S. Mishra
L. Scarabosio
J. Šukys

Spring Term 2014

Numerical Methods for Partial Differential Equations

ETH Zürich
D-MATH

Homework Problem Sheet 3

Introduction.

This assignment if fully devoted to the Finite Element Method in 2D.
The first problem concerns the implementation of Linear Finite Elements for the diffusion equation with Dirichlet boundary conditions. The implementation will be developed step by step under the perspective of "finite element assembly" of the Galerkin matrix and the right-handside error.
The second problem is aimed to discretize the same Dirichlet problem but this time by means of *quadratic finite elements*.
Particular attention is given in the problems to the convergence properties of the solution.

In the online handout you can find the mesh data structures that you need to test your routines and perform the convergence studies.
Every file `*.mat` refers to a mesh and contains a struct. For the convergence studies, the meshes are ordered in increasing order for number of degrees of freedom.
Each struct, let's call it `Mesh`, contains the following fields:

- `Mesh.Coordinates`: $N_V \times 2$ array, with N_V the number of vertices, containing the vertex coordinates;
- `Mesh.Edges`: $N_E \times 2$ array, with N_E the number of edges; the i-th row contains the *indices* of the two vertices connected by the edge i;
- `Mesh.Elements`: $N_{El} \times 3$ array, with N_{El} the number of elements; the i-th row contains the *indices* of the three vertices of the element i;
- `Mesh.BdFlags`: $N_E \times 2$ array, with N_E the number of edges, containing the edge boundary flags; the convention is that the boundary flag is 0 is the edge is an interior edge, it is negative for boundary condition flags;
- `Mesh.Vert2Edge`: $N_V \times N_V$ array, with N_V the number of vertices; `Mesh.Vert2Edge(i,j)` contains the index of the edge connecting the vertices i and j.

To load the mesh data structures, in MATLAB you can use the command `load`, in Python the code would be

```
from scipy.io import loadmat
Mesh = loadmat(path_to_file)
print Mesh['Coordinates']
print Mesh['Edges']
```

Problem 3.1 2D Linear Finite Elements (Core problem)

We consider the problem

$$-\operatorname{div}(D(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{3.1.1}$$

$$u(\boldsymbol{x}) = g(\boldsymbol{x}) \quad \text{on } \partial\Omega \tag{3.1.2}$$

where D is uniformly positive and bounded in Ω, g is a continuous function of $\partial\Omega$ and $f \in L^2(\Omega)$.

We solve (3.1.1)-(3.1.2) by means of Galerkin discretization based on piecewise linear finite elements on triangular meshes of Ω.

(3.1a) Write the variational formulation for (3.1.1)-(3.1.2), specifying the bilinear form and the linear form.

Solution: We multiply both the left handside and right handside of (3.1.1) by a test function v. Applying Green's formula for integration by parts on the left handside we get:

$$-\int_\Omega \operatorname{div}(D(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega D(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} D(\boldsymbol{x})\frac{\partial u}{\partial \boldsymbol{n}}(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}.$$

Since u satisfies Dirichlet boundary conditions, the test functions belong to $H_0^1(\Omega)$ and thus the boundary integral in the above expression vanishes. The variational formulation results then:

$$\text{Find } u \in V = \left\{w \in H^1(\Omega) : w = g \text{ on } \partial\Omega\right\} \text{ such that}$$

$$\underbrace{\int_\Omega D(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{\mathsf{a}(u,v)} = \underbrace{\int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{l(v)} \text{ for all } v \in H_0^1(\Omega)$$

(3.1b) Show that the solution to the variational formulation in subproblem (3.1a) exists and is unique when $g = 0$.

Solution: We have to show that the bilinear form is continuous on $H_0^1(\Omega) \times H_0^1(\Omega)$ and coercive on $H_0^1(\Omega)$, and that the linear form is continuous on $H_0^1(\Omega)$.

- Continuity of $\mathsf{a}(\cdot,\cdot)$:

  $$|\mathsf{a}(u,v)| \le \|D(\boldsymbol{x})\|_{L^\infty(\Omega)}\,|u|_{H^1(\Omega)}|v|_{H^1(\Omega)},$$

  where we used the Cauchy-Schwarz inequality and the fact that the coefficient D is bounded in Ω.

- Coercivity of $\mathsf{a}(\cdot,\cdot)$:

  $$|\mathsf{a}(u,u)| \ge \inf_{\boldsymbol{x}\in\Omega} D(\boldsymbol{x})\,|u|^2_{H^1(\Omega)},$$

  where we used that D is uniformly positive on Ω.

- Continuity of the $l(\cdot)$:

  $$|l(v)| \le \|f\|_{L^2(\Omega)}\|u\|_{L^2(\Omega)} \le C_P\|f\|_{L^2(\Omega)}|u|_{H^1(\Omega)},$$

  obtained using the Cauchy-Schwarz inequality and the Poincaré inequality (with Poincaré constant C_P).

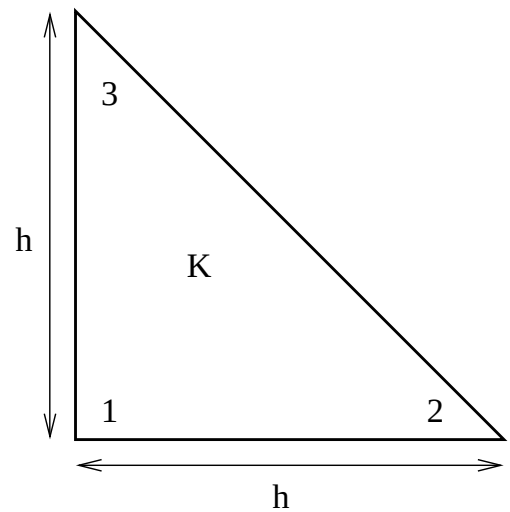


Figure 3.1: Reference element for 2D linear finite elements ($h = 1$).

(3.1c) Implement the function

```
shap = shap_LFE(x)
```

which computes the the value of the three local shape functions $\lambda_i(\boldsymbol{x})$, $i = 1, 2, 3$, on the reference element depicted in Fig.3.1 at the points x (a $N \times 2$ matrix, where each rows contains the coordinates of a point), and returns the values in the $N \times 3$ matrix `shap` (each row corresponding to the evaluation of the basis functions in a point).

Solution: See Listing 3.1 for the code.

Listing 3.1: Implementation for `shap_LFE`

```
function shap = shap_LFE(x)

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

  shap = zeros(size(x,1),3);

  shap(:,1) = 1-x(:,1)-x(:,2);
  shap(:,2) = x(:,1);
  shap(:,3) = x(:,2);

return
```

(3.1d) Implement the function

```
shap = grad_shap_LFE(x)
```

which returns the values of the derivatives of local shape functions $\lambda_i(\boldsymbol{x})$, $i = 1, 2, 3$. The input argument `x` follows the same convention as in shap_LFE(x), while the output `shap` is a $N \times 6$ matrix containing the gradients of the shape functions evaluated at the N points (the first two columns contain the gradient of λ_1, and so on).

Solution: See Listing 3.2 for the code.

Listing 3.2: Implementation for `grad_shap_LFE`

```
function grad_shap = grad_shap_LFE(x)

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

  % Initialize constants

  nPts = size(x,1);

  % Preallocate memory

  grad_shap = zeros(nPts,6);

  % Compute values of gradients

  grad_shap(:,1:2) = -ones(nPts,2);
  grad_shap(:,3) = ones(nPts,1);
  grad_shap(:,6) = ones(nPts,1);

return
```

(3.1e) Implement the routine STIMA_Heat_LFE to compute the element (stiffness) matrices. The function header is

```
Aloc = STIMA_Heat_LFE(Vertices, QuadRule, FHandle)
```

Here, Vertices is a 3×2-vector providing the coordinates of the element vertices, `QuadRule.w` is a vector with quadrature weights and `QuadRule.x` is a vector with quadrature points relative to $\hat{K}$. The function should return a 3×3 matrix Aloc containing the element stiffness matrix. FHandle is a handle to the function D.

