

UPPSALA UNIVERSITY



ADVANCED NUMERICAL METHODS

1TD050

Assignment 1

Author:
Csongor HORVÁTH

October 9, 2023

Introduction

In this report we are investigating the numerical solution of the following given PDE problem:

Let Ω be a fixed (open) domain in \mathbb{R}^2 , with boundary $\partial\Omega$ over a time interval $[0, T]$ with initial time zero and the final time T . We are interested in solving the following time-dependent scalar conservation laws:

$$\begin{aligned}\partial_t u + \nabla \cdot \mathbf{f}(u) &= 0, & (\mathbf{x}, t) &\in \Omega \times (0, T], \\ u(\mathbf{x}, 0) &= u_0(\mathbf{x}), & \mathbf{x} &\in \Omega,\end{aligned}$$

with appropriate boundary conditions. Here u represents the unknown variable, $f \in \mathcal{C}^1(\mathbb{R}, \mathbb{R}^2)$ is the flux term and $u_0 \in L^\infty(\mathbb{R}^2)$ is given initial data. Let $0 = t_0 < t_1 < \dots < t_N = T$ be a sequence of discrete time steps with associated time intervals $I_n = (t_{n-1}, t_n]$ of length $k_n = t_n - t_{n-1}$, $n = 1, 2, \dots, N$. Let

$$\mathcal{X}_h := \{v : v \in H^1(\Omega), v(\mathbf{x}) - \text{cont. pw. linear in } \Omega\},$$

be a finite element space consisting of continuous piecewise linear polynomials on a mesh $\mathcal{T}_h = \{K\}$ of mesh-size $h(\mathbf{x})$ and let $V_{h,0}$ be the space of all functions in \mathcal{X}_h vanishing on $\partial\Omega$. Next, let us denote by $U_0 = \hat{\pi}_h u_0$ the interpolation of the initial data into the finite element space \mathcal{X}_h . We are now ready to formulate the following finite element approximation: for $n = 1, 2, \dots, N$ find $U_n \in \mathcal{X}_h$ such that

$$\frac{1}{k_n} (U_n - U_{n-1}, v) + \frac{1}{2} (\nabla \cdot (\mathbf{f}(U_{n-1}) + \mathbf{f}(U_n)), v) = 0, \quad \forall v \in \mathcal{X}_h, \quad (1)$$

where $U_n = U_h(t_n)$ is the solution at the discrete time steps t_n . The implicit Crank-Nicholson time-stepping is used for the time discretization.

Note: In the tasks we will use $\Omega = \{\mathbf{x} : x_1^2 + x_2^2 \leq 1\}$, and Dirichlet boundary condition.

Part I

Problem 1.1

In this task we should implement a Matlab code to solve the finite element approximation given in (1). The implementation should be done with the following values: $u_0(\mathbf{x}, 0) = \frac{1}{2} \left(1 - \tanh \left(\frac{(x_1 - x_1^0)^2 + (x_2 - x_2^0)^2}{r_0^2} - 1 \right) \right)$ is the initial data with $r_0 = 0.25$, $(x_1^0, x_2^0) = (0.3, 0)$. Additionally we should use $T = 1$, $CFL = 0.5$ and create a solution using $h_{max} = \frac{1}{8}$ and $h_{min} = \frac{1}{16}$ mesh size.

First let's rewrite the flux term of equation (1) in the given way. So use $\nabla \cdot \mathbf{f}(u) := \mathbf{f}'(u) \cdot \nabla u$, assuming $\mathbf{f}'(u) = 2\pi(-x_2, x_1)$. Then knowing that ∇ is a linear operator, we get this:

$$\frac{1}{k_n} (U_n - U_{n-1}, v) + \frac{1}{2} (\mathbf{f}'(U_{n-1}) \cdot \nabla U_{n-1} + \mathbf{f}'(U_n) \cdot \nabla U_n, v) = 0, \quad \forall v \in \mathcal{X}_h,$$

After this step, we should create the system of linear equations from this FEM formulation using the appropriate basis functions. Then we can solve the equation system for U_{n+1} in case U_n is given using the given u_0 value.

We can write U_n in the form $U_n = \sum \xi_i \varphi_i$, where the sum is taken from the φ_i basis function of \mathcal{X}_h space. And in the form of the basis function the statement that $\forall v \in \mathcal{X}_h$, is equivalent with the statement that the equation holds for all basis function. From this we get the following for of the equation:

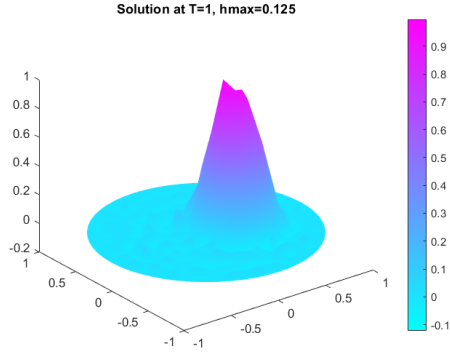
$$\frac{1}{k_n} \sum_{N_i \in \mathcal{N}_h} (\xi_{i,n} - \xi_{i,n-1}) \cdot \int_{\Omega} \varphi_j \varphi_i d\mathbf{x} + \frac{1}{2} \sum_{N_i \in \mathcal{N}_h} (\xi_{i,n} + \xi_{i,n-1}) \cdot \int_{\Omega} \mathbf{f}'(u) \nabla \varphi_j \varphi_i d\mathbf{x} = 0 \quad \forall i$$

Now we can establish the M mass matrix from the values of $\int_{\Omega} \varphi_j \varphi_i$ and C convection matrix with values from $\int_{\Omega} \mathbf{f}'(u) \nabla \varphi_j \varphi_i$. Using these matrices we can write the equation of FEM with implicit Crank-Nicholson time-discretization in the following form, which solution can be implemented easily:

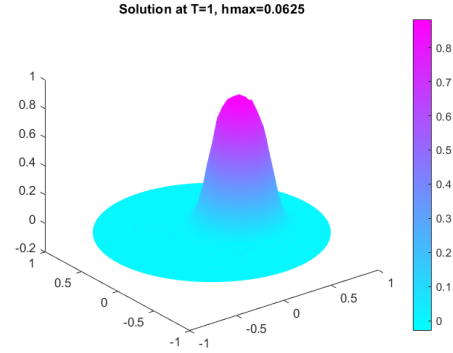
$$\frac{1}{k_n} M(\boldsymbol{\xi}_n - \boldsymbol{\xi}_{n-1}) + \frac{1}{2} C(\boldsymbol{\xi}_n + \boldsymbol{\xi}_{n-1}) = 0$$

Notes on the implementation: For the calculation of mass and convection matrix I follow the instructions given in the book *The Finite Element Method: Theory, Implementation, and Applications*

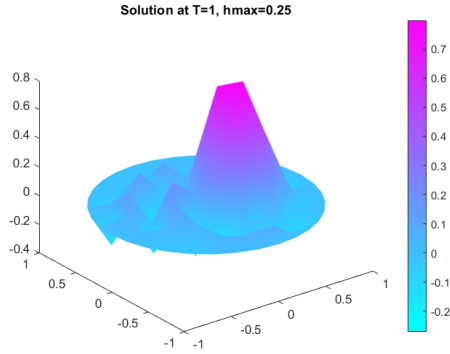
The plots of the final solutions at $T = 1$ with $hmax \in \{\frac{1}{8}, \frac{1}{16}\}$ can be found in Figure 1. As it can be seen the problem is a conservation problem, so the solution is the initial data in the given case. It can be seen that solution contains some noise for both meshes over time for both mesh. This means that even if we start from the exact solution, over time some noise occurs in the numerical solution of the equation even with finer meshes. Now let's move to the next part, where we investigate the size of the generated error.



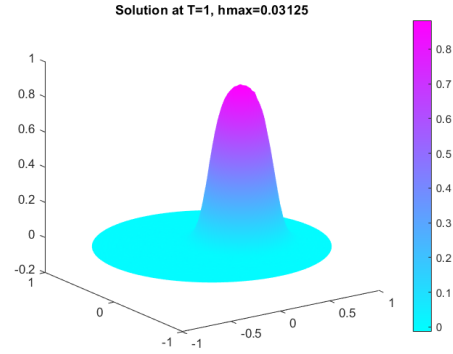
(a) Solution for $hmax = 1/8$



(b) Solution for $hmax = 1/16$



(c) Solution for $hmax = 1/4$



(d) Solution for $hmax = 1/32$

Figure 1: FEM solutions of the PDE with smooth initial data

Problem 1.2

For calculating the error, we need to know the exact solution. As I mentioned in this case that is the initial data. So the error can be calculated $e = u - U_n = u_0 - U_n \approx U_0 - U_n =: e_n$. As it is given in the assignment the norm of the error can be calculated in the following way in the n th time step: $\|e\|_{L^2(\Omega)} = \left(\int_{\Omega} e^2 dx\right)^{\frac{1}{2}} \approx \sqrt{e_n^T M e_n}$, where M is the mass matrix.

