R. Hiptmair
L. Scarabosio
E. Fonn

Spring Term 2013

Numerical Methods for Partial Differential Equations

ETH Zürich
D-MATH

Homework Problem Sheet 12

Introduction.

Problem 12.1 Sign Conditions for Galerkin Matrix

As in [NPDE, Sect. 7.2], we consider the non-dimensional two-dimensional stationary convection-diffusion equation, *cf.* [NPDE, Eq. (7.2.1)]

$$-\epsilon\Delta u(\mathbf{x}) + \mathbf{v}\cdot\mathbf{grad}\, u(\mathbf{x}) = f(\mathbf{x}) \qquad \text{in } \Omega\,. \tag{12.1.1}$$

We know from [NPDE, Ex. 7.2.19] and [NPDE, Ex. 7.2.24] that the standard Galerkin discretization of (12.1.1) is haunted by spurious oscillations in the singularly perturbed case $\epsilon \ll 1$ (assuming $\|\mathbf{v}\| \approx 1$). This manifests itself in a blatant violation of the discrete maximum principle, remember the discussion following [NPDE, Ex. 7.2.19]. In this problem we are going to observe the breakdown of the discrete maximum principle as ϵ decreases.

We will be using homogeneous Dirichlet boundary conditions $u(\mathbf{x}) = 0$ on $\partial\Omega$, a constant velocity of

$$\mathbf{v} = \begin{pmatrix}\cos\theta\\ \sin\theta\end{pmatrix}, \quad \theta \in [0, 2\pi),$$

and Ω will be the unit square. Assume $\mathcal{M}$ is a regular mesh as shown in Figure 12.1. For the rest of this problem, let h be the side length of a short edge of a triangle.

(12.1a) For linear Lagrangian finite elements ($V_N = \mathcal{S}^0_{1,0}(\mathcal{M})$), compute the element matrices for the triangles of the mesh $\mathcal{M}$.

HINT: We do *not* use upwind quadrature. You only need to express the element matrix for two types of triangles (i) triangles with the right angle at the top-left, (ii) triangles with the right angle at the bottom-right. For consistency enumerate all vertices counter-clockwise. See [NPDE, Sect. 3.2.5] for the computations concerning the diffusive part.

(12.1b) Specify one row of the global Galerkin matrix corresponding to an internal node.

HINT: Use a global node numbering where nodes are numbered from the bottom left, towards the right and then upwards (lexikographic numbering) as in [NPDE, Fig. 130]. Recall the assembly of a Galerkin matrix as explained in [NPDE, Sect. 3.2.5].

(12.1c) How small can ϵ be without compromising the discrete maximum principle.

HINT: Recall that the sign conditions [NPDE, Eq. (5.7.9)]–[NPDE, Eq. (5.7.11)] for the Galerkin matrix provide sufficient conditions for the discrete maximum principle to hold.