HINT: Use `grad_shap_LFE` to compute the gradients of the shape functions.

Solution: See Listing 3.3 for the code.

Listing 3.3: Implementation for `STIMA_Heat_LFE`

```
function Aloc = STIMA_Heat_LFE(Vertices,QuadRule,FHandle)

%   Copyright 2005-2005 Patrick Meury & Kah Ling Sia
```

```
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

  nPoints = size(QuadRule.w,1);

  % Preallocate memory

  Aloc = zeros(3,3);

  % Compute element mapping

  bK = Vertices(1,:);
  BK = [Vertices(2,:)-bK; Vertices(3,:)-bK];
  inv_BK = inv(BK);
  det_BK = abs(det(BK));

  TK = det_BK*transpose(inv_BK)*inv_BK;

  x = QuadRule.x*BK+ones(nPoints,1)*bK;

  % Compute element stiffness matrix

  FVal = FHandle(x);
  grad_N = grad_shap_LFE(QuadRule.x);

  Aloc(1,1) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,1:2)*TK),2));
  Aloc(1,2) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,3:4)*TK),2));
  Aloc(1,3) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,5:6)*TK),2));
  Aloc(2,2) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,3:4)).*(grad_N(:,3:4)*TK),2));
  Aloc(2,3) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,3:4)).*(grad_N(:,5:6)*TK),2));
  Aloc(3,3) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,5:6)).*(grad_N(:,5:6)*TK),2));

  % Update lower triangular part

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

```

```
return
```

(3.1f) Implement the routine `LOAD_LFE` to compute the *element vector*. The function header is

```
Lloc = LOAD_LFE(Vertices, QuadRule, FHandle)
```

and follows the same convention as `STIMA_Heat_LFE`.

Solution: See Listing 3.4 for the code.

Listing 3.4: Implementation for `LOAD_LFE`

```
function Lloc = LOAD_LFE(Vertices,QuadRule,FHandle)

% Initialize constants

nPoints = size(QuadRule.w,1);

% Preallocate memory

Lloc = zeros(3,1);

% Compute element mapping

bK = Vertices(1,:);
BK = [Vertices(2,:)-bK; Vertices(3,:)-bK];
inv_BK = inv(BK);
det_BK = abs(det(BK));

x = QuadRule.x*BK+ones(nPoints,1)*bK;

% Compute element load vector

FVal = FHandle(x);
N = shap_LFE(QuadRule.x);

Lloc(1) = sum(QuadRule.w.*FVal.*N(:,1))*det_BK;
Lloc(2) = sum(QuadRule.w.*FVal.*N(:,2))*det_BK;
Lloc(3) = sum(QuadRule.w.*FVal.*N(:,3))*det_BK;
```

(3.1g) Implement a function

```
A = assemMat_LFE(Mesh,EHandle,varargin)
```

that assembles the Galerkin matrix A given the mesh structure `Mesh` and a routine `EHandle` to assemble the element matrix. In your implementation make a call `EHandle(Vertices, varargin{:})`, where `Vertices` are the coordinates of an element K_i. Here for `EHandle`

= STIMA_Heat_LFE the variable argument list varargin should carry the parameters QuadRule, FHandle.

HINT: Use the sparse format to store the matrix A.

Solution: See Listing 3.5 for the code.

Listing 3.5: Implementation for assemMat_LFE

```
function A = assemMat_LFE(Mesh,EHandle,varargin)

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

  % Initialize constants

  nElements = size(Mesh.Elements,1);

  % Preallocate memory

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

  % 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,varargin{:});

    % Add contributions to stiffness matrix

    Iloc = idx(ones(3,1),:)';
    I(loc) = Iloc(:);

    Jloc = idx(ones(1,3),:);
    J(loc) = Jloc(:);

    A(loc) = Aloc(:);
```

```
    loc = loc+9;

  end

    A = sparse(I,J,A);

return
```

(3.1h) Implement a function

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

to compute the right-hand side vector L given the mesh structure `Mesh`, the quadrature rule via `QuadRule` and a handle to the function f via `FHandle`.

HINT: Proceed as for `assemMat_LFE` and use `LOAD_LFE`.

Solution: See Listing 3.6 for the code.

Listing 3.6: Implementation for `assemLoad_LFE`

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

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

  % Initialize constants

  nCoordinates = size(Mesh.Coordinates,1);
  nElements = size(Mesh.Elements,1);

  % Preallocate memory

  L = zeros(nCoordinates,1);

  % Assemble element contributions

  for i = 1:nElements

    % Extract vertices

    vidx = Mesh.Elements(i,:);
    Vertices = Mesh.Coordinates(vidx,:);

    % Compute load data

    Lloc = LOAD_LFE(Vertices,QuadRule,FHandle);
```

```

    % Add contributions to global load vector

    L(vidx(1)) = L(vidx(1)) + Lloc(1);
    L(vidx(2)) = L(vidx(2)) + Lloc(2);
    L(vidx(3)) = L(vidx(3)) + Lloc(3);

  end

return
```

(3.1i) Implement a routine

```
[U,FreeDofs] = assemDir_LFE(Mesh,BdFlag,GHandle)
```

which accepts in input the mesh, the flag `BdFlag` associated to the Dirichlet boundary and the function handle `GHandle` to the boundary data $g(\boldsymbol{x})$. As output, this function should return the degrees of freedom which are not on the Dirichlet boundary, and initialize the solution vector `U` incoporating the Dirichlet boundary conditions for the entries of `U` associated to nodes on the Dirichlet boundary.
In this problem the convention is that the boundary flag is -1 if the edge is on the Dirichlet boundary.

Solution: See Listing 3.7 for the code.

Listing 3.7: Implementation for `assemDir_LFE`

```
function [U,FreeDofs] = assemDir_LFE(Mesh,BdFlag,GHandle)

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

  % Initialize constants

  nCoordinates = size(Mesh.Coordinates,1);
  U = zeros(nCoordinates,1);

  % Extract Dirichlet nodes

  Loc = find(Mesh.BdFlags==BdFlag);
  DNodes = unique([Mesh.Edges(Loc,1); Mesh.Edges(Loc,2)]);

  % Compute Dirichlet boundary conditions

  U(DNodes) = GHandle(Mesh.Coordinates(DNodes,:));

```

```
  % Compute set of free dofs

  FreeDofs = setdiff(1:nCoordinates,DNodes);

return
```

(3.1j) Implement a function

`plot_LFE(U,Mesh)`

to plot the FE solution given the vector of coefficients `U` and the structure `Mesh` containing the field `Mesh.Coordinates`.

Solution: See Listing 3.8 for the code.

Listing 3.8: Implementation for `plot_LFE`

```
function plot_LFE(U,Mesh)

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

  % Initialize constants

  OFFSET = 0.05;

  % Compute axes limits

  XMin = min(Mesh.Coordinates(:,1));
  XMax = max(Mesh.Coordinates(:,1));
  YMin = min(Mesh.Coordinates(:,2));
  YMax = max(Mesh.Coordinates(:,2));
  XLim = [XMin XMax] + OFFSET*(XMax-XMin)*[-1 1];
  YLim = [YMin YMax] + OFFSET*(YMax-YMin)*[-1 1];

  % Generate figure

    % Compute color axes limits

    CMin = min(U);
    CMax = max(U);
    if (CMin < CMax)          % or error will occur in set
       function
      CLim = [CMin CMax] + OFFSET*(CMax-CMin)*[-1 1];
    else
      CLim = [1-OFFSET 1+OFFSET]*CMin;
```