Now we should add this to the Matlab implementation and run the simulation for $h_{max} \in \{\frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}\}$ and calculate the α convergence rate. The errors of the solution can be seen in Figure 2 for the smallest and largest mesh size. As it can be seen the error is getting less with using smaller mesh size.

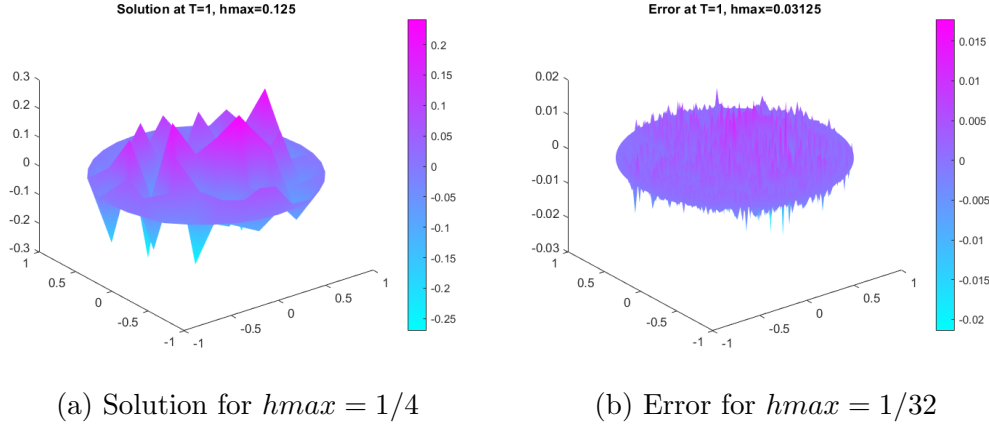


Figure 2: FEM, error of the PDE at $T = 1$, smooth initial data

The $\alpha = 1.18$ convergence rate parameter is calculated based on the error norms and a linear interpolation of the log values. The log-log plot of the errors can be seen in Figure 3 together with the h_{max} versus h_{max}^α plot.

Actually with sufficient parameter it should be possible to reach second order convergence for the FEM algorithm with the smooth initial data. To investigate this I modified the time stepping parameter CFL and changed it to 0.1 from 0.5. After this I got a much better convergence rate with $\alpha = 1.74$, which can be seen in Figure 4. This is much closer to the expected second order convergence.

(Note: with the same change there wasn't any significant improvement on the

α convergence rate with the non-smooth initial data).

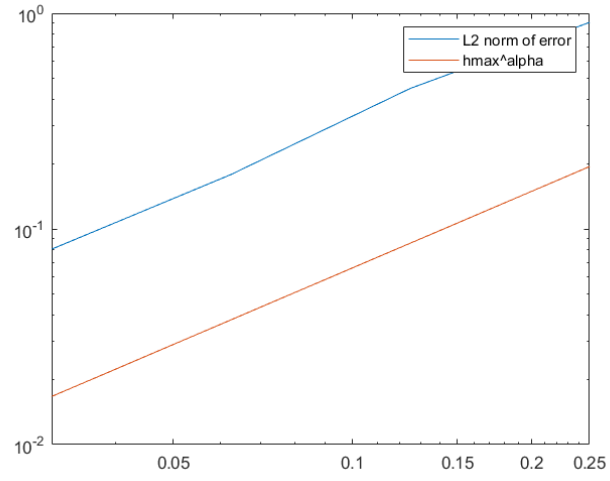


Figure 3: FEM $\alpha = 1.18$ convergence rate with smooth initial data

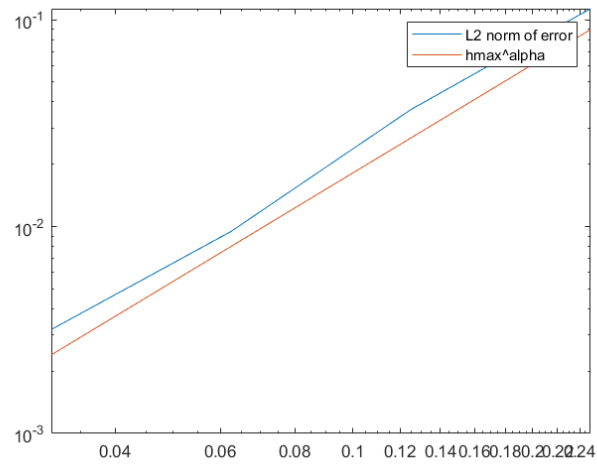


Figure 4: FEM $\alpha = 1.74$ convergence rate, smooth initial data, $CFL = 0.1$

Problem 1.3

In this part the task was to solve the same equation as in the previous two parts with different initial data.

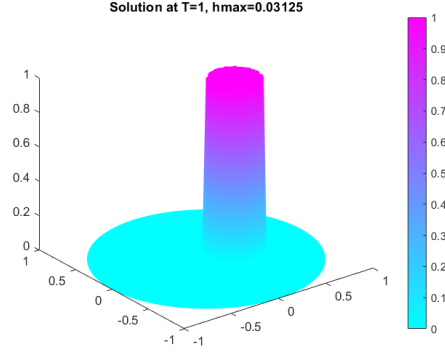


Figure 5: The non-smooth initial data

The theoretical formulation of the problems work the same way with different u_0 data. So the only thing to complete this task was to change the initial data in the MATLAB code and create similar figures as before.

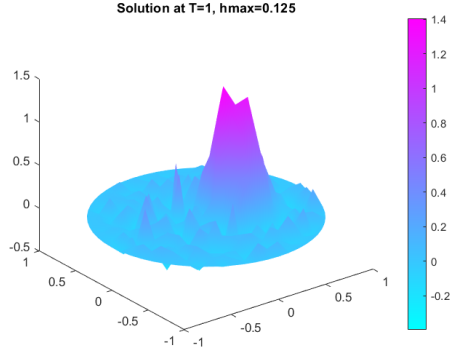
Now the initial data is set to:

$$u_0(\mathbf{x}, 0) = \begin{cases} 1 & \text{if } (x_1 - x_1^0)^2 + (x_2 - x_2^0)^2 \leq r_0^2 \\ 0 & \text{otherwise} \end{cases}$$

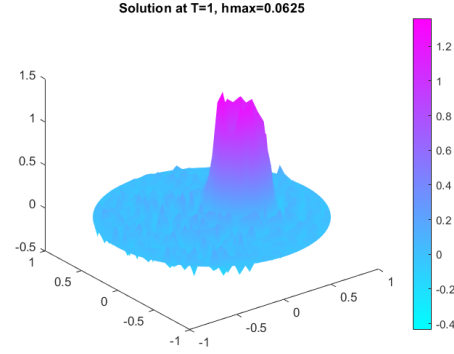
So now the initial data is non-smooth and therefore the simulation is expected to result in worse results. As for a reference the initial data is plotted in Figure 5.

The plots for the solutions at the final time ($T = 1$) with different mesh sizes can be seen in Figure 6. As it could be expected these results are more noisy then the ones in the case of the smooth initial data. But as the mesh is getting finer, the solution is getting better and better. To measure this phenomenon now let's look at the error plots and the convergence rate.

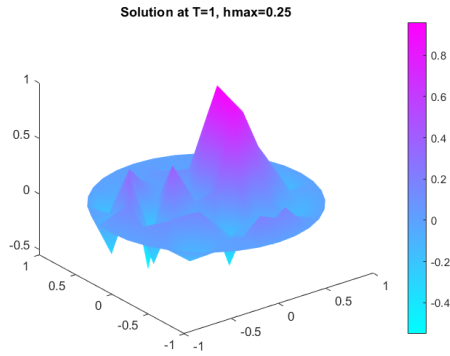
The errors for the smallest and largest mesh is plotted in Figure 7. The shape of the error is similar to the previous case, but the scale is much larger as it



