

Homework Problem Sheet 5

Introduction. The first problem is the implementation of Crank-Nicolson time stepping scheme coupled with linear finite element discretization of the heat equation. You will reuse most of the FEM code that you have written for the Problem 1 of the Exercise sheet 3. The second problem is the “introductory” problem for the *hyperbolic* partial differential equations, the method of characteristics and the finite volume method using upwinding, which will be introduced in the lectures on May 5th-6th.

Problem 5.1 Parabolic Timestepping with Crank-Nicolson (Core problem)

Let $\Omega := (0, 1)^2$ and consider the problem

$$\begin{aligned}\frac{\partial u}{\partial t} - \Delta u &= f && \text{in } (0, T] \times \Omega, \\ u &= g && \text{on } (0, T] \times \partial\Omega, \\ u &= u_0 && \text{on } \{0\} \times \Omega,\end{aligned}$$

where f, g and u_0 are given such $u(t, \mathbf{x}) = \cos(2\pi x_1) \sin(t\pi x_2)$ is the exact solution.

(5.1a) Derive the variational formulation for this parabolic problem.

HINT: Fix $t \in (0, T)$ and integrate by parts in \mathbf{x} to obtain conditions on $u(t) \in H^1(\Omega)$.

Solution: Fixing $t \in (0, T)$ and integration by parts in \mathbf{x} , yields for $v \in V = H_0^1(\Omega)$,

$$\begin{aligned}\int_{\Omega} \frac{\partial u}{\partial t} v \, d\mathbf{x} - \int_{\Omega} \Delta u v \, d\mathbf{x} &= \int_{\Omega} f v \, d\mathbf{x} \\ \Rightarrow \int_{\Omega} \frac{\partial u}{\partial t} v \, d\mathbf{x} + \int_{\Omega} \text{grad } u \cdot \text{grad } v \, d\mathbf{x} &= \int_{\Omega} f v \, d\mathbf{x}.\end{aligned}$$

The variational formulation reads:

Find $u(t) \in H^1(\Omega)$, $\frac{\partial u}{\partial t} \in L^2(\Omega)$, $u(t)|_{\partial\Omega} = g$ such that

$$\begin{aligned}\int_{\Omega} \frac{\partial u}{\partial t} v \, d\mathbf{x} + \int_{\Omega} \text{grad } u \cdot \text{grad } v \, d\mathbf{x} &= \int_{\Omega} f v \, d\mathbf{x} \quad \forall v \in V, \\ \int_{\Omega} u(0) v \, d\mathbf{x} &= \int_{\Omega} u_0 v \, d\mathbf{x} \quad \forall v \in V,\end{aligned}\tag{5.1.1}$$

(5.1b) Show that the initial value problem arising from a spatial discretization of the variational formulation using piecewise linear finite elements with basis functions $\{b_N^i\}_i$ is given by

$$\begin{aligned} \mathbf{M} \frac{d}{dt} \vec{\mu}(t) + \mathbf{A} \vec{\mu}(t) &= \mathbf{F}(t), \\ \mathbf{M} \vec{\mu}(0) &= \vec{\mu}_0, \end{aligned} \quad (5.1.2)$$

where $\vec{\mu}(t)$ is the finite element coefficient vector, $\vec{\mu}_0 = \vec{\mu}(0)$, \mathbf{F} is the time-dependent load vector

$$F_i(t) = \int_{\Omega} f(t, \mathbf{x}) b_N^i(\mathbf{x}) d\mathbf{x},$$

and \mathbf{M} and \mathbf{A} are the mass- and Galerkin matrices respectively

$$M_{ji} = \int_{\Omega} b_N^i(\mathbf{x}) b_N^j(\mathbf{x}) d\mathbf{x}, \quad A_{ji} = \int_{\Omega} \text{grad } b_N^i(\mathbf{x}) \cdot \text{grad } b_N^j(\mathbf{x}) d\mathbf{x}.$$

Solution: The discretization of (5.1.1) reads: Find $u_N(t) \in V_N$ such that $(u_N^0 - u_0(x), v_N) = 0$, $\forall v_N \in V_N$, and

$$\left(\frac{\partial u_N(t)}{\partial t}, v_N \right) + (\text{grad } u_N(t), \text{grad } v_N) = (f(t), v_N) \quad \forall v_N \in V_N. \quad (5.1.3)$$

This is equivalent to a matrix equation: Let b_1, \dots, b_N be a basis of V_N and set $u_N(t, x) = \sum_{i=1}^N \mu_i(t) b_i(x) \in V_N$. Let $\vec{\mu}(t) = \{\mu_i(t)\}_{i=1}^N$ denote the vector of coefficients, $F_i(t) = (f(t), b_i)$ and $\mu_{0,i} = (u_0, b_i)$. Then (5.1.3) can be written in matrix form

$$\begin{aligned} \mathbf{M} \frac{\partial}{\partial t} \vec{\mu}(t) + \mathbf{A} \vec{\mu}(t) &= \mathbf{F}(t), \\ \mathbf{M} \vec{\mu}(0) &= \vec{\mu}_0, \end{aligned}$$

where $M_{ji} = (b_i, b_j)_{L^2(\Omega)}$ and $A_{ji} = (\text{grad } b_i, \text{grad } b_j)_{L^2(\Omega)}$.