```
    end

    % Plot real finite element solution
    % Create new figure, if argument 'fig' is not specifiied
    % Otherwise this argument is supposed to be a figure handle
    fig = figure('Name','Linear finite elements');

    patch('faces', Mesh.Elements, ...
          'vertices', [Mesh.Coordinates(:,1) Mesh.Coordinates(:,2) U], ...
          'CData', U, ...
          'facecolor', 'interp', ...
          'edgecolor', 'none');
    set(gca,'XLim',XLim,'YLim',YLim,'CLim',CLim,'DataAspectRatio',[1 1 1]);

return
```

(3.1k) Implement a function

```
err = L2Err_LFE(Mesh,u,QuadRule,FHandle)
```

that computes the L^2-error of the FEM function given by the coefficient vector `u` and the mesh `Coordinates` to the exact solution given as the function handle `FHandle`.

HINT: Proceed computing the local contributions element-wise and then summing them up to get the total error.

Solution: See Listing 3.9 for the code.

Listing 3.9: Implementation for `L2Err_LFE`

```
function err = L2Err_LFE(Mesh,u,QuadRule,FHandle)

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

  % Intialize constants

  nPts = size(QuadRule.w,1);
  nElements = size(Mesh.Elements,1);

  % Precompute shape functions

  N = shap_LFE(QuadRule.x);

```

```
  % Compute discretization error

  err = 0;
  for i = 1:nElements

    % Extract vertex numbers

    vidx = Mesh.Elements(i,:);

    % Compute element mapping

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

    % Transform quadrature points

    x = QuadRule.x*BK+ones(nPts,1)*bK;

    % Evaluate solutions

    u_EX = FHandle(x);
    u_FE = u(vidx(1))*N(:,1) + u(vidx(2))*N(:,2) +
       u(vidx(3))*N(:,3);

    % Compute error on current element

    err = err+sum(QuadRule.w.*abs(u_EX-u_FE).^2)*det_BK;

  end
  err = sqrt(err);

return
```

(3.1l) Implement a function

```
err = H1SErr_LFE(Mesh,u,QuadRule,FHandle)
```

that computes the H^1-seminorm error of the FEM function given by the coefficient vector `u` and the mesh `Coordinates` to the exact solution gradient given as the function handle `FHandle`.

HINT: Proceed element-wise as in subproblem subproblem (3.1k).

Solution: See Listing 3.10 for the code.

Listing 3.10: Implementation for `H1SErr_LFE`

```
function [err,locerr] = H1SErr_LFE(Mesh,u,QuadRule,FHandle)
```

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

  % Initialize constants

  nPts = size(QuadRule.w,1);
  nElements = size(Mesh.Elements,1);

  % Precompute gradients of shape functions

  grad_N = grad_shap_LFE(QuadRule.x);

  % Compute discretization error

  err = 0;
  for i= 1:nElements

    % Extract vertex numbers

    vidx = Mesh.Elements(i,:);

    % Compute element mapping

    bK = Mesh.Coordinates(vidx(1),:);
    BK = [Mesh.Coordinates(vidx(2),:)-bK;
       Mesh.Coordinates(vidx(3),:)-bK];
    inv_BK = inv(BK);
    det_BK = abs(det(BK));

    % Transform quadrature points

    x = QuadRule.x*BK+ones(nPts,1)*bK;

    % Evaluate solutions

    grad_u_EX = FHandle(x);
    grad_u_FE = (u(vidx(1))*grad_N(:,1:2)+ ...
                 u(vidx(2))*grad_N(:,3:4)+ ...
                 u(vidx(3))*grad_N(:,5:6))*transpose(inv_BK);

    % Compute error on the current element

    err = err +
       sum(QuadRule.w.*sum(abs(grad_u_FE-grad_u_EX).^2,2))*det_BK;
```

```
  end
  err = sqrt(err);

return
```

(3.1m) Implement a function

```
[N,l2,h1s] = main_LFE(Mesh)
```

to compute the FE solution `U` to (3.1.1) with coefficient $D(x) = 1$ and exact solution $u_{\text{ex}} = \cos(2\pi x)\cos(2\pi y)$ on the square $\Omega = (0,1)^2$. The function should return the number `N` of *degrees of freedom* (in this case the nodes which are *not* on the boundary) and the L^2-norm and H^1-seminorm errors.
Inside the function, use the routine implemented in task (3.1j) to plot the solution.
As quadrature rule, use the sixth-order quadrature rule `P7O6()` given in the handout.

Solution: See Listing 3.11 for the code.

Listing 3.11: Implementation for `main_LFE`

```
function [N,L2err,H1Serr] = main_LFE(Mesh)

DHandle = @(x) 1;
FHandle = @(x) 8*pi^2*cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));
Uex = @(x) cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));
Ugradex = @(x) [-2*pi*sin(2*pi.*x(:,1)).*cos(2*pi.*x(:,2))
   -2*pi*cos(2*pi.*x(:,1)).*sin(2*pi.*x(:,2))];

% Assemble stiffness matrix and load vector
A = assemMat_LFE(Mesh, @STIMA_Heat_LFE, P7O6(),DHandle);
L = assemLoad_LFE(Mesh,P7O6(),FHandle);

% Incorporate Dirichlet boundary data

[U,FreeDofs] = assemDir_LFE(Mesh,-1,Uex);
L = L - A*U;

% Solve the linear system
U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);

% Plot solution
plot_LFE(U,Mesh);colorbar;

% Compute errors
L2err = L2Err_LFE(Mesh,U,P7O6(),Uex);
H1Serr = H1SErr_LFE(Mesh,U,P7O6(),Ugradex);
N = length(FreeDofs);
```

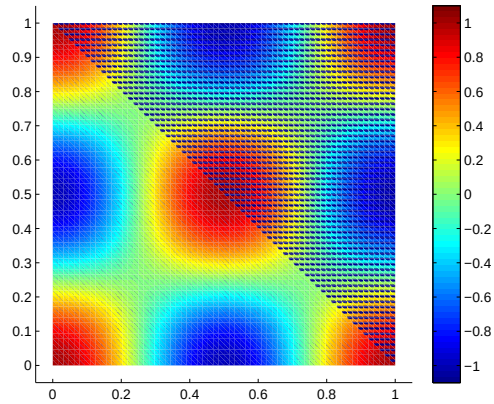


Figure 3.2: Solution plot for subproblem (3.1n).

(3.1n) Run the routine implementated in task (3.1m) to produce a plot of the solution. For the mesh, load the mesh `Square5.mat` given in the handout.

Solution: See Fig.3.2 for the plot.

(3.1o) Consider again the case $u_{\text{ex}} = \cos(2\pi x)\cos(2\pi y)$ and $D(x) = 1$ on the unit square. Implement a script called `cvg_LFE` to perform the convergence study for the error in the L^2-norm and H^1-seminorm.
Produce loglog plots of the errors versus the number of degrees of freedom.
Use the meshes contained in the file `Square.zip` given in the handout.
Which rates of convergence do you observe?

HINT: You may use the function `main_LFE` implemented in task (3.1m).

Solution: See Listing 3.12 for the code and Fig.3.3 for the convergence plots. To estimate the convergence rates, we neglect the error for the first mesh, because from the plots we can see that the error behavior for the first mesh is in the preasymptotic regime. Doing so, the convergence rates that we get are 0.9547 for the L^2-norm and 0.4783 for the H^1-seminorm. These rates are with respect to the number of degrees of freedom; they correspond to rates of respectively 1.9 and 0.96 with respect to the meshwidth h.