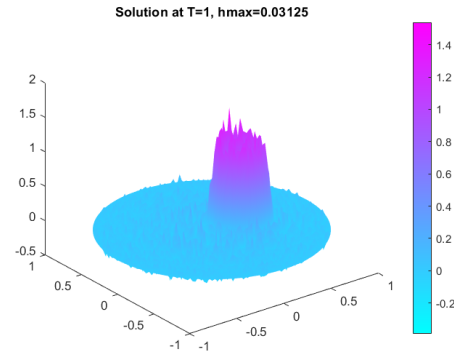
(a) Solution for $hmax = 1/8$



(b) Solution for $hmax = 1/16$



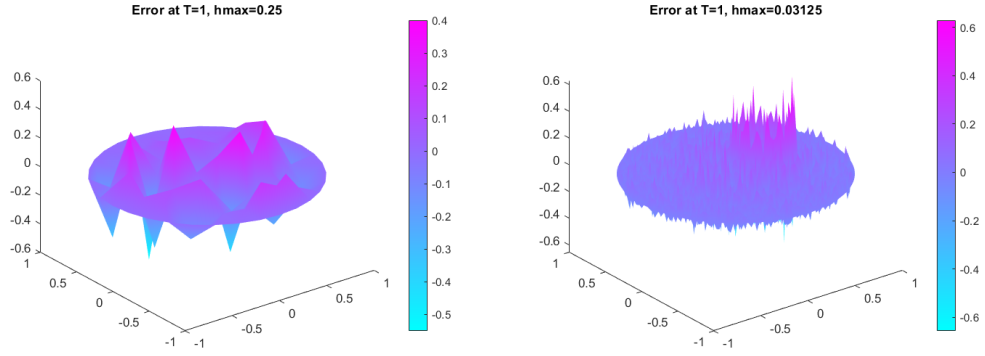
(c) Solution for $hmax = 1/4$



(d) Solution for $hmax = 1/32$

Figure 6: FEM solutions of the PDE, non-smooth initial data

can be expected. Now let's look at the convergence rate plotted in Figure 8. The calculated convergence rate is $\alpha = 0.28$, which is much less, than in the previous case. So as it was expected from the numerical solutions we also get that the convergence rate is much worse in this case as in the smooth version of the problem.



(a) Solution for $hmax = 1/4$

(b) Error for $hmax = 1/32$

Figure 7: FEM, error of the PDE at $T = 1$, non-smooth initial data

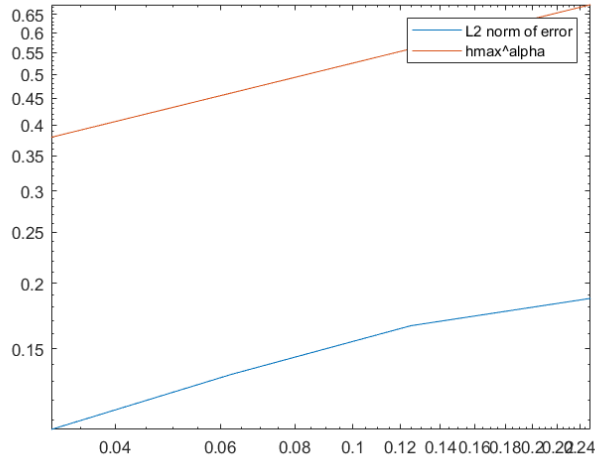


Figure 8: FEM $\alpha = 0.28$ convergence rate for the non-smooth initial data

Part II

In part 2, we will investigate methods for stabilizing the solution for the same problem as the one in part 1. To this end the SUPG and RV method for this problem will be introduced and implemented for both smooth and non smooth initial data.

Problem 2.1

First we have to create the SUPG formulation of the original problem:

$$\begin{aligned}\partial_t u + \nabla \cdot \mathbf{f}(u) &= 0, & (\mathbf{x}, t) &\in \Omega \times (0, T], \\ u(\mathbf{x}, 0) &= u_0(\mathbf{x}), & \mathbf{x} &\in \Omega,\end{aligned}$$

Reformulate the problem with the given information $\nabla \mathbf{f}(u) = \mathbf{f}'(u) \cdot \nabla u$.

$$\begin{aligned}\partial_t u + \mathbf{f}'(u) \cdot \nabla u &= 0, & (\mathbf{x}, t) &\in \Omega \times (0, T], \\ u(\mathbf{x}, 0) &= u_0(\mathbf{x}), & \mathbf{x} &\in \Omega,\end{aligned}$$

Now we can get the SUPG formulation of the problem by testing with $v + \delta \mathbf{f}'(v) \cdot \nabla v$. Find $u_h \in X_h$ s.t.:

$$(\partial_t u_h + \mathbf{f}'(u_h) \cdot \nabla u_h, v) + (\partial_t u_h + \mathbf{f}'(u_h) \cdot \nabla u_h, \delta \mathbf{f}'(v) \cdot \nabla v) = 0, \quad \forall v \in X_h \quad (2)$$

Note: Here $\mathbf{f}'(u_h)$ only depends on the position x , therefore it is a linear PDE, even though in this form $\mathbf{f}'(u_h)$ seems to be dependent on the solution. So using the implicit Cranck-Nicholson time-stepping, we get the following formulation:

$$\frac{1}{k}(U_n - U_{n-1}, v + \delta \mathbf{f}'(v) \cdot \nabla v) + \frac{1}{2}(\mathbf{f}'(U_n) \cdot \nabla U_n + \mathbf{f}'(U_{n-1}) \cdot \nabla U_{n-1}, v + \delta \mathbf{f}'(v) \cdot \nabla v) = 0$$

To prove stability we will use the form written in (2) and use $v = u$ to make stability estimates.

$$\frac{1}{2} \partial_t \|u_h\|^2 + \|\sqrt{\delta} \mathbf{f}'(u_h) \nabla u_h\|^2 + (\mathbf{f}'(u_h) \nabla u_h, u_h) + (\partial_t u_h, \delta \mathbf{f}'(u_h) \nabla u_h) = 0$$

With the notations of the lectures we get that since $\bar{\beta} = \mathbf{f}'$ is divergence free in this case as $\frac{x_2}{\partial x_1} = \frac{x_1}{\partial x_2} = 0$. So $\nabla \cdot \mathbf{f}' = 0$. Therefore the same prove works

as in the lecture to show that $(\mathbf{f}' \cdot \nabla u_h, u_h) = 0$ as it was given in the lecture for $(\bar{\beta} \cdot \nabla u_h, u_h)$ with the homogeneous Dirichlet boundary condition.

For the other term we can show the same thing using the fully discrete form. So in the fully discrete form we get $(\frac{U_n - U_{n-1}}{k_n}, \delta \mathbf{f}'(\mathbf{x}) \nabla \frac{U_n + U_{n-1}}{2}) = 0$.

Therefore in the fully discrete form we showed stability by gaining $\frac{1}{2} \partial_t \|U_h\|^2 + \|\sqrt{\delta} \mathbf{f}'(x) \delta U_h\| = 0$ in the fully discrete equivalent.

Therefore we showed stability.

Problem 2.2 - 2.3

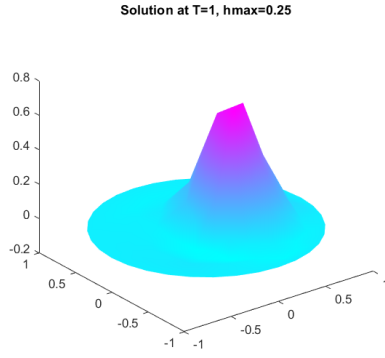
The SUPG (Streamline-Upwind-Petrov-Galerkin) method

First let's discuss and implement the SUPG method. From the the previous discussion we obtain the following discretized system to be implemented:

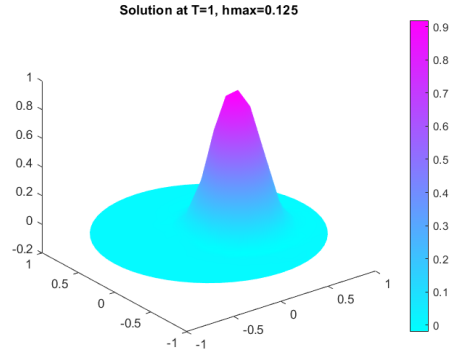
$$\xi_{n+1} \left(\frac{M}{k_n} + \frac{C}{2} + \frac{SM}{k_n} + \frac{SC}{2} \right) = \xi_n \left(\frac{M}{k_n} - \frac{C}{2} + \frac{SM}{k_n} - \frac{SC}{2} \right)$$

Here M and C are the mass and convection matrices from part 1. $SM_{ij} = (\varphi_j, \delta \mathbf{f}'(N_i) \nabla \varphi_i)$, and $SC_{ij} = (\mathbf{f}'(N_j) \nabla \varphi_j, \delta \mathbf{f}'(N_i) \nabla \varphi_i)$. Here $\mathbf{f}'(N_i)$ is used to show that the value of \mathbf{f}' is only space dependent in this case, and N_i denotes the node of the triangulation corresponding to the φ_i basis functions location. From here the implementation is straightforward using discrete integration to calculating the matrices.

