linearly away from the corner nodes (cf. ssec:asseMathp 5.1.7).
- `main_2`: explained in detail above
- `main_3`: Computes the discretization error for different refinement steps to check convergence rates. The computational domain is L-shaped.
- `main_4`: Convergence rates for $p-$refinement with an analytic solution on an L-shaped computational domain.
- `main_5`: Solves the poisson equation on a square using *hp*-finite elements.
- `main_6`: Convergence rate for *hp*-FEM on a square.

- `main_7`: Convergence rate for p-refinement on a square.
- `main_8`: Convergence rate for p-refinement on an L-shaped domain.

9.5 Convection Diffusion problems

The implementation of special methods for solving convection diffusion equations with a dominating convective term can be found in the folder `Examples/DiffConv`. In addition to the finite element based methods there is also an implementation of the finite volume method availlable (cf. Chapter chapt:fvm_conv_diff 10).

The convection diffusion equations we look at have no time dependecy and are of the form

$$-a\Delta u + c \cdot \nabla u = f, \tag{9.4}$$

where $f : \Omega \to \mathbb{R}$ and $c : \Omega \to \mathbb{R}^2$. We assume a to be a real number. The problems we want to investigate have a dominating convective term, i.e. a should be small compared to the magnitude of the velocity field c. In the implementation the diffusion constant is of order 10^{-10}, while both components of c are of order 10^0. Different methods are implemented for the discretization of the convection term $c \cdot \nabla u$ in the equation. It is well known that for this kind of problem a standard finite element discretization yields very bad results even though the results on convergence rates apply. This is due to large constants in the estimates. The weak form is given by

$$\int_\Omega (a\nabla u \cdot \nabla v + (c \cdot \nabla u)v)\, dx = \int_\Omega fv\, dx. \tag{9.5}$$

Apart from the standard finite element approach we want to discuss two other methods to overcome the difficulties.

9.5.1 SUPG-method

The second approach is the so called *streamline upwind Petrov-Galerkin method (SUPG-method)*. For details see KA03 [?], S08 [?]. The basic idea is to use the identy

$$\sum_K \delta_K \langle -a\Delta u + c \cdot \nabla u, \tau(v)\rangle = \sum_K \delta_K \langle f, \tau(v)\rangle_K, \tag{9.6}$$ eq:supgg

where $\langle \cdot, \cdot \rangle_K$ denotes the L^2 inner product on the elment K. The identity above holds in the L^2 sense and requires further assumptions on the smoothness of the solution u. Furthermore the identity is true for any choice of parameters δ_K. The summation runs over all elements of the mesh. The mapping $\tau : L^2 \to L^2$ assigns a function to every test function v. This identity is added to the standard finite element discretization for stabilization purposes. The SUPG-method correlates to the choice $\tau(v) = c \cdot \nabla v$. For the special case of linear finite elements the bilinear form can then be written as

$$a(u, v) = \int_\Omega (a\nabla u \cdot \nabla v)\, dx + \langle c \cdot \nabla u, v\rangle + \sum_K \delta_K \langle c \cdot \nabla u, c \cdot \nabla v\rangle_K. \tag{9.7}$$ eq:supg

where $\langle \cdot, \cdot \rangle$ denotes the L^2 inner product on the computational domain Ω. The corresponding left hand side is given by the linear functional

$$l(v) = \langle f, v \rangle + \sum_K \delta_K \langle f, c \cdot \nabla v \rangle_K \tag{9.8}$$

9.5.2 Upwinding methods

Finally the third approach is done by so called upwinding schemes to discretize the convection term. The diffusion term is discretized in the standard finite element manner. Here we shall only give a short outline of the idea, more information can be found in [?]. To evaluate the stiffness matrix corresponding to the convective term $\langle c \cdot \nabla u, v \rangle$ we have to approximate integrals of the form

$$\int_T (c \cdot \nabla \varphi_j) \varphi_i \, dx, \tag{9.9}$$

eq:upw1

where φ_i are basis functions in the used finite element space. In the standard finite element approach this is done by Gaussian integration. Here we also use an integration rule on an element, which in general can be formulated as

$$\int_T f(x) \, dx \approx |T| \sum_i w_i f(x_i), \tag{9.10}$$

for given weights and evaluation points. In the case of the convective term the integrand from (9.9) can be inserted in the integration formula. Doing this the term $(c \cdot \nabla \varphi_j)(x_i)$ appears. For points in the interior of the element this value is simply computed using the shape functions. For points on the boundary $\nabla \varphi_j$ might have a jump. An upwinding triangle for the boundary point x is then defined as a triangle T where the vector $-c(x)$ points into. Possibly there are more upwinding triangles but in this case one can just use any of them. Then take $\nabla \varphi_j(x)$ to be the limiting value coming from the corresponding upwinding triangle.