(5.1c) For an initial value problem

$$\frac{\partial}{\partial t} \mathbf{y} = \mathbf{h}(t, \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0,$$

let the time-stepping scheme be given for $m = 0, \dots, K$ by the *Crank-Nicolson scheme*

$$\mathbf{y}^{(m+1)} = \mathbf{y}^{(m)} + \frac{1}{2} \Delta t (\mathbf{h}(t_m, \mathbf{y}^{(m)}) + \mathbf{h}(t_{m+1}, \mathbf{y}^{(m+1)})),$$

with initial value $\mathbf{y}^{(0)} = \mathbf{y}_0$, time step $\Delta t := T/K$ and time points $t_m := m\Delta t$.

Show that the Crank-Nicolson scheme applied to (5.1.2) gives the following linear system:

$$\left(\mathbf{M} + \frac{1}{2} \Delta t \mathbf{A} \right) \vec{\mu}^{(m+1)} = \left(\mathbf{M} - \frac{1}{2} \Delta t \mathbf{A} \right) \vec{\mu}^{(m)} + \frac{1}{2} \Delta t (\mathbf{F}_{m+1} + \mathbf{F}_m), \quad (5.1.4)$$

where $\mathbf{F}_m = \mathbf{F}(t_m)$.

Solution: Applying the Crank-Nicolson scheme to the matrix equation yields

$$\begin{aligned}\frac{\partial}{\partial t}\vec{\mu}(t) &= \mathbf{M}^{-1}(\mathbf{F}(t) - \mathbf{A}\vec{\mu}(t)) \\ \Rightarrow \quad \mathbf{M}\vec{\mu}^{(m+1)} &= \mathbf{M}\vec{\mu}^{(m)} + \frac{\Delta t}{2}(\mathbf{F}_m - \mathbf{A}\vec{\mu}^{(m)} + \mathbf{F}_{m+1} - \mathbf{A}\vec{\mu}^{(m+1)}) \\ \Rightarrow \quad \left(\mathbf{M} + \frac{\Delta t}{2}\mathbf{A}\right)\vec{\mu}^{(m+1)} &= \left(\mathbf{M} - \frac{\Delta t}{2}\mathbf{A}\right)\vec{\mu}^{(m)} + \frac{\Delta t}{2}(\mathbf{F}_m + \mathbf{F}_{m+1}).\end{aligned}$$

(5.1d) Next, we aim to implement the linear FEM with Crank-Nicolson time stepping (5.1c). For this purpose, we will reuse the routines from the Problem 1 of the Exercise sheet 3. The Galerkin matrix assembly routines

```
Aloc = STIMA_Heat_LFE(Vertices, QuadRule, FHandle)
A = assemMat_LFE(Coordinates, EHandle, varargin)
```

can be reused without any modifications.

Modify the load vector assembly routines

```
Lloc = LOAD_LFE(Vertices, QuadRule, FHandle)
L = assemLoad_LFE(Coordinates, QuadRule, FHandle)
```

to compute the *time-dependent* load vector \mathbf{F} in (5.1.2).

HINT: For the detailed description of the above functions, refer to the Problem 1 of the Exercise sheet 3.

Solution: See Listing 5.1 and Listing 5.2 for the codes.

Listing 5.1: Implementation for LOAD_LFE

```
1  function Lloc = LOAD_LFE(Vertices, QuadRule, FHandle, varargin)
2
3  % Initialize constants
4
5  nPoints = size(QuadRule.w, 1);
6
7  % Preallocate memory
8
9  Lloc = zeros(3, 1);
10
11 % Compute element mapping
12
13 bK = Vertices(1, :);
14 BK = [Vertices(2, :)-bK; Vertices(3, :)-bK];
15 inv_BK = inv(BK);
16 det_BK = abs(det(BK));
17
18 x = QuadRule.x*BK+ones(nPoints, 1)*bK;
```

```

19
20 % Compute element load vector
21
22 FVal = FHandle(x,varargin{:});
23 N = shap_LFE(QuadRule.x);
24
25 Lloc(1) = sum(QuadRule.w.*FVal.*N(:,1))*det_BK;
26 Lloc(2) = sum(QuadRule.w.*FVal.*N(:,2))*det_BK;
27 Lloc(3) = sum(QuadRule.w.*FVal.*N(:,3))*det_BK;