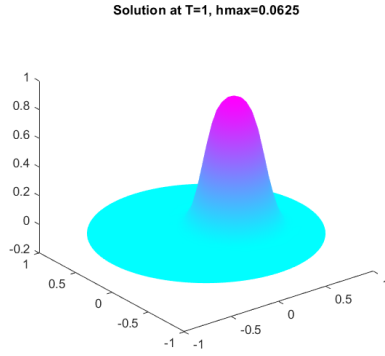
The plots for the smooth initial data can be found in Figures 9, 10 and 11. While the plots for the non-smooth initial data can be found in Figures 12,13, 14. As from the plots and error it can be seen the constant 1 initial data parts stays 0 and there is basically no noise to it only close to the "interesting" part of the function. Also the non-smooth data act much better in this case than with the original FEM. This is not surprising as this method is stable. Also we get higher convergence rates with $\alpha = 1.82$ and $\alpha = 0.29$ for the two cases. This means that the SUPG method is also superior to basic FEM in the rate of convergence.



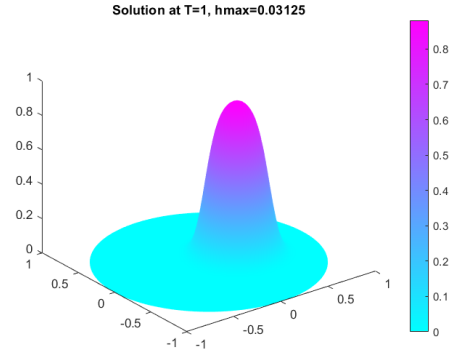
(a) Solution for $hmax = 1/4$



(b) Solution for $hmax = 1/8$

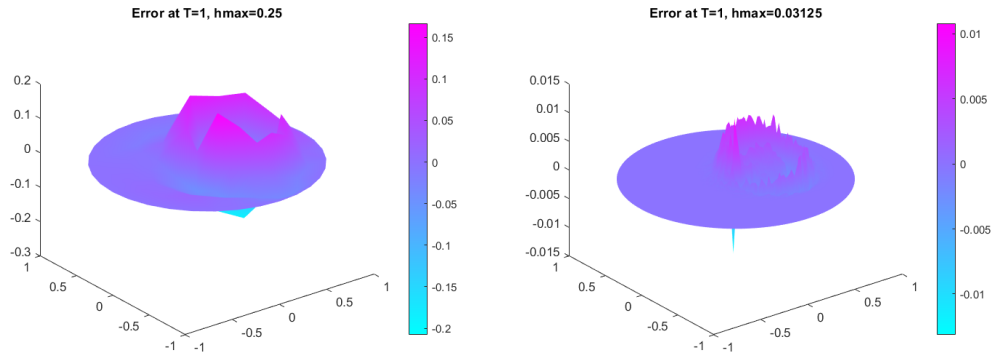


(c) Solution for $hmax = 1/16$



(d) Solution for $hmax = 1/32$

Figure 9: SUPG solutions of the PDE, smooth initial data



(a) Error for $hmax = 1/4$

(b) Error for $hmax = 1/32$

Figure 10: SUPG error of the PDE, smooth initial data

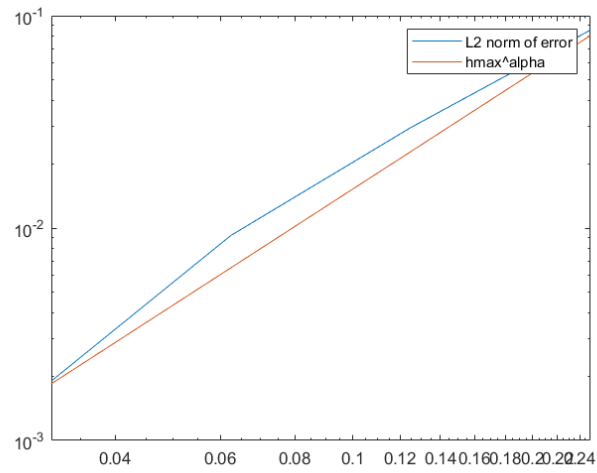
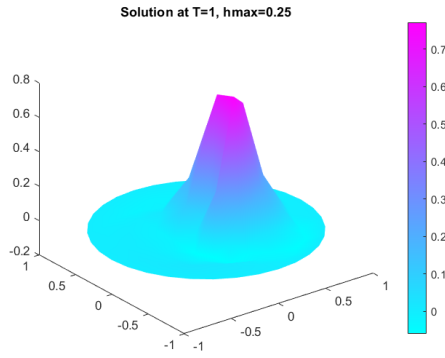
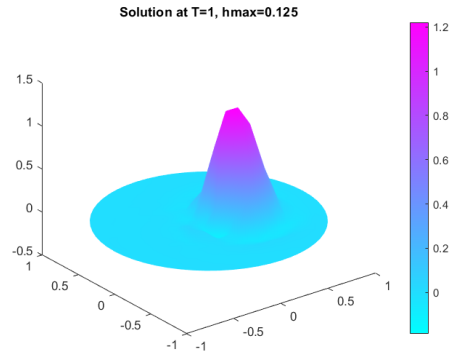


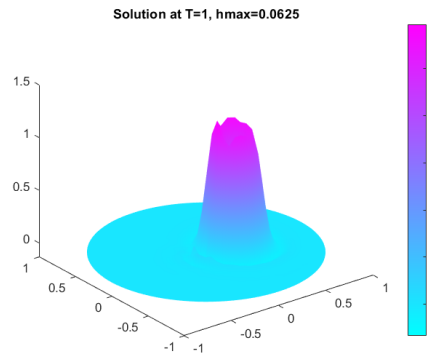
Figure 11: $\alpha = 1.82$ convergence rate with SUPG method, smooth initial data



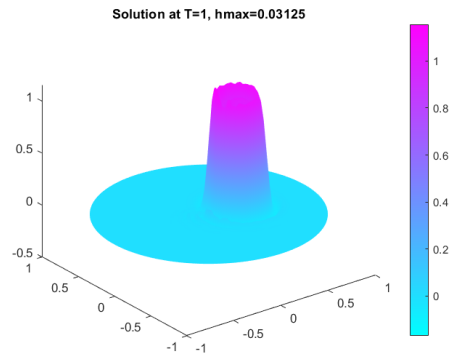
(a) Solution for $hmax = 1/4$



(b) Solution for $hmax = 1/8$

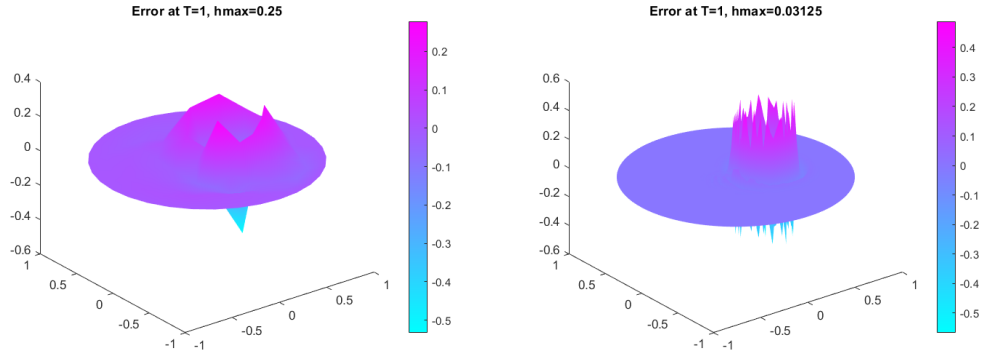


(c) Solution for $hmax = 1/16$



(d) Solution for $hmax = 1/32$

Figure 12: SUPG solutions of the PDE, non-smooth initial data



(a) Error for $hmax = 1/4$

(b) Error for $hmax = 1/32$

Figure 13: SUPG error of the PDE, non-smooth initial data

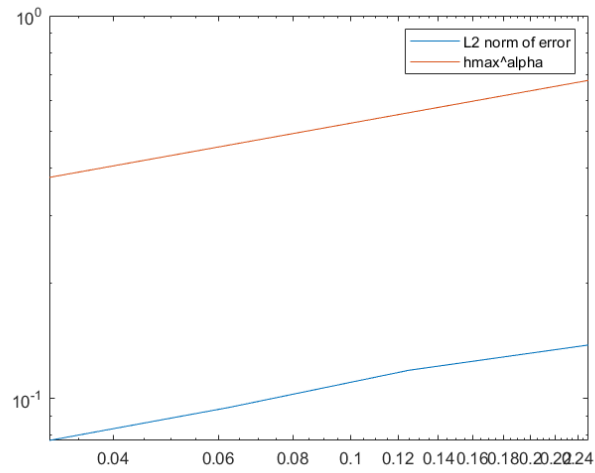


Figure 14: $\alpha = 0.29$ convergence rate with SUPG method, non-smooth initial data

The RV (Residual based artificial viscosity) method

The RV method was given in the task description. In this case we add an additional term to the equation.