This kind of upwinding method is implemented for linear and quadratic finite elements. We will have a closer look at the easier linear finite elements. The quadratic case is treated similarly, but involves more quadrature points. In the reference routine we want to look at the quadrature points are placed on the mid points of the edges and the corresponding weights are all 1/3. When computing the stiffness matrix on an element the convection contribution for two basis functions has to be evaluated as shown in 9.9. By the definition of the upwind quadrature the term $(c \cdot \nabla \varphi_j)(m)$ evaluates to the same value for both triangles T_1 and T_2 sharing the edge with the midpoint m. This can be expoited when computing the local stiffness matrices by computing the value $(|T_1| + |T_2|)/3 \; (c \cdot \nabla \varphi_j)(m) \varphi_i(m)$. The code for computing the stiffness on an element can be found below.

```
function UP_loc = STIMA_UPLFE(Vertices,vHandle, mass, varargin)
%   Copyright 2007 Holger Heumann
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland
```

```
UP_loc = zeros(3,3);

x=[0.5 0; 0.5 0.5; 0 0.5];

% Compute element mapping

a1 = Vertices(1,:);
a2 = Vertices(2,:);
a3 = Vertices(3,:);

m1=(a2+a1)/2;
m2=(a3+a2)/2;
m3=(a1+a3)/2;

bK = a1;
BK = [a2-bK; ...
      a3-bK];
det_BK = abs(det(BK));
inv_BK = inv(BK);

% Compute gradient of local shape functions

gradN=grad_shap_LFE(x);
for i=1:2:6
    gradN(:,[i i+1])=gradN(:,[i i+1])*inv_BK';
end

% Extract nodal vectors

v = -vHandle(x*BK + ones(3,1)*a1);

%Compute Lie derivative scheme for first
% midpoint using upwind quadrature
yhat = (m1+v(1,:)-bK)*inv_BK;
if(yhat(2) >= 0)
  elem= mass(1)/2*[-gradN(1,[1 2])*v(1,:)',...
              -gradN(1,[3 4])*v(1,:)',...
              -gradN(1,[5 6])*v(1,:)'];
  UP_loc(1,:) = elem;
  UP_loc(2,:) = elem;
end
if (yhat(2)==0)
   UP_loc(1,:)=1/2* UP_loc(1,:);
   UP_loc(2,:)=1/2* UP_loc(1,:);
end

% Compute Lie derivative scheme for second
% midpoint using upwind quadrature
yhat = (m2+v(2,:)-bK)*inv_BK;
if(sum(yhat) <= 1)
  elem = mass(2)/2*[-gradN(2,[1 2])*v(2,:)', ...
               -gradN(2,[3 4])*v(2,:)', ...
               -gradN(2,[5 6])*v(2,:)'];
  UP_loc(3,:) = UP_loc(3,:)+elem;
```

```
61    UP_loc(2,:) = UP_loc(2,:)+elem;
62  end
63  if (sum(yhat)==1)
64     UP_loc(2,:)=1/2* UP_loc(2,:);
65     UP_loc(3,:)=1/2* UP_loc(3,:);
66  end
67
68  % Compute Lie derivative scheme for third midpoint
69  % using upwind quadrature
70  yhat = (m3+v(3,:)-bK)*inv_BK;
71  if(yhat(1) >= 0)
72    elem = mass(3)/2*[-gradN(3,[1 2])*v(3,:)', ...
73                      -gradN(3,[3 4])*v(3,:)', ...
74                      -gradN(3,[5 6])*v(3,:)'];
75    UP_loc(3,:) = UP_loc(3,:)+elem;
76    UP_loc(1,:) = UP_loc(1,:)+elem;
77  end
78  if (yhat(1)==0)
79     UP_loc(1,:)=1/2* UP_loc(1,:);
80     UP_loc(3,:)=1/2* UP_loc(3,:);
81  end
82
83  return
```

The input parameter `mass` is a vector of length 3. The i-th entry contains one third of the area of the two elements sharing the local edge i. In the lines 38-51 the local computations are done for the first midpoint m_1, which lies in between the vertices a_1 and a_2. If the current triangle is an upwinding triangle with respect to m_1 we can compute the value

$$\frac{|T_1|+|T_2|}{3}(c\cdot\nabla\varphi_j)(m_1)\varphi_i(m_1). \tag{9.11}$$

The basis functions corresponding to the nodes a_1 and a_2 evaluate to $1/2$ while the one corresponding to a_3 is zero in m_1. This leads to the computations in lines 42-46. If both triangles are upwinding triangles the contribution has to be multiplied by $1/2$, which is done in the lines 49 and 50. The same computations are perfomed for the other midpoints.

