POLITECNICO DI TORINO

Master Degree
in Mathematical Engineering

Master Thesis

Solution of Second Order Elliptic Equations on Polygonal/Polyhedral Meshes by the Virtual Element Method with Applications in Geosciences

Supervisor
Prof. Stefano Berrone

Candidate
Federica Milinanni

Co-supervisor
Prof. Marco Verani

Academic Year 2019-2020

Summary

The Reaction-Convection-Diffusion equation is a second order elliptic partial differential equation that can be used to model many different physical phenomena. A method to solve this class of problems is the Virtual Element Method (VEM).

In this thesis we focus on two dimensional problems defined on polygonal meshes, considering both the stationary and evolutive case. The aim of this thesis is to analyze and implement the stabilization methods Streamline Upwind Petrov-Galerkin (SUPG) and Mass Lumping in the particular case of Virtual Element space of order $k = 1$.

Numerical results show the positive stabilization effect of SUPG and Mass Lumping when the problem is characterized respectively by very large Péclet and very low Karlowitz numbers. Moreover, an error analysis on an easy stationary problem shows that the stabilization methods preserve the rate of convergence of VEM. Finally, a simulation of a realistic geophysical evolutive problem is carried out to show the performance of the method on a domain characterized by a high geometrical complexity.

Acknowledgements

I would like to thank
Prof. Stefano Berrone and Dr. Andrea Borio
for their continuous and valuable support
and the passion they put and transmit through their work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many physical phenomena are mathematically described through partial differential equations, and one of the most used methods to solve numerically these problems is the Finite Element Method (FEM). This method requires the generation of a mesh to discretize the space where the problem is defined. A good approximation of the solution depends on the quality of the mesh, whose generation could be computationally expensive when certain geometrical constraints must be satisfied.

A physical application where the meshing process becomes very costly is the simulation of single phase flows in porous media with an embedded Discrete Fracture Network (DFN). Indeed, these fractures, represented as two-dimensional objects, are usually generated stochastically, and it is common to meet very narrow angles between them. The problem lies in the fact that the quality of a FEM mesh is related to the angles of the discretization of the space, in particular, small angles should be avoided in order to have a good approximation of the solution. To overcome this problem, in recent years the FEM has been generalized in methods such as the Virtual Element Method (VEM), which relaxes some geometrical constraints and allows the use of a general polygonal or polyhedral mesh.

In this thesis we will focus on the solution of a second order elliptic equation in a two-dimensional domain by means of the VEM of order $k = 1$. We will consider, in particular, the Reaction-Convection-Diffusion equation, which can be used to describe a scalar field such as the temperature or the concentration of a pollutant in the fractures of the above mentioned DFN. In this work we will take into account both the stationary and evolutive problems.

As the name suggests, the equation that we are considering presents the three terms of reaction, convection and diffusion. Each of them can be predominant over the others. We are in particular interested in the situations of convection-dominated and reaction-dominated problems, which can be identified respectively from the values of the Péclet and the Karlowitz numbers. In these cases the numerical solution could show oscillations that are not present in the exact solution. Their source does not lie in the model, but it is of a numerical nature instead. This problem can be fixed through a stabilization Streamline Upwind Petrov Galerkin (SUPG) in the convection-dominated case and with Mass Lumping in the reaction-dominated case.

The work is organized as follows: after a description of the stationary and evolutive Reaction-Convection-Diffusion problem in Section 2 and its variational formulation in 3, in Section 4 it is presented the VEM of order $k = 1$ applied to this particular problem. The core of the thesis lies in Section 6, where the stabilization methods SUPG and Mass Lumping are introduced and analyzed, and their implementation in C++ is discussed. To compare their behavior to the non-stabilized numerical method, in Section 7.2 we consider a simple DFN composed by two fractures, where we ran simulations on a stationary and an evolutive problem depending on three different parameters: a reaction coefficient, a transmissivity coefficient and a diffusion coefficient. We show the results of numerical simulations carried out for different combinations of the parameters, in order to isolate the cases of convection-dominated and reaction-dominated problems. In so doing, it is possible to determine when it is appropriate to apply a certain stabilization method. Besides, in this section we show through an a posteriori error analysis that the order of convergence of non-stabilized VEM is preserved also when applying SUPG and Mass Lumping. Finally, a realistic geophysical simulation on a large scale DFN is presented in Section 7.3 to show the applicability of the method on a practical problem characterized by a high geometrical complexity.

