Uppsala University

Advanced Numerical Methods

1TD050

Assignment 2
Nonlinear conservation laws and goal-oriented adaptivity

Author:
Csongor Horváth

October 26, 2023

Part A - Nonlinear conservation law

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:

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

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.

In this case we will use a non-linear $\boldsymbol{f}$ function for the problem and investigate different methods solving the problem.

In this part, we are interested in solving the so-called KPP rotating wave problem, where the flux and initial data in (1) are defined as

$$\boldsymbol{f}(u) = (\sin u, \cos u), \quad u_0(\boldsymbol{x}) = \begin{cases} \frac{14\pi}{4}, & \text{if } \sqrt{x^2+y^2} \leq 1, \\ \frac{\pi}{4}, & \text{otherwise.} \end{cases}$$

The computation for this problem is the square $[-2, 2] \times [-2.5, 1.5]$ and the problem is usually solved until $T = 1$. The solution of the KPP problem contains a strongly rotating shock wave. This phenomenon is difficult to capture numerically.

Problem A.1

In this part we will use the Picard iteration to deal with the non-linear term. After the GFEM method we will use the SUPG and RV method for stabilization. The implicit Crank-Nicholson method will be used for time discretization.

GFEM Formulation

First let's formulate the GFEM problem. After space and time discretization we get the following problem same as in the first assignment:

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

Now as $\boldsymbol{f'}$ is non-linear, we use the Picard iteration for U_n over time and space. we get the following Picard iteration. Given $U_0^n = U_{n-1}$, until convergence reached find U_n^{k+1} from

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

From here the linear systems are easy to create. Let $M = \{M\}_{i,j} = (\varphi_j, \varphi_i)$ be the mass matrix and $C(\xi) = (\boldsymbol{f'}(\xi_j)\nabla\varphi_j, \varphi_i)$ is the convection matrix, which depends on an approximation of the solution ξ. Here ξ_j denotes the corresponding value of ξ to the j element of the triangulation. So we get the following iteration to implement:

For $n = 0, 1, \ldots$ until $T = 1$ reached solve the following Picardi iteration: $\xi_{n+1}^0 = \xi_n$, for $i = 0, 1, \ldots$ until convergence reached solve

$$\left(\frac{M}{k} + \frac{C(\xi_{n+1}^i)}{2}\right)\xi_{n+1}^{i+1} = \left(\frac{M}{k} - \frac{C(\xi_n)}{2}\right)\xi_n$$

If convergence reached after i iteration make $\xi_{n+1} = \xi_{n+1}^i$.

SUPG method - Formulation

For the SUPG method we also have a similar starting equation as in the previous assignment, but as now we want to do Picard iteration we shall

use the following modified test function $v + \delta \boldsymbol{f'}(U_{n-1})\nabla v$ instead of $v + \delta \boldsymbol{f'}(U_{n-1})\nabla v$, as the non-linear part have to stay U dependent for the Picard iteration to make sense and the U_{n-1} is the easier choice. So we have the following non-linear equation.

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

From here we get the following Picard iteration for all $n \Longrightarrow n+1$ time step with the matrices M_{ij} mass matrix,
Let be $\xi_{n+1}^0 = \xi_n$. For $i = 0, 1 \ldots$ until convergence:

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

When convergence reached $\xi : n+1 = \xi_{n+1}^i$
Here M and C are the mass and convection matrices similarly to the GFEM part. $SM_{ij}(\xi) = (\varphi_j, \delta \boldsymbol{f'}(\xi_i)\nabla\varphi_i)$, and $SC_{ij}(\xi_1, \xi_2) = (\boldsymbol{f'}(\xi_{1,j})\nabla\varphi_j, \delta \boldsymbol{f'}(\xi_{2,i})\nabla\varphi_i)$.

RV method - Formulation

Similarly to the previous methods the RV methods formulation with Picard iteration for the non-linear term can be easily constructed starting from the formulation used for the linear function in the previous assignment.

The previous formulation was:

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

From this we get the following Picard iteration for each time step $n \mapsto n+1$:
Let be $\xi_{n+1}^0 = \xi_n$. For $i = 0, 1, \ldots$ until convergence reached:

$$\left(\frac{M}{k_n} + \frac{C(\xi_{n+1}^i)}{2} + \frac{R(\xi_n), \xi_{n-1}}{2}\right)\xi_{n+1}^{i+1} = \left(\frac{M}{k_n} - \frac{C(\xi_n)}{2} - \frac{R(\xi_n, \xi_{n-1})}{2}\right)\xi_n$$

Here M, $C(\xi)$ are matrices as before. And the R matrix stores the corresponding values of $\epsilon_j(\xi_n, \xi_{n-1})\nabla\varphi_j \cdot \nabla\varphi_i$, where the $\epsilon(\xi_n, \xi_{n-1})$ is calculated the same way as in the linear case using different f'.

Simulation results and figures

For simulations we had to use a small time step as for larger steps the Picard iteration doesn't converge even in the beginning. The chosen time step is proportional to the value of $hmax$. I choose $k_n = CFL \cdot hmax$ with $CFL = 0.05$ in most case. And I choose $TOL = 10^{-3}$ for the Picard iterations limit. Then run the simulations. The resulting figures can be found in Figure 2, with the simple GFEM being in (2a, 2b), the SUPG being in (2c, 2d) and the RV method in (2e, 2f). It can be seen that the GFEM produce a lot of noise at the original boundary, but even the SUPG has relatively noisy solutions. And knowing how the solution looks like both the GFEM and the SUPG which is stable produce kind of rubbish solutions. This shows us that even the stable methods not guarantee to produce the expected real solution, but only a numerical solution, which can be different. On the other hand the RV method produce quite good results with faster run times. So the main thing in this part is that even when we can prove stability we can't be sure to produce good results with our methods. Additionally we learned that the method with Picard iteration has poor performance in respect to run time, as in each time step there should be an inner iteration where in each step we should establish matrices and solve linear system of equations. Additionally from this assignment experience for the convergence in Picard method we should apply tiny time steps and even if we could apply larger steps, we could expect in those case that the Picard iteration converge more slowly.

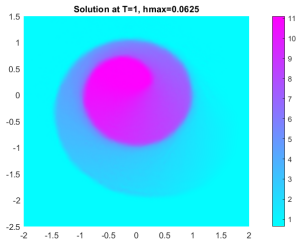
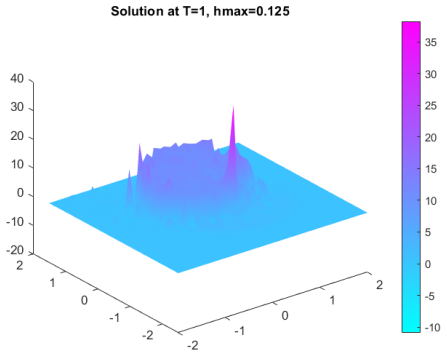
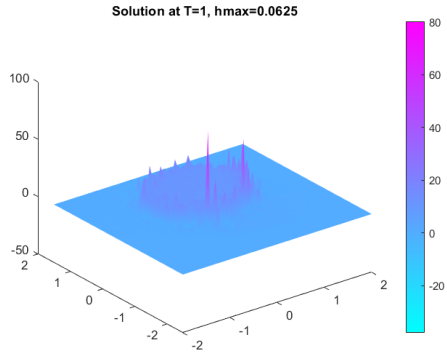


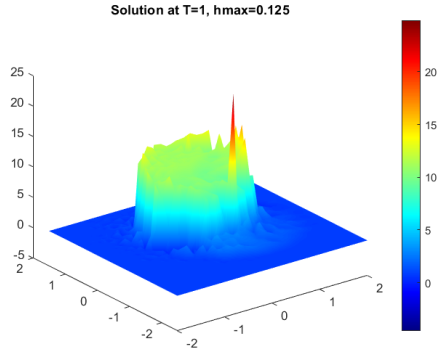
Figure 1: The solution from above with RV $hmax = 1/16$



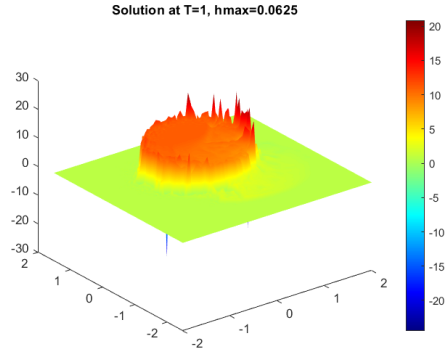
(a) GFEM solution for $hmax = 1/8$, $TOL = 10^{-3}$, $CFL = 0.05$



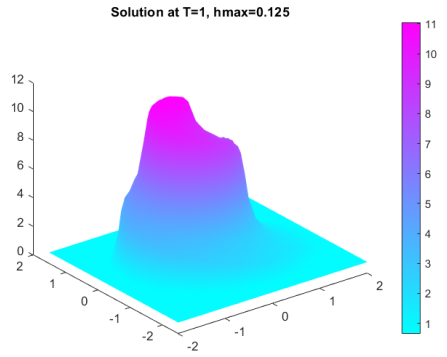
(b) GFEM solution for $hmax = 1/16$, $TOL = 10^{-3}$, $CFL = 0.05$



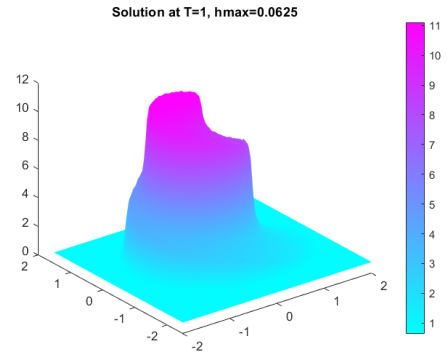
(c) SUPG solution for $hmax = 1/8$, $TOL = 10^{-3}$, $CFL = 0.05$



(d) SUPG solution for $hmax = 1/16$, $TOL = 10^{-3}$, $CFL = 0.05$



(e) RV solution for $hmax = 1/8$, $TOL = 10^{-3}$, $CFL = 0.05$



(f) RV solution for $hmax = 1/16$, $TOL = 10^{-3}$, $CFL = 0.1$

Figure 2: Solution of the KPP rotating wave problem at $T = 1$

Problem A.2

In this part we want to use the 4th order Runge-Kutta method for solving the problem. This is an explicit method. In general this method can be written as if there is a given proble

$$\frac{du(t)}{dt} = f(y(t), t), \qquad \text{with } u\left(t_0\right) = u_0$$

(assume we only have an approximation to $u\left(t_0\right)$, which we call $u^*\left(t_0\right)$).

$$\begin{aligned}
k_1 &= f\left(u^*\left(t_0\right), t_0\right) \\
k_2 &= f\left(u^*\left(t_0\right) + k_1 \frac{h}{2}, t_0 + \frac{h}{2}\right) \\
k_3 &= f\left(u^*\left(t_0\right) + k_2 \frac{h}{2}, t_0 + \frac{h}{2}\right) \\
k_4 &= f\left(u^*\left(t_0\right) + k_3 h, t_0 + h\right)
\end{aligned}$$

Then we get the value of $u^*\left(t_0 + h\right)$

$$u^*\left(t_0 + h\right) = u^*\left(t_0\right) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} h$$

And we can use this at each time step. So our task is to write the different methods formulation into the form $\partial_t u_h = g(u_h)$ as the PDE is not time dependent. Note that in the simplest form the equation was given by $\partial_t u = -f'(u) \cdot \nabla u$, but as ∇u can't be calculated directly, we need to apply the explicit RK4 method to the WF form of the equations. Therefore it makes sense to implement different version of RK4 to the different WF discussed above.

RK-GFEM - formulation

From the WF of GFEM we can write the following equation to $\partial_t \xi$

$$M(\partial_t \xi) = -C(\xi)\xi$$

Here M and C matrices as above. From this we can implement the 4th order RK method for this equation.

RK-SUPG - formulation

Similarly from the WF of SUPG we can write the following equation to $\partial_t \xi$

$$(M + SM(\xi))(\partial_t \xi) = -(C(\xi) + SC(\xi))\xi$$

Here M, C, SM and SC matrices as above. From this we can implement the 4th order RK method for this equation.

RK-RV - formulation

Similarly from the WF of RV we can write the following equation to $\partial_t \xi$

$$M(\partial_t \xi) = -(C(\xi) + R(\xi))\xi$$

Here M, C, and R matrices as above. From this we can implement the 4th order RK method for this equation.