$$\begin{aligned} \frac{1}{k_n} (U_n - U_{n-1}, v) + \frac{1}{2} (\nabla \cdot (\mathbf{f}(U_{n-1}) + \mathbf{f}(U_n)), v) \\ + \frac{1}{2} (\varepsilon_n(U_{n-1}) \nabla (U_n + U_{n-1}), \nabla v) = 0, \quad \forall v \in \mathcal{X}_h, \end{aligned}$$

Here the artificial viscosity $\varepsilon_n := \varepsilon_h(t_n)$ is computed as

$$\varepsilon_{n,K} = \min \left(C_{\text{vel}} h_K \beta_K, C_{\text{RV}} h_K^2 \frac{\|R_{\text{RV}}\|_{\infty,K}}{\|U_n - \bar{U}_n\|_{\infty,\Omega}} \right),$$

for each cell $K \subset \mathcal{T}_h$. Here $R(U_n) = \frac{1}{k_n} (U_n - U_{n-1}) + \nabla \cdot \mathbf{f}(U_n)$, and $\|U_n - \bar{U}_n\|_{\infty,\Omega}$ is a normalization term, with \bar{U}_n being the space average of the solution over Ω , h_K is the mesh-size of the element K , β_K denotes the local element wave speed that is computed as

$$\beta_K \equiv \|\mathbf{f}'(U_n)\|_{L^\infty(K)} = \max_{N_i \in K, i=0,1,2} \left(\left[(f'_1(U_n))^2 + (f'_2(U_n))^2 \right]^{\frac{1}{2}} (N_i) \right)$$

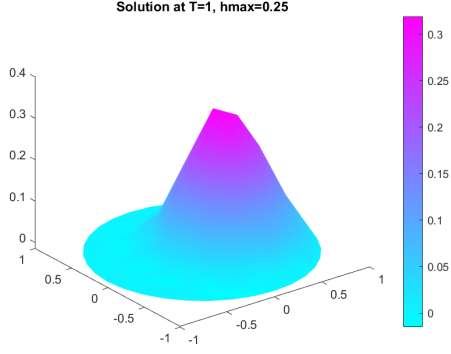
, where $N_i, i = 0, 1, 2$ is the nodes of the element K , $C_{\text{vel}} = 0.25$ and $C_{\text{RV}} = 1$ are stabilization parameters. The standard Galerkin solution can be obtained by setting $C_{\text{vel}} = 0$.

From these given information it is straightforward to do the implementation. All we need is one function to calculate $\varepsilon_{n,K}$ and one to calculate the matrix for the new term.

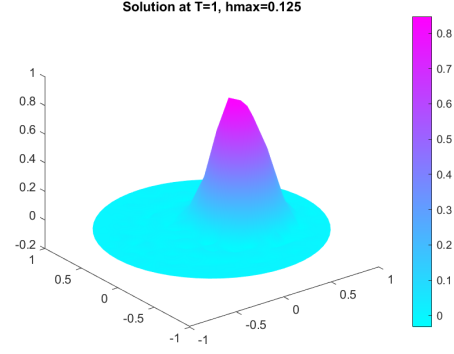
Note: From (R_u, v) the value of R_u can be calculated as $R(U_n) = M^{-1}(M(\frac{U_n - U_{n-1}}{k_n}) + CU_n)$ based on the given form.

The plots for the smooth initial data can be found in Figures 15, 16 and 18. While the plots for the non-smooth initial data can be found in Figures 19, 20, 22. In this case for both the smooth and non-smooth data the plots for small mesh sizes are worse than for FEM and SUPG, but as the mesh get finer, the results start to act nicely. Also the convergence rates are here

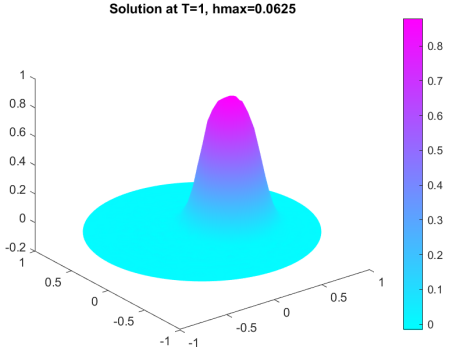
the bests with $\alpha = 1.97$ and $\alpha = 0.53$. The $\epsilon(u)$ values at the final time is also plotted in Figure 17 and 21. Though this plots are not accurate as ϵ is $\tau_h = \{K\} \mapsto \mathcal{R}$ and the plots are done at the nodal points with calculating an average from the neighbouring elements ϵ value.



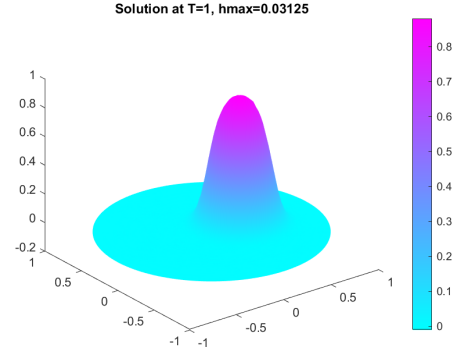
(a) Solution for $hmax = 1/4$



(b) Solution for $hmax = 1/8$

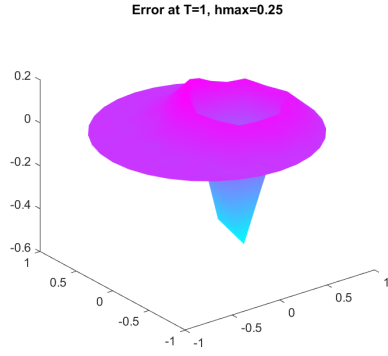


(c) Solution for $hmax = 1/16$

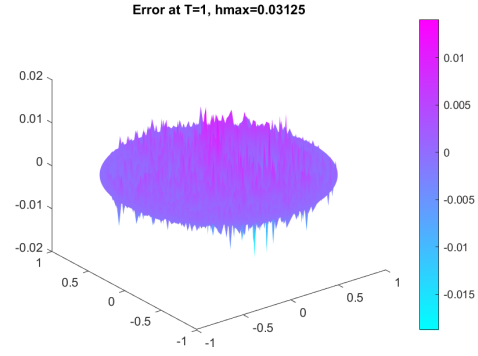


(d) Solution for $hmax = 1/32$

Figure 15: RV solutions of the PDE, smooth initial data

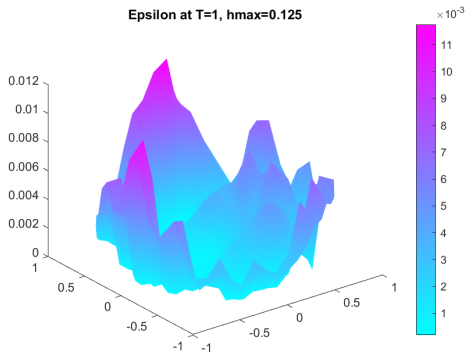


(a) Error for $hmax = 1/4$

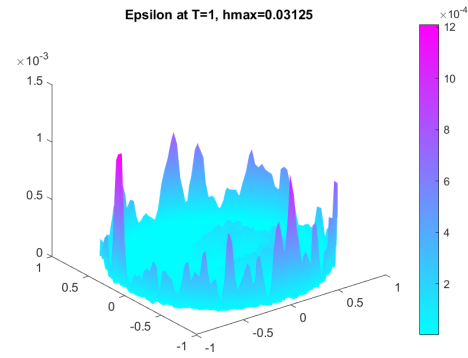


(b) Error for $hmax = 1/32$

Figure 16: RV, error of the PDE, smooth initial data



(a) $\epsilon(U_n), T = 1$ for $hmax = 1/4$



(b) $\epsilon(U_n), T = 1$ for $hmax = 1/32$

Figure 17: Value of $\epsilon(U)$ at the end of the RV simulation, smooth initial data