Chapter 2

Reaction-Convection-Diffusion Equation

In several scientific fields, such as chemistry, fluid dynamics and biology, the behavior of some scalar physical quantities can be modeled through the Reaction-Convection-Difffusion equation

$$\frac{\partial u}{\partial t} - \nabla \cdot (\epsilon \nabla u) + \nabla \cdot (\beta u) + \sigma u = f$$

The scalar physical quantity $u : [0, T] \times \Omega \to \mathbb{R}^d$ can describe, for example, the temperature field of a fluid at a certain point $x \in \Omega$ of the space, at a certain time $t \in [0, T]$. Each term in the equation refers to a particular aspect of the behavior of the field.

- The evolutive term $\frac{\partial u}{\partial t}$ models the evolution in time of the scalar field.
- The diffusive term $-\nabla \cdot (\epsilon \nabla u)$ describes the spread of the examined quantity in the medium with a diffusivity given by the smooth function $\epsilon : \Omega \to \mathbb{R}$, with $\epsilon(x) \geq \epsilon_0 > 0, \quad \forall x \in \Omega$.
- The convective term $\nabla \cdot (\beta u)$ represents the advection of the scalar quantity in the domain Ω due to the smooth convective field of the medium: $\beta : \Omega \to \mathbb{R}^d$. In many applications, for example when we are modeling a physical quantity in a fluid flow, the vector field β is divergence-free (the flow is incompressible). Mathematically $\nabla \cdot \beta = 0$, therefore $\nabla \cdot (\beta u) = (\nabla \cdot \beta)u + \beta \cdot \nabla u = \beta \cdot \nabla u$.
- The reaction term σu models the local production of the considered physical quantity. For example, if we are analyzing the concentration of a pollutant in a fluid, chemical reactions can occur and they can induce a change in space and time of the quantity. We consider σ as a smooth function $\sigma : \Omega \to \mathbb{R}$.
- The forcing term $f : \Omega \to \mathbb{R}$ provides energy to the system. It represents, for example, the dispensation of heat to a fluid.

When the term $\frac{\partial u}{\partial t}$ is not considered, then the partial differential equation takes the form

$$-\nabla \cdot (\epsilon \nabla u) + \nabla \cdot (\beta u) + \sigma u = f$$

It represents a stationary problem and the solution does not depend on time. This equation is coupled with boundary conditions, which can be of three different types:

- Dirichlet boundary conditions: the values of the solution are imposed on the Dirichlet boundary $\Gamma_D \subseteq \partial\Omega$
$$u = g_D, \quad \Gamma_D$$
- Neumann boundary conditions: on the portion of the boundary $\Gamma_N \subseteq \partial\Omega$ it is fixed the out-flux of the quantity u
$$\epsilon \frac{\partial u}{\partial \hat{n}} = g_N, \quad \text{on } \Gamma_N$$
- Robin boundary conditions on $\Gamma_R \subseteq \partial\Omega$, which are a weighted combination of the previous two.

In order to guarantee a well-posedness of the problem, boundary conditions (also of different types) have to be imposed on the whole boundary $\partial\Omega$, meaning that $\Gamma_D \cup \Gamma_N \cup \Gamma_R = \partial\Omega$, and $\Gamma_D \cap \Gamma_N = \emptyset$, $\Gamma_D \cap \Gamma_R = \emptyset$, and $\Gamma_N \cap \Gamma_R = \emptyset$.

When, instead, also the temporal term is taken into consideration, then the problem is said to be evolutive. If we study the problem in the temporal interval $[0, T]$, boundary conditions must be specified $\forall t \in [0, T]$, and also an initial conditions has to be imposed:

$$u(0, x) = u_0(x), \quad x \in \Omega$$

The solution of the Reaction-Convection-Diffusion equation could be a complex problem, and in many cases it is also unknown how to solve it analytically. Therefore, numerical methods have been developed in order to compute an approximate solution of the partial differential equation.

Chapter 3

Variational Formulation

3.1 Stationary Problem

One way to solve numerically partial differential equations is to convert the differential problem into a system of linear algebraic equations, which is a mathematical problem that we know how to solve efficiently. We can perform this conversion by means of the variational formulation of the differential problem.