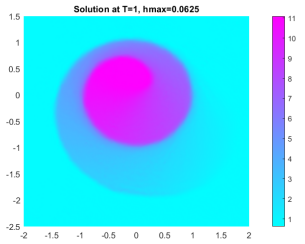
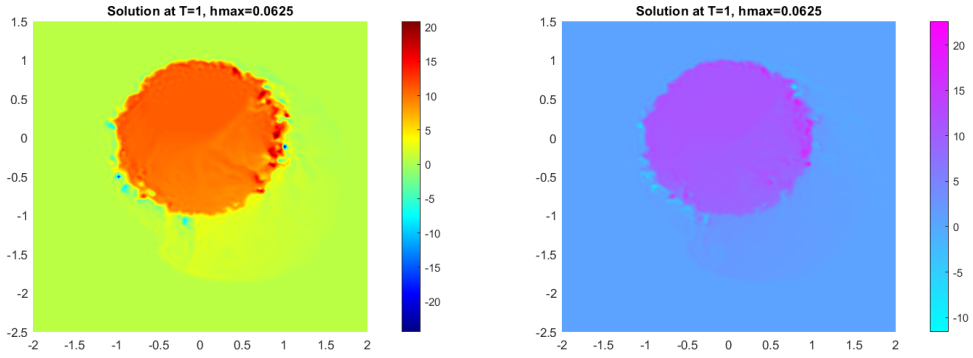


Figure 3: The solution from above with RK4-RV $hmax = 1/16$

Simulation result and figures

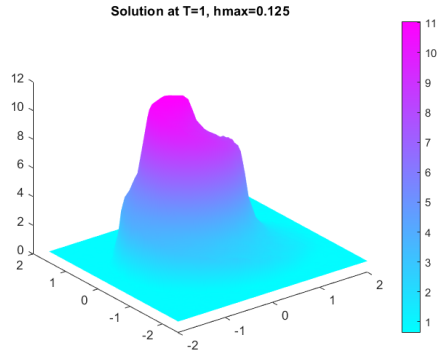
After implementing the above methods the results can be seen in Figure 9, with (5a,5b) containing the RK4-GFEM solutions, (5c,5d) containing the RK4-SUPG method and (5e, 5f) containing the RK4-RV methods solution. For simulations at each version $k_n = CFL * hmax$, $CFL = 0.05$ was used. Note that there is restriction on the times steps quadratic on the mesh size parameter. Note that with using the same time step the produced result are quite similar as in the Piccard iteration case, but the RK4 has much faster run times with similar time steps. Therefore here it is performs better in run time and the results is also better. For showing this I made a more colored version of the SUPG methods as the improvement shows there the most. The comparison can be seen at Figure 4. It can be seen that the RK4 version produce better result, where the wave form can be seen much clearer, while it can't really be seen at the Picard version.



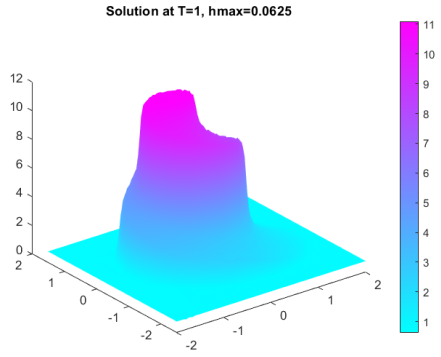
(a) SUPG solution for $hmax = 1/16$, $CFL = 0.05$

(b) RK4-SUPG solution for $hmax = 1/16$, $CFL = 0.05$

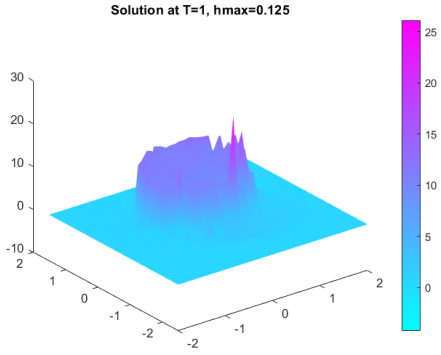
Figure 4: The solution from above with RK4-SUPG and SUPG $hmax = 1/16$



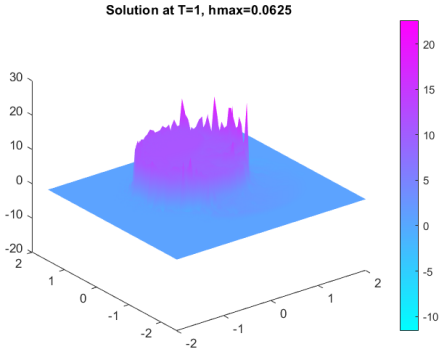
(a) RK4-GFEM solution for $hmax = 1/8$, $CFL = 0.05$



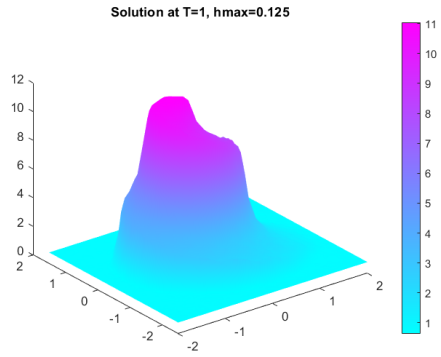
(b) RK4-GFEM solution for $hmax = 1/16$, $CFL = 0.05$



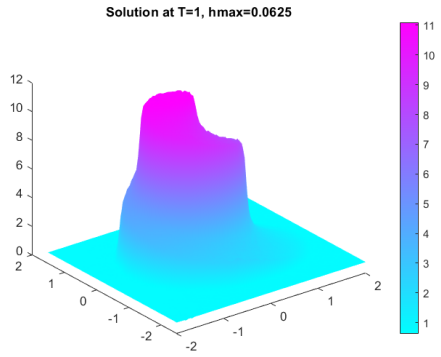
(c) RK4-SUPG solution for $hmax = 1/8$, $CFL = 0.05$



(d) RK4-SUPG solution for $hmax = 1/16$, $CFL = 0.05$



(e) RK4-RV solution for $hmax = 1/8$, $CFL = 0.05$



(f) RK4-RV solution for $hmax = 1/16$, $CFL = 0.1$

Figure 5: Solution of the KPP rotating wave problem at $T = 1$ using 4th order Runge-Kutta methods for time stepping

Summary

In conclusion the Picard iteration and the RK4 methods both seems a fine way of dealing with non-linear PDEs. The RK4 version produce better results as it was demonstrated with the SUPG method. But the major different is between the performance of the different formulation methods. For the given problem RV clearly performs the best, with being the only one producing considerably good results. And thus I consider RV with RK4 the best due to precision and computational speed to solve this problem from the tried methods.

Note: After adding Dirichlet boundary condition the simulation becomes stable at the edge and therefore the results in the end are much less noisy. The figures shows the results with boundary condition now.

Part B - Goal-oriented adaptivity

In this part we investigate the goal-oriented adaptive solution of the advection-diffusion equation. The goal-orientation can be used to make error estimate in an average sense. This is can be useful when we can't prove point wise convergation, but it is enough to show that a solution is good in a general sense. In this example we will use the goal-oriented error estimate to refine the grid we will do it for different goal-functions, which will be representing averages in different regions. This can be useful for application when a larger system is modelled, but we only interested in the solution in a smaller part. Now let's look at the given problem.

We consider the advection-diffusion equation in $\Omega = \{\boldsymbol{x} : x_1^2 + x_2^2 \leq 1\}$, $\boldsymbol{f}'(u) := \boldsymbol{\beta}(\boldsymbol{x}) = 2\pi(-x_2, x_1)$, a linear diffusion term has the form $-\nabla \cdot (\varepsilon \nabla u), \varepsilon \geq 0$, with boundary condition $u(x,t) = 0, \forall x \in \partial\Omega$. Th given primal problem is

$$\begin{aligned} \partial_t u + \boldsymbol{\beta} \cdot \nabla u - \nabla \cdot (\varepsilon \nabla u) &= 0, && (\boldsymbol{x}, t) \in \boldsymbol{\Omega} \times [0, T), \\ u &= 0, && (\boldsymbol{x}, t) \in \partial\Omega \times [0, T), \\ u(\boldsymbol{x}, 0) &= u_0(\boldsymbol{x}), && \boldsymbol{x} \in \Omega, \end{aligned}$$

With initial condition

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

where, with $r_0 = 0.25, (x_1^0, x_2^0) = (0.3, 0)$.

From here we could formulate the relation of the error ($e = u - U$). So an equation in the for $A(e) = -R(U)$. Then from here construct the adjoint problem $A^* z = \psi$. This is not necessary as the dual problem was already given in the problem description. The dual problem: find $z := z(x,t)$ s.t.

$$\begin{aligned} -\partial_t z - \boldsymbol{\beta} \cdot \nabla z - \nabla \cdot (\varepsilon \nabla z) &= \psi_\Omega, && (\boldsymbol{x}, t) \in \boldsymbol{\Omega} \times [T, 0), \\ z &= 0, && (\boldsymbol{x}, t) \in \partial\Omega \times [T, 0), \\ z(\boldsymbol{x}, T) &= 0, && \boldsymbol{x} \in \Omega, \end{aligned}$$

where $\psi_{\mathrm{n}} := \psi_\Omega(x,t)$ is given functions with $\psi_{\mathrm{n}} \in L_2(\Omega \times [0,T])$.

From here the target function is defined as

$$\mathcal{M}(u) = \int_{\Omega \times [0,T]} ut(\Omega) dx dt.$$

Problem B.1

In this part we should prove the following theorem for the above described advection-diffusion equation and it's dual problem.

Theorem 1. For the finite element approximation $U(t)$:
1. the following error representation holds:

$$|\mathcal{M}(u) - \mathcal{M}(U)| = \int_0^T \sum_{K \in T_h} \int_K \mathcal{R}(U(t)) \left(z(t) - \pi_h z(t)\right) \mathrm{d}x \ \mathrm{d}t,$$

2. the following error estimates holds:

$$|\mathcal{M}(u) - \mathcal{M}(U)| \leq C \int_0^T \sum_{K \in T_h} h_K \|\mathcal{R}(U(t))\|_{L^2(K)} \|Dz(t)\|_{L^2(K)} \mathrm{d}t,$$

where U is finite element approximation of u, π_h is the standard interpolation function, $\mathcal{R}(U)$ is the residual, Dz the first derivative of z, h_K is the cell diameter.

Now similarly as we did at the 8th lecture we can formulate a general equation for the error estimate in the target function. Let $Q = \Omega \times [0, T)$ an integration domain.

$$\mathcal{M}(u) - \mathcal{M}(U) = \int_Q (\underbrace{(u-U)}_{e}) \psi_\Omega d\bar{x}dt = \int_Q e\psi_\Omega d\bar{x}dt$$

$$= \int_Q e\left(-\partial_t z - \bar{\beta}\cdot\nabla z - \nabla\cdot(\varepsilon\nabla z)\right) d\bar{x}dt$$

{integrate by parts}

$$= \int_Q \left[\partial_t e + \bar{\beta}\cdot\nabla e - \nabla\cdot(\varepsilon\nabla e)\right] z d\bar{x}dt$$

$$+ \int_\Omega ez\Big|_0^T d\bar{x} - \int_{\partial Q} \bar{\beta}\cdot\bar{n}\underbrace{e}_{=0} z\, d\bar{s}dt - \int_{\partial Q} \varepsilon\partial_n z \underbrace{e}_{=0} d\bar{s}dt + \int_{\partial Q} \varepsilon\partial_n e \underbrace{z}_{=0} d\bar{s}dt$$

$$= \int_Q [\underbrace{\partial_t u + \bar{\beta}\cdot\nabla u - \nabla\cdot(\varepsilon\nabla u)}_{=\boldsymbol{f}} - \left(\partial_t U + \bar{\beta}\cdot\nabla U - \nabla\cdot(\varepsilon\nabla U)\right)] z d\bar{x}dt$$

$$= \int_Q -R(U)\, z d\bar{x}dt.$$

So we obtained the following equality$\mathcal{M}(u) - \mathcal{M}(U) = \int_Q -R(U)\, z\, d\bar{x}dt$.