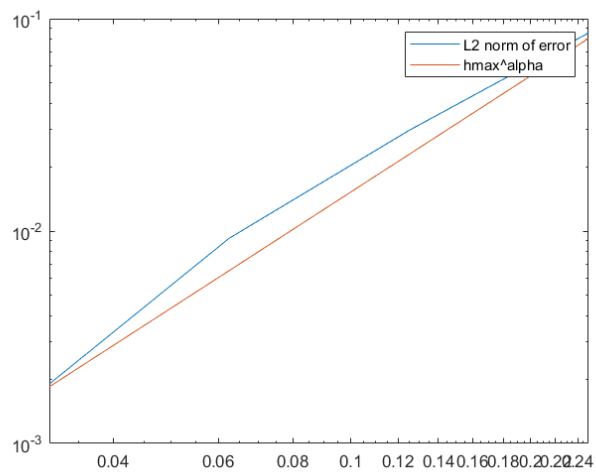
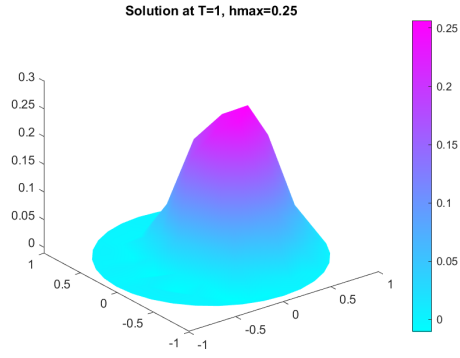
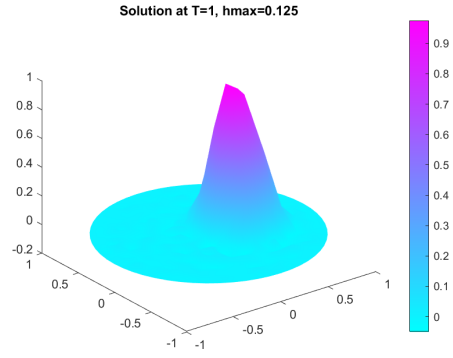


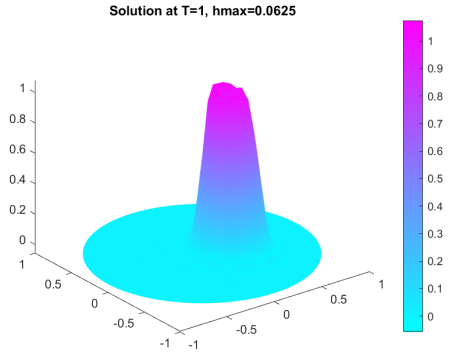
Figure 18: $\alpha = 1.97$ convergence rate with RV method, smooth initial data



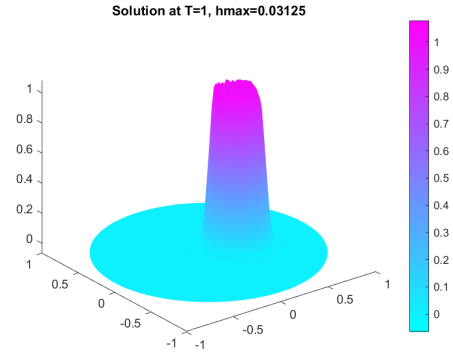
(a) Solution for $hmax = 1/4$



(b) Solution for $hmax = 1/8$

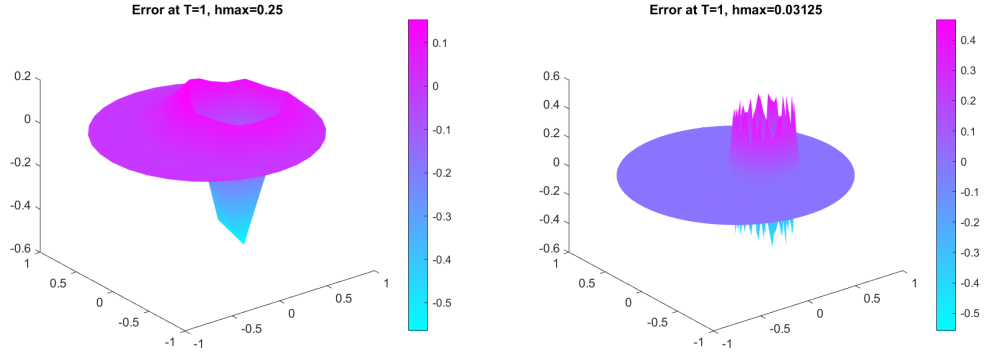


(c) Solution for $hmax = 1/16$



(d) Solution for $hmax = 1/32$

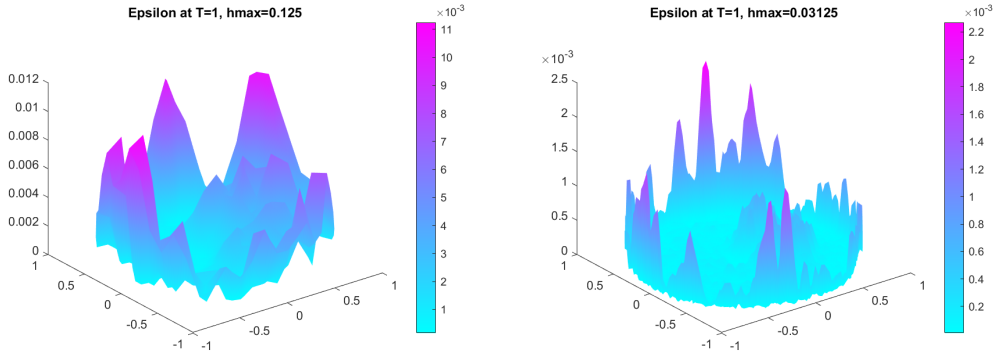
Figure 19: RV solutions of the PDE, non-smooth initial data



(a) Error for $hmax = 1/4$

(b) Error for $hmax = 1/32$

Figure 20: RV, error of the PDE, non-smooth initial data



(a) $\epsilon(U_n), T = 1$ for $hmax = 1/4$

(b) $\epsilon(U_n), T = 1$ for $hmax = 1/32$

Figure 21: Value of $\epsilon(U)$ at the end of the RV simulation, non-smooth initial data

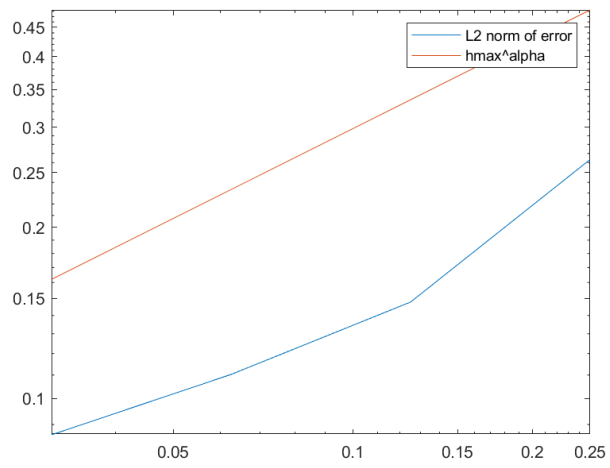


Figure 22: $\alpha = 0.53$ convergence rate with RV method, non-smooth initial data

Summary

With this we concluded the analysis and the numerical modelling of the given tasks. We investigated and implemented the basic FEM, the SUPG and RV method to solve the problem. What we obtained is that the FEM works better with smooth initial data, than with the noncontinuous, resulting better measured convergence rate and much less error. These solutions are numerically not stable, therefore we wanted to use methods, where stability can be proven. We implemented the SUPG and RV methods, which are stable and results in better convergence rate and better solutions at least for the non-smooth data the results are much better with these methods, while for the smooth data the FEM worked just fine.

Appendix

Matlab code for parts 1.1 – 1.3: *fem.m*

```
1 %% Given parameters
2 T=1;
3 CFL = 0.5;
4 r0 = 0.25;
5 x1_0 = 0.3;
6 x2_0 = 0;
7
8 %Used geometry
9 g = @circleg ;
10
11
12
13 %% Problem 1 – numerical fem solutions and error +
    convergence rate calculation
14 %hmax values to run the simulation for
15 hmax_list = [1/4, 1/8, 1/16, 1/32];
16 initial_data = [1, 2];
17
18 %Iteration on possible hmax values
19
20 for init = initial_data
21     i=1;
22     for hmax = hmax_list
23         %mesh data
24         [p ,e , t ] = initmesh (g , 'hmax' , hmax); %
            function call to create mesh
25         x1=p(1,:)'; % vector of x1 coordinates
26         x2=p(2,:)'; % vector of x2 coordinates
27
28         % Set up the matrices
29         df=ddf(x1,x2)';
30         M = MassAssembler2D(p,t);
31         C = ConvectionAssembler2D(p,t,df(1,:),df(2,:));
32
```

```

33 %Initial setup
34
35 if init==1
36 U_0 = 1 / 2 * (1 - tanh((((x1 - x1_0).^2 + (x2 -
      x2_0).^2) / (r0^2)) - 1));
37 elseif init==2
38 U_00 = ((x1 - x1_0).^2 + (x2 - x2_0).^2) < r0^2;
39 U_0=double(U_00);
40 end
41
42 U=U_0;
43
44 %Time integration Crank–Nicholson
45 time = 0;
46 inf_norm = max(abs(df(1,:))+abs(df(2,:)));
47 kn = CFL * hmax/inf_norm*1.0; % timestep – not
      depends on U_n due to f'
48
49 while time<T
50     %Set up equation matrix / vector
51     A=((M/kn) + (C/2));
52     b=((M/kn) * U - (C/2) * U);
53
54     %Boundary conditions
55     I = eye( length ( p ));
56     A(e(1,:) ,:) =I(e(1,:) ,:);
57     b(e(1,:) ) = 0;
58
59     %Solve equation
60     U = A\b;
61
62     time = time + kn;
63 end
64
65
66 %Plot solution
67 figure()
68 pdeplot(p,e,t,"XYData",U, 'Zdata',U);