The assembly of the global stiffness matrix for the upwinding discretization of the convective term is done in the routine `assemMat_UpLFE`.

9.5.3 The driver routines

As reference driver routine we chose `main_errorOFlfe`, the code of which can be found below.

```
% Initialize constants



JIG=2;
d1=1;                  %impact of supg modification
d2=1;
a=10^-10               %amount of diffusivity
NREFS =7;
```

```
% select test code
d=getData(5);

QuadRule = P706();

Mesh.Coordinates =d.Coordinates;
Mesh.Elements = d.Elements;
Mesh = add_Edges(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
sh.BdFlags(Loc) = d.boundtype;
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);

err=zeros(NREFS,5);
err1=zeros(NREFS,5);
h=zeros(NREFS,1);
Dofs=zeros(NREFS,1);

for i = 1:NREFS
 %refine Mesh
 Mesh = refine_REG(Mesh);
 Mesh=add_Edge2Elem(Mesh);


 %Laplace
 A = assemMat_LFE(Mesh,@STIMA_Lapl_LFE,P303());

 UP4=assemMat_UpLFE(Mesh,d.V_Handle);
 B_supg=assemMat_LFE(Mesh,@STIMA_SUPG_LFE,P303(),...
           d.V_Handle,a,d1,d2);
 B=assemMat_LFE(Mesh,@STIMA_Conv_LFE,d.V_Handle,P704());

 A_u4=a*A+(UP4);
 A_supg=a*A+B+B_supg;
 A_s=a*A+B;

 L = assemLoad_LFE(Mesh,P303(),d.SOL_Handle,a);
 L_supg=assemLoad_LFE_SUPG(Mesh,P303(),...
          d.V_Handle,d.SOL_Handle,a,d1,d2);

 %Incorporate Dirichlet boundary data

 [U_s,FreeDofs] = assemDir_LFE(Mesh,[-1 -2],d.U_EX_Handle,a);
 U_u4=U_s;
 U_supg=U_s;

 L_u4 = L - A_u4*U_u4;
 L_s = L - A_s*U_s;
 L_supg = L+L_supg - A_supg*U_supg;

 % Solve the linear system

 U_u4(FreeDofs) = A_u4(FreeDofs,FreeDofs)\L_u4(FreeDofs);
```

```
63   U_s(FreeDofs) = A_s(FreeDofs,FreeDofs)\L_s(FreeDofs);
64   U_supg(FreeDofs)=A_supg(FreeDofs,FreeDofs)\L_supg(FreeDofs);
65
66   err1(i,4)=L2Err_LFE(Mesh,U_u4,P706(),d.U_EX_Handle,0,a);
67   err1(i,5)=L2Err_LFE(Mesh,U_s,P706(),d.U_EX_Handle,0,a);
68   err1(i,6)=L2Err_LFE(Mesh,U_supg,P706(),d.U_EX_Handle,0,a);
69
70   Dofs(i)=size(Mesh.Coordinates,1);
71   plot_LFE(U_u4,Mesh); colorbar
72 end
73
74 %%%%%%%%%%%%%%%%
75 % plot computed errors...
76 %%%%%%%%%%%%%%%%
77
78 clear all;
```

The driver routine above compares the convergence rates of 3 different methods. The stiffness matrix for the standard finite element discretization is computed as the sum of the discretized convective and diffusive term in line 44. The right hand side for this method is assembled in line 46.

The matrix that is added to the original stiffness matrix based on the additional term apearing in (9.7) correlated with the SUPG method is computed in line 38. The final system matrix is then computed in line 43. The right hand side also consists of two contributions now, the original right hand side and the right hand side of (9.6). The computation is done in the lines 47 and 58. The solution is then computed in line 64.

The stiffness matrix resulting from an upwind discretization is computed in line 37, the corresponding finite element solution is computed in line 62.