Now using the Galerkin orthogonality, so that the residual ($R(U)$) is orthogonal to the finite element space in which we are working ($U \in V_h$). And we use that $\pi z \in V_h$. Now we obtain:

$$|\mathcal{M}(u) - \mathcal{M}(U)| = \left|\int_Q -R(U)\,(z - \pi z)\, d\bar{x}dt\right| = \int_0^T \sum_{K\in T_h} \int_K \mathcal{R}(U(t))\,(z(t) - \pi_h z(t))\,\mathrm{d}x\ \mathrm{d}t$$

From here using Cauchy-Schwarz inequality and the standard error estimates for the interpolations error and integrating the gained element wise constant function in the elements, so multiplying it with the area. We obtain the 2nd part of the theorem:

$$|\mathcal{M}(u) - \mathcal{M}(U)| \leq C \int_0^T \sum_{K\in T_h} h_K \|\mathcal{R}(U(t))\|_{L^2(K)} \|Dz(t)\|_{L^2(K)} \mathrm{d}t$$

With this both part of the theorem is proven.

Problem B.2

In this part we should obtain the GFEM formulation of the dual problem. It is similar as what we did in the first assignment. So use $\nabla \cdot \boldsymbol{f}(u) := \boldsymbol{f'}(u) \cdot \nabla u$, assuming $\boldsymbol{f'}(u) = 2\pi(-x_2, x_1)$. Then knowing that ∇ is a linear operator, we get the following GFEM formulation from the WF after time discretization: knowing Z_{n-1} we are solving for Z_n from $Z_0 = T$ until $Z_n = 0$:

$$\frac{1}{k_n}(Z_{n-1} - Z_n, v) - \frac{1}{2}(\boldsymbol{f'}(Z_{n-1}) \cdot \nabla Z_{n-1} + \boldsymbol{f'}(Z_n) \cdot \nabla Z_n, v) \\ - \frac{1}{2}(\epsilon \nabla Z_{n-1} + \epsilon \nabla Z_n, \nabla v) = \psi_\Omega, \quad \forall v \in \mathcal{X}_h$$

From here we can obtain the equations in the linear system by fully discretizing the space using $Z = \sum_j \xi_j \varphi_j$ and $\psi_\Omega = \sum_i \zeta_i \varphi_i$:

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

In pure matrix form we obtained:

$$\frac{1}{k_n} \boldsymbol{M}(\boldsymbol{\xi_{n-1}} - \boldsymbol{\xi_n}) - \frac{1}{2} \boldsymbol{C}(\boldsymbol{\xi_n} + \boldsymbol{\xi_{n-1}}) - \frac{1}{2} \boldsymbol{S}(\boldsymbol{\xi_n} + \boldsymbol{\xi_{n-1}}) = \boldsymbol{\zeta}$$

Where $\boldsymbol{M}, \boldsymbol{C}$ as before and $\boldsymbol{S}$ the stiffness matrix is obtained from $\int_\Omega \epsilon \nabla \varphi_j \nabla \varphi_i \, d\boldsymbol{x}$.

Problem B.3

In this part we did an implementation of the following adaptive goal oriented algorithm.

Using the following error indicator for each cell K :

$$\eta_K(t) = Ch_K \|\mathcal{R}(U(t))\|_{L^2(K)} \|Dz(t)\|_{L^x(K)}.$$

Algorithm

1. Initialize coarse grid $\mathcal{T}_N$, choose TOL > 0, and set $N := 0$.
2. Solve primal problem.
3. Solve the dual problem.
4. For every $K \in \mathcal{T}_N$: compute a posteriori error estimator η_K
5. If $\mathcal{E} = \int_0^T \sum_{K \in T_\mathrm{A}} \eta_K(t) dt <$ TOL : STOP
6. Refine 10% of cells with the largest $\int_0^T \eta_K(t) \mathrm{d}t$.
7. Set $N := N + 1$ and goto step 2 .

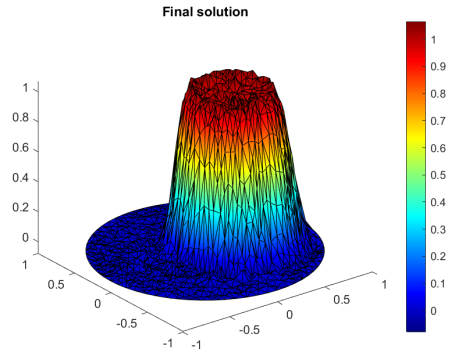
Notes on the implementation: For implementation the primal and dual solver from the GFEM formulation is straight forward. As the task asked I added an RV stabilization term for both the primal and dual solver. The more interesting part is the computation of the error in the cells. For the residual and derivative calculation I used the FEM form of this, so for residual I wrote a fem formulation and computed it from there. The WF for this $(R(U), v) = (\partial_t U + \nabla \cdot \boldsymbol{f'}(U) - \nabla \epsilon \nabla U, v)$. Similarly the derivative can be calculated as Cz, where C is the convection matrix. For the norms in the elements I used the following form $\|f\|_K = area(K) * \frac{f(n_1)+f(n_2)+f(n_3)}{3}$, where n_i is the nodes of the element. And finally instead of the given way of refinement, so refining the elements with at least eta value of the 80% of the max eta value I refined the largest 10% of eta values in the elements.

Now discuss the simulation results. The different simluation results can be found below in figure 7, 9, 11, 13, 15 containing the results of the different goal function. For each goal function I plotted the following: final U, z the primal and dual solutions, the final mesh, the final error and the error after one iteration. The final error indicators $R(U)$, Dz as a sum over time. And the asked loglog plot.

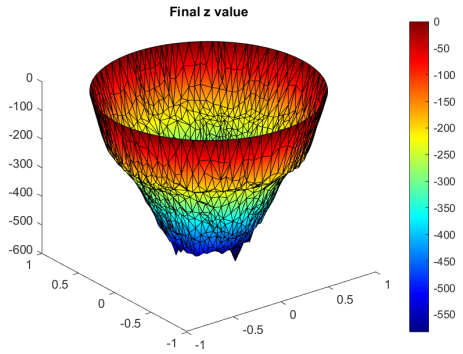
As it can be seen all produces relatively good result. But the mesh get most dense in the parts where the goal of refinement was. As from the $R(U)$ and Dz we can see the $R(U)$ errors are largest around the edge of the non-smooth starting data and the Dz values are largest in the goal oriented parts with large negative values. Also the error at the end doesn't say much, but after the first iteration it has a higher values in the goal oriented parts as it could be expected. After some refinement it is not so easy to guess where the largest error is coming from. From the convergence plot we can see a slower convergence at the end.

Figures for the goal function

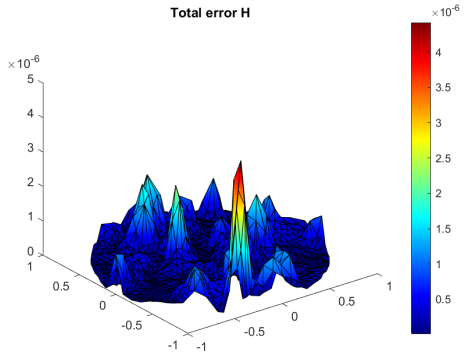
the average error in the domain, i.e., $x_0 = (0, 0), r_0 = 1$



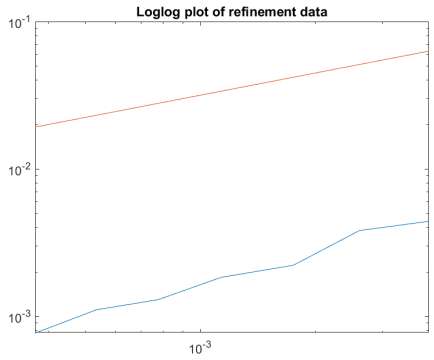
(a) Final fem solution



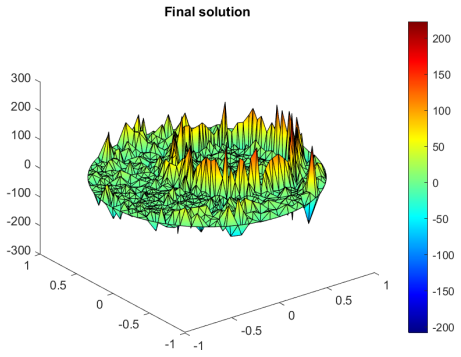
(b) Final dual solution



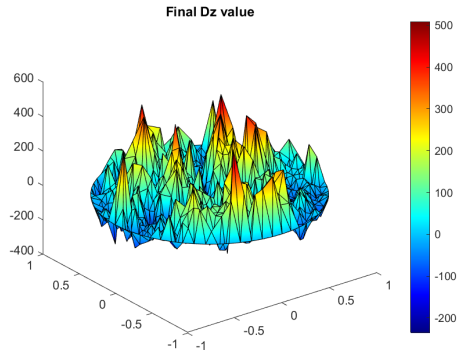
(c) Final error H



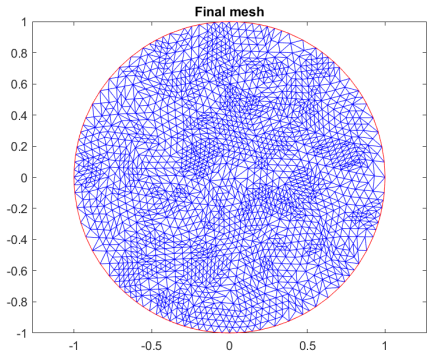
(d) Convergence plot



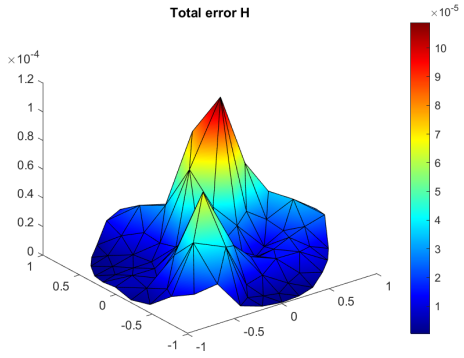
(a) Final $R(U)$ values sum over time



(b) Final Dz values sum over time



(c) Final mesh

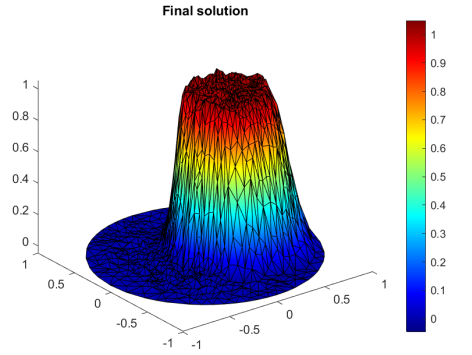


(d) Error H at $N = 1$

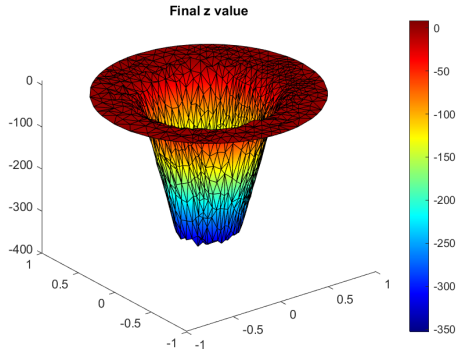
Figure 7: Plots for goal function 1.

Figures for the goal function 2

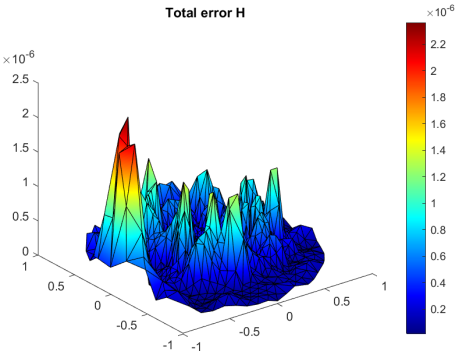
the average error in a circle with center $x_0 = (0.3, 0)$ and radius $r_0 = 0.25$.