```

Listing 5.2: Implementation for `assemLoad_LFE`

```

1 function L = assemLoad_LFE(Mesh,QuadRule,FHandle,varargin)
2
3 % Copyright 2005-2005 Patrick Meury
4 % SAM - Seminar for Applied Mathematics
5 % ETH-Zentrum
6 % CH-8092 Zurich, Switzerland
7
8 % Initialize constants
9
10 nCoordinates = size(Mesh.Coordinates,1);
11 nElements = size(Mesh.Elements,1);
12
13 % Preallocate memory
14
15 L = zeros(nCoordinates,1);
16
17 % Assemble element contributions
18
19 for i = 1:nElements
20
21 % Extract vertices
22
23 vidx = Mesh.Elements(i,:);
24 Vertices = Mesh.Coordinates(vidx,:);
25
26 % Compute load data
27
28 Lloc = LOAD_LFE(Vertices,QuadRule,FHandle,varargin{:});
29
30 % Add contributions to global load vector
31
32 L(vidx(1)) = L(vidx(1)) + Lloc(1);
33 L(vidx(2)) = L(vidx(2)) + Lloc(2);
34 L(vidx(3)) = L(vidx(3)) + Lloc(3);
35
36 end

```

```

37
38 return

```

(5.1e) Implement the local mass matrix routine

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

which will be used in the `assemMat_LFE` routine for the assembly of the global mass matrix M in (5.1.2).

HINT: Modify the existing routine `Aloc = STIMA_Heat_LFE`.

Solution: See [Listing 5.3](#) for the code.

Listing 5.3: Implementation for MASS_LFE

```

1 function Mloc = MASS_LFE(Vertices,varargin)
2 % MASS_LFE Element mass matrix.
3 %
4 %   MLOC = MASS_LFE(VERTICES) computes the element mass
5 %   matrix using linear
6 %   Lagrangian finite elements.
7 %
8 %   VERTICES is 3-by-2 matrix specifying the vertices of the
9 %   current element
10 %   in a row wise orientation.
11 %
12 %   Example:
13 %
14 %   Mloc = MASS_LFE(Vertices);
15 %
16 %   Copyright 2005-2005 Patrick Meury
17 %   SAM - Seminar for Applied Mathematics
18 %   ETH-Zentrum
19 %   CH-8092 Zurich, Switzerland
20
21 % Compute element mapping
22
23 bK = Vertices(1,:);
24 BK = [Vertices(2,:)-bK; ...
25       Vertices(3,:)-bK];
26 det_BK = abs(det(BK));
27
28 % Compute local mass matrix
29
30 Mloc = det_BK/24*[2 1 1; 1 2 1; 1 1 2];
return

```

(5.1f) Implement a function

```
U = Crank_Nicolson_LFE(Mesh, K, T, G_HANDLE, F_HANDLE, U0_HANDLE)
```

to compute the FE solution (vector of coefficients U) using the K iterations of the Crank-Nicolson time stepping scheme (5.1.4) up to a specified time T .

HINT: For *each* iteration of the Crank-Nicolson time stepping, you will need to solve (numerically) the resulting linear system for the coefficients of U .

HINT: For efficiency, construct the matrices $M + \frac{1}{2}\Delta t A$ and $M - \frac{1}{2}\Delta t A$ only once.

HINT: Use the supplied function `P7O6` for any quadrature that you might need.

HINT: For the implementation of the Dirichlet boundary conditions, reuse the routine

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

from the Problem 1 of the Exercise sheet 3 - you will need to modify it to accept an additional argument t indicating the time t .

Solution: See [Listing 5.4](#) for the code.