The plot of the errors is omited in this code sample since it does not involve any code that is related to the presented methods. The output of the `main_error0Flfe` can be found in Figure 9.3. Other driver routines in this folder shall be explained in the following.

- `LFE_invNorms` compares different discretization methods for different amount of diffusivity.
- `main_error` compares different discretization methods for different mesh widths. For upwinding the vertices are used as quadrature points.
- `main_error0Fqfe` compares different discretization methods for different mesh widths. Quadratic finite elments are used and the quadrature points are the vertices, the midpoints and one center point.
- `QFE_invNorms` compares different discretization methods for different amount of diffusivity using quadratic finite elements.

Figure 9.3: Convergence rates for a convection diffusion problem fig:convdiff

Chapter 10

Finite Volume Method

chapt:fvm_conv_diff

10.1 Finite Volume Code for Solving Convection/Diffusion Equations

sect:fvm_conv_diff

This section details code written for the LehrFEM library during the summer of 2007 by Eivind Fonn, for solving convection/diffusion equations using the Finite Volumes approach.

10.1.1 Background

The equation in question is

$$-\nabla \cdot (k\nabla u) + \nabla \cdot (cu) + ru = f$$

on some domain $\Omega \subset \mathbb{R}^2$. Here, $k, r, f : \Omega \to \mathbb{R}$, and $c : \Omega \to \mathbb{R}^2$. k is the diffusivity and should be positive everywhere. c is the velocity field.

In addition to the above, one can specify Dirichlet and Neumann boundary conditions on various parts of $\partial\Omega$.

10.1.2 Mesh Generation and Plotting

The FV mesh generation builds upon FE meshes. Given a FE mesh, use the `add_MidPoints` function to add the data required for the dual mesh:

```
>> mesh = add_MidPoints(mesh,method);
```

`method` is a string specifying which dual mesh method to use. The two options are `barycentric` and `orthogonal`. If `method` is not specified, the `barycentric` method will be used. For the `orthogonal` method, the mesh cannot include any obtuse triangles, and if any such triangle exists, an error will occur.

The full mesh can be plotted using the following call:

```
>> plot_Mesh_LFV(mesh);
```

10.1.3 Local computations

As in the case of finite elements the local stiffness matrix is computed for every element. This is done in three different functions, which will be discussed below

Diffusion term

In the function `STIMA_GenLapl_LFV` the local stiffness matrix comming from the diffusive term $-\nabla \cdot (k\nabla u)$ is computed. The code can be found below.

```
function aLoc = STIMA_GenLapl_LFV(vertices,midPoints,...
      center,method,bdFlags,kHandle,varargin)
%   Copyright 2007-2007 Eivind Fonn
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

% Compute required coefficients

m = zeros(3,3);
m(1,2) = norm(midPoints(1,:)-center);
m(1,3) = norm(midPoints(3,:)-center);
m(2,3) = norm(midPoints(2,:)-center);
m = m + m';

mu = zeros(3,3);
mu(1,2) = kHandle((midPoints(1,:)+center)/2);
mu(1,3) = kHandle((midPoints(3,:)+center)/2);
mu(2,3) = kHandle((midPoints(2,:)+center)/2);
mu = mu + mu';

mum = mu.*m;

sigma = zeros(3,2);
sigma(1,:) = confw(novec(vertices([2 3],:)),vertices(1,:)-vertices(2,:));
sigma(2,:) = confw(novec(vertices([1 3],:)),vertices(2,:)-vertices(3,:));
sigma(3,:) = confw(novec(vertices([1 2],:)),vertices(3,:)-vertices(1,:));
sigma(1,:) = sigma(1,:)/trialtv(vertices);
sigma(2,:) = sigma(2,:)/trialtv([vertices(2,:);vertices([1 3],:)]);
sigma(3,:) = sigma(3,:)/trialtv([vertices(3,:);vertices(1:2,:)]);

% Compute stiffness matrix

aLoc = zeros(3,3);
for i=1:3
   for j=1:3
      % Calculate aLoc(i,j)
      for l=[1:(i-1) (i+1):3]
         aLoc(i,j) = aLoc(i,j) -...
         mum(i,l)*dot(sigma(j,:),...
           confw(novec([midPoints(midpt([i l]),:);center]),...
           vertices(l,:)-vertices(i,:)));
      end
   end
end
```

```
47 % Helping functions
48
49 function s = midpt(idx)
50    idx = sort(idx);
51    if idx==[1 2]
52       s = 1;
53    elseif idx==[2 3]
54       s = 2;
55    else
56       s = 3;
57    end
58 end
59
60 function v = confw(v,t)
61    if dot(t,v)<0
62       v = -v;
63    end
64 end
65
66 function out = novec(v)
67    out = v(2,:)-v(1,:);
68    if norm(out) ~= 0
69       out = [-out(2) out(1)]/norm(out);
70    end
71 end
72
73 function out = trialtv(v)
74      out = norm(v(2,:)-v(1,:) + ...
75            dot(v(3,:)-v(2,:),v(1,:)-v(2,:))/...
76            (norm(v(3,:)-v(2,:))^2)*(v(3,:)-v(2,:)));
77 end
78
79 end
```