Listing 3.12: Implementation for `cvg_LFE`

```
N = [];
l2err = [];
h1serr = [];

for i=1:7
%   Mesh = load(['Square' num2str(i) '.mat']);
%   [Ndof l2 h1] = main_LFE(Mesh);
   Mesh = load(['LShape' num2str(i) '.mat']);
   [Ndof l2 h1] = mainLshaped_LFE(Mesh);
   N = [N Ndof];
   l2err = [l2err l2];
   h1serr = [h1serr h1];
```

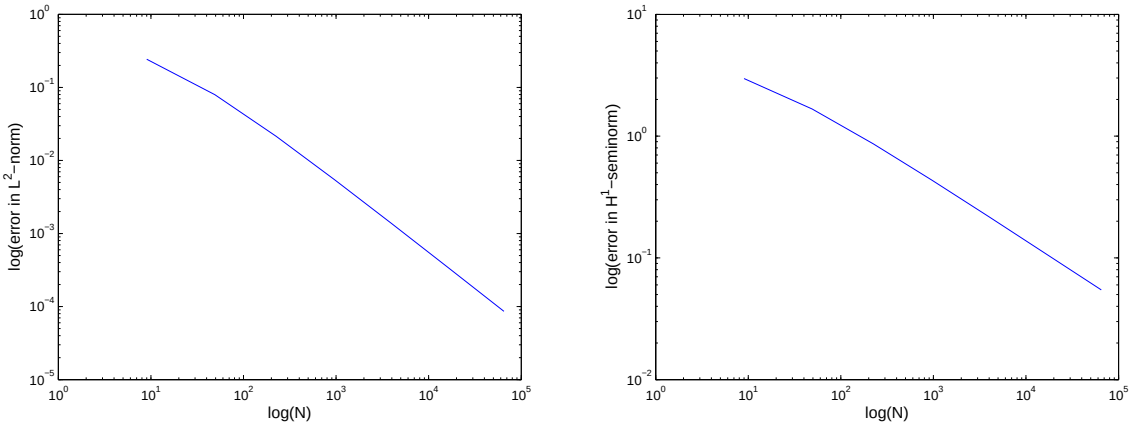


Figure 3.3: Convergence plots for subproblem (3.1n).

```
end

loglog(N,l2err)
figure
loglog(N,h1serr)
```

We are now going to solve (3.1.1)-(3.1.2) on the L-shaped domain $\Omega = (-1,1)^2 \setminus ((0,1) \times (-1,0))$, as depicted in Fig.3.4.
We consider the case that the exact solution is, in polar coordinates, $u = r^{\frac{2}{3}} \sin(\frac{2}{3}\varphi)$, for $(r,\varphi) \in [0,1) \times [0,2\pi)$. The right-handside is then $f = 0$ and the boundary data $g = u_{|\partial\Omega}$.

(3.1p) Implement a function

```
uex = uex_LShap_L2(x)
```

to compute the exact solution $u = r^{\frac{2}{3}} \sin(\frac{2}{3}\varphi)$ given the $N \times 2$ vector of point coordinates `x`, and store the values in the column vector `uex`.

Solution: See Listing 3.13 for the code.

Listing 3.13: Implementation for `uex_LShap_L2`

```
function uex = uex_LShap_L2(x)

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

  % Compute polar coordinates

  r = sqrt(x(:,1).^2 + x(:,2).^2);
  theta = atan2(x(:,2),x(:,1));
```

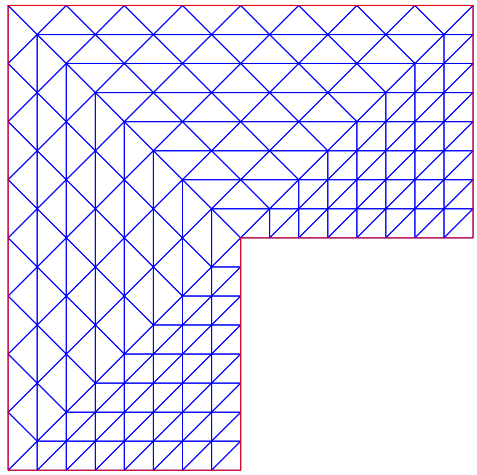


Figure 3.4: Domain for subproblems (3.1p)-(3.1r)

```
  Loc = theta < 0;
  theta(Loc) = theta(Loc) + 2*pi;

  % Compute the exact solution for L2 norm

  uex = r.^(2/3).*sin(2*theta/3);

return
```

.

(3.1q) Implement a function

```
uex = uex_LShapH1S(x)
```

to compute the gradient of the exact solution $u = r^{\frac{2}{3}} \sin(\frac{2}{3}\varphi)$ given the $N \times 2$ vector of point coordinates `x`, and store the values in the $N \times 2$ vector `uex`.

Solution: See Listing 3.14 for the code.

Listing 3.14: Implementation for `uex_LShapH1S`

```
function uex = uex_LShap_H1S(x)

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

  % Compute polar coordinates
```

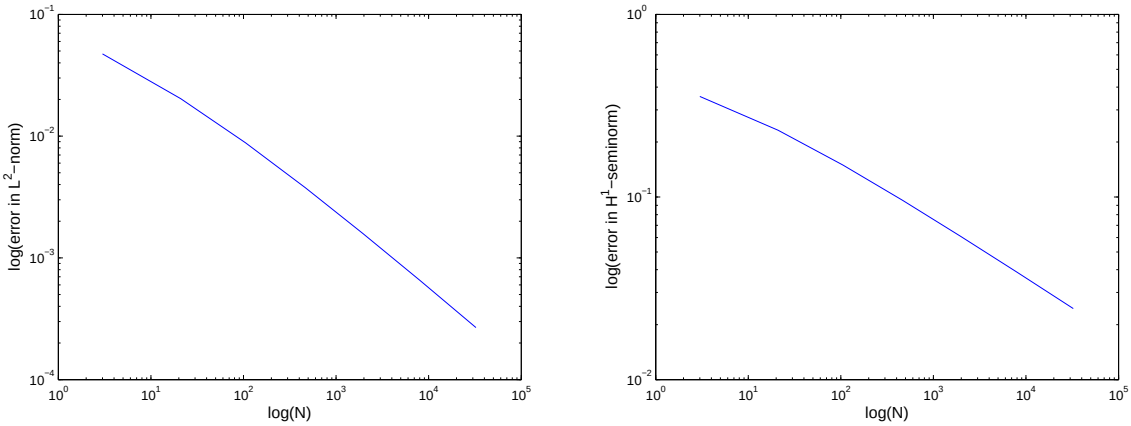


Figure 3.5: Convergence plots for subproblem (3.1r).

```
  r = sqrt(x(:,1).^2 + x(:,2).^2);
  theta = atan2(x(:,2),x(:,1));
  Loc = theta < 0;
  theta(Loc) = theta(Loc) + 2*pi;

  % Compute the exact solution for H1 semi norm

  uex = 2/3*(r.^(-1/3)*ones(1,2)).*[-sin(theta/3)
     cos(theta/3)];

return
```

.

(3.1r) Modify the routine `main_LFE` implemented in subproblem (3.1m) and the script `cvg_LFE` implemented in subproblem (3.1o) to perform the convergence study for the L-shaped domain. Use the meshes contained in the zip file `Lshape.zip` given in the handout. Which rates of convergence to you observe?
Give a motivation for your results.

Solution: See Fig.3.5 for the convergence plots. The convergence rates with respect to the number of degrees of freedom are 0.5628 for the L^2-norm and 0.2919 for the H^1-seminorm, corresponding to rates of around 1 and 0.6 respectively with respect to the meshsize h. The reason for these lower rates is that the gradient of the solution has a singularity at the origin.