We firstly choose a functional space V in which we want to find a solution $u \in V$ to our problem. If we consider the stationary Reaction-Convection-Diffusion problem with homogeneous Dirichlet boundary conditions in a bounded, convex, polygonal domain $\Omega \subset \mathbb{R}^2$:

$$\begin{cases} \nabla \cdot (-\epsilon(x)\nabla u + \beta(x)u) + \sigma(x)u = f(x) & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

then a natural choice for V is the functional space $H_0^1(\Omega)$. To obtain the weak formulation of the problem, we multiply both sides of the equation with a generic test function $v \in H_0^1(\Omega)$ and integrate on the domain Ω.

By integrating by parts and considering the homogeneous boundary conditions, we get the variational formulation of the problem:

$$\begin{cases} \text{Find } u \in H_0^1(\Omega) \text{ such that} \\ B(u,v) = (f,v) \quad \forall v \in H_0^1(\Omega) \end{cases}$$

where $(\cdot,\cdot)$ represents the $L^2(\Omega)$ scalar product and

$$B(u,v) := a(u,v) + b(u,v) + c(u,v) \tag{3.1}$$

with:

$$a(u,v) := \int_\Omega \epsilon \nabla u \cdot \nabla v dx \tag{3.2}$$

$$b(u,v) := -\int_\Omega u(\beta \cdot \nabla v) dx \tag{3.3}$$

$$c(u,v) := \int_\Omega \sigma u v dx \tag{3.4}$$

By assuming

- $f \in H^{-1}(\Omega)$,
- $\|u\|_{1,\Omega} \leq C\|f\|_{-1,\Omega}$,
- $\|u\|_{2,\Omega} \leq C\|f\|_{0,\Omega}$

for a constant $C > 0$ independent of f, then the bilinear form $B(\cdot,\cdot)$ is continuous and coercive, thus the existence and uniqueness of the solution $u \in H_0^1(\Omega)$ is guaranteed (da Veiga et al. [2014]).

If the problem is not characterized by homogeneous Dirichlet boundary conditions, we can transform the problem into an equivalent one, which shows homogeneous Dirichlet boundary conditions and can be, therefore, solved as described before.

In the case of inhomogeneous Dirichlet boundary conditions, the problem is characterized by

$$u = g_D \quad \text{on } \Gamma_D$$

with $g_D \in H^{\frac{1}{2}}(\Gamma_D)$.

We define $\mathcal{R}_{g_D} \in H^1(\Omega)$ as the function in $H^1(\Omega)$ such that $\gamma_{\Gamma_D}(\mathcal{R}_{g_D}) = g_D$ and $\gamma_{\partial\Omega\setminus\Gamma_D}(\mathcal{R}_{g_D}) = 0$, where $\gamma_{\partial\Omega} : H^1(\Omega) \to H^{\frac{1}{2}}(\partial\Omega)$ represents the trace operator.
If we let

$$u = u_0 + \mathcal{R}_{g_D} \tag{3.5}$$

We can convert the problem to

$$\begin{cases} \nabla \cdot (-\epsilon \nabla u + \beta u) + \sigma u = f, & \text{in } \Omega \\ u_0 = 0, & \text{on } \partial\Omega \end{cases}$$

that can be rearranged as

$$\begin{cases} \nabla \cdot (-\epsilon \nabla u_0 + \beta u_0) + \sigma u_0 = f - \nabla \cdot (-\epsilon \nabla \mathcal{R}_{g_D} + \beta \mathcal{R}_{g_D}) - \sigma \mathcal{R}_{g_D}, & \text{in } \Omega \\ u_0 = 0, & \text{on } \partial\Omega \end{cases}$$

If it is possible to find a function $\mathcal{R}_{g_D}$ with these properties, then we can solve the problem in u_0, which is characterized by homogeneous Dirichlet boundary conditions, and it has the following variational formulation:

$$\begin{cases} \text{Find } u_0 \in H_0^1(\Omega) \text{ such that} \\ B(u_0, v) = (f, v) - B(\mathcal{R}_{g_D}, v) \quad \forall v \in H_0^1(\Omega) \end{cases}$$

After solving this problem, we reconstruct the solution to the original problem recalling (3.5).