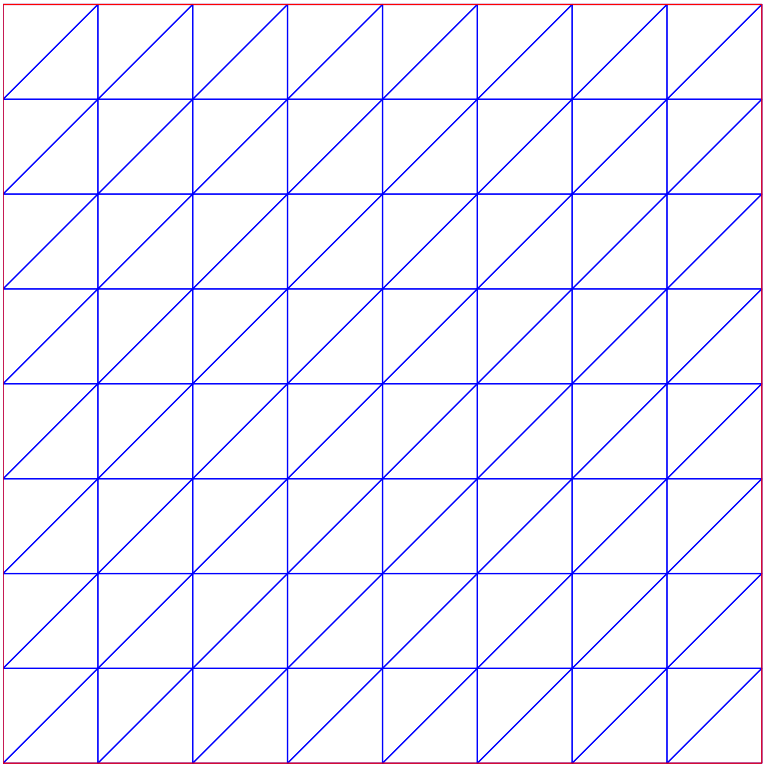

Figure 12.1: Mesh for Problem 12.1

(12.1d) Can we get the sign conditions [NPDE, Eq. (5.7.9)]–[NPDE, Eq. (5.7.11)] of the discrete maximum principle satisfied for an arbitrary velocity $\mathbf{v}$ by introducing a proper *artificial diffusion*, i. e. by choosing ϵ, as a function of h, large enough?

HINT: the sign conditions require $\mathbf{A}_{i,i} > 0$, $\mathbf{A}_{i,j} \leq 0$ for $j \neq i$, and $\mathbf{A}_{i,i} \geq \sum_{j \neq i} |\mathbf{A}_{i,j}|$ for each i.

Problem 12.2 Upwind Quadrature (Core problem)

In [NPDE, Sect. 7.2.2.1] we discovered that the strategy of *upwind quadrature* makes it possible to generalize the idea of upwinding, that is, the use of backward difference quotients to discretize convective terms, to higher dimensions.

In this problem we discuss the implementation of upwind quadrature for the stationary convection-diffusion equation,

$$-\epsilon \Delta u(\mathbf{x}) + \mathbf{v}(\mathbf{x}) \cdot \operatorname{grad} u(\mathbf{x}) = 0 \qquad \text{in } \partial\Omega, \tag{12.2.1}$$

with Dirichlet boundary conditions $u(\mathbf{x}) = g(\mathbf{x})$ on $\partial\Omega$. Here, Ω will be the unit square, and

$$g(\mathbf{x}) = \begin{cases} \frac{1}{2} - \left|x_1 - \frac{1}{2}\right| & x_2 = 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathbf{v}(\mathbf{x}) = \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix}.$$

The equation models the temperature distribution $u = u(\boldsymbol{x}, t)$ in a fluid streaming with stationary velocity $\mathbf{v}$.

(12.2a) Show that $\mathbf{v}$ describes the flow of an incompressible fluid according to [NPDE, Def. 7.1.7].

HINT: Incompressible fluid flows satisfy $\operatorname{div} \mathbf{v}(\mathbf{x}) = 0$ everywhere (see [NPDE, Thm. 7.1.12]).

(12.2b) Show that the streamlines of the flow are (parts of) circles centered at the origin.

HINT: The velocity field should always be tangent to the streamlines.

(12.2c) As $\epsilon \to 0$, the convection term will dominate the diffusion term (singular perturbation). What do you expect will happen in the limit? Will there be a boundary layer somewhere in the domain? Where?

HINT: Reading [NPDE, Sect. 7.2.1] might offer crucial clues.

(12.2d) The file `main_p4.m` solves (12.2.1) with $\epsilon = 10^{-4}$ using the standard Galerkin approach, see [NPDE, Rem. 7.2.2], and linear finite elements. Run the code. What do you see?