Listing 3.15: Testcalls for Problem 3.1

```
Mesh = load(['Square' num2str(1) '.mat']);
DHandle = @(x) 1;
FHandle = @(x) 8*pi^2*cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));
Uex = @(x) cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));

fprintf('\n\n##shap_LFE:')
```

```
shap_LFE([0.3 0.6])

fprintf('\n\n##grad_shap_LFE:')
grad_shap_LFE([0.4 0.4])

fprintf('\n\n##STIMA_Heat_LFE:')
STIMA_Heat_LFE([0 0; 1 1/4; 1/8 1],P7O6(),DHandle)

fprintf('\n\n##LOAD_LFE:')
LOAD_LFE([0 0; 1 1/4; 1/8 1],P7O6(),FHandle)

fprintf('\n\n##assemMat_LFE:')
A = assemMat_LFE(Mesh, @STIMA_Heat_LFE, P7O6(),DHandle);
A = full(A);
A(1:6,1:6)
A(20:25,20:25)

fprintf('\n\n##assemLoad_LFE:')
L = assemLoad_LFE(Mesh,P7O6(),FHandle);
L(1:3)

fprintf('\n\n##assemDir_LFE:')
[U,FreeDofs]=assemDir_LFE(Mesh,-1,Uex);
FreeDofs
```

Listing 3.16: Output for Testcalls for Problem 3.1

```
>> test_call

##shap_LFE:
ans =

    0.1000    0.3000    0.6000

##grad_shap_LFE:
ans =

    -1    -1     1     0     0     1

##STIMA_Heat_LFE:
ans =

    0.6855   -0.3306   -0.3548
   -0.3306    0.5242   -0.1935
   -0.3548   -0.1935    0.5484

##LOAD_LFE:
ans =

    1.2638
```

```
    2.3698
    2.5917

##assemMat_LFE:
ans =

     1     0     0     0     0     0
     0     1     0     0     0     0
     0     0     1     0     0     0
     0     0     0     1     0     0
     0     0     0     0     2     0
     0     0     0     0     0     2

ans =

     4    -1    -1     0     0     0
    -1     4     0     0     0     0
    -1     0     4     0     0     0
     0     0     0     4     0    -1
     0     0     0     0     4    -1
     0     0     0    -1    -1     4

##assemLoad_LFE:
ans =

    0.6366
    0.9995
    0.6366

##assemDir_LFE:
FreeDofs =

     7    12    18    20    21    22    23    24    25
```

Problem 3.2 2D Quadratic Finite Elements

We consider the problem

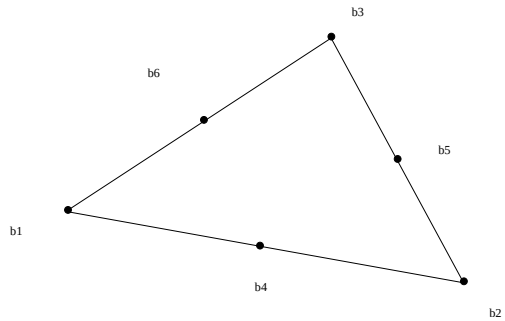
$$-\operatorname{div}(D(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) = f(\boldsymbol{x}) \quad \text{in } \Omega \subset \mathbb{R}^2 \tag{3.2.1}$$

$$u(\boldsymbol{x}) = g(\boldsymbol{x}) \quad \text{on } \partial\Omega \tag{3.2.2}$$

where D is uniformly positive and bounded in Ω, g is a continuous function of $\partial\Omega$ and $f \in L^2(\Omega)$.

We solve (3.2.1)-(3.2.2) by means of Galerkin discretization based on *piecewise quadratic* finite elements on triangular meshes of Ω.

For quadratic finite elements with affine element mapping a specific choice of element shape functions is given by



$$
\begin{aligned}
b_1(\mathbf{x}) &:= -\lambda_1(\mathbf{x})(1-2\lambda_1(\mathbf{x})),\\
b_2(\mathbf{x}) &:= -\lambda_2(\mathbf{x})(1-2\lambda_2(\mathbf{x})),\\
b_3(\mathbf{x}) &:= -\lambda_3(\mathbf{x})(1-2\lambda_3(\mathbf{x})),\\
b_4(\mathbf{x}) &:= 4\lambda_1(\mathbf{x})\lambda_2(\mathbf{x}),\\
b_5(\mathbf{x}) &:= 4\lambda_2(\mathbf{x})\lambda_3(\mathbf{x}),\\
b_6(\mathbf{x}) &:= 4\lambda_3(\mathbf{x})\lambda_1(\mathbf{x}).
\end{aligned}
$$

(3.2a) Implement the function

```
shap = shap_QFE(x)
```

which computes the the value of the six local shape functions $\lambda_i(\boldsymbol{x})$, $i = 1, \ldots, 6$, on the reference element at the points x (a $N \times 2$ matrix, where each rows contains the coordinates of a point), and returns the values in the $N \times 6$ matrix `shap` (each row corresponding to the evaluation of the basis functions in a point).

Solution: See Listing 3.17 for the code.

Listing 3.17: Implementation for `shap_QFE`

```
function shap = shap_QFE(x)

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

  shap = zeros(size(x,1),6);

  shap(:,1) = (1-x(:,1)-x(:,2)).*(1-2*x(:,1)-2*x(:,2));
  shap(:,2) = x(:,1).*(2*x(:,1)-1);
  shap(:,3) = x(:,2).*(2*x(:,2)-1);
  shap(:,4) = 4*x(:,1).*(1-x(:,1)-x(:,2));
  shap(:,5) = 4*x(:,1).*x(:,2);
  shap(:,6) = 4*x(:,2).*(1-x(:,1)-x(:,2));

return
```

(3.2b) Implement the function

```
shap = grad_shap_QFE(x)
```

to compute the gradients of the shape functions, following the same convention as in `grad_shap_LFE`.

Solution: See Listing 3.18 for the code.

Listing 3.18: Implementation for grad_shap_QFE

```
function grad_shap = grad_shap_QFE(x)

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

  grad_shap = zeros(size(x,1),12);

  grad_shap(:,1) = 4*x(:,1)+4*x(:,2)-3;
  grad_shap(:,2) = 4*x(:,1)+4*x(:,2)-3;
  grad_shap(:,3) = 4*x(:,1)-1;
  grad_shap(:,6) = 4*x(:,2)-1;
  grad_shap(:,7) = 4*(1-2*x(:,1)-x(:,2));
  grad_shap(:,8) = (-4)*x(:,1);
  grad_shap(:,9) = 4*x(:,2);
  grad_shap(:,10) = 4*x(:,1);
  grad_shap(:,11) = (-4)*x(:,2);
  grad_shap(:,12) = 4*(1-x(:,1)-2*x(:,2));

return
```

(3.2c) Implement the routine STIMA_Heat_QFE to compute the element (stiffness) matrices. The function header is

```
Aloc = STIMA_Heat_QFE(Vertices, QuadRule, FHandle)
```

and the conventions are the same as in STIMA_Heat_LFE.

Solution: See Listing 3.19 for the code.