Listing 5.4: Implementation for Crank_Nicolson_LFE

```
1
2 % Copyright 2014 Jonas Sukys
3 % Adapted from Christoph Winter, 2008
4 % SAM - Seminar for Applied Mathematics
5 % ETH-Zentrum
6 % CH-8092 Zurich, Switzerland
7
8 function [ U, Ndofs ] = Crank_Nicolson_LFE ( Mesh, K, T,
9       G_HANDLE, F_HANDLE, U0_HANDLE )
10
11 % compute time step size
12 dt = T/K;
13
14 % set DHandle for matrix assembly routines
15 DHandle = @(x) 1;
16
17 % pre-compute mass and stiffness matrices
18 M = assemMat_LFE (Mesh, @MASS_LFE, P7O6(), DHandle);
19 A = assemMat_LFE (Mesh, @STIMA_Heat_LFE, P7O6(), DHandle);
20
21 % pre-compute matrices that will be required for the linear
22 % system
23 S1 = M + 0.5*dt * A;
24 S2 = M - 0.5*dt * A;
25
26 % compute initial data
```

```

25 u_old = assemLoad_LFE (Mesh, P7O6(), U0_HANDLE);
26 u_old = M \ u_old;
27
28 % compute load vector (at time 0)
29 F_old = assemLoad_LFE (Mesh, P7O6(), F_HANDLE, 0);
30
31 % start time-stepping
32 for i = 1:K
33
34     % assemble the new load vector (at time i*dt)
35     F_new = assemLoad_LFE (Mesh, P7O6(), F_HANDLE, i*dt);
36
37     % compute the RHS of the linear system
38     rhs = S2 * u_old + dt/2 * (F_new + F_old);
39
40     % incorporate Dirichlet boundary data (at time i*dt)
41     [u_new, FreeDofs] = assemDir_LFE (Mesh, -1, G_HANDLE,
42                                     i*dt);
43     rhs = rhs - S1 * u_new;
44
45     % solve the linear system
46     u_new (FreeDofs) = S1(FreeDofs,FreeDofs) \ rhs(FreeDofs);
47
48     % update vectors
49     u_old = u_new;
50     F_old = F_new;
51 end
52
53 % output the result
54 U = u_new;
55 Ndofs = length(FreeDofs);
56 end

```

(5.1g) Implement a function

```
U = plot_CrankNicolson_LFE()
```

which computes the solution using CrankNicolson_LFE for $T = 0.5, 1.0, 1.5, 2.0$ and plots it using the plot_LFE routine from the Problem 1 of the Exercise sheet 3 (you may modify the plot_LFE routine to indicate the time T in the title). Use the mesh Square3.mat and $K = 100$ time steps (for $T = 2$; use proportionally smaller K for other values of T). For each time $T = 0.5, 1.0, 1.5, 2.0$, you can recompute the solution from $t = 0$.

Does your implementation of the Crank-Nicolson time stepping scheme approximate the exact solution correctly?

Solution: See [Listing 5.5](#) for the code and [Figure 5.1](#) for the plots of the approximated solution.

Listing 5.5: Implementation for plot_Crank_Nicolson_LFE

```

1
2 % Copyright 2014 Jonas Sukys
3 % Adapted from Christoph Winter, 2008
4 % SAM - Seminar for Applied Mathematics
5 % ETH-Zentrum
6 % CH-8092 Zurich, Switzerland
7
8 function plot_Crank_Nicolson_LFE ()
9
10 % problem data
11 T = [0.5, 1, 1.5, 2]; % Final times
12 G_HANDLE = @g; % Dirichlet boundary data
13 F_HANDLE = @f; % Right-hand side source term
14 U0_HANDLE = @u0; % Initial data
15 UEX_HANDLE = @uex; % Exact solution
16
17 % mesh
18 Mesh = load ('Square3.mat');
19
20 % number of time steps
21 K = [25, 50, 75, 100];
22
23 figure ('Name', 'Linear finite elements');
24 % compute and plot the solutions
25 for i = 1:length(T)
26     subplot (2,2,i);
27     [U, ~] = Crank_Nicolson_LFE (Mesh, K(i), T(i), G_HANDLE,
28         F_HANDLE, U0_HANDLE);
29     plot_LFE (U, Mesh);
30     title (['solution at time t=' num2str (T(i))]);
31     colorbar;
32 end
33
34 print -depsc '../fig/plot.eps'
35
36 end

```

(5.1h) Implement a function

```
U = conv_Crank_Nicolson_LFE()
```

which computes the solution for $T = 1$ using Crank_Nicolson_LFE on the series of meshes

```
Square1.mat - Square4.mat
```

and plots the $L^2(\Omega)$ -error convergence. What type of convergence and what order do you observe?