In the lines 10-14 the lengths of the pieces of the dual mesh on the current element are computed. In the lines 16-20 the approximation of the function k on the boundary of the dual element is computed. The lines 24-30 compute the gradient of the basis functions on the current element. Finally the stiffness matrix is computed in the lines 34-45. Therefore the approximation of the integrals over the pieces of boundary of the dual mesh are added using the computed gradients of the basis functions.

In the lines 47-77 helping functions are implemented. They are used to compute the gradient and to find the correct number of the midpoint lying on an edge.

For the discretization of the convective term the function `STIMA_GenGrad_LFV` is used. The code of which can be found below.

```
function aLoc = STIMA_GenGrad_LFV(vertices,midPoints,center,method,...
           bdFlags,cHandle,kHandle,rHandle,conDom,varargin)
%   Copyright 2007-2007 Eivind Fonn
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland
```

```
% Compute required coefficients

m = zeros(3,3);
m(1,2) = norm(midPoints(1,:)-center);
m(1,3) = norm(midPoints(3,:)-center);
m(2,3) = norm(midPoints(2,:)-center);
m = m + m';

qr = gauleg(0, 1, 4, 1e-6);
c = zeros(3,2);
for i=1:length(qr.w)
    c(1,:) = c(1,:) + qr.w(i)*cHandle(midPoints(1,:)+...
            qr.x(i)*(center-midPoints(1,:)));
    c(2,:) = c(2,:) + qr.w(i)*cHandle(midPoints(2,:)+...
            qr.x(i)*(center-midPoints(2,:)));
    c(3,:) = c(3,:) + qr.w(i)*cHandle(midPoints(3,:)+...
            qr.x(i)*(center-midPoints(3,:)));
end

gamma = zeros(3,3);
gamma(1,2) = dot(confw(novec([midPoints(1,:); center]),...
      vertices(2,:)-vertices(1,:)),c(1,:));
gamma(2,3) = dot(confw(novec([midPoints(2,:); center]),...
      vertices(3,:)-vertices(2,:)),c(2,:));
gamma(1,3) = dot(confw(novec([midPoints(3,:); center]),...
      vertices(3,:)-vertices(1,:)),c(3,:));
gamma = gamma - gamma';

if conDom
    mu = ones(3,3);
    mu(1,2) = kHandle((midPoints(1,:)+center)/2);
    mu(2,3) = kHandle((midPoints(2,:)+center)/2);
    mu(1,3) = kHandle((midPoints(3,:)+center)/2);
    mu = mu + mu';

    d = zeros(3,3);
    d(1,2) = norm(vertices(1,:)-vertices(2,:));
    d(2,3) = norm(vertices(2,:)-vertices(3,:));
    d(1,3) = norm(vertices(3,:)-vertices(1,:));
    d = d + d';

    z = gamma.*d./mu;
    r = zeros(3,3);
    for i=1:3, for j=1:3
        r(i,j) = rHandle(z(i,j));
    end; end;
else
    r = ones(3,3)/2;
end

a = m.*gamma.*r;
b = m.*gamma;
```

```
62 % Compute stiffness matrix
63 
64 aLoc = b'-a';
65 for i=1:3
66      aLoc(i,i) = sum(a(i,:));
67 end
68 
69 aLoc = aLoc';
70 
71 % Helping functions
72 
73 function v = confw(v,t)
74      if dot(v,t)<0
75           v = -v;
76      end
77 end
78 
79 function out = novec(v)
80      out = v(2,:)-v(1,:);
81         if norm(out) ~= 0
82             out = [-out(2) out(1)]/norm(out);
83         end
84 end
85 
86 end
```

In the lines 10-14 the lengths of the pieces of the dual mesh on the current element are computed.