Listing 3.19: Implementation for STIMA_Heat_QFE

```
function Aloc = STIMA_Heat_QFE(Vertices,QuadRule,FHandle)

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

  % Initialize constants

  nPoints = size(QuadRule.w,1);

  % Preallocate memory

  Aloc = zeros(6,6);
```

```
% Compute element mapping

BK = [Vertices(2,:)-Vertices(1,:);
   Vertices(3,:)-Vertices(1,:)];
bK = Vertices(1,:);
inv_BK = inv(BK);
det_BK = abs(det(BK));

TK = det_BK*transpose(inv_BK)*inv_BK;

x = QuadRule.x*BK+ones(nPoints,1)*bK;

% Compute element stiffness matrix

FVal = FHandle(x);
grad_N = grad_shap_QFE(QuadRule.x);

Aloc(1,1) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,1:2)*TK),2));
Aloc(1,2) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,3:4)*TK),2));
Aloc(1,3) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,5:6)*TK),2));
Aloc(1,4) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,7:8)*TK),2));
Aloc(1,5) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,9:10)*TK),2));
Aloc(1,6) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,1:2)).*(grad_N(:,11:12)*TK),2));

Aloc(2,2) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,3:4)).*(grad_N(:,3:4)*TK),2));
Aloc(2,3) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,3:4)).*(grad_N(:,5:6)*TK),2));
Aloc(2,4) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,3:4)).*(grad_N(:,7:8)*TK),2));
Aloc(2,5) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,3:4)).*(grad_N(:,9:10)*TK),2));
Aloc(2,6) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,3:4)).*(grad_N(:,11:12)*TK),2));

Aloc(3,3) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,5:6)).*(grad_N(:,5:6)*TK),2));
Aloc(3,4) =
   sum(QuadRule.w.*FVal.*sum((grad_N(:,5:6)).*(grad_N(:,7:8)*TK),2));
```

```
  Aloc(3,5) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,5:6)).*(grad_N(:,9:10)*TK),2));
  Aloc(3,6) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,5:6)).*(grad_N(:,11:12)*TK),2));

  Aloc(4,4) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,7:8)).*(grad_N(:,7:8)*TK),2));
  Aloc(4,5) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,7:8)).*(grad_N(:,9:10)*TK),2));
  Aloc(4,6) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,7:8)).*(grad_N(:,11:12)*TK),2));

  Aloc(5,5) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,9:10)).*(grad_N(:,9:10)*TK),2));
  Aloc(5,6) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,9:10)).*(grad_N(:,11:12)*TK),2));

  Aloc(6,6) =
     sum(QuadRule.w.*FVal.*sum((grad_N(:,11:12)).*(grad_N(:,11:12)*TK),2))

  % Fill in lower triangular part

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

return
```

(3.2d) Implement the routine `LOAD_QFE`

```
Lloc = LOAD_QFE(Vertices, QuadRule, FHandle)
```

to compute the *element vector*, which follows the same convention as `LOAD_LFE`.

Solution: See Listing 3.20 for the code.

Listing 3.20: Implementation for `LOAD_QFE`

```
function Lloc = LOAD_QFE(Vertices,QuadRule,FHandle)

% Initialize constants

nPoints = size(QuadRule.w,1);

% Preallocate memory

Lloc = zeros(6,1);

% Compute element mapping
```

```
  bK = Vertices(1,:);
  BK = [Vertices(2,:)-bK; Vertices(3,:)-bK];
  inv_BK = inv(BK);
  det_BK = abs(det(BK));

  x = QuadRule.x*BK+ones(nPoints,1)*bK;

  % Compute element load vector

  FVal = FHandle(x);
  N = shap_QFE(QuadRule.x);

  Lloc(1) = sum(QuadRule.w.*FVal.*N(:,1))*det_BK;
  Lloc(2) = sum(QuadRule.w.*FVal.*N(:,2))*det_BK;
  Lloc(3) = sum(QuadRule.w.*FVal.*N(:,3))*det_BK;
  Lloc(4) = sum(QuadRule.w.*FVal.*N(:,4))*det_BK;
  Lloc(5) = sum(QuadRule.w.*FVal.*N(:,5))*det_BK;
  Lloc(6) = sum(QuadRule.w.*FVal.*N(:,6))*det_BK;
```

We now consider the assembly part.
The global order of basis function for quadratic finite elements is the following:

- first the basis functions associated to the vertices are stored; the basis function associated to the vertex with index i is $b_N^i(\boldsymbol{x})$;
- then, the basis functions associated to the midpoints, i.e. to the edges, are considered: the basis function associated to the edge i is $b_N^{N_V+i}(\boldsymbol{x})$, with N_V the number of vertices.

(3.2e) Implement a function

```
A = assemMat_QFE(Mesh,EHandle,varargin)
```

that assembles the Galerkin matrix A and follows the same convention as `assem_Mat_LFE`.

HINT: Use the sparse format to store the matrix A.
The field `Mesh.Vert2Edge` may be useful.

Solution: See Listing 3.21 for the code.

Listing 3.21: Implementation for `assemMat_QFE`

```
function A = assemMat_QFE(Mesh,EHandle,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);
nElements = size(Mesh.Elements,1);

% Preallocate memory

I = zeros(36*nElements,1);
J = zeros(36*nElements,1);
A = zeros(36*nElements,1);

% Assemble element contributions

loc = 1:36;
for i = 1:nElements

  % Extract vertices of the current element

  vidx = Mesh.Elements(i,:);
  Vertices = Mesh.Coordinates(vidx,:);

  % Compute element contributions

  Aloc = EHandle(Vertices,varargin{:});

  % Extract global edge numbers

  idx = [vidx ...
         Mesh.Vert2Edge(Mesh.Elements(i,1),Mesh.Elements(i,2))+nCoordina
            ...
         Mesh.Vert2Edge(Mesh.Elements(i,2),Mesh.Elements(i,3))+nCoordina
            ...
         Mesh.Vert2Edge(Mesh.Elements(i,3),Mesh.Elements(i,1))+nCoordina

  % Add contributions to global matrix

  Iloc = idx(ones(6,1),:)';
  I(loc) = Iloc(:);

  Jloc = idx(ones(1,6),:);
  J(loc) = Jloc(:);

  A(loc) = Aloc(:);
  loc = loc+36;

end

```

```
  % Assign output arguments

  A = sparse(I,J,A);

return
```

(3.2f) Implement a function

```
L = assemLoad_QFE(Mesh,QuadRule,FHandle)
```

to compute the right-hand side vector L, following the same conventions as in `assemLoad_LFE`.

Solution: See Listing 3.22 for the code.

Listing 3.22: Implementation for `assemLoad_QFE`

```
function L = assemLoad_QFE(Mesh,QuadRule,FHandle)

%   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);
  nElements = size(Mesh.Elements,1);
  nEdges = size(Mesh.Edges,1);

  % Preallocate memory

  L = zeros(nCoordinates+nEdges,1);

  eidx = zeros(1,3);
  for i = 1:nElements

    % Extract vertices

    vidx = Mesh.Elements(i,:);
    eidx(1) = Mesh.Vert2Edge(vidx(1),vidx(2)) + nCoordinates;
    eidx(2) = Mesh.Vert2Edge(vidx(2),vidx(3)) + nCoordinates;
    eidx(3) = Mesh.Vert2Edge(vidx(3),vidx(1)) + nCoordinates;

    % Compute load data

    Lloc =
       LOAD_QFE(Mesh.Coordinates(vidx,:),QuadRule,FHandle);

```

```
    % Add contributions to global load vector

    L(vidx(1)) = L(vidx(1)) + Lloc(1);
    L(vidx(2)) = L(vidx(2)) + Lloc(2);
    L(vidx(3)) = L(vidx(3)) + Lloc(3);
    L(eidx(1)) = L(eidx(1)) + Lloc(4);
    L(eidx(2)) = L(eidx(2)) + Lloc(5);
    L(eidx(3)) = L(eidx(3)) + Lloc(6);

  end

return
```

(3.2g) Implement a routine

```
[U,FreeDofs] = assemDir_QFE(Mesh,BdFlag,GHandle)
```

following the same principles as `assemDir_LFE`.

Solution: See Listing 3.23 for the code.

Listing 3.23: Implementation for `assemDir_QFE`

```
function [U,FreeDofs] = assemDir_QFE(Mesh,BdFlag,GHandle)

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

  % Intialize constants

  nCoordinates = size(Mesh.Coordinates,1);
  nEdges = size(Mesh.Edges,1);

  U = zeros(nCoordinates+nEdges,1);

    % Extract Dirichlet nodes

    DEdges = find(Mesh.BdFlags==BdFlag);
    DNodes = unique([Mesh.Edges(DEdges,1);
       Mesh.Edges(DEdges,2)]);

    % Compute midpoints of all edges

    MidPoints = 1/2*(Mesh.Coordinates(Mesh.Edges(DEdges,1),:)
       + ...
                     Mesh.Coordinates(Mesh.Edges(DEdges,2),:));
```

```

    % Compute Dirichlet boundary conditions

    U(DNodes) = GHandle(Mesh.Coordinates(DNodes,:));
    U(DEdges+nCoordinates) = GHandle(MidPoints);

  % Compute set of free dofs

  FreeDofs = [setdiff(1:nCoordinates,DNodes) ...
              setdiff(1:nEdges,DEdges) + nCoordinates];

return
```

(3.2h) Implement a function

```
err = L2Err_QFE(Mesh,u,QuadRule,FHandle)
```

that computes the L^2-error of the FEM function given by the coefficient vector `u` and the mesh `Mesh` to the exact solution given as the function handle `FHandle`.