We will now attempt to cure the instability in the method for small ϵ. To this end, we will implement *upwind quadrature*, also known as *Tabata scheme*, for the convective part (the diffusive part will be unchanged). In this scheme, the local element contribution is

$$\int_K (\mathbf{v}(\mathbf{x}) \cdot \operatorname{grad} u_N(\mathbf{x})) w_N(\mathbf{x}) \mathrm{d}\mathbf{x} \approx \frac{|K|}{3} \sum_{l=1}^{3} (\mathbf{v}(\mathbf{a}^l) \cdot (\operatorname{grad} u_N)^+(\mathbf{a}^l)) w_N(\mathbf{a}^l),$$

where $\mathbf{a}^l$ are the vertices of the triangle K, and where $(\operatorname{grad} u_N)^+(\mathbf{a}^l)$ is *not* the gradient of u_N in the triangle K, but rather the gradient of u_N in the triangle *upstream* from K (the one in the direction of $-\mathbf{v}(\mathbf{a}^l)$). Note that this is unambigously defined, as the gradient of a piecewise linear function is constant on each triangle. More details are given in [NPDE, Sect. 7.2.2.1].

(12.2e) Show that according to the above scheme, we have (for the *global* convective bilinear form b),

$$\mathsf{b}(b_N^i, b_N^j) = m(\mathbf{a}^j) \left(\mathbf{v}(\mathbf{a}^j) \cdot (\operatorname{grad} b_N^i)^+(\mathbf{a}^j) \right),$$

where b_N^i is the nodal basis function of $\mathcal{S}_1^0(\mathcal{M})$ associated with node $\boldsymbol{a}^i$ of the a triangular mesh $\mathcal{M}$ and $m(\mathbf{a}^j)$ is defined as the *mass* of vertex j, i.e.

$$m(\mathbf{a}^j) = \frac{1}{3} \sum_{\mathbf{a}^j \in K} |K|, \tag{12.2.2}$$

where the sum goes over all triangles containing $\mathbf{a}^j$, see also [NPDE, Eq. (7.2.26)]

(12.2f) The upwind quadrature method does not completely fit the concept of local assembly as introduced in [NPDE, Sect. 3.5.3], because $\mathsf{b}(b_N^i, b_N^j)$ cannot be written as the sum of contributions of individual cells. Nevertheless, after the vertex masses have been precomputed, there is a smart way to use local assembly to initialize the convective part $\mathbf{B}$ of the system matrix.

Write a MATLAB function

```
Aloc = STIMA_Up_LFE(Vertices, VHandle, Mass)
```

that computes the "local" contribution of a triangle K (defined by the 3×2 matrix `Vertices`), given a function handle `VHandle` to the velocity $\mathbf{v}$ and the vertex masses as given by (12.2.2). By "local" contribution, we mean, that for each j:

1. Check if $-\mathbf{v}(\mathbf{a}^j)$ points *into* the triangle K. Do this by transforming $\mathbf{a}^j - \mathbf{v}(\mathbf{a}^j)$ to the unit triangle, where it should be easier to check.

2. *If* it does, compute $\mathsf{b}(b_K^i, b_K^j)$ for each i, where b_K^i designates a local shape function (barycentric coordinate function).

3. If it does *not*, skip the current triangle (no contribution).

4. If $-\mathbf{v}(\mathbf{a}^j)$ points along an edge, divide the corresponding elements by 2 (because these will get a contribution from two different elements).

In this way, you are actually evaluating the *global* bilinear form $\mathsf{b}(b_N^i, b_N^j)$, but only for those j where K is the upwind triangle.

HINT: Note that b is not symmetric, so take extra care in getting the matrix right: $\mathbf{B}_{i,j} = \mathsf{b}(b^j, b^i)$.

(12.2g) Write a MATLAB function

```
A = assemMat_UP_LFE(Mesh, VHandle)
```

that assembles the local contributions from the previous problem into the matrix `A`. You can reuse code from `assemMat_LFE.m` if you wish.

HINT: The most crucial difference is that you will have to compute the vertex masses (12.2.2). For this, simply allocate a vector of zeros, loop through the elements, compute their areas (for example by taking the determinant of the affine transformation), and add into your vector.