10.1.4 Assembly

Assembly of the stiffness matrices and load vector is done much the same way as in FE, i.e. assemble the stiffness matrices term-by-term, sum them up to get the full matrix. Assemble the load vector in the normal way, then incorporate Neumann boundary data, followed by Dirichlet boundary data.

For the lazy, there is a wrapper function for all of the above:

```
>> [A,U,L,fd] = assemMat_CD_LFV(mesh,k,c,r,d,n,f,cd,out);
```

This function assembles all the matrices and vectors required at once, without any hassle. Here follows a list of the input arguments:

- `mesh` The mesh struct (see last section).
- `k` Function handle for the k-function.
- `c` Function handle for the c-function.
- `r` Function handle for the r-function.
- `d` Function handle for the Dirichlet boundary data.
- `n` Function handle for the Neumann boundary data.

- `f` Function handle for the f-function.
- `cd` If equal to 1, specifies that the problem is convection-dominated, and that appropriate upwinding techniques should be applied. If 0, specifies that the problem is diffusion-dominated. If left out, the program itself will determine whether the problem is convection- or diffusion-dominated.
- `out` If equal to 1, the program will write output to the console. This is useful for large meshes, to keep track of the program.

If any of the function handles are left out or set equal to the empty matrix, it is assumed they are zero and will play no part.

The output are as follows:

- `A` The stiffness matrix.
- `U` The solution vector, containing Dirichlet boundary data.
- `L` The load vector.
- `fd` A vector with the indices of the free nodes.

From the above, the solution can be obtained by:

```
>> U(fd) = A(fd,fd)\L(fd);
```

The solution vector `U` contains the approximate values of the solution u at the nodes given in the mesh.

10.1.5 Error Analysis

Given a mesh `mesh`, a FV solution vector `U` and a function handle `u` to the exact solution, the error can be measured in three different norms ($\|\cdot\|_1$, $\|\cdot\|_2$ and $\|\cdot\|_\infty$) using the following calls:

```
>> L1err = L1Err_LFV(mesh,U,qr,u);
>> L2err = L2Err_LFV(mesh,U,qr,u);
>> Linferr = LInfErr_LFV(mesh,U,u);
```

Here, `qr` is any quadrature rule for the reference element, i.e. `qr = P7O6()`.

10.1.6 File-by-File Description

Here follows a list of the files which were written as part of this project, and a description of each.

- `Assembly/assemDir_LFV.m`
  `[U,fd] = assemDir_LFV(mesh,bdFlags,fHandle)` incorporates the Dirichlet boundary conditions given by the function `fHandle` in the Finite Volume solution vector `U`. `fd` is a vector giving the indices of the vertices with no Dirichlet boundary data (i.e. the free vertices).

- `Assembly/assemLoad_LFV.m`
  `L = assemLoad_LFV(mesh,fHandle)` assembles the load vector `L` for the data given by the function `fHandle`.

- `Assembly/assemMat_CD_LFV.m`
  Wrapper function for assembly of convection/diffusion problems. See above for description.

- `Assembly/assemMat_LFV.m`
  `A = assemMat_LFV(mesh,fHandle,varargin)` assembles the global stiffness matrix `A` from the local element contributions given by the function `fHandle` and returns it in a sparse format. `fHandle` is passed the extra input arguments given by `varargin`.

- `Assembly/assemNeu_LFV.m`
  `L = assemNeu_LFV(mesh,bdFlags,L,qr,fHandle)` incorporates Neumann boundary conditions in the load vector `L` as given by the function `fHandle`. `qr` is any 1D quadrature rule. Note: In the general diffusion problem $-\nabla \cdot (k\nabla u) = f$, with Neumann boundary data $\frac{\partial u}{\partial n} = g$, the function h you need to pass to the Neumann assembly function is $h(x) = g(x)k(x)$.

- `Element/STIMA_GenGrad_LFV.m`
  `A = STIMA_GenGrad_LFV(v,mp,cp,m,bd,cH,kH,rH,cd)` calculates the local element contribution to the stiffness matrix from the term $\nabla \cdot (cu)$. Here, `v` is a 3-by-2 matrix giving the vertices of the element, `mp` and `cp` give the midpoints on each of the edges as well as the centerpoint (that is, the dual mesh geometry). `m` is the method used to generate the dual mesh (not used at the current time), `bd` are boundary flags for the edges (not used at the current time). `cH` is the c-function, `kH` is the k-function from the diffusion term and `rH` is a proper upwinding function. `cd` is 1 if the problem is convection-dominated and 0 otherwise. `kH` and `rH` are only used if `cd=1`. If not, they need not be specified.