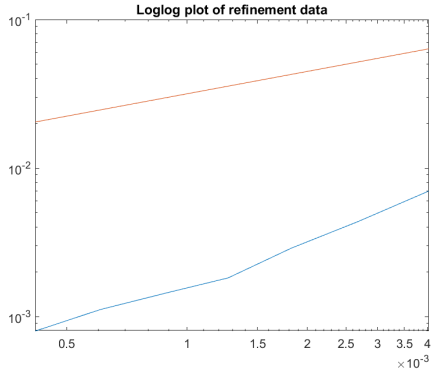
(a) Final fem solution



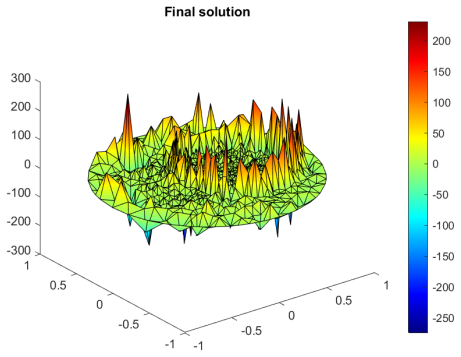
(b) Final dual solution



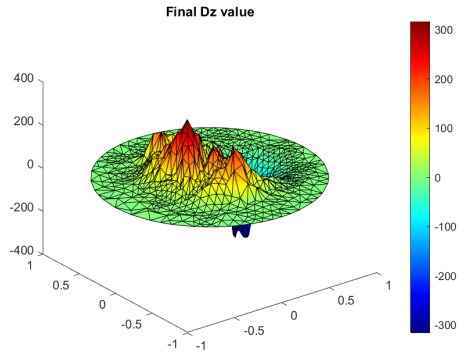
(c) Final error H



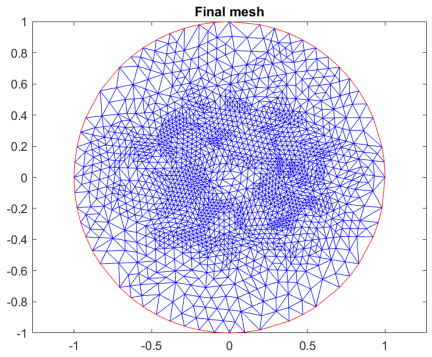
(d) Convergence plot



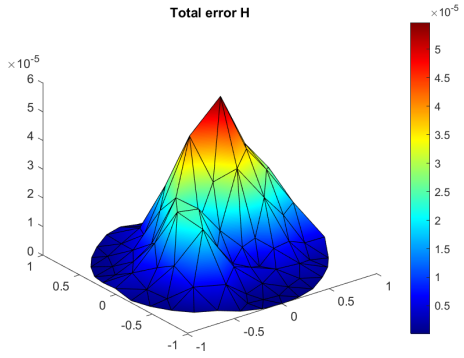
(a) Final $R(U)$ values sum over time



(b) Final Dz values sum over time



(c) Final mesh

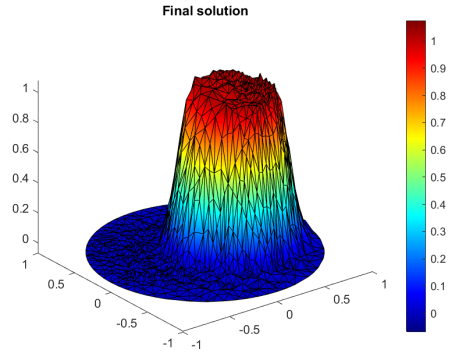


(d) Error H at $N = 1$

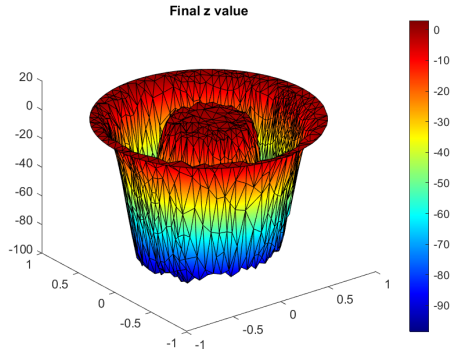
Figure 9: Plots for goal function 2.

Figures for the goal function 3

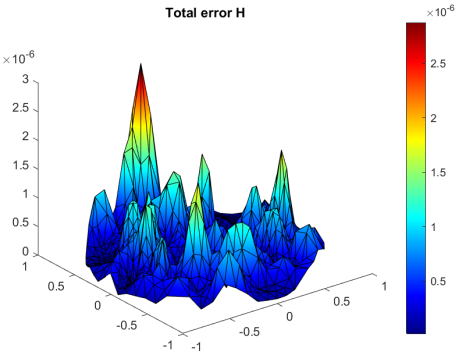
the average error in a circle with center $x_0 = (0.6, 0)$ and radius $r_0 = 0.15$.



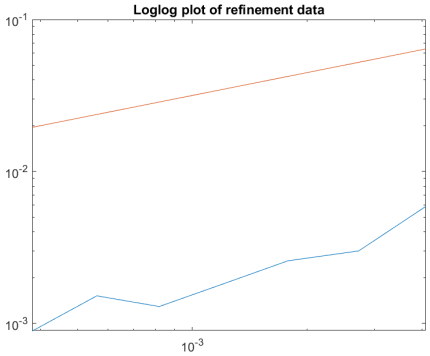
(a) Final fem solution



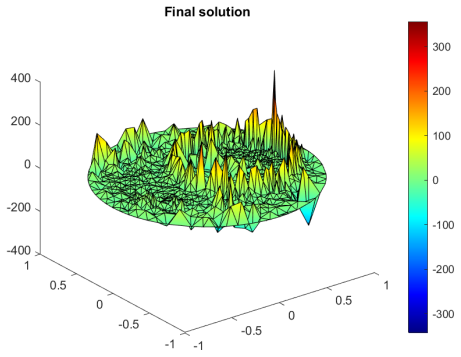
(b) Final dual solution



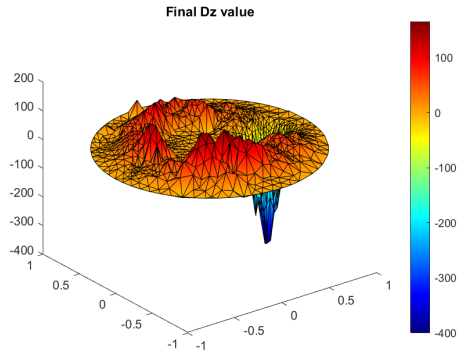
(c) Final error H



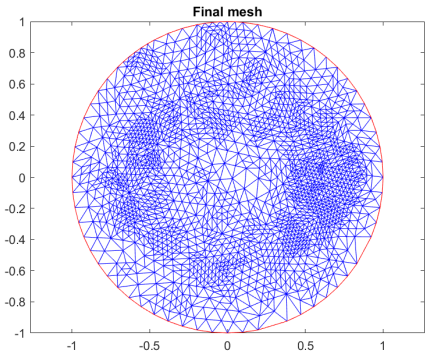
(d) Convergence plot



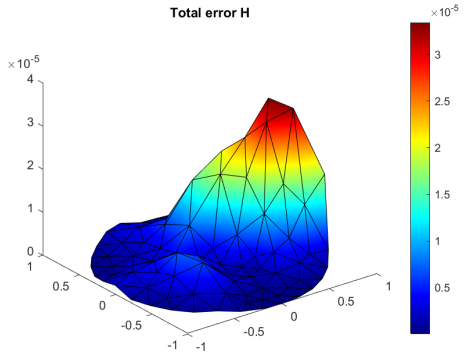
(a) Final $R(U)$ values sum over time



(b) Final Dz values sum over time



(c) Final mesh

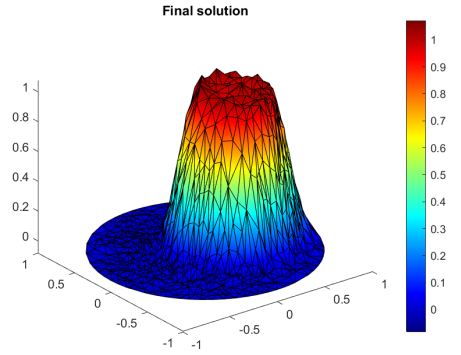


(d) Error H at $N = 1$

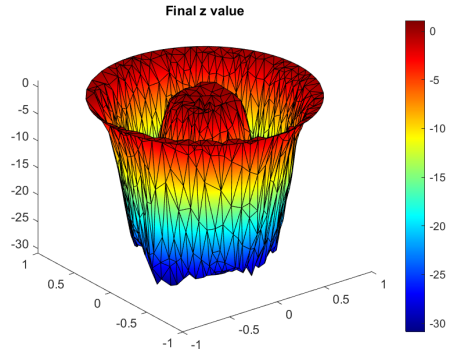
Figure 11: Plots for goal function 3.

Figures for the goal function 4

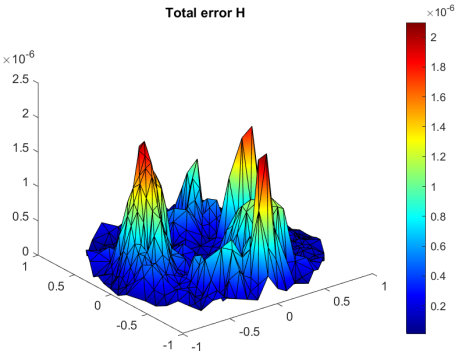
the average error in a circle with center $x_0 = (-0.6, 0)$ and radius $r_0 = 0.15$.



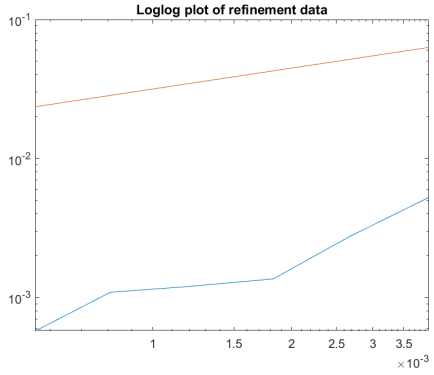
(a) Final fem solution



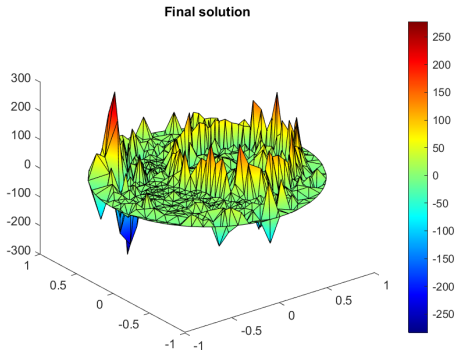
(b) Final dual solution



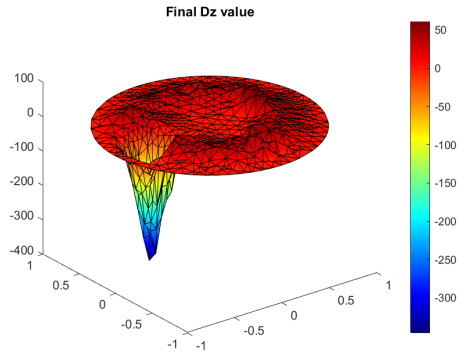
(c) Final error H



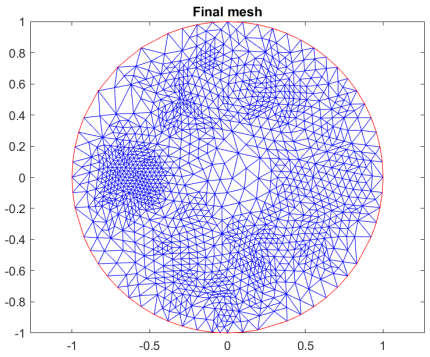
(d) Convergence plot



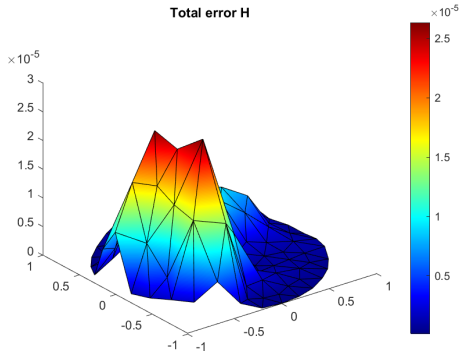
(a) Final $R(U)$ values sum over time



(b) Final Dz values sum over time



(c) Final mesh

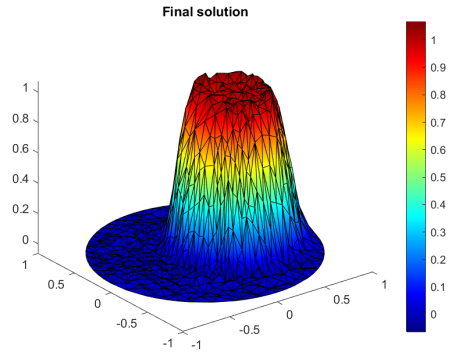


(d) Error H at $N = 1$

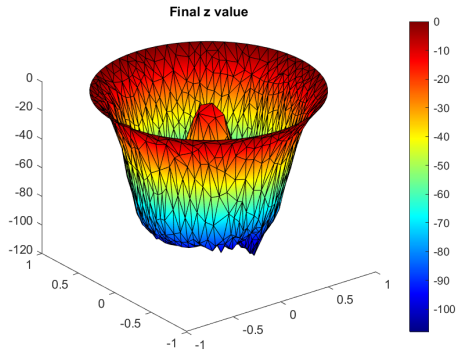
Figure 13: Plots for goal function 4.