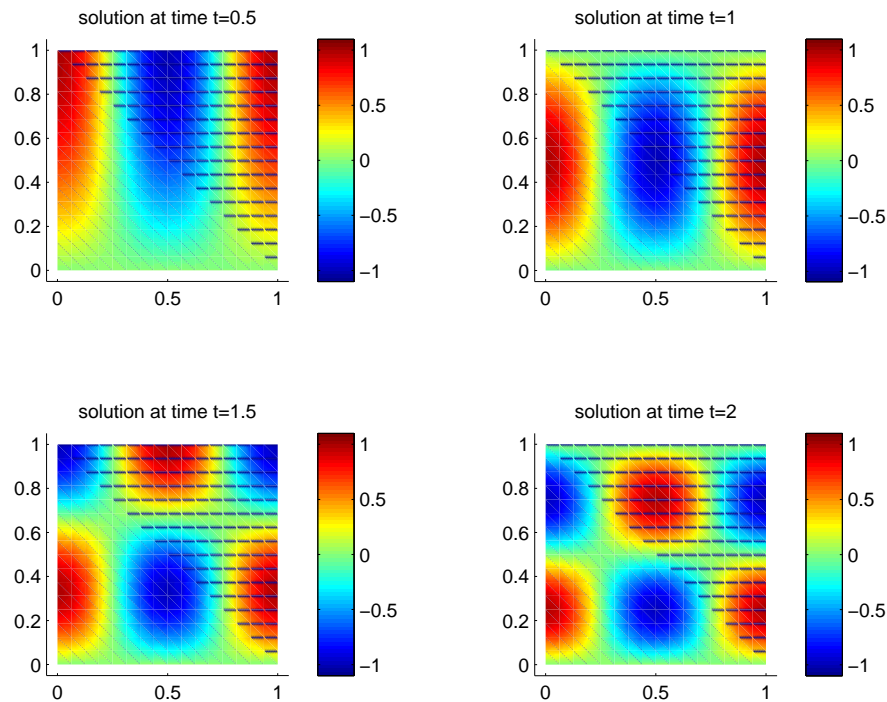


Figure 5.1: FEM approximations of the solution to the heat equation using the Crank-Nicolson time-stepping scheme.

Use the supplied function `P706` for any quadrature that you might need. For each level of mesh refinement, use the number of timesteps needed to balance the errors from time and space discretization.

HINT: Use the provided `L2Err_LFE` routine to compute the $L^2(\Omega)$ -error of the solution.

Solution: To equilibrate the time-stepping and spatial discretization errors in $\mathcal{O}(\Delta t^2 + \Delta x^2)$, we require $\Delta t \sim \Delta x$, i.e. $\Delta t \sim \sqrt{N}$, where N is the total number of degrees of freedom. See [Listing 5.6](#) for the code and [Figure 5.2](#) for the convergence plot. We observe the first order convergence with respect to the number of degrees of freedom, which asymptotically scales as $\mathcal{O}(\Delta t^2)$, or, equivalently as $\mathcal{O}(\Delta x^2)$, as expected.

Listing 5.6: Implementation for `conv_Crank_Nicolson_LFE`

```

1
2 % Copyright 2014 Jonas Sukys
3 % Adapted from Christoph Winter, 2008
4 % SAM - Seminar for Applied Mathematics
5 % ETH-Zentrum
6 % CH-8092 Zurich, Switzerland
7
8 function conv_Crank_Nicolson_LFE ()
9
10 % problem data
11 T = 1; % Final time
12 G_HANDLE = @g; % Dirichlet boundary data

```

```

13 F_HANDLE = @f;           % Right-hand side source term
14 U0_HANDLE = @u0;         % Initial data
15 UEX_HANDLE = @uex;       % Exact solution
16
17 NREFS = 4;
18
19 for j = 1:NREFS
20
21     % load mesh and print some info
22     Mesh = load(['Square' int2str(j) '.mat']);
23     fprintf('Number of mesh points %4d\n',
24             size(Mesh.Coordinates,1))
25
26     % compute the required number of time steps
27     K = ceil(sqrt(size(Mesh.Coordinates,1))*T);
28
29     % solve
30     [U, Ndofs(j)] = Crank_Nicolson_LFE (Mesh, K, T, G_HANDLE,
31                                         F_HANDLE, U0_HANDLE);
32
33     % compute errors
34     L2err(j) = L2Err_LFE(Mesh,U,P706(),@(x)UEX_HANDLE(x,T));
35 end
36
37 loglog(Ndofs,L2err);
38 title('Error convergence for the Crank-Nicolson linear finite
39       elements')
40 xlabel('Ndofs')
41 ylabel('L^2 error')
42 print -depsc '../fig/conv.eps'
43 end

```

Problem 5.2 Transport in One Dimension

Consider the one-dimensional linear transport equation:

$$\begin{aligned}
 U_t + (a(x)U)_x &= 0, & \forall (x,t) \in \mathbb{R} \times \mathbb{R}_+, \\
 U(x,0) &= U_0(x), & \forall x \in \mathbb{R},
 \end{aligned}
 \tag{5.2.1}$$

with coefficient $a(x) \in C^1(\mathbb{R})$.