```

```

69     title(" Solution  at T="+T+", hmax="+hmax)
70
71     %Error
72     error=U-U_0;
73     L2E(i)=sqrt( error '*M*error );
74     figure()
75     pdeplot (p ,[], t , 'XYData' ,error , 'ZData' ,error , '
        ColorBar ' , 'on')
76     title(" Error  at T="+T+", hmax="+hmax)
77
78     i=i+1;
79 end
80
81 % Plot Conv
82 p = polyfit(log(hmax_list),log(L2E),1);
83 alpha = p(1)
84 figure ()
85 %xlabel ('hmax')
86 %ylabel('L2E')
87 loglog( hmax_list , L2E , 'DisplayName' , 'L2 norm of
        error ')
88 hold on
89 loglog( hmax_list , hmax_list.^alpha , 'DisplayName' ,
        'hmax\^alpha ')
90 hold off
91 legend show
92
93 end
94
95
96
97
98
99
100 %% Assembler functions based on book (The Finite
    Element Method:Theory, Implementation, and
    Applications %%%%%%%%%%)
101

```

```

102 % derivative f
103 function df = ddf(x1, x2)
104     df = 2 * pi * [-x2, x1];
105 end
106
107 % Convection Matrix Assembler
108 function C = ConvectionAssembler2D(p, t, bx, by)
109     np=size(p,2);
110     nt=size(t,2);
111     C=sparse(np,np);
112     for i=1:nt
113         loc2glb=t(1:3,i);
114         x=p(1,loc2glb);
115         y=p(2,loc2glb);
116         [area,b,c]=HatGradients(x,y);
117         bxmid=mean(bx(loc2glb));
118         bymid=mean(by(loc2glb));
119         CK=ones(3,1)*(bxmid*b+bymid*c) '* area/3;
120         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
121     end
122 end
123
124 %Mass Matrix Assembler
125 function M = MassAssembler2D(p, t)
126     np = size(p,2); % number of nodes
127     nt = size(t,2); % number of elements
128     M = sparse(np,np); % allocate mass matrix
129     for K = 1:nt % loop over elements
130         loc2glb = t(1:3,K); % local-to-global map
131         x = p(1,loc2glb); % node x-coordinates
132         y = p(2,loc2glb); % y
133         area = polyarea(x,y); % triangle area
134         MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
            element mass matrix
135         M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
            add element masses to M
136     end
137 end

```

```

138
139 %Gradients
140 function [area,b,c] = HatGradients(x,y)
141 area=polyarea(x,y);
142 b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
143 c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
144 end

```

Matlab code for SUPG method: *SUPG.m*

```
1 %% Given parameters
2 T=1;
3 CFL = 0.5;
4 r0 = 0.25;
5 x1_0 = 0.3;
6 x2_0 = 0;
7
8
9 %Used geometry
10 g = @circleg ;
11
12
13
14 %% Problem 1 – numerical fem solutions and error +
    convergence rate calculation
15 %hmax values to run the simulation for
16 hmax_list = [1/4, 1/8, 1/16, 1/32];
17 initial_data = [1, 2];
18
19 %Iteration on possible hmax values
20
21 for init = initial_data
22     i=1;
23     for hmax = hmax_list
24         %mesh data
25         delta=0.05*hmax;
26         [p ,e , t ] = initmesh (g , 'hmax' , hmax); %
            function call to create mesh
27         x1=p(1,:)'; % vector of x1 coordinates
28         x2=p(2,:)'; % vector of x2 coordinates
29
30         % Set up the matrices
31         df=ddf(x1,x2)';
32         M = MassAssembler2D(p,t);
33         C = ConvectionAssembler2D(p,t,df(1,:),df(2,:));
34         SC = SCAssembler2D(p,t,df(1,:),df(2,:),delta);
```

```

35     SM = delta*C';
36
37     %Initial setup
38
39     if init==1
40         U_0 = 1 / 2 * (1 - tanh((((x1 - x1_0).^2 + (x2 -
           x2_0).^2) / (r0^2)) - 1));
41     elseif init==2
42         U_00 = ((x1 - x1_0).^2 + (x2 - x2_0).^2) < r0^2;
43         U_0=double(U_00);
44     end
45
46     U=U_0;
47
48     %Time integration Crank–Nicholson
49     time = 0;
50     inf_norm = max(abs(df(1,:))+abs(df(2,:)));
51     kn = CFL * hmax/inf_norm*1.0; % timestep – not
       depends on U_n due to f'
52
53     while time<T
54         %Set up equatddion matrix / vector
55         A=((M/kn) + (C/2) + (SM/kn) + (SC/2));
56         b=((M/kn) - (C/2) + (SM/kn) - (SC/2)) * U;
57
58         %Boundary conditions
59         I = eye( length ( p ));
60         A(e(1,:) ,:) =I(e(1,:) ,:);
61         b(e(1,:) ) = 0;
62
63         %Solve equation
64         U = A\b;
65
66         time = time + kn;
67     end
68
69
70     %Plot solution

```

```

71     figure()
72     pdeplot(p,e,t,"XYData",U,'Zdata',U);
73     title("Solution at T="+T+", hmax="+hmax)
74
75     %Error
76     error=U-U_0;
77     L2E(i)=sqrt(error'*M*error);
78     figure()
79     pdeplot(p,[],t,'XYData',error,'ZData',error,'
        ColorBar','on')
80     title("Error at T="+T+", hmax="+hmax)
81
82     i=i+1;
83 end
84
85 % Plot Conv
86 p = polyfit(log(hmax_list),log(L2E),1);
87 alpha = p(1)
88 figure()
89 xlabel('hmax')
90 ylabel('L2E')
91 loglog(hmax_list, L2E, 'DisplayName', 'L2 norm of
    error')
92 hold on
93 loglog(hmax_list, hmax_list.^alpha, 'DisplayName',
    'hmax\^alpha')
94 hold off
95 legend show
96
97 end
98
99
100
101
102
103
104 %% Assembler functions based on book (The Finite
    Element Method:Theory, Implementation, and

```



```

Applications %%%%%%%%%%
105
106 % derivative f
107 function df = ddf(x1, x2)
108     df = 2 * pi * [-x2, x1];
109 end
110
111 % Convection Matrix Assembler
112 function C = ConvectionAssembler2D(p, t, bx, by)
113     np=size(p,2);
114     nt=size(t,2);
115     C=sparse(np,np);
116     for i=1:nt
117         loc2glb=t(1:3,i);
118         x=p(1,loc2glb);
119         y=p(2,loc2glb);
120         [area,b,c]=HatGradients(x,y);
121         bxmid=mean(bx(loc2glb));
122         bymid=mean(by(loc2glb));
123         CK=ones(3,1)*(bxmid*b+bymid*c) '* area/3;
124         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
125     end
126 end
127
128 %Mass Matrix Assembler
129 function M = MassAssembler2D(p, t)
130     np = size(p,2); % number of nodes
131     nt = size(t,2); % number of elements
132     M = sparse(np,np); % allocate mass matrix
133     for K = 1:nt % loop over elements
134         loc2glb = t(1:3,K); % local-to-global map
135         x = p(1,loc2glb); % node x-coordinates
136         y = p(2,loc2glb); % y
137         area = polyarea(x,y); % triangle area
138         MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
            element mass matrix
139         M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
            add element masses to M

```

```

140     end
141 end
142
143 % SC matrix assembly
144 function A = SCAssembler2D(p,t,bx, by, delta)
145 np = size(p,2);
146 nt = size(t,2);
147 A = sparse(np,np); % allocate stiffness matrix
148 for K = 1:nt
149     loc2glb = t(1:3,K); % local-to-global map
150     x = p(1,loc2glb); % node x-coordinates
151     y = p(2,loc2glb); % node y-
152     [area,b,c] = HatGradients(x,y);
153     bxmid=mean(bx(loc2glb));
154     bymid=mean(by(loc2glb));
155     AK = delta*(bxmid^2*b*b'+bymid^2*c*c'+bxmid*bymid*(b*c
        '+c*b'))*area; % element stiffness matrix
156     A(loc2glb,loc2glb) = A(loc2glb,loc2glb)+ AK; % add
        element stiffnesses to A
157 end
158 end
159
160 %Gradients
161 function [area,b,c] = HatGradients(x,y)
162 area=polyarea(x,y);
163 b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
164 c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
165 end