Figures for the goal function 5

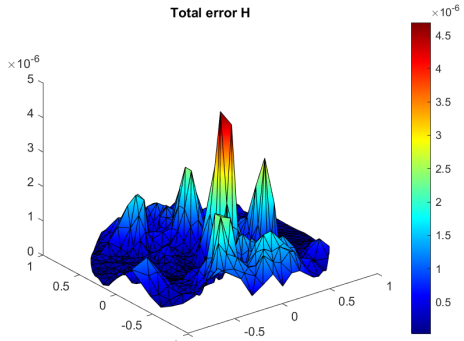
the average error in two circles with centers and radii $x_0 = (0, 0.6)$ and $r_0 = 0.15$, as well as $x_0 = (0, -0.55)$ and $r_0 = 0.35$.



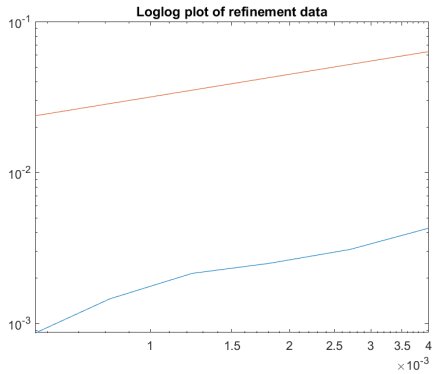
(a) Final fem solution



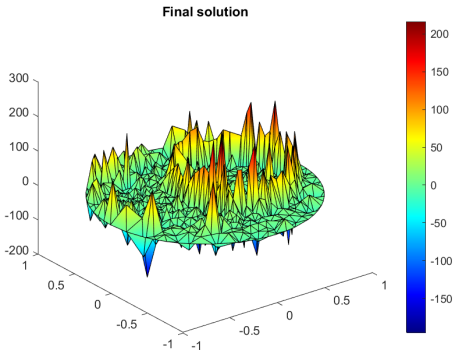
(b) Final dual solution



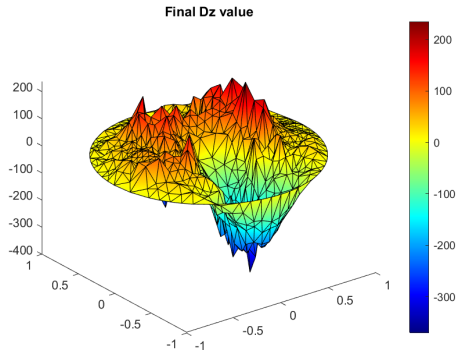
(c) Final error H



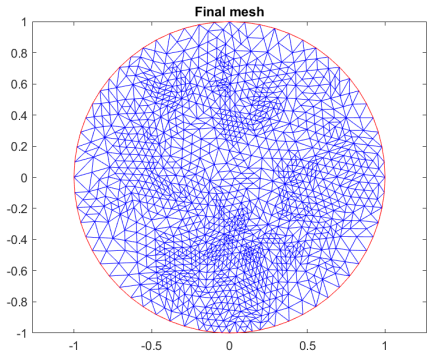
(d) Convergence plot



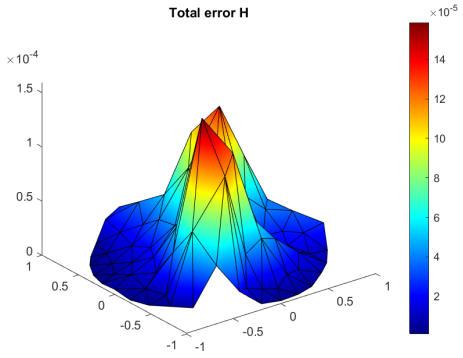
(a) Final $R(U)$ values sum over time



(b) Final Dz values sum over time



(c) Final mesh



(d) Error H at $N = 1$

Figure 15: Plots for goal function 5.

Summary

In summary the goal oriented method can be used to refine meshes with a goal of making the error small in a given region. As from the plots we can see we can generate quite good results with this methods.

Appendix

Matlab code for parts GFEM with Picard iteration

```
%% Given parameters
T=1;
CFL=0.05;
TOL=10^(-3);

%Used geometry
g=Rectangle(-2,-2.5,2,1.5);


%% Nonlinear GFEM
%hmax values to run the simulation for
hmax_list = [1/8, 1/16];

%Iteration on possible hmax value

for hmax = hmax_list
    %mesh data
    [p ,e , t ] = initmesh (g , 'hmax' , hmax); %
       function call to create mesh
    x=p(1,:)'; % vector of x1 coordinates
    y=p(2,:)'; % vector of x2 coordinates

    % Set mass matrix
    M = MassAssembler2D(p,t);

    %Initial setup
    U_0=initial_data(x,y)';
    U_n=U_0;
    U_i=U_0;
    U_new=U_0;


figure()
pdeplot(p,e,t,"XYData",U_n,'Zdata',U_n);
```

```
34 title("Solution at T="+T+", hmax="+hmax)
35     %Time integration Crank-Nicholson
36     time = 0;
37
38
39     kn = CFL * hmax; % Time steps
40
41
42     while time<T
43         df_n=ddf(U_n);
44         C_n=ConvectionAssembler2D(p,t,df_n(:,1),df_n
               (:,2));
45         r=1;
46         while r>TOL
47                 df_i=ddf(U_i);
48                     C=ConvectionAssembler2D(p,t,df_i
                           (:,1),df_i(:,2));
49                 A=M/kn+C/2;
50                 b=(M/kn)*U_n-(C_n/2)*U_n;
51
52                 %Boundary conditions
53                 I = eye( length ( p ));
54                 A(e(1,:) ,:) =I(e(1,:) ,:);
55                 b(e(1,:)) = pi/4;
56
57                 U_new=A\b;
58                 r=norm(U_i-U_new,2);
59                 disp(['The value of r is: ', num2str(r)
                       ])
60                 U_i=U_new;
61         end
62         time=time+kn;
63         disp(['The value of time is: ', num2str(time)])
64         U_n=U_new;
65     end
66
67
68     %Plot solution
```

```
69      figure()
70      pdeplot(p,e,t,"XYData",U_n,'Zdata',U_n);
71      title("Solution at T="+T+", hmax="+hmax)
72
73     %Error
74      error=U_n-U_0;
75      figure()
76      pdeplot (p ,[],t,'XYData',error,'ZData',error,'
            ColorBar','on')
77      title("Error at T="+T+", hmax="+hmax)
78
79
80  end
81
82
83
84
85
86
87  %% Assembler functions based on book (The Finite
        Element Method:Theory, Implementation, and
        Applications %%%%%%%%%%%%
88
89  %Gradients
90  function [area,b,c] = HatGradients(x,y)
91  area=polyarea(x,y);
92  b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
93  c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
94  end
95
96  % derivative f
97  function df=ddf(u)
98      df =[cos(u),-sin(u)];
99  end
100
101 %initial
102 function init=initial_data(x,y)
103     init=[];
```

```
104     for i=1:length(x)
105         if sqrt(x(i).^2 +y(i).^2)<=1
106             init(i)=14*pi/4;
107         else
108             init(i)=pi/4;
109         end
110     end
111 end
112
113
114 % Convection Matrix Assembler
115 function C = ConvectionAssembler2D(p,t,bx,by)
116     np=size(p,2);
117     nt=size(t,2);
118     C=sparse(np,np);
119     for i=1:nt
120         loc2glb=t(1:3,i);
121         x=p(1,loc2glb);
122         y=p(2,loc2glb);
123         [area,b,c]=HatGradients(x,y);
124         bxmid=mean(bx(loc2glb));
125         bymid=mean(by(loc2glb));
126         CK=ones(3,1)*(bxmid*b+bymid*c)'*area/3;
127         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
128     end
129 end
130
131 %Mass Matrix Assembler
132 function M = MassAssembler2D(p,t)
133     np = size(p,2); % number of nodes
134     nt = size(t,2); % number of elements
135     M = sparse(np,np); % allocate mass matrix
136     for K = 1:nt % loop over elements
137         loc2glb = t(1:3,K); % local-to-global map
138         x = p(1,loc2glb); % node x-coordinates
139         y = p(2,loc2glb); % y
140         area = polyarea(x,y); % triangle area
```

```
141             MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; % element mass matrix
142             M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+MK; % add element masses to M
143         end
144     end
145
146
147     function r = Rectangle(xmin,ymin,xmax,ymax)
148     r=[2 xmin xmax ymin ymin 1 0;
149     2 xmax xmax ymin ymax 1 0;
150     2 xmax xmin ymax ymax 1 0;
151     2 xmin xmin ymax ymin 1 0]';
152     end
```

Matlab code for parts SUPG with Picard iteration

```
%% Given parameters
T=1;
CFL=0.05;
TOL=10^(-3);

%Used geometry
g=Rectangle(-2,-2.5,2,1.5);



%% Problem 1 - numerical fem solutions and error + convergence rate calculation
%hmax values to run the simulation for
hmax_list = [1/16];

%Iteration on possible hmax values
for hmax = hmax_list
    delta=0.05*hmax;
        %mesh data
    [p ,e , t ] = initmesh (g , 'hmax' , hmax); % function call to create mesh
    x=p(1,:)'; % vector of x1 coordinates
    y=p(2,:)'; % vector of x2 coordinates

    % Set mass matrix
    M = MassAssembler2D(p,t);

    %Initial setup
    U_0=initial_data(x,y)';
    U_n=U_0;
    U_i=U_0;
    U_new=U_0;


    %Time integration Crank-Nicholson
    time = 0;
```

```
35      kn = CFL * hmax;
36
37      while time<T
38          %Set up equatddion matrix / vector
39          df_n=ddf(U_n);
40          C_n=ConvectionAssembler2D(p,t,df_n(:,1),df_n
                (:,2));
41          SC_n = SCAssembler2D(p,t,df_n(:,1),df_n(:,2),
                delta);
42          SM_n = delta*C_n';
43          r=1;
44          while r>TOL
45          df_n=ddf(U_i);
46          C_i=ConvectionAssembler2D(p,t,df_n(:,1),df_n
                (:,2));
47          SC_i = SCAssembler2D(p,t,df_n(:,1),df_n(:,2),
                delta);
48
49          A=((M/kn) + (C_i/2) + (SM_n/kn) + (SC_i/2));
50          b=((M/kn) - (C_n/2) + (SM_n/kn) - (SC_n/2)) *
                U_n;
51
52          %Boundary conditions
53          I = eye( length ( p ));
54          A(e(1,:) ,:) =I(e(1,:) ,:);
55          b(e(1,:)) = pi/4;
56
57          %Solve equation
58              U_new=A\b;
59              r=norm(U_i-U_new,2);
60              disp(['The value of r is: ', num2str(r)])
61              U_i=U_new;
62          end
63          time=time+kn;
64          disp(['The value of time is: ', num2str(time)])
65          U_n=U_new;
66      end
67
```

```
68
69      %Plot solution
70      figure()
71      pdeplot(p,e,t,"XYData",U_n,'Zdata',U_n,'ColorBar','
            on','ColorMap','jet');
72      title("Solution at T="+T+", hmax="+hmax)
73
74      %Error
75      error=U_n-U_0;
76      figure()
77      pdeplot (p ,[],t,'XYData',error,'ZData',error,'
            ColorBar','on','ColorMap','jet')
78      title("Error at T="+T+", hmax="+hmax)
79
80  end
81
82
83
84
85
86
87  %% Assembler functions based on book (The Finite
        Element Method:Theory, Implementation, and
        Applications %%%%%%%%%%%%%
88
89  % derivative f
90  function df=ddf(u)
91      df =[cos(u),-sin(u)];
92  end
93
94  %initial
95  function init=initial_data(x,y)
96      init=[];
97      for i=1:length(x)
98          if sqrt(x(i).^2 +y(i).^2)<=1
99              init(i)=14*pi/4;
100         else
101             init(i)=pi/4;
```