- `Element/STIMA_GenLapl_LFV.m`
  `A = STIMA_GenLapl_LFV(v,mp,cp,m,bd,kH)` calculates the local element contribution to the stiffness matrix from the term $-\nabla \cdot (k\nabla u)$. The arguments are the same as above.

- `Element/STIMA_ZerOrd_LFV.m`
  `A = STIMA_ZerOrd_LFV(v,mp,cp,m,bd,rH)` calculates the local element contribution to the stiffness matrix from the term ru. The arguments are the same as above, except that `rH` is the function handle for the r-function.

- `Errors/L1Err_LFV.m`
  `e = L1Err_LFV(mesh,U,qr,fHandle)` calculates the discretization error in the norm $\|\cdot\|_1$, between the finite volume approximation given by `U` (the solution vector), and the exact solution given by the function `fHandle`. `qr` is any appropriate quadrature rule on the reference element.

L2Err_LFV

- `Errors/L2Err_LFV.m`
  `e = L2Err_LFV(mesh,U,qr,fHandle)` calculates the discretization error in the norm $\|\cdot\|_2$. All the arguments are as above.

LInfErr_LFV

- Errors/LInfErr_LFV.m
  `e = LInfErr_LFV(mesh,U,fHandle)` calculates the discretization error in the norm $\|\cdot\|_\infty$. All the arguments are as above.
- MeshGen/add_MidPoints.m
  `mesh = add_MidPoints(mesh,method)` incorporates the dual mesh data required for the finite volume method into the mesh `mesh`. `method` may be either `'barycentric'` or `'orthogonal'`. See earlier description.
- Plots/plot_Mesh_LFV.m
  `plot_Mesh_LFV(mesh)` plots the base mesh and the dual mesh.

10.1.7 Driver routines

The driver routines for solving convection diffusion equations can be found in the folder `/Examples/FVOL`. There are 3 different example problems implemented, which we shall discuss in more detail. The files `exp_*` are functions, which solve the problems for the given input parameters. The functions `exp_run_*` contain executeable code and call the corresponding `exp_*` function to solve the problem for different settings. Below the code of `exp_run_1` is included.

Example 1

```
exp1_epsrange = [1e-4 5e-4 1e-3 5e-3 1e-2 5e-2 1e-1 5e-1];
refrange = 1:6;
exp1_meshes = [];
exp1_solutions = [];
exp1_errors_L1 = [];
exp1_errors_L2 = [];
exp1_errors_Linf = [];
exp1_hrange = [];

for r=1:size(refrange,2)
    for e=1:size(exp1_epsrange,2)
        disp(['Running test ' num2str((e-1)+...
              size(exp1_epsrange,2)*(r-1)+1) ' of ' ...
              num2str(size(exp1_epsrange,2)*size(refrange,2))]);
        [errs, mw, msh, u] =...
              exp_1(exp1_epsrange(e), refrange(r), 0, 0);
        exp1_errors_L1(e,r) = errs(1);
        exp1_errors_L2(e,r) = errs(2);
        exp1_errors_Linf(e,r) = errs(3);
        exp1_solutions(e,r).sol = u;
    end

    exp1_meshes(r).Coordinates = msh.Coordinates;
    exp1_meshes(r).Elements = msh.Elements;
    exp1_meshes(r).ElemFlag = msh.ElemFlag;
    exp1_meshes(r).Edges = msh.Edges;
    exp1_meshes(r).Vert2Edge = msh.Vert2Edge;
    exp1_meshes(r).Max_Nodes = msh.Max_Nodes;
    exp1_meshes(r).BdFlags = msh.BdFlags;
    exp1_meshes(r).CenterPoints = msh.CenterPoints;
```

```
31      exp1_meshes(r).Type = msh.Type;
32      exp1_meshes(r).MidPoints = msh.MidPoints;
33      exp1_hrange(r) = mw;
34  end
35
36
37
38  save 'exp1_lfv.mat' exp1_*
```

There are two loops that start in the lines 10 and 11 control the variable ϵ and the number of mesh refinements respectively. A call to the function `exp_1` in line 15 computes the finite volume solution of the equation

$$-\epsilon\Delta u + u_x = 1, \tag{10.1}$$

on the triangle $\Omega = \{(x, y) \mid 0 \leq x \leq 1,\ -x \leq y \leq x\}$. In every step different values for ϵ and different mesh widths (refinement steps) are handed to `eps_1`. The other two input parameters are flags, which are here not set to suppress output and plotting. The code of the function `eps_1` will be explained in more detail below.