(12.2h) Edit `main_p4.m` so that it uses the upwind quadrature scheme instead. Run your code. Will it produce a "better" solution than that obtained in subproblem (12.2d). How small $\epsilon > 0$ can you handle now?

Listing 12.1: Testcalls for Problem 12.2

```
VHandle = @(x,varargin) [-x(:,2), x(:,1)];

fprintf('\n##STIMA_Up_LFE');
Up_loc=STIMA_Up_LFE([1/2,1;1,1;1,1/2],VHandle,1/3*[1,1,1])

clear Mesh
Mesh.Coordinates = [0 0; 1 0; 1 1; 0 1];
Mesh.Elements = [1 2 3; 1 3 4];
Mesh = add_Edges(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);
Mesh = add_Edge2Elem(Mesh);
Mesh = refine_REG(Mesh);
fprintf('\n#assemMat_Up_LFE');
assemMat_Up_LFE(Mesh, VHandle)
```

Listing 12.2: Output for Testcalls for Problem 12.2

```
>> test_call

##STIMA_Up_LFE
Up_loc =

    0.6667   -0.3333   -0.3333
         0         0         0
         0         0         0

#assemMat_Up_LFE
ans =

   (9,3)       -0.1250
   (6,5)       -0.1250
   (6,6)        0.2500
   (9,6)       -0.0625
   (6,8)       -0.1250
   (9,9)        0.1875
```

Problem 12.3 Upwind Finite Volume Method

In this problem we will study another stable method for approximately solving the convection-diffusion equation with a general velocity field, not necessarily divergence-free:

$$\operatorname{div}(\epsilon \,\mathbf{grad}\, u + \boldsymbol{v}u) = f, \tag{12.3.1}$$

in Ω and with Dirichlet boundary conditions $u = g$ on $\partial\Omega$. See [NPDE, Sect. 4.2] for details on boundary value problems for (12.3.1).

The discretization scheme adheres to the spirit of finite volume methods, see [NPDE, Sect. 4.2] and proceeds as follows: given a control volume C_i ($\rightarrow$ [NPDE, Sect. 4.2.1], [NPDE, Eq. (4.2.2)]), we state the balance law encoded in (12.3.1) for it:

$$\int_{\partial C_i} \mathbf{j}(u_N) \cdot \boldsymbol{n} \mathrm{d}S \approx \sum_{K \in U_i} \Psi(\mu_i, \mu_k) = \int_{C_i} f \mathrm{d}\boldsymbol{x}\ , \tag{12.3.2}$$

where U_i is the set of control volumes adjacent to C_i, and $\Psi(\mu_i, \mu_k)$ is the *numerical flux* from C_i to C_k. The coefficient μ_i agrees with the value $u_N(\boldsymbol{p}_i)$ of the approximate solution $u_N \in \mathcal{S}_1^0(\mathcal{M})$ in the vertex $\boldsymbol{p}_i$ of the primal mesh. In this problem triangular (primal) meshes and *orthogonal dual meshes* (Voronoi cells) will be used, see [NPDE, Eq. (4.2.3)].

Finding a suitable expression for the numerical flux is key to constructing a good finite volume method. Here the idea is to "project" the PDE onto the line connecting the two vertices at the center of adjacent control volumes. This projection yields a scalar linear ODE, which can be solved analytically. From this analytic solution we recover an approximate flux.