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

```
function A = SCAssembler2D(p,t,bx, by, delta)
np = size(p,2);
nt = size(t,2);
A = sparse(np,np); % allocate stiffness matrix
for K = 1:nt
loc2glb = t(1:3,K); % local-to-global map
x = p(1,loc2glb); % node x-coordinates
y = p(2,loc2glb); % node y-
[area,b,c] = HatGradients(x,y);
bxmid=mean(bx(loc2glb));
bymid=mean(by(loc2glb));
AK = delta*(bxmid^2*b*b'+bymid^2*c*c'+bxmid*bymid*(b*c
    '+c*b'))*area; % element stiffness matrix
A(loc2glb,loc2glb) = A(loc2glb,loc2glb)+ AK; % add
    element stiffnesses to A
end
end


%Gradients
function [area,b,c] = HatGradients(x,y)
area=polyarea(x,y);
b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
end

function r = Rectangle(xmin,ymin,xmax,ymax)
r=[2 xmin xmax ymin ymin 1 0;
2 xmax xmax ymin ymax 1 0;
2 xmax xmin ymax ymax 1 0;
2 xmin xmin ymax ymin 1 0]';
end
```

Matlab code for parts RV with Picard iteration

```
%% Given parameters
T=1;
CFL=0.1;
TOL=10^(-3);
C_vel=0.5;
C_RV=4;

%Used geometry
g=Rectangle(-2,-2.5,2,1.5);



%% Problem 1 - numerical fem solutions and error +
   convergence rate calculation
%hmax values to run the simulation for
hmax_list = [1/8,
     1/16];

%Iteration on possible hmax values
for hmax = hmax_list
          %mesh data
     [p ,e , t ] = initmesh (g , 'hmax' , hmax); %
        function call to create mesh
     x=p(1,:)'; % vector of x1 coordinates
     y=p(2,:)'; % vector of x2 coordinates

     % Set mass matrix
     M = MassAssembler2D(p,t);

     %Initial setup
     U_0=initial_data(x,y)';
     U_n=U_0;
     U_i=U_0;
     U_new=U_0;
     U_old=U_0;

```

```
35      nt = size(t,2); hk=[]
36      for K = 1:nt % for each element
37          loc2glb = t(1:3,K); % local-to-global map
38          x = p(1,loc2glb); % node x-coordinates
39          y = p(2,loc2glb); % node y-coordinates
40          hk(K)=min([norm([x(1)-x(2),y(1)-y(2)]),norm([x
                (2)-x(3),y(2)-y(3)]),norm([x(3)-x(1),y(3)-y
                (1)]), 2*norm([x(1)-sum(x)/3,y(1)-sum(y)/3])
                ]); % diameter
41      end
42
43      %Time integration Crank-Nicholson
44      time = 0;
45      kn = CFL * hmax; % timstep - not depends on U_n due
            to f'
46
47      while time<T
48          df_n=ddf(U_n);
49          C_n=ConvectionAssembler2D(p,t,df_n(:,1),df_n
                (:,2));
50          R_n = ResidualAssembler(p,t, ddf(U_n), hk, U_n,
                U_old, M, C_n, kn,C_vel,C_RV);
51          %Set up equatddion matrix / vector
52          b=((M/kn) - (C_n/2) - (R_n/2)) * U_n;
53
54         r=1;
55          while r>TOL
56          df_n=ddf(U_i);
57          C_i=ConvectionAssembler2D(p,t,df_n(:,1),df_n
                (:,2));
58
59          A=((M/kn) + (C_i/2) + (R_n/2));
60
61
62          %Boundary conditions
63          I = eye( length ( p ));
64          A(e(1,:) ,:) =I(e(1,:) ,:);
65          b(e(1,:)) = pi/4;
```

```
66
67
68          %Solve equation
69               U_new=A\b;
70               r=norm(U_i-U_new,2);
71               disp(['The value of r is: ', num2str(r)])
72               U_i=U_new;
73          end
74          time=time+kn;
75          disp(['The value of time is: ', num2str(time)])
76          U_old=U_n;
77          U_n=U_new;
78      end
79
80
81     %Plot solution
82      figure()
83      pdeplot(p,e,t,"XYData",U_n,'Zdata',U_n);
84      title("Solution at T="+T+", hmax="+hmax)
85
86      %Error
87      error=U_n-U_0;
88      figure()
89      pdeplot (p ,[],t,'XYData',error,'ZData',error,'
           ColorBar','on')
90      title("Error at T="+T+", hmax="+hmax)
91
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
103 % Convection Matrix Assembler
104 function C = ConvectionAssembler2D(p,t,bx,by)
105     np=size(p,2);
106     nt=size(t,2);
107     C=sparse(np,np);
108     for i=1:nt
109         loc2glb=t(1:3,i);
110         x=p(1,loc2glb);
111         y=p(2,loc2glb);
112         [area,b,c]=HatGradients(x,y);
113         bxmid=mean(bx(loc2glb));
114         bymid=mean(by(loc2glb));
115         CK=ones(3,1)*(bxmid*b+bymid*c)'*area/3;
116         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
117     end
118 end
119
120 %Mass Matrix Assembler
121 function M = MassAssembler2D(p,t)
122     np = size(p,2); % number of nodes
123     nt = size(t,2); % number of elements
124     M = sparse(np,np); % allocate mass matrix
125     for K = 1:nt % loop over elements
126         loc2glb = t(1:3,K); % local-to-global map
127         x = p(1,loc2glb); % node x-coordinates
128         y = p(2,loc2glb); % y
129         area = polyarea(x,y); % triangle area
130         MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
                element mass matrix
131         M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
                add element masses to M
132     end
133 end
134
135 % SC matrix assembly
136 function R = ResidualAssembler(p,t, df,hk, U, U_old, M,
```

```
      C, kn, C_vel,C_RV)
np = size(p,2);
nt = size(t,2);
R = sparse(np,np); % allocate matrix
RU=M\(1/kn*M*(U-U_old)+C*U);
for K = 1:nt % for each element
loc2glb = t(1:3,K); % local-to-global map
x = p(1,loc2glb); % node x-coordinates
y = p(2,loc2glb); % node y-coordinates
[area,b,c] = HatGradients(x,y);
dx=df(loc2glb,1);
dy=df(loc2glb,2);
h_k=hk(K);
b_k=max([norm([dx(1),dy(1)]),norm([dx(2),dy(2)]),norm([
    dx(3),dy(3)])]);
U_norm=max(abs(U-mean(U)));
Rk=max(abs(RU(loc2glb)));

e=min([C_vel*h_k*b_k,C_RV*h_k^2*Rk/U_norm]);
AK = e*(b*b'+c*c')*area; % element stiffness matrix
R(loc2glb,loc2glb) = R(loc2glb,loc2glb)+ AK; % add
    element stiffnesses to A
end
end

%Gradients
function [area,b,c] = HatGradients(x,y)
area=polyarea(x,y);
b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
end

% derivative f
function df=ddf(u)
     df =[cos(u),-sin(u)];
end

%initial
```

```
172 function init=initial_data(x,y)
173     init=[];
174     for i=1:length(x)
175         if sqrt(x(i).^2 +y(i).^2)<=1
176             init(i)=14*pi/4;
177         else
178             init(i)=pi/4;
179         end
180     end
181 end
182 function r = Rectangle(xmin,ymin,xmax,ymax)
183 r=[2 xmin xmax ymin ymin 1 0;
184 2 xmax xmax ymin ymax 1 0;
185 2 xmax xmin ymax ymax 1 0;
186 2 xmin xmin ymax ymin 1 0]';
187 end
```

Matlab code for parts GFEM with RK4

```
%% Given parameters
T=1;
CFL=0.05;

%Used geometry
g=Rectangle(-2,-2.5,2,1.5);


%% Nonlinear GFEM
%hmax values to run the simulation for
hmax_list = [1/8,1/16];

%Iteration on possible hmax value

for hmax = hmax_list
    %mesh data
    [p,e,t] = initmesh(g, 'hmax', hmax); %
       function call to create mesh
    x=p(1,:)'; % vector of x1 coordinates
    y=p(2,:)'; % vector of x2 coordinates

    % Set mass matrix
    M = MassAssembler2D(p,t);

    %Initial setup
    U=initial_data(x,y)';



    %Time integration Crank-Nicholson
    time = 0;
    kn = CFL * hmax; % Time steps

    while time<T
        df_n=ddf(U);
        C_n=ConvectionAssembler2D(p,t,df_n(:,1),df_n
```

```
            (:,2));
36          b=-C_n*U;
37
38
39
40
41          %Boundary conditions
42          I = eye( length ( p ));
43          M(e(1,:) ,:) =I(e(1,:) ,:);
44          b(e(1,:)) = pi/4;
45
46          k1=M\b;
47          b1=-C_n*(U+k1*kn/2);
48          b1(e(1,:)) = pi/4;
49          k2=M\b1;
50          b2=-C_n*(U+k2*kn/2);
51          b2(e(1,:)) = pi/4;
52          k3=M\b2;
53          b3=-C_n*(U+k3*kn);
54          b3(e(1,:)) = pi/4;
55          k4=M\b3;
56
57          U=U+kn/6*(k1+2*k2+2*k3+k4);
58          U(e(1,:))=pi/4;
59          time=time+kn;
60          disp(['The value of time is: ', num2str(time)])
61      end
62
63
64      %Plot solution
65      figure()
66      pdeplot(p,e,t,"XYData",U,'Zdata',U);
67      title("Solution at T="+T+", hmax="+hmax)
68
69
70
71  end
72
```

```
73
74
75
76
77
78 %% Assembler functions based on book (The Finite
       Element Method:Theory, Implementation, and
       Applications %%%%%%%%%%%%
79
80 %Gradients
81 function [area,b,c] = HatGradients(x,y)
82 area=polyarea(x,y);
83 b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
84 c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
85 end
86
87 % derivative f
88 function df=ddf(u)
89     df =[cos(u),-sin(u)];
90 end
91
92 %initial
93 function init=initial_data(x,y)
94     init=[];
95     for i=1:length(x)
96         if sqrt(x(i).^2 +y(i).^2)<=1
97             init(i)=14*pi/4;
98         else
99             init(i)=pi/4;
100        end
101    end
102 end
103
104
105 % Convection Matrix Assembler
106 function C = ConvectionAssembler2D(p,t,bx,by)
107     np=size(p,2);
108     nt=size(t,2);
```

```
109     C=sparse(np,np);
110     for i=1:nt
111         loc2glb=t(1:3,i);
112         x=p(1,loc2glb);
113         y=p(2,loc2glb);
114         [area,b,c]=HatGradients(x,y);
115         bxmid=mean(bx(loc2glb));
116         bymid=mean(by(loc2glb));
117         CK=ones(3,1)*(bxmid*b+bymid*c)'*area/3;
118         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
119     end
120 end
121
122 %Mass Matrix Assembler
123 function M = MassAssembler2D(p,t)
124     np = size(p,2); % number of nodes
125     nt = size(t,2); % number of elements
126     M = sparse(np,np); % allocate mass matrix
127     for K = 1:nt % loop over elements
128         loc2glb = t(1:3,K); % local-to-global map
129         x = p(1,loc2glb); % node x-coordinates
130         y = p(2,loc2glb); % y
131         area = polyarea(x,y); % triangle area
132         MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
                element mass matrix
133         M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
                add element masses to M
134     end
135 end
136
137
138 function r = Rectangle(xmin,ymin,xmax,ymax)
139 r=[2 xmin xmax ymin ymin 1 0;
140 2 xmax xmax ymin ymax 1 0;
141 2 xmax xmin ymax ymax 1 0;
142 2 xmin xmin ymax ymin 1 0]';
143 end
```