Solution: See Listing 3.24 for the code.

Listing 3.24: Implementation for `L2Err_QFE`

```
function err = L2Err_QFE(Mesh,u,QuadRule,FHandle)

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

  % Intialize constants

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

  % Precompute shape function values at the quedrature points

  N = shap_QFE(QuadRule.x);

  % Compute discretization error

  err = 0;
  eidx = zeros(1,3);
  for i = 1:nElements

    % Extract vertex and edge numbers
```

```
    vidx = Mesh.Elements(i,:);
    eidx(1) = Mesh.Vert2Edge(vidx(1),vidx(2)) + nCoordinates;
    eidx(2) = Mesh.Vert2Edge(vidx(2),vidx(3)) + nCoordinates;
    eidx(3) = Mesh.Vert2Edge(vidx(3),vidx(1)) + nCoordinates;

    % Compute element mapping

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

    % Transform quadrature points

    x = QuadRule.x*BK+ones(nPts,1)*bK;

    % Evaluate solutions

    u_EX = FHandle(x);
    u_FE = u(vidx(1))*N(:,1) + u(vidx(2))*N(:,2) +
       u(vidx(3))*N(:,3) + ...
           u(eidx(1))*N(:,4) + u(eidx(2))*N(:,5) +
              u(eidx(3))*N(:,6);

    % Compute error on current element

    err = err+sum(QuadRule.w.*abs(u_EX-u_FE).^2)*det_BK;

  end
  err = sqrt(err);

return
```

(3.2i) Implement a function

```
err = H1SErr_QFE(Mesh,u,QuadRule,FHandle)
```

that computes the H^1-seminorm error of the FEM function given by the coefficient vector `u` and the mesh `Mesh` to the exact solution gradient given as the function handle `FHandle`.

Solution: See Listing 3.25 for the code.

Listing 3.25: Implementation for `H1SErr_QFE`

```
function err = H1SErr_QFE(Mesh,u,QuadRule,FHandle)

%   Copyright 2005-2005 Patrick Meury & Kah Ling Sia
```

```
%   SAM - Seminar for Applied Mathematics
%   ETH-Zentrum
%   CH-8092 Zurich, Switzerland

  % Initialize constants

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

  % Precompute gradients of shape functions

  grad_N = grad_shap_QFE(QuadRule.x);

  % Compute discretization error

  err = 0;
  eidx = zeros(1,3);
  for i= 1:nElements

    % Extract vertex and edge numbers

    vidx = Mesh.Elements(i,:);
    eidx(1) = Mesh.Vert2Edge(vidx(1),vidx(2)) + nCoordinates;
    eidx(2) = Mesh.Vert2Edge(vidx(2),vidx(3)) + nCoordinates;
    eidx(3) = Mesh.Vert2Edge(vidx(3),vidx(1)) + nCoordinates;

    % Compute element mapping

    bK = Mesh.Coordinates(vidx(1),:);
    BK = [Mesh.Coordinates(vidx(2),:)-bK;
       Mesh.Coordinates(vidx(3),:)-bK];
    inv_BK = inv(BK);
    det_BK = abs(det(BK));

    % Transform quadrature points

    x = QuadRule.x*BK+ones(nPts,1)*bK;

    % Evaluate solutions

    grad_u_EX = FHandle(x);
    grad_u_FE = (u(vidx(1))*grad_N(:,1:2) + ...
                 u(vidx(2))*grad_N(:,3:4) + ...
                 u(vidx(3))*grad_N(:,5:6) + ...
                 u(eidx(1))*grad_N(:,7:8) + ...
                 u(eidx(2))*grad_N(:,9:10) + ...
```

```
                    u(eidx(3))*grad_N(:,11:12))*transpose(inv_BK);

    % Compute error on the current element

    err =
       err+sum(QuadRule.w.*sum(abs(grad_u_FE-grad_u_EX).^2,2))*det_BK;

  end
  err = sqrt(err);

return
```

(3.2j) Implement a function

```
[N,l2,h1s] = main_QFE(Mesh)
```

to compute the FE solution `U` to (3.2.1) with coefficient $D(x) = 1$ and exact solution $u_{\text{ex}} = \cos(2\pi x)\cos(2\pi y)$ on the square $\Omega = (0,1)^2$. The routine follows the same conventions as `main_LFE`, but this time you don't need to plot the solution.
Again, as quadrature rule, use the sixth-order quadrature rule `P7O6()` given in the handout.

Solution: See Listing 3.26 for the code.

Listing 3.26: Implementation for `main_QFE`

```
function [N,L2err,H1Serr] = main_QFE(Mesh)

DHandle = @(x) 1;
FHandle = @(x) 8*pi^2*cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));
Uex = @(x) cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));
Ugradex = @(x) [-2*pi*sin(2*pi.*x(:,1)).*cos(2*pi.*x(:,2))
   -2*pi*cos(2*pi.*x(:,1)).*sin(2*pi.*x(:,2))];

% Assemble stiffness matrix and load vector
A = assemMat_QFE(Mesh, @STIMA_Heat_QFE, P7O6(),DHandle);
L = assemLoad_QFE(Mesh,P7O6(),FHandle);

% Incorporate Dirichlet boundary data

[U,FreeDofs] = assemDir_QFE(Mesh,-1,Uex);
L = L - A*U;

% Solve the linear system
U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);

% Compute errors
L2err = L2Err_QFE(Mesh,U,P7O6(),Uex);
H1Serr = H1SErr_QFE(Mesh,U,P7O6(),Ugradex);
```

```
N = length(FreeDofs);
```

(3.2k) Consider again the case $u_{\text{ex}} = \cos(2\pi x)\cos(2\pi y)$ and $D(x) = 1$ on the unit square. Implement a script called `cvg_QFE` to perform the convergence study for the error in the L^2-norm and H^1-seminorm.
Produce loglog plots of the errors versus the number of degrees of freedom.
Use the meshes contained in the file `Square.zip` given in the handout.
Which rates of convergence do you observe?

HINT: Use the function `main_QFE` implemented in task (3.2j).

Solution: See Listing 3.27 for the code and Fig.3.6 for the convergence plots. The convergence rates that we get are 1.4623 for the L^2-norm and 0.9593 for the H^1-seminorm. These rates are with respect to the number of degrees of freedom; they correspond to rates of respectively 2.92 and 1.92 with respect to the meshwidth h.

Listing 3.27: Implementation for `cvg_QFE`

```
N = [];
l2err = [];
h1serr = [];

for i=1:7
%   Mesh = load(['Square' num2str(i) '.mat']);
%   [Ndof l2 h1] = main_QFE(Mesh);
   Mesh = load(['LShape' num2str(i) '.mat']);
   [Ndof l2 h1] = mainLshaped_QFE(Mesh);
   N = [N Ndof];
   l2err = [l2err l2];
   h1serr = [h1serr h1];
end

loglog(N,l2err)
figure
loglog(N,h1serr)
```

Now we consider again (3.2.1)-(3.2.2) on the L-shaped domain $\Omega = (-1,1)^2 \setminus ((0,1) \times (-10))$. We take again the case that the exact solution is, in polar coordinates, $u = r^{\frac{2}{3}} \sin(\frac{2}{3}\varphi)$, for $(r,\varphi) \in [0,1) \times [0,2\pi)$.