In detail, we start from the expression

$$\Psi(\mu_i, \mu_k) = |\boldsymbol{f}_{ik}| J(\mu_i, \mu_k),$$

where $|\boldsymbol{f}_{ik}|$ is the length of the common interface Γ_{ik} between C_i and C_k, and

$$J(\mu_i, \mu_k) = \epsilon w'(0) + \widehat{v}(\boldsymbol{m}_{ik}, i, k)w(0),$$

where $\widehat{v}(\boldsymbol{x}, i, k)$ is the length of the projection of $\boldsymbol{v}(\boldsymbol{x})$ onto the vector $\boldsymbol{p}_k - \boldsymbol{p}_i$, i.e. it is the component of the velocity directed in the $i \to k$ direction (note it is a scalar), and $w = w(\xi) : [0, d_{ik}] \mapsto \mathbb{R}$ solves a linear scalar 2-point boundary value problem "along the vector $\boldsymbol{p}_k - \boldsymbol{p}_i$",

$$(\epsilon w'(\xi) + \widehat{v}(\boldsymbol{m}_{ik}, i, k)w(\xi))' = 0, \tag{12.3.3}$$

where $\boldsymbol{m}_{ik}$ is the midpoint between $\boldsymbol{p}_i$ and $\boldsymbol{p}_k$, with boundary conditions

$$w(0) = \mu_i \quad , \quad w(d_{ik}) = \mu_k,$$

where d_{ik} is the length of the vector $\boldsymbol{p}_k - \boldsymbol{p}_i$.

(12.3a) Treating, $\widehat{v}$, μ_i and μ_k as parameters, show that the analytical solution to (12.3.3) is:

- if $\widehat{v} \neq 0$:

$$w(\xi) = \frac{1}{\widehat{v}}(C_1 - C_2 e^{-\frac{\widehat{v}}{\epsilon}\xi}) \tag{12.3.4}$$

  with $C_1 = \widehat{v}\mu_i + C_2$ and $C_2 = \frac{\widehat{v}(\mu_k - \mu_i)}{1 - e^{-\frac{\widehat{v}}{\epsilon}d_{ik}}}$

- if $\widehat{v} = 0$:

$$w(\xi) = \frac{\mu_k - \mu_i}{d_{ik}}\xi + \mu_i \tag{12.3.5}$$

(12.3b) Give a formula for $\Psi(\mu_i, \mu_k)$.

(12.3c) Refresh your knowledge about the construction of Voronoi dual meshes, see [NPDE, Sect. 4.2.2].

(12.3d) We will use LehrFEM to generate a finite element mesh for this problem, and then build a Voronoi dual mesh for finite volumes, as described in [NPDE, Sect. 4.2.2]. Implement a LehrFEM function

```
cc = computeCircumcenters(Mesh)
```

that takes a LehrFEM mesh `Mesh`, and computes a $P \times 2$ matrix of circumcenters of each element. Output an error if you detect an obtuse triangle (the position of the circumcenter relative to the triangle can be checked with barycentric coordinates), *cf.* [NPDE, Rem. 4.2.4].

HINT: The circumcenter of a triangle is the intersection of the perpendicular bisectors of each edge.

(12.3e) Implement a LehrFEM function

```
STIMA_FV_CD(eps, v, Mesh, i, cc)
```

that computes and returns a 3×3 "element" matrix for triangle i. Here, integration should be performed over edges Γ_{jk} where j, k are vertices of triangle i, but of course, only over the parts of the edges that lie within the triangle! It should take as arguments the value of ϵ, a function handle for $\boldsymbol{v}$, the finite element mesh, the triangle number and the vector of circumcenters.

HINT: The "element matrix point of view" on finite volume methods was developed in [NPDE, Sect. 4.2.3].
Furthermore, be careful in not divinding by zero when implementing the formula for the fluxes.

(12.3f) Implement a function

```
[A, L, U, FreeDofs] = assemble_FV_XD(Mesh, eps, v, g, f)
```

that assembles "everything" needed for a finite volume computation of this problem: the global matrix `A`, the right-hand side `L`, the solution vector `U` containing the Dirichlet boundary data and the vector `FreeDofs` listing the free nodes. It requires a mesh `Mesh`, the value of ϵ, and function handles for $\boldsymbol{v}$, g and f. For computation of the right-hand side of (12.3.2), use one-point numerical quadrature,

$$\int_{C_i} f \mathrm{d}\boldsymbol{x} \approx |C_i| f(\boldsymbol{p}_i).$$

HINT: You will have to find an efficient way to calculate the area $|C_i|$.