Matlab code for parts SUPG with RK4

```
%% Given parameters
T=1;
CFL=0.05;

%Used geometry
g=Rectangle(-2,-2.5,2,1.5);


%% Nonlinear GFEM
%hmax values to run the simulation for
hmax_list = [1/8,1/16];

%Iteration on possible hmax value

for hmax = hmax_list
    %mesh data
    [p,e,t] = initmesh(g, 'hmax', hmax); %
       function call to create mesh
    x=p(1,:)'; % vector of x1 coordinates
    y=p(2,:)'; % vector of x2 coordinates

    % Set mass matrix
    M = MassAssembler2D(p,t);

    %Initial setup
    U=initial_data(x,y)';



    %Time integration Crank-Nicholson
    time = 0;
    kn = CFL * hmax; % Time steps
    delta=0.05*hmax;
    while time<T
        df_n=ddf(U);
        C_n=ConvectionAssembler2D(p,t,df_n(:,1),df_n
```

```
        (:,2));
36          SC_n = SCAssembler2D(p,t,df_n(:,1),df_n(:,2),
                delta);
37          SM_n=delta*C_n';
38          A=M+SM_n;
39          b=-(C_n+SC_n)*U;
40
41          I = eye( length ( p ));
42          A(e(1,:) ,:) =I(e(1,:) ,:);
43          b(e(1,:)) = pi/4;
44
45          k1=A\b;
46          b1=-(C_n+SC_n)*(U+k1*kn/2);
47          b1(e(1,:)) = pi/4;
48          k2=A\b1;
49          b2=-(C_n+SC_n)*(U+k2*kn/2);
50          b2(e(1,:)) = pi/4;
51          k3=A\b2;
52          b3=-(C_n+SC_n)*(U+k3*kn);
53          b3(e(1,:)) = pi/4;
54          k4=A\b3;
55
56          U=U+kn/6*(k1+2*k2+2*k3+k4);
57          U(e(1,:)) = pi/4;
58          time=time+kn;
59          disp(['The value of time is: ', num2str(time)])
60      end
61
62
63      %Plot solution
64      figure()
65      pdeplot(p,e,t,"XYData",U,'Zdata',U);
66      title("Solution at T="+T+", hmax="+hmax)
67
68
69
70  end
71
```

```
72
73
74
75
76
77 %% Assembler functions based on book (The Finite
       Element Method:Theory, Implementation, and
       Applications %%%%%%%%%%%%
78
79 %Gradients
80 function [area,b,c] = HatGradients(x,y)
81 area=polyarea(x,y);
82 b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
83 c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
84 end
85
86 % derivative f
87 function df=ddf(u)
88     df =[cos(u),-sin(u)];
89 end
90
91 %initial
92 function init=initial_data(x,y)
93     init=[];
94     for i=1:length(x)
95         if sqrt(x(i).^2 +y(i).^2)<=1
96             init(i)=14*pi/4;
97         else
98             init(i)=pi/4;
99         end
100    end
101 end
102
103
104 % Convection Matrix Assembler
105 function C = ConvectionAssembler2D(p,t,bx,by)
106     np=size(p,2);
107     nt=size(t,2);
```

```
108     C=sparse(np,np);
109     for i=1:nt
110         loc2glb=t(1:3,i);
111         x=p(1,loc2glb);
112         y=p(2,loc2glb);
113         [area,b,c]=HatGradients(x,y);
114         bxmid=mean(bx(loc2glb));
115         bymid=mean(by(loc2glb));
116         CK=ones(3,1)*(bxmid*b+bymid*c)'*area/3;
117         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
118     end
119 end
120
121 %Mass Matrix Assembler
122 function M = MassAssembler2D(p,t)
123     np = size(p,2); % number of nodes
124     nt = size(t,2); % number of elements
125     M = sparse(np,np); % allocate mass matrix
126     for K = 1:nt % loop over elements
127         loc2glb = t(1:3,K); % local-to-global map
128         x = p(1,loc2glb); % node x-coordinates
129         y = p(2,loc2glb); % y
130         area = polyarea(x,y); % triangle area
131         MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
                element mass matrix
132         M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
                add element masses to M
133     end
134 end
135
136 % SC matrix assembly
137 function A = SCAssembler2D(p,t,bx, by, delta)
138 np = size(p,2);
139 nt = size(t,2);
140 A = sparse(np,np); % allocate stiffness matrix
141 for i = 1:nt
142 loc2glb = t(1:3,i); % local-to-global map
143 x = p(1,loc2glb); % node x-coordinates
```

```
144 y = p(2,loc2glb); % node y-
145 [area,b,c] = HatGradients(x,y);
146 bxmid=mean(bx(loc2glb));
147 bymid=mean(by(loc2glb));
148 AK = delta*(bxmid^2*b*b'+bymid^2*c*c'+bxmid*bymid*(b*c
      '+c*b'))*area; % element stiffness matrix
149 A(loc2glb,loc2glb) = A(loc2glb,loc2glb)+ AK; % add
      element stiffnesses to A
150 end
151 end
152
153
154 function r = Rectangle(xmin,ymin,xmax,ymax)
155 r=[2 xmin xmax ymin ymin 1 0;
156 2 xmax xmax ymin ymax 1 0;
157 2 xmax xmin ymax ymax 1 0;
158 2 xmin xmin ymax ymin 1 0]';
159 end
```

Matlab code for parts RV with RK4

```
%% Given parameters
T=1;CFL=0.05;
C_vel=0.5;
C_RV=4;

%Used geometry
g=Rectangle(-2,-2.5,2,1.5);



%% Problem
%hmax values to run the simulation for
hmax_list = [1/8,1/16];

%Iteration on possible hmax values
for hmax = hmax_list
        %mesh data
    [p ,e , t ] = initmesh (g , 'hmax' , hmax); %
       function call to create mesh
    x=p(1,:)'; % vector of x1 coordinates
    y=p(2,:)'; % vector of x2 coordinates

    % Set mass matrix
    M = MassAssembler2D(p,t);

    %Initial setup
    U=initial_data(x,y)';
    U_old=U;

    nt = size(t,2); hk=[]
    for K = 1:nt % for each element
        loc2glb = t(1:3,K); % local-to-global map
        x = p(1,loc2glb); % node x-coordinates
        y = p(2,loc2glb); % node y-coordinates
        hk(K)=min([norm([x(1)-x(2),y(1)-y(2)]),norm([x
            (2)-x(3),y(2)-y(3)]),norm([x(3)-x(1),y(3)-y
```

```
        (1)]), 2*norm([x(1)-sum(x)/3,y(1)-sum(y)/3])
        ]); % diameter
35  end
36
37  %Time integration Crank-Nicholson
38  time = 0;
39  kn = CFL * hmax;
40
41  while time<T
42      df=ddf(U);
43      C=ConvectionAssembler2D(p,t,df(:,1),df(:,2));
44      R = ResidualAssembler(p,t, ddf(U), hk, U, U_old
            , M, C, kn,C_vel,C_RV);
45      %Set up equatddion matrix / vector
46
47      %Boundary conditions
48      I = eye( length ( p ));
49      M(e(1,:) ,:) =I(e(1,:) ,:);
50      b=((C+R)*U);
51      b(e(1,:)) = pi/4;
52
53
54          k1=-M\b;
55          b1=-(C+R)*(U+k1*kn/2);
56          b1(e(1,:)) = pi/4;
57
58          k2=M\b1;
59          b2=-(C+R)*(U+k2*kn/2);
60          b2(e(1,:)) = pi/4;
61          k3=M\b2;
62          b3=-(C+R)*(U+k3*kn);
63          b3(e(1,:)) = pi/4;
64          k4=M\b3;
65          % SOlution
66          U_old=U;
67          U=U+(kn/6)*(k1+2*k2+2*k3+k4);
68          U(e(1,:)) = pi/4;
69
```

```
70          time=time+kn;
71          disp(['The value of time is: ', num2str(time)])
72          U_prev=U_new;
73      end
74
75
76      %Plot solution
77       figure()
78       pdeplot(p,e,t,"XYData",U,'Zdata',U);
79       title("Solution at T="+T+", hmax="+hmax)
80
81
82
83
84  end
85
86
87
88
89
90
91  %% Assembler functions based on book (The Finite
        Element Method:Theory, Implementation, and
        Applications %%%%%%%%%%%%
92
93
94  % Convection Matrix Assembler
95  function C = ConvectionAssembler2D(p,t,bx,by)
96       np=size(p,2);
97       nt=size(t,2);
98       C=sparse(np,np);
99       for i=1:nt
100          loc2glb=t(1:3,i);
101          x=p(1,loc2glb);
102          y=p(2,loc2glb);
103          [area,b,c]=HatGradients(x,y);
104          bxmid=mean(bx(loc2glb));
105          bymid=mean(by(loc2glb));
```

```
106             CK=ones(3,1)*(bxmid*b+bymid*c)'*area/3;
107             C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
108         end
109     end
110
111     %Mass Matrix Assembler
112     function M = MassAssembler2D(p,t)
113         np = size(p,2); % number of nodes
114         nt = size(t,2); % number of elements
115         M = sparse(np,np); % allocate mass matrix
116         for K = 1:nt % loop over elements
117             loc2glb = t(1:3,K); % local-to-global map
118             x = p(1,loc2glb); % node x-coordinates
119             y = p(2,loc2glb); % y
120             area = polyarea(x,y); % triangle area
121             MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
                    element mass matrix
122             M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
                    add element masses to M
123         end
124     end
125
126     % SC matrix assembly
127     function R = ResidualAssembler(p,t, df,hk, U, U_old, M,
            C, kn, C_vel,C_RV)
128     np = size(p,2);
129     nt = size(t,2);
130     R = sparse(np,np); % allocate matrix
131     RU=M\(1/kn*M*(U-U_old)+C*U);
132     for K = 1:nt % for each element
133     loc2glb = t(1:3,K); % local-to-global map
134     x = p(1,loc2glb); % node x-coordinates
135     y = p(2,loc2glb); % node y-coordinates
136     [area,b,c] = HatGradients(x,y);
137     dx=df(loc2glb,1);
138     dy=df(loc2glb,2);
139     h_k=hk(K);
140     b_k=max([norm([dx(1),dy(1)]),norm([dx(2),dy(2)]),norm([
```

```
    dx(3),dy(3)])]);
U_norm=max(abs(U-mean(U)));
Rk=max(abs(RU(loc2glb)));

e=min([C_vel*h_k*b_k,C_RV*h_k^2*Rk/U_norm]);
AK = e*(b*b'+c*c')*area; % element stiffness matrix
R(loc2glb,loc2glb) = R(loc2glb,loc2glb)+ AK; % add
    element stiffnesses to A
end
end

%Gradients
function [area,b,c] = HatGradients(x,y)
area=polyarea(x,y);
b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
end

% derivative f
function df=ddf(u)
    df =[cos(u),-sin(u)];
end

%initial
function init=initial_data(x,y)
    init=[];
    for i=1:length(x)
        if sqrt(x(i).^2 +y(i).^2)<=1
            init(i)=14*pi/4;
        else
            init(i)=pi/4;
        end
    end
end
function r = Rectangle(xmin,ymin,xmax,ymax)
r=[2 xmin xmax ymin ymin 1 0;
2 xmax xmax ymin ymax 1 0;
2 xmax xmin ymax ymax 1 0;
```

```
177  2 xmin xmin ymax ymin 1 0]';
178  end
```