(3.2l) Modify the routine `main_QFE` implemented in subproblem (3.2j) and the script `cvg_QFE` implemented in subproblem (3.2k) to perform the convergence study for the L-shaped domain. Use the meshes contained in the zip file `Lshape.zip` given in the handout. Which rates of convergence to you observe?
Compare your results with the case of Linear Finite Elements and give a motivation for the behavior that you observe.

HINT: You may use the routines `uex_LShap_L2(x)` and `uex_LShapH1S(x)` implemented for the previous problem for the computation of the exact solution and its gradient.

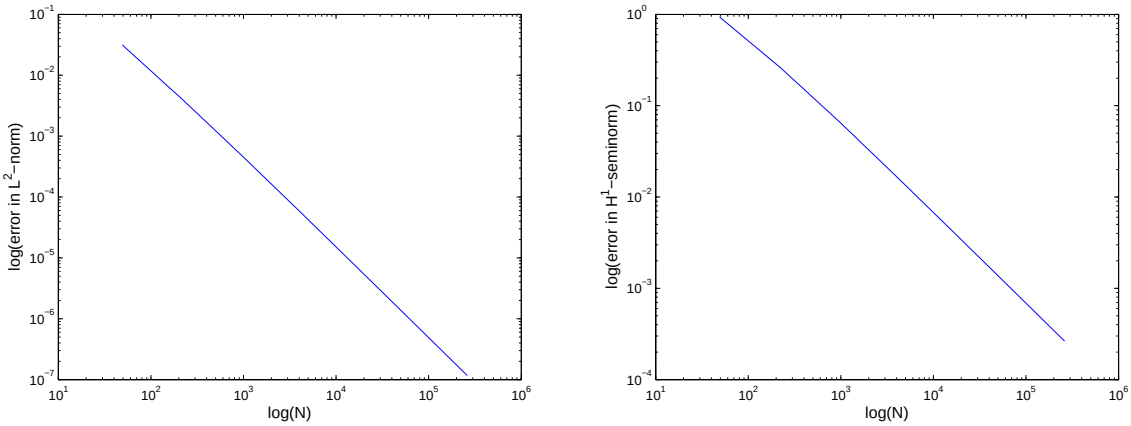


Figure 3.6: Convergence plots for subproblem (3.2k).

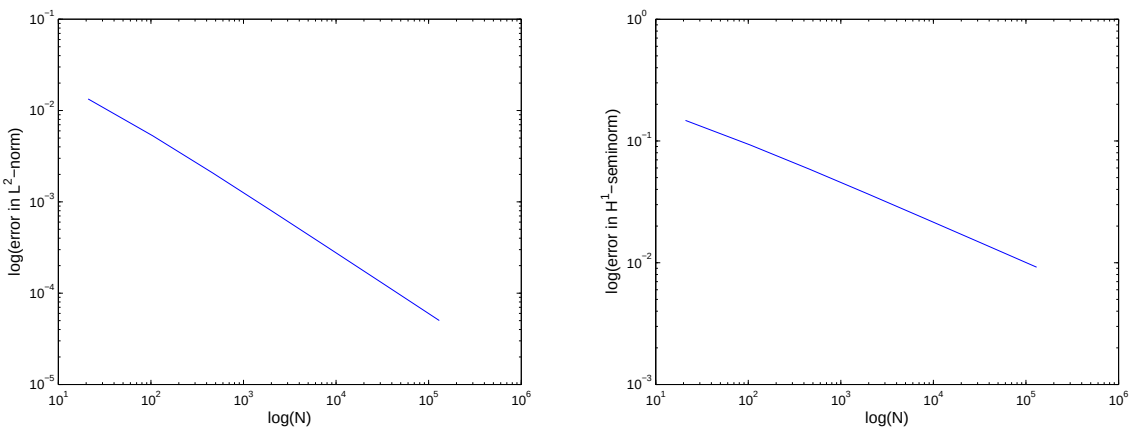


Figure 3.7: Convergence plots for subproblem (3.2l).

Solution: See Fig.3.7 for the convergence plots. The convergence rates with respect to the number of degrees of freedom are 0.6436 for the L^2-norm and 0.3195 for the H^1-seminorm, corresponding to rates of around 1.29 and 0.64 respectively with respect to the meshsize h. The reason for these lower rates is that the gradient of the solution has a singularity at the origin. Comparing with the results with the linear finite element discretization, we cannot see a significant improvement in the convergence rates because, since the solution is not smooth enough, increasing the polynomial order does not lead to an improvement of the convergence.

Listing 3.28: Testcalls for Problem 3.2

```
Mesh = load(['Square' num2str(1) '.mat']);
DHandle = @(x) 1;
FHandle = @(x) 8*pi^2*cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));
Uex = @(x) cos(2*pi.*x(:,1)).*cos(2*pi.*x(:,2));

fprintf('\n\n##shap_QFE:')
shap_QFE([0.3 0.6])

```

```
fprintf('\n\n##grad_shap_QFE:')
grad_shap_QFE([0.4 0.8])

fprintf('\n\n##STIMA_Heat_QFE:')
STIMA_Heat_QFE([0 0; 1 0; 0 1],P7O6(),DHandle)

fprintf('\n\n##LOAD_QFE:')
LOAD_QFE([0 0; 1 0; 0 1],P7O6(),FHandle)

fprintf('\n\n##assemMat_QFE:')
A = assemMat_QFE(Mesh, @STIMA_Heat_QFE, P7O6(),DHandle);
A = full(A);
A(1:3,1:3)
A(26:28,26:28)

fprintf('\n\n##assemLoad_QFE:')
L = assemLoad_QFE(Mesh,P7O6(),FHandle);
L(1:3)
L(26:28)

fprintf('\n\n##assemDir_QFE:')
[U,FreeDofs]=assemDir_QFE(Mesh,-1,Uex);
FreeDofs
```

Listing 3.29: Output for Testcalls for Problem 3.2

```
test_call

##shap_QFE:
ans =

   -0.0800   -0.1200    0.1200    0.1200    0.7200    0.2400

##grad_shap_QFE:
ans =

    1.8000    1.8000    0.6000         0         0    2.2000
       -2.4000   -1.6000    3.2000    1.6000   -3.2000   -4.0000

##STIMA_Heat_QFE:
ans =

    1.0000    0.1667    0.1667   -0.6667   -0.0000   -0.6667
    0.1667    0.5000         0   -0.6667   -0.0000    0.0000
    0.1667         0    0.5000    0.0000   -0.0000   -0.6667
   -0.6667   -0.6667    0.0000    2.6667   -1.3333   -0.0000
   -0.0000   -0.0000   -0.0000   -1.3333    2.6667   -1.3333
   -0.6667    0.0000   -0.6667   -0.0000   -1.3333    2.6667

##LOAD_QFE:
```

```
ans =

    1.0920
    0.1993
    0.1993
   -1.7408
    5.2648
   -1.7408

##assemMat_QFE:
ans =

    1.0000         0         0
         0    1.0000         0
         0         0    1.0000

ans =

    2.6667   -0.0000         0
   -0.0000    2.6667         0
         0         0    5.3333

##assemLoad_QFE:
ans =

    0.0590
    0.1889
    0.0590

ans =

    0.5776
    0.5776
    0.7577

##assemDir_QFE:
FreeDofs =

  Columns 1 through 20

     7    12    18    20    21    22    23    24    25    28    34
            36    39    41    42    44    45    46    47    48

  Columns 21 through 40

    49    50    53    54    57    58    59    60    61    62    63
          64    65    66    67    68    69    70    71    72

  Columns 41 through 49

```

```
72      73      74      75      76      77      78      79      80      81
```

Published on March 31.

To be submitted on April 14.

Last modified on August 18, 2014