(12.3g) To test our method we will use a test problem where $\boldsymbol{v}(\boldsymbol{x}) = (1, 1/2 - x_2)$ and $u(\boldsymbol{x}) = \sin(4\pi x_2)\mathrm{e}^{-x_1}$. Compute the corresponding f. For the Dirichlet data g, simply use the restriction of u to the boundary.

(12.3h) Implement a MATLAB script `main_cd.m` that computes the finite volume solution on a sequence of meshes of the unit square generated in the following way:

1. The base mesh is a very coarse mesh generated "by hand" with two elements and four vertices.
2. This mesh is then refined using the LehrFEM function `refine_REG`...
3. ...and then smoothed using the LehrFEM function `smooth`.

By calling `refine_REG` and then `smooth` a number of times, we can generate a sequence of increasingly fine meshes. Note that the mesh will need to have boundary flags set before you call `smooth`, see [LehrFEM, Sect. 1] for details.

(12.3i) Study the convergence of $\|u - u_N\|_{L^2}$ for a suitable sequence of meshes and $\epsilon = 10^{-4}, 10^{-6}, 10^{-8}$. Plot the errors and compute the rates of convergence.

Listing 12.3: Testcalls for Problem 12.3

```
clear Mesh
Mesh.Coordinates = [0 0; 1 0; 1 1; 0 1];
Mesh.Elements = [1 2 3; 1 3 4];
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
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 = refine_REG(Mesh);
    for k = 1:5
        Loc = get_BdEdges(Mesh);
        FixedVertices = Mesh.Edges(Loc,:);
        FixedVertices = unique(FixedVertices);
        FixedPos = zeros(size(Mesh.Coordinates,1));
        FixedPos(FixedVertices)=1;
        Mesh = smooth(Mesh,FixedPos);
    end

eps = 1e-6;

v = @(x,varargin)[ones(size(x,1),1), .5-x(:,2)];
u = @(x,varargin)sin(4*pi*x(:,2)).*exp(-x(:,1));
g = @(x,varargin)u(x);
f = @(x,varargin)(((.5-x(:,2)).*cos(4*pi*x(:,2)).*exp(-x(:,1))) -
   eps*(1+16*pi^2)*u(x));

fprintf('\n##computeCircumcenters');
cc=computeCircumcenters(Mesh)

fprintf('\n##STIMA_FV_CD');
STIMA_FV_CD(eps,v,Mesh,1,cc)

fprintf('\n##assemble_FV_XD');
assemble_FV_XD(Mesh,eps,v,g,f)
```

Listing 12.4: Output for Testcalls for Problem 12.3

```
>> test_call

##computeCircumcenters
cc =

    0.2500    0.2500
    0.7500    0.2500
    0.7500    0.7500
    0.7500    0.2500
    0.2500    0.2500
    0.7500    0.7500
    0.2500    0.7500
```

```
    0.2500    0.7500

##STIMA_FV_CD
ans =

         0    0.2500         0
         0   -0.2500    0.0625
         0         0   -0.0625

##assemble_FV_XD
ans =

   (2,2)       -0.2500
   (5,2)        0.2500
   (3,3)       -0.2500
   (9,3)        0.2500
   (1,5)        0.2500
   (5,5)       -0.2500
   (5,6)        0.1250
   (6,6)       -0.7500
   (7,6)        0.5000
   (9,6)        0.1250
   (1,7)        0.0625
   (4,7)        0.0625
   (7,7)       -0.1250
   (2,8)        0.0625
   (3,8)        0.0625
   (6,8)        0.5000
   (8,8)       -0.6250
   (4,9)        0.2500
   (9,9)       -0.2500
```

Problem 12.4 One-Dimensional Convection-Diffusion Problem (Core problem)

In this problem we come across a rather strange method for avoiding the spurious oscillations that usually beset the standard Galerkin discretization of singularly perturbed convection-diffusion problems. This method relies on imposing Dirichlet boundary conditions weakly in the variational formulation and is known as consistently penalized stabilized Galerkin method.