Matlab code for the goal oriented parts

```
%% Given parameters
T=1;
TOL=10^(-3);
maxIter = 200;

CFL=0.1;
C_vel=0.25;
C_RV=1;
eps=0.000;
C_eta=0.4;

FUNCTION=2;

for FUNCTION = 1:5
    if FUNCTION==1
        C_eta=0.1;
    elseif FUNCTION==5
        C_eta=0.15;
    end

%initial data parameter
r0 = 0.25;
x1_0 = 0.3;
x2_0 = 0;

%goal oriented function parameters
switch FUNCTION
    case 1
        psi_x = 0;
        psi_y = 0;
        psi_r = 1;
    case 2
        psi_x = 0.3;
        psi_y = 0;
        psi_r = 0.25;
    case 3
```

```
37          psi_x = 0.6;
38          psi_y = 0;
39          psi_r = 0.15;
40      case 4
41          psi_x = -0.6;
42          psi_y = 0;
43          psi_r = 0.15;
44      case 5
45          psi_x = 0;
46          psi_y = 0.6;
47          psi_r = 0.15;
48
49          psi2_x = 0;
50          psi2_y = -0.55;
51          psi2_r = 0.35;
52  end
53
54  %Used geometry
55  g = @circleg;
56
57
58  %% Adaptive goal-oriented method
59  %hmax values to run the simulation for
60  hmax = 1/4;
61  eta=[1,1];
62
63
64  [p,e,t] = initmesh(g, 'hmax', hmax); %function
        call to create mesh
65  N=1;
66  N_list = zeros(maxIter,1);
67  E_list= zeros(maxIter, 1);
68  while sum(eta) > TOL && N < maxIter
69      x=p(1,:)'; % vector of x1 coordinates
70      y=p(2,:)'; % vector of x2 coordinates
71      df=ddf(x,y)';
72
73      %initial data
```

```
74      U_00 = ((x - x1_0).^2 + (y - x2_0).^2) <r0;
75      U_0=double(U_00);
76
77      Z_0= zeros(size(x));
78      psi = exp(-((x - psi_x).^2 + (y - psi_y).^2)/psi_r
            ^2);
79      if FUNCTION ==5
80       psi2=exp(-((x - psi2_x).^2 + (y - psi2_y).^2)/
            psi2_r^2);
81       psi=psi+psi2;
82      end
83
84     %time step
85      inf_norm = max(abs(df(1,:))+abs(df(2,:)));
86      kn = CFL * hmax/inf_norm*1.0; % timstep - not
           depends on U_n due to f'
87
88      % Set mass matrix
89      M = MassAssembler2D(p,t);
90      C = ConvectionAssembler2D(p,t,df(1,:),df(2,:));
91      S = StiffnessAssembler2D(p,t,eps);
92
93      nt = size(t,2); hk=[]; bk=[];
94      for K = 1:nt % for each element
95           loc2glb = t(1:3,K); % local-to-global map
96           x_ = p(1,loc2glb); % node x-coordinates
97           y_ = p(2,loc2glb); % node y-coordinates
98           hk(K)=min([norm([x_(1)-x_(2),y_(1)-y_(2)]),norm
                ([x_(2)-x_(3),y_(2)-y_(3)]),norm([x_(3)-x_
                (1),y_(3)-y_(1)]), 2*norm([x_(1)-sum(x_)/3,
                y_(1)-sum(y_)/3])]); % diameter
99           dx=df(1,loc2glb);
100          dy=df(2,loc2glb);
101          bk(K)=max([norm([dx(1),dy(1)]),norm([dx(2),dy
                (2)]),norm([dx(3),dy(3)])]);
102     end
103
104
```

```
105     %% Primal time integration Crank-Nicholson
106     U = U_0; U_old=U;
107     tim_len = floor(T/kn);
108     kn = T/tim_len;
109
110     U_norm=zeros(length(x),tim_len);
111     for i = 1:tim_len
112
113         RU=M\(1/kn*M*(U-U_old)+C*U-S*U);
114         R = ResidualAssembler(p,t, bk, hk, U, RU,C_vel,
                C_RV);
115
116         %Set up equation matrix / vector
117         A=((M/kn) + (C/2) - (S/2) + (R/2));
118         b=((M/kn) - (C/2) + (S/2) - (R/2))*U;
119
120         %Boundary conditions
121         I = eye( length ( p ));
122         A(e(1,:) ,:) =I(e(1,:) ,:);
123         b(e(1,:)) = 0;
124
125         %Solve equation
126         U_old=U;
127         U = A\b;
128         U(e(1,:))=0;
129
130         %Store Data
131         %%%%%%%%%%%%%%%%%%%%%%%%%%%% TO DO
              %%%%%%%%%%%%%%%%%%%%%%%%%%%%5
132         U_norm(:,i)=M\((M/kn+C-S)*U-(M/kn)*U_old);
133         U_norm(e(1,:),i)=0;
134     end
135
136
137 %% Dual time integration Crank-Nicholson
138     Z = Z_0; Z_old=Z;
139     inf_norm = max(abs(df(1,:))+abs(df(2,:)));
140     DZ=zeros(length(x),tim_len);
```

```
141
142     for i = 1:tim_len
143
144         RU=M\(1/kn*M*(Z_old-Z)-C*Z-S*Z);
145         R = ResidualAssembler(p,t, bk, hk, Z, RU,C_vel,
                C_RV);
146         %Set up equation matrix / vector
147         A=((M/kn) + (C/2) + (S/2) +(R/2));
148         b=((M/kn) - (C/2) - (S/2) - (R/2))*Z - psi;
149
150         %Boundary conditions
151         I = eye( length ( p ));
152         A(e(1,:) ,:) =I(e(1,:) ,:);
153         b(e(1,:)) = 0;
154
155         %Solve equation
156         Z_old=Z;
157         Z = A\b;
158         Z(e(1,:))=0;
159
160         %Store Data
161         %%%%%%%%%%%%%%%%%%%%%%%%%%%% TO DO
              %%%%%%%%%%%%%%%%%%%%%%%%%%%5
162         DZ(:,tim_len-i+1) = C*Z;
163         DZ(e(1,:),tim_len-i+1)=0;
164     end
165
166     %eta calculation
167     np = size(p,2); % number of nodes
168     nt = size(t,2); % number of elements
169     eta = zeros(nt,1); % allocate mass matrix
170     for K = 1:nt % loop over elements
171         loc2glb = t(1:3,K); % local-to-global map
172         x = p(1,loc2glb); % node x-coordinates
173         y = p(2,loc2glb); % y
174         area = polyarea(x,y); % triangle area
175         hk=min([norm([x(1)-x(2),y(1)-y(2)]),norm([x(2)-
              x(3),y(2)-y(3)]),norm([x(3)-x(1),y(3)-y(1)])
```

```
                , 2*norm([x(1)-sum(x)/3,y(1)-sum(y)/3])]); %
                 diameter
176           eta(K) = C_eta*hk*kn*area*sum((sum(U_norm(
                  loc2glb,:).^2,1).^0.5).*(sum(DZ(loc2glb,:)
                  .^2,1).^0.5))/3;
177           %if max(x.^2+y.^2)>0.8
178           %     eta(K)=0;
179           %end
180       end
181
182       N_list(N)=size(t,2);
183       E_list(N)=(sum(eta));
184       N=N+1;
185
186
187   %Plot epsilon U_n
188    if i == tim_len
189       np = size(p,2);
190       nt = size(t,2);
191       plot = zeros(np); % allocate matrix
192            occurances = zeros(np);
193
194       for K = 1:nt % for each element
195       loc2glb = t(1:3,K); % local-to-global map
196       x = p(1,loc2glb); % node x-coordinates
197       y = p(2,loc2glb); % node y-coordinates
198       [area,b,c] = HatGradients(x,y);
199       plot(loc2glb) = plot(loc2glb)+ eta(K)*[1;1;1]; %
200       occurances(loc2glb(1)) = occurances(loc2glb(1))+1;
201       occurances(loc2glb(2)) = occurances(loc2glb(2))+1;
202       occurances(loc2glb(3)) = occurances(loc2glb(3))+1;
203       end
204       plot=plot./occurances;
205       plt=figure();
206       pdeplot (p ,[],t,'XYData',plot,'ZData',plot,'
             ColorMap','jet','Mesh','on')
207       title("Total error H");
208       saveas(plt,"H_v"+FUNCTION+".png");
```

```
209
210 end
211
212 %
213
214     % adaptive mesh refinement :
215     ind=floor(length(eta)*0.1);
216     % select elements for refinement
217     sorted_eta=sort(eta, 'descend');
218     elements = find(ismember(eta, sorted_eta(1:ind)));
219     % refine elements using regular refinement
220     po=p;to=t;eo=e;
221     [p ,e , t ] = refinemesh (g ,p ,e ,t , elements , '
          regular' );
222 %  figure();
223 % pdemesh(p,e,t)
224     disp(sum(eta));
225
226
227
228 end
229
230 %% Plot
231 %
232 plt=figure();
233 pdemesh(p,e,t);
234 title("Final mesh");
235 saveas(plt,"mesh_v"+FUNCTION+".png");
236
237
238 plt=figure();
239 pdeplot(po,eo,to,'XYData',U,'ZData',U,'ColorMap','jet',
      'Mesh','on');
240 title("Final solution");
241 saveas(plt,"U_v"+FUNCTION+".png");
242
243 plt=figure();
244 pdeplot(po,eo,to,'XYData',sum(U_norm,2),'ZData',sum(
```

```
    U_norm,2),'ColorMap','jet','Mesh','on');
title("Final solution");
saveas(plt,"RU_v"+FUNCTION+".png");

plt=figure();
pdeplot(po,eo,to,'XYData',Z,'ZData',Z,'ColorMap','jet',
    'Mesh','on');
title("Final z value");
saveas(plt,"z_v"+FUNCTION+".png");

plt=figure();
pdeplot(po,eo,to,'XYData',sum(DZ,2),'ZData',sum(DZ,2),'
    ColorMap','jet','Mesh','on');
title("Final Dz value");
saveas(plt,"Dz_v"+FUNCTION+".png");

plt = figure();
loglog (1./ N_list(2:N-1) , E_list(2:N-1) , 1./ N_list
    (2:N-1) , 1./ sqrt ( N_list(2:N-1) ))
title("Loglog plot of refinement data");
saveas(plt,"NE_v"+FUNCTION+".png");


end


%% Assembler functions based on book (The Finite
    Element Method:Theory, Implementation, and
    Applications %%%%%%%%%%%%

%Gradients
function [area,b,c] = HatGradients(x,y)
area=polyarea(x,y);
b=[y(2)-y(3); y(3)-y(1); y(1)-y(2)]/2/area;
c=[x(3)-x(2); x(1)-x(3); x(2)-x(1)]/2/area;
end

% derivative f
```

```
277 function df = ddf(x1, x2)
278     df = 2 * pi * [-x2, x1];
279 end
280
281 %initial
282 function init=initial_data(x,y)
283     init=[];
284     for i=1:length(x)
285         if sqrt(x(i).^2 +y(i).^2)<=1
286             init(i)=14*pi/4;
287         else
288             init(i)=pi/4;
289         end
290     end
291 end
292
293
294 % Convection Matrix Assembler
295 function C = ConvectionAssembler2D(p,t,bx,by)
296     np=size(p,2);
297     nt=size(t,2);
298     C=sparse(np,np);
299     for i=1:nt
300         loc2glb=t(1:3,i);
301         x=p(1,loc2glb);
302         y=p(2,loc2glb);
303         [area,b,c]=HatGradients(x,y);
304         bxmid=mean(bx(loc2glb));
305         bymid=mean(by(loc2glb));
306         CK=ones(3,1)*(bxmid*b+bymid*c)'*area/3;
307         C(loc2glb,loc2glb)=C(loc2glb,loc2glb)+CK;
308     end
309 end
310
311 %Mass Matrix Assembler
312 function M = MassAssembler2D(p,t)
313     np = size(p,2); % number of nodes
314     nt = size(t,2); % number of elements
```

```
315     M = sparse(np,np); % allocate mass matrix
316     for K = 1:nt % loop over elements
317         loc2glb = t(1:3,K); % local-to-global map
318         x = p(1,loc2glb); % node x-coordinates
319         y = p(2,loc2glb); % y
320         area = polyarea(x,y); % triangle area
321         MK = [2, 1, 1; 1, 2, 1; 1, 1, 2]/12*area; %
                element mass matrix
322         M(loc2glb,loc2glb) = M(loc2glb,loc2glb)+ MK; %
                add element masses to M
323     end
324 end
325
326 %Stiffness Assembler
327 function A = StiffnessAssembler2D(p,t,a)
328 np = size(p,2);
329 nt = size(t,2);
330 A = sparse(np,np); % allocate stiffness matrix
331 for K = 1:nt
332 loc2glb = t(1:3,K); % local-to-global map
333 x = p(1,loc2glb); % node x-coordinates
334 y = p(2,loc2glb); % node y-
335 [area,b,c] = HatGradients(x,y);
336 xc = mean(x); yc = mean(y); % element centroid
337 AK = a*(b*b'+c*c')*area; % element stiffness matrix
338 A(loc2glb,loc2glb) = A(loc2glb,loc2glb)+ AK; % add
      element stiffnesses to A
339 end
340 end
341
342 % SC matrix assembly
343 function R = ResidualAssembler(p,t, bk,hk, U, RU, C_vel
      ,C_RV)
344 np = size(p,2);
345 nt = size(t,2);
346 R = sparse(np,np); % allocate matrix
347
348 for K = 1:nt % for each element
```

```
349 loc2glb = t(1:3,K); % local-to-global map
350 x = p(1,loc2glb); % node x-coordinates
351 y = p(2,loc2glb); % node y-coordinates
352 [area,b,c] = HatGradients(x,y);
353
354 h_k=hk(K);
355 b_k=bk(K);
356 U_norm=max(abs(U-mean(U)));
357 Rk=max(abs(RU(loc2glb)));
358
359 e=min([C_vel*h_k*b_k,C_RV*h_k^2*Rk/U_norm]);
360 AK = e*(b*b'+c*c')*area; % element stiffness matrix
361 R(loc2glb,loc2glb) = R(loc2glb,loc2glb)+ AK; % add
      element stiffnesses to A
362 end
363 end
```