In the last line all the results including errors, solution and mesh information are saved to the file `exp1_lfv.mat`.

Now we shall discuss the solver routine for this example, the code can be found below.

```
function [errs, mw, msh, U] = exp_1(eps, nRef, plotting, output)
if nargin < 2 || isempty(nRef)
    nRef = 0;
end
if nargin < 3 || isempty(plotting)
    plotting = 0;
end
if nargin < 4 || isempty(output)
    output = 0;
end

out('Generating mesh.');
msh = load_Mesh('mesh_exp_1_coords.dat', 'mesh_exp_1_elements.dat');
msh.ElemFlag = ones(size(msh.Elements,1),1);
msh = add_Edges(msh);
loc = get_BdEdges(msh);
msh.BdFlags = zeros(size(msh.Edges,1),1);
msh.BdFlags(loc) = -ones(size(loc));
for i=1:nRef
    msh = refine_REG(msh);
end
msh = add_MidPoints(msh, 'barycentric');

u = @(x,varargin)(x(1)-(exp(-(1-x(1))/eps)-...
     exp(-1/eps))/(1-exp(-1/eps)));
k = @(x,varargin)eps;
c = @(x,varargin)[1 0];
r = [];
d = @(x,varargin)(u(x,varargin));
```

```
30 n = [];
31 f = @(x,varargin)1;
32 conDom = 1;
33
34 [A, U, L, fd] =...
35     assemMat_CD_LFV(msh, k, c, r, d, n, f, conDom, output);
36
37 out('Solving system.');
38 U(fd) = A(fd,fd)\L(fd);
39
40 if plotting
41     out('Plotting.');
42     plot_LFE(U, msh);
43 end
44
45 out('Calculating errors.');
46 errs = [L1Err_LFV(msh, U, P706(), u);...
47      L2Err_LFV(msh, U, P706(), u); LInfErr_LFV(msh, U, u)];
48 out(['Discretization errors: ' num2str(errs(1)) ',...
49     ' num2str(errs(2)) ', ' num2str(errs(3)) '.']);
50
51 out('Calculating meshwidth.');
52 mw = get_MeshWidth(msh);
53
54
55 % Helping functions
56 function out(text, level)
57     if output
58         if nargin < 2 || isempty(level),
59             level = 1;
60         end
61         for i=1:level
62             text = ['- ' text];
63         end
64         disp([text]);
65     end
66 end
67 end
```

- lines 2-10: the output and plotting flags are set, if they are not part of the input already
- lines 12-22: generate and refine mesh, in line 22 the midpoints for the finite volume method are computed
- lines 23-32: define all the input variables for the finite volume method, i.e. the functions for defining the differential equation and for defining the Dirichlet and Neumann data; the variable `conDom` is a flag specifying if the problem is convergence dominated
- lines 40-45: plot the solution if the `plotting` flag is set
- lines 47-54: compute the discretization errors and print them in case the flag `output` is set

- lines 57-68: the function `out` controls the output in the command window

Furthermore there the function `main_1` solves uses the `eps_1` function to solve one example problem. The input parameters can be given to the function in the beginning. The solution and the underlying mesh will be plotted.

Example 2

The structure of Example 2 is the same as in the first one. The equation that is solved is given by

$$-\epsilon\Delta u + u_x + u_y = 0, \tag{10.2}$$

with dirichlet boundary condition on the square $[0,1]^2$. The exact solution we are looking for is in this case given by

$$u(x,y) = \begin{cases} 1 & \text{if } x > y \\ 0.5 & \text{if } x = y \\ 0 & \text{if } x < y. \end{cases} \tag{10.3}$$

On the boundary Dirichlet conditions are set.

Example 3

For the third example there is no wrapper function `eps_3_run` but only the function `eps_3`, which solves the problem

$$-\Delta u + \nabla \cdot (u\nabla\Psi) = 0, \tag{10.4}$$

where $\Psi = \Psi(r) = \frac{1}{1+e^{a(r-1)}}$. The radius is given by $r = \sqrt{x^2+y^2}$ and the computational domain is given by $[0,1]^2$. The last two input parameters of the function are used to control plotting and output. The input variable `a` is needed in the definition of the function Ψ. The variable `nRef` determines the number of refinement steps.

For this example there is, as for the experiments before the function `main_3`, which solves the problem for the parameters specified in the beginning of the file.