(5.2a) Write down the equation for characteristics of (5.2.1). Use it to derive an expression for the exact solution.

HINT: Assume that a is an increasing function of x .

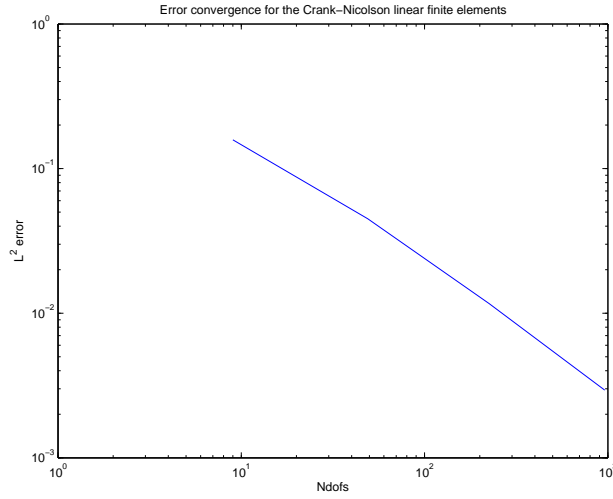


Figure 5.2: Convergence plot for the Crank-Nicolson method.

Solution: The characteristics equation for the equation (5.2.1) are,

$$\frac{dx}{dt} = a(x), \quad \frac{dU}{dt} = -a_x(x)U,$$

integrating the first equation we get,

$$\int_0^t \frac{1}{a(x)} dx = t + c,$$

here c can be calculated using $x(t = 0) = x_0$. As we do not have an expression for $a(x)$, we can not write x explicitly as function of t .

Similarly integrating second equation of the characteristics we get,

$$\log U = \int_0^t -a_x(x(t)) dt + c_1$$

taking exponential both sides we get,

$$U = C \exp \left(- \int_0^t a_x(x(t)) dt \right)$$

and C can be derived using initial conditions.

REMARK: Note that equation $\frac{dx}{dt} = a(x)$, will have unique solution if $a(x)$ is globally Lipschitz. On other hand solution set still may not fill the whole $x - t$ space.

(5.2b) Let $U(x, t)$ be a smooth solution of (5.2.1), that decays to zero at infinity. Then show that U satisfies the energy bound

$$\int_{\mathbb{R}} U^2(x, T) dx \leq e^{CT} \int_{\mathbb{R}} U_0^2(x) dx, \quad (5.2.2)$$

for all $T > 0$, with constant C depending on $\|a\|_{C^1}$.

Solution: Rewriting the equation, we get,

$$U_t + a(x)U_x = -a_x U.$$

Multiplying this with U , results in,

$$UU_t + a(x)UU_x = -a_x U^2,$$

with the chain rule,

$$\begin{aligned} \left(\frac{U^2}{2}\right)_t + \left(a(x)\frac{U^2}{2}\right)_x - a_x(x)\frac{U^2}{2} &= -a_x U^2, \\ \left(\frac{U^2}{2}\right)_t + \left(a(x)\frac{U^2}{2}\right)_x &= -a_x(x)\frac{U^2}{2}. \end{aligned}$$

integrating over space variable,

$$\frac{d}{dt} \int_{\mathbb{R}} \left(\frac{U^2}{2}\right) dx + \int_{\mathbb{R}} \left(a(x)\frac{U^2}{2}\right)_x dx = - \int_{\mathbb{R}} a_x(x) \frac{U^2}{2} dx$$

as U , vanish at infinity, we have

$$\frac{d}{dt} \int_{\mathbb{R}} \left(\frac{U^2}{2}\right) dx = - \int_{\mathbb{R}} a_x(x) \frac{U^2}{2} dx.$$

Using regularity of a ,

$$\frac{d}{dt} \int_{\mathbb{R}} \left(\frac{U^2}{2}\right) dx \leq \|a\|_{C^1} \int_{\mathbb{R}} \frac{U^2}{2} dx.$$

Using Gronwall inequality, we get the required result.

(5.2c) Consider the equation (5.2.1) on the domain $D = (0, 1)$ with periodic boundary conditions and $a = -1$. Implement a stable numerical scheme to simulate (5.2.1). Plot the results at $T = 1$ and 200 mesh cells for the following initial conditions:

Smooth Solution

$$U_0(x) = \sin(2\pi x) \tag{5.2.3}$$

Non-smooth Solution

$$U_0(x) = \begin{cases} 1, & \text{if } x < 0.5, \\ 0, & \text{otherwise.} \end{cases} \tag{5.2.4}$$

Solution: The exact solution at time $T = 1$ for the above initial condition,

$$U(x, T = 1) = U_0(x).$$

The upwind scheme with $a = -1$ is

$$U_i^{n+1} = U_i^n + \frac{\Delta t}{\Delta x} (U_{i+1}^n - U_i^n).$$

See codes 5.7 - 5.9 and Figure 5.3 for the exact and the approximate FVM solutions.