```

Matlab code for RV method: *RV.m*

```
1 %% Given parameters
2 T=1;
3 CFL = 0.5;
4 r0 = 0.25;
5 x1_0 = 0.3;
6 x2_0 = 0;
7 C_vel=0.25;
8 C_RV=1;
9
10
11 %Used geometry
12 g = @circleg ;
13
14
15
16 %% Problem 1 – numerical fem solutions and error +
    convergence rate calculation
17 %hmax values to run the simulation for
18 hmax_list = [1/4,1/8,1/16,1/32];
19 initial_data = [1,2];
20
21 %Iteration on possible hmax values
22
23 for init = initial_data
24     i=1;L2E=[];
25     for hmax = hmax_list
26         disp("hmax: "+hmax);
27         %mesh data
28         [p ,e , t ] = initmesh (g , 'hmax' , hmax); %
            function call to create mesh
29
30         x1=p(1,:)'; % vector of x1 coordinates
31         x2=p(2,:)'; % vector of x2 coordinates
32
33         % Set up the matrices
34         df=ddf(x1,x2)';
```

```

35 M = MassAssembler2D(p,t);
36 C = ConvectionAssembler2D(p,t,df(1,:),df(2,:));
37
38 nt = size(t,2); hk=[]; bk=[];
39 for K = 1:nt % for each element
40     loc2glb = t(1:3,K); % local-to-global map
41     x = p(1,loc2glb); % node x-coordinates
42     y = p(2,loc2glb); % node y-coordinates
43     hk(K)=min([norm([x(1)-x(2),y(1)-y(2)]),norm([x
44         (2)-x(3),y(2)-y(3)]),norm([x(3)-x(1),y(3)-y
45         (1)]), 2*norm([x(1)-sum(x)/3,y(1)-sum(y)/3]
46         ]]); % diameter
47     dx=df(1,loc2glb);
48     dy=df(2,loc2glb);
49     bk(K)=max([norm([dx(1),dy(1)]),norm([dx(2),dy
50         (2)]),norm([dx(3),dy(3)])]);
51 end
52
53 %Initial setup
54
55 if init==1
56     U_0 = 1 / 2 * (1 - tanh((((x1 - x1_0).^2 + (x2 -
57         x2_0).^2) / (r0^2)) - 1));
58 elseif init==2
59     U_00 = ((x1 - x1_0).^2 + (x2 - x2_0).^2) < r0^2;
60     U_0=double(U_00);
61 end
62
63 U=U_0;
64 U_old=U_0;
65
66 %Time integration Crank-Nicholson
67 time = 0;
68 inf_norm = max(abs(df(1,:))+abs(df(2,:)));
69 kn = CFL * hmax/inf_norm*1.0; % timestep - not
70     depends on U_n due to f'
71
72 while time<T

```

```

67         %disp(time);
68         R = ResidualAssembler(p,t, bk, hk, U, U_old, M,
        C, kn, C_vel, CRV);
69         %Set up equatddion matrix / vector
70         A=((M/kn) + (C/2) + (R/2));
71         b=((M/kn) - (C/2) - (R/2)) * U;
72
73         %Boundary conditions
74         I = eye( length ( p ));
75         A(e(1,:),:),:)=I(e(1,:),:),:);
76         b(e(1,:),) = 0;
77
78         %Solve equation
79         U_old = U;
80         U = A\b;
81
82         time = time + kn;
83     end
84
85
86     %Plot solution
87     plt=figure();
88     pdeplot(p,e,t,"XYData",U, 'Zdata',U);
89     title("Solution at T="+T+", hmax="+hmax)
90     saveas(plt,"RV_plot_"+init+"_"+hmax+".png")
91
92     %Error
93     error=U-U_0;
94     L2E(i)=sqrt(error'*M*error);
95     err=figure();
96     pdeplot(p,[],t,'XYData',error,'ZData',error,'
        ColorBar','on')
97     title("Error at T="+T+", hmax="+hmax)
98     saveas(err,"RV_err_"+init+"_"+hmax+".png")
99
100
101     %Plot epsilon U_n
102     np = size(p,2);

```

```

103     nt = size(t,2);
104     eps = zeros(np); % allocate matrix
105     occurances = zeros(np);
106     RU=M\ (1/kn*M*(U-U_old)+C*U);
107     for K = 1:nt % for each element
108         loc2glb = t(1:3,K); % local-to-global map
109         x = p(1,loc2glb); % node x-coordinates
110         y = p(2,loc2glb); % node y-coordinates
111         [area,b,c] = HatGradients(x,y);
112         h_k=hk(K);
113         b_k=bk(K);
114         U_norm=max(abs(U-mean(U)));
115         Rk=max(abs(RU(loc2glb)));
116         ep=min([C_vel*h_k*b_k,C_RV*h_k^2*Rk/U_norm]);
117         eps(loc2glb) = eps(loc2glb)+ 1/3*ep*[1;1;1]; % add
            element stiffnesses to A
118         occurances(loc2glb(1)) = occurances(loc2glb(1))+1;
119         occurances(loc2glb(2)) = occurances(loc2glb(2))+1;
120         occurances(loc2glb(3)) = occurances(loc2glb(3))+1;
121     end
122     eps=eps./ occurances;
123     pdeplot (p,[],t,'XYData',eps,'ZData',eps,'ColorBar
        ', 'on')
124     title("Epsilon at T="+T+", hmax="+hmax)
125     saveas(err,"RV_eps_"+init+"_"+hmax+".png")
126
127     i=i+1;
128 end
129
130 % Plot Conv
131 p = polyfit(log(hmax_list),log(L2E),1);
132 alpha = p(1)
133 alp=figure();
134 xlabel('hmax')
135 ylabel('L2E')
136 loglog(hmax_list, L2E, 'DisplayName', 'L2 norm of
        error')
137 hold on

```

```

138 loglog( hmax_list , hmax_list.^alpha , 'DisplayName' ,
        'hmax\^alpha')
139 hold off
140 legend show
141     saveas( alp , "RV_conv_"+init+".png")
142
143 end
144
145
146
147
148
149
150 %% Assembler functions based on book (The Finite
    Element Method:Theory, Implementation, and
    Applications %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
151
152
153 % Convection Matrix Assembler
154 function C = ConvectionAssembler2D(p,t,bx,by)
155     np=size(p,2);
156     nt=size(t,2);
157     C=sparse(np,np);
158     for i=1:nt
159         loc2glb=t(1:3,i);
160         x=p(1,loc2glb);
161         y=p(2,loc2glb);
162         [area,b,c]=HatGradients(x,y);
163         bxmid=mean(bx(loc2glb));
164         bymid=mean(by(loc2glb));
165         CK=ones(3,1)*(bxmid*b+bymid*c) '* area/3;
166         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
167     end
168 end
169
170 %Mass Matrix Assembler
171 function M = MassAssembler2D(p,t)
172     np = size(p,2); % number of nodes

```

```

173     nt = size(t,2); % number of elements
174     M = sparse(np,np); % allocate mass matrix
175     for K = 1:nt % loop over elements
176         loc2glb = t(1:3,K); % local-to-global map
177         x = p(1,loc2glb); % node x-coordinates
178         y = p(2,loc2glb); % y
179         area = polyarea(x,y); % triangle area
180         MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
            element mass matrix
181         M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
            add element masses to M
182     end
183 end
184
185 % SC matrix assembly
186 function R = ResidualAssembler(p,t, bk,hk, U, U_old, M,
    C, kn, C_vel,CRV)
187 np = size(p,2);
188 nt = size(t,2);
189 R = sparse(np,np); % allocate matrix
190 RU=M\ (1/kn*M*(U-U_old)+C*U);
191 for K = 1:nt % for each element
192     loc2glb = t(1:3,K); % local-to-global map
193     x = p(1,loc2glb); % node x-coordinates
194     y = p(2,loc2glb); % node y-coordinates
195     [area,b,c] = HatGradients(x,y);
196
197     h_k=hk(K);
198     b_k=bk(K);
199     U_norm=max(abs(U-mean(U)));
200     Rk=max(abs(RU(loc2glb)));
201
202     e=min([C_vel*h_k*b_k,CRV*h_k^2*Rk/U_norm]);
203     AK = e*(b*b'+c*c')*area; % element stiffness matrix
204     R(loc2glb,loc2glb) = R(loc2glb,loc2glb)+ AK; % add
        element stiffnesses to A
205 end
206 end

```



```

207
208 %Gradients
209 function [area,b,c] = HatGradients(x,y)
210 area=polyarea(x,y);
211 b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
212 c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
213 end
214
215 % derivative f
216 function df = ddf(x1, x2)
217     df = 2 * pi * [-x2, x1];
218 end

```