For $\epsilon > 0$ we consider the one-dimensional convection diffusion problem on $\Omega = (0,1)$

$$-\epsilon \frac{\mathrm{d}^2 u}{\mathrm{d}x^2} + \frac{\mathrm{d}u}{\mathrm{d}x} = f(x) \quad \text{in } (0,1), \quad u(0) = u(1) = 0. \tag{12.4.1}$$

The following variational formulation has been suggested for (12.4.1): seek $u \in H^2((0,1))$ such

that

$$\int_0^1 \epsilon \frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x} + \frac{\mathrm{d}u}{\mathrm{d}x}v\,\mathrm{d}x + u(0)v(0) + \epsilon\Big(\frac{\mathrm{d}u}{\mathrm{d}x}(0)v(0) - \frac{\mathrm{d}u}{\mathrm{d}x}(1)v(1) + u(0)\frac{\mathrm{d}v}{\mathrm{d}x}(0) - u(1)\frac{\mathrm{d}v}{\mathrm{d}x}(1) + \alpha u(0)v(0) + \alpha u(1)v(1)\Big) = \int_0^1 fv\,\mathrm{d}x, \quad (12.4.2)$$

for all $v \in H^2((0,1))$. Here $\alpha > 0$ is a parameter.

(12.4a) Show that a smooth solution of (12.4.1) will also solve (12.4.2).

(12.4b) Compute the linear system of equations arising from the Galerkin finite element discretization of (12.4.2) by means of piecewise linear Lagrangian finite elements on an equidistant grid with meshwidth $h := \frac{1}{N}$, $N \in \mathbb{N}$. Use the trapezoidal rule for the approximate evaluation of the integrals.

(12.4c) Write a MATLAB function

```
u = solve2pcdbvp(N,epsilon,f_hd)
```

that solves (12.4.1) with the discretization introduced in subproblem (12.4b) using N grid cells. The argument `f_hd` is a function handle of type `@(x)` providing the function f. The vector `u` is to return the values of the finite element solution at the nodes of the mesh. Choose $\alpha = \frac{10}{h}$.

HINT: The function `solve2pcdbvpRef.p` supplies a reference implementation of `solve2pcdbvp`.

(12.4d) For $f \equiv 1$, $\epsilon = 0.01$, create a suitable plot of the error norm

$$\mathrm{err}(N) = \left(\frac{1}{N}\sum_{j=1}^{N-1} \left|(u - u_N)(jh)\right|^2\right)^{\frac{1}{2}}$$

and use it to describe qualitatively and quantitatively the convergence of the method in this norm.

HINT: The exact solution in the case $f \equiv 1$ is

$$u(x) = x + \frac{\exp(\frac{x-1}{\epsilon}) - \exp(-\frac{1}{\epsilon})}{\exp(-\frac{1}{\epsilon}) - 1}.$$

Listing 12.5: Testcalls for Problem 12.4

```
1 fprintf('\n##solve2pcdbvp')
2 solve2pcdbvp(10, 0.01, @(x) 1)
```

Listing 12.6: Output for Testcalls for Problem 12.4

```
1 >> test_call
2
3 ##solve2pcdbvp
4 ans =
5
```

```
   -0.0051
    0.1143
    0.1865
    0.3282
    0.3656
    0.5595
    0.5186
    0.8300
    0.6130
    1.1885
    0.4602
```

Published on May 15.

To be submitted on May 21. **MATLAB:** Submit all files in the online system. Include the files that generate the plots. Label all your plots. Include commands to run your functions. Comment on your results.

References

[NPDE] Lecture Slides for the course "Numerical Methods for Partial Differential Equations", SVN revision # 55075.

[NCSE] Lecture Slides for the course "Numerical Methods for CSE".

[LehrFEM] LehrFEM manual.

Last modified on May 22, 2013