Listing 5.7: Implementation for solve

```

1 function [ x, u, dx ] = solve( u0, l, r, T, n )
2
3 dx = abs(r-l)/n;
4 dt = 0.99*dx;           % required < 1 for stability
5
6 x = l:dx:r;             % space discretization
7 x = [l-dx x r+dx];      % ghost cells for bc
8
9 u = u0(x);
10
11 for t=dt:dt:T
12     v = u;
13     v = apply_periodic_bc(v);
14     for j = 2:length(x)-1
15         u(j) = v(j) + dt/dx*(v(j+1) - v(j));
16     end
17 end
18
19 u = u(2:end-1);
20 x = x(2:end-1);
21
22 end

```

Listing 5.8: Implementation for apply_periodic_bc

```

1 function [ out ] = apply_periodic_bc( u )
2
3 out = u;
4
5 out(1) = out(end - 1);
6 out(end) = out(2);
7
8 end

```

Listing 5.9: Implementation for main

```

1 function [] = main()
2
3 N = 100*(2.^(0:6));
4 n = N(2);
5
6 u0 = @(x) sin(2*pi*x);
7 u_e = @(x) u0(x+1);
8
9 [ x, u, dx ] = solve( u0, 0, 1, 1, n );
10
11 figure;

```

```

12 plot (x,u,'ro',x,u_e(x));
13 title ('Solution for smooth case');
14 xlabel ('x');
15 ylabel ('u(x,1)');
16 legend ('approximation','exact solution');
17
18 print -depsc '../fig/plot_smooth.eps'
19
20 [err_L1, err_Li] = geterrors (u0, N);
21
22 figure;
23 loglog (N, err_L1, N, err_Li);
24 title ('Errors for smooth case');
25 legend ('L^1((0,1)) norm','L^\infty((0,1)) norm');
26 xlabel ('no. of cells');
27 ylabel ('error');
28
29 print -depsc '../fig/conv_smooth.eps'
30
31 u0 = @(x) 1*((x-floor(x))<0.5);
32 u_e = @(x) u0(x+1);
33
34 [ x, u, dx ] = solve( u0, 0, 1, 1, n );
35 figure;
36 plot (x,u,'ro',x,u_e(x));
37 title ('Solution for non-smooth case');
38 xlabel ('x');
39 ylabel ('u(x,1)');
40 legend ('approximation','exact solution');
41 axis ([-0.1 1.1 -0.1 1.1]);
42
43 print -depsc '../fig/plot_disc.eps'
44
45 [err_L1, err_Li] = geterrors (u0, N);
46 figure;
47 loglog (N, err_L1, N, err_Li);
48 title ('Errors for non-smooth case');
49 legend ('L^1((0,1)) norm','L^\infty((0,1)) norm');
50 xlabel ('no. of cells');
51 ylabel ('error');
52
53 print -depsc '../fig/conv_disc.eps'
54
55 end

```

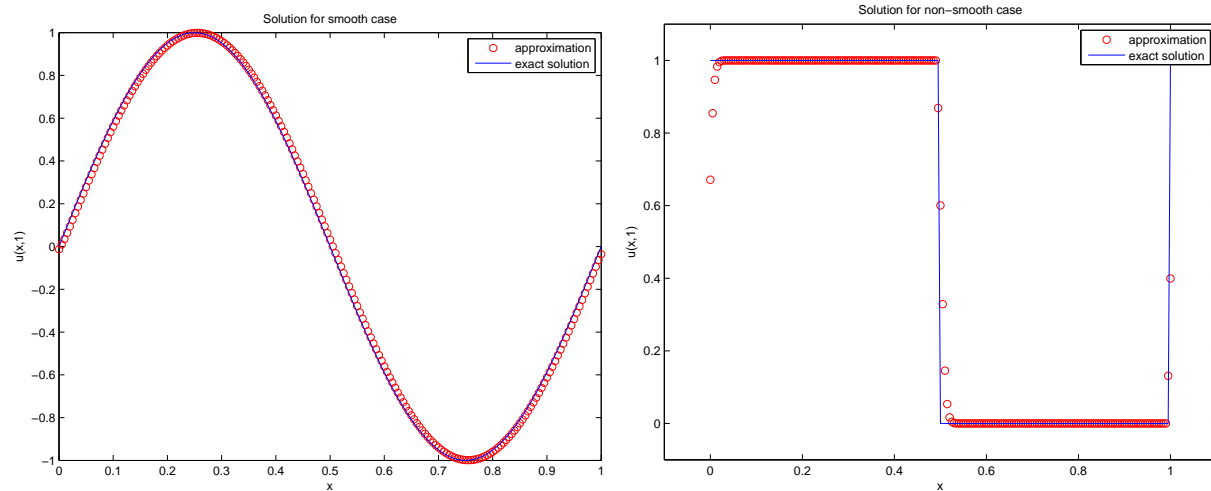


Figure 5.3: Exact and FVM approximations of smooth (left) and discontinuous (right) solutions.

(5.2d) Plot $L^1(D)$ and $L^\infty(D)$ errors vs numbers of cells (for no. of cells 100, 200, 400, 800, 1600, 3200, 6400). Use exact solution derived in sub-problem (5.2a) to calculate the errors. Comment the observed results. In which case does the method converge? What is the convergence rate?

Solution:

See codes 5.10 and 5.9. For the error convergence plots, see Figure 5.4. For the *smooth* case, we observe the *first order* convergence both in L^1 and L^∞ errors. For the *discontinuous* solutions, the L^∞ -error is *not* converging, and the L^1 error is converges with the *lower rate* $1/2$.

Listing 5.10: Implementation for `geterrors`

```
1 function [err_L1, err_Li] = geterrors( u0, N )
2
3 u_e = @(x) u0(x+1);
4
5 err_L1 = [];
6 err_Li = [];
7
8 for n = N
9     [x, u, dx] = solve(u0, 0, 1, 1, n);
10    err_L1 = [err_L1 get_L1(u, u_e(x), dx)]; %#ok<AGROW>
11    err_Li = [err_Li get_Li(u, u_e(x), dx)]; %#ok<AGROW>
12 end
13
14 end
```

Listing 5.11: Implementation for `get_L1`

```
1 function [ err ] = get_L1( u, u_e, dx )
2
3 err = dx*sum(abs(u-u_e));
4
```

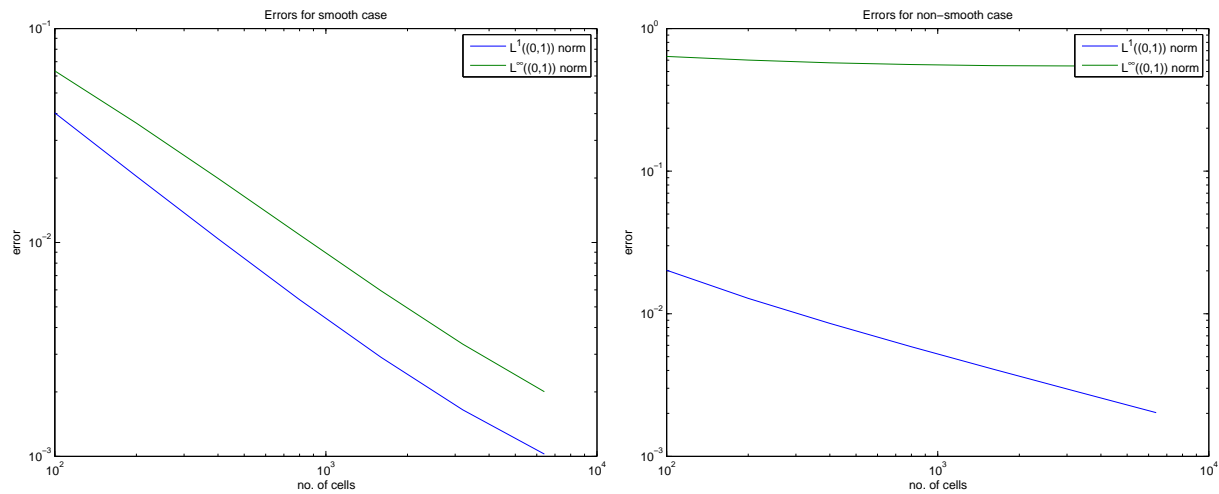


Figure 5.4: L^1 and L^∞ error convergence of the smooth (left) and the discontinuous (right) solutions. For the *smooth* case, we observe the *first* order convergence both in L^1 and L^∞ errors. For the *discontinuous* solutions, the L^∞ -error is *not* converging, and the L^1 error is converges with the *lower* rate $1/2$.

5 **end**

Listing 5.12: Implementation for `get_Li`

```

1 function [ err ] = get_Li( u, u_e, dx )
2
3 err = max(abs(u-u_e));
4
5 end

```

Published on April 30th.

To be submitted on May 12th.

Last modified on May 1, 2014