In the case of Neumann conditions

$$\epsilon \frac{\partial u}{\partial \hat{n}} = g_N, \quad \text{on } \Gamma_N$$

when we integrate by parts to get the variational formulation of the problem, we obtain new terms due to the boundary condition:

$$\begin{cases} \text{Find } u \in H_0^1(\Omega) \text{ such that} \\ B(u,v) = (f,v) + (g_N, \gamma_{\Gamma_N}(v))_{\Gamma_N} \quad \forall v \in H_0^1(\Omega) \end{cases}$$

Similarly, Robin boundary conditions are performed.

3.2 Evolutive Problem

In a similar way we can approach the evolutive problem of Reaction-Convection-Diffusion, whose equation is given by

$$\frac{\partial u}{\partial t} - \nabla \cdot (\epsilon \nabla u) + \nabla \cdot (\beta u) + \sigma u = f$$

Also in this case, we can reformulate a problem with generic boundary conditions (homogeneous or inhomogeneous Dirichlet, Neumann or Robin boundary conditions) in a problem with homogeneous Dirichlet conditions. Therefore, we can reduce our analysis to the problem:

$$\begin{cases} \frac{\partial u}{\partial t} - \nabla \cdot (\epsilon \nabla u) + \nabla \cdot (\beta u) + \sigma u = f, & \text{on } [0,T] \times \Omega \\ u_D(t,x) = 0, & \text{on } [0,T] \times \partial\Omega \\ u(0,x) = u_0(x), & \text{on } \Omega \end{cases}$$

Given the homogeneous Dirichlet boundary conditions, we can look for solutions in the space $L^2([0,T], H_0^1(\Omega))$. Besides, for the problem to be well-defined, we assume

$$f, \frac{\partial u}{\partial t} \in L^2\left([0,T], H^{-1}(\Omega)\right)$$

We can solve the problem through the Crank-Nicolson method - a finite difference method that shows a second-order convergence in time.
Following this method, we firstly discretize the temporal domain $[0,T]$ into N intervals of size $\Delta t = \frac{T}{N}$. Thus, each time step is given by $t_k = k\Delta t, \quad k = 0, \dots, N$. We denote with u^k and f^k respectively the solution and the forcing term at time step t_k. We discretize the evolutive term as

$$\frac{u^{k+1} - u^k}{\Delta t}$$

and we average the spatial terms and the forcing term in the time steps t_k and t_{k+1}:

$$\begin{aligned} &\frac{u^{k+1} - u^k}{\Delta t} + \\ &+ \frac{1}{2}\left(\left(-\nabla \cdot (\epsilon \nabla u^k) + \nabla \cdot (\beta u^k) + \sigma u^k\right) + \left(-\nabla \cdot (\epsilon \nabla u^{k+1}) + \nabla \cdot (\beta u^{k+1}) + \sigma u^{k+1}\right)\right) = \\ &= \frac{1}{2}\left(f^k + f^{k+1}\right) \end{aligned}$$

We can now multiply the equation by a generic test function $v \in H_0^1(\Omega)$ and integrate on the spatial domain Ω. Considering the homogeneous Dirichlet boundary conditions, we integrate by parts and we obtain the variational formulation of the evolutive problem:

$$\begin{cases} \text{Find } u \in L^2\left([0,T], H_0^1(\Omega)\right) \text{ such that} \\ \frac{u^{k+1}-u^k}{\Delta t} + \frac{1}{2}\left(B(u^k, v) + B(u^{k+1}, v)\right) = \frac{1}{2}\left((f^k, v) + (f^{k+1}, v)\right), \quad \forall v \in H_0^1(\Omega) \end{cases} \tag{3.6}$$

To solve this problem numerically, we will consider the initial condition u_0 and recursively we will determine u^{k+1} given u^k. It is therefore useful to rearrange (3.6) as

$$\begin{cases} \text{Find } u \in L^2\left([0,T], H_0^1(\Omega)\right) \text{ such that} \\ (u^{k+1}, v) + \frac{\Delta t}{2} B(u^{k+1}, v) = (u^k, v) - \frac{\Delta t}{2} B(u^k, v) + \frac{\Delta t}{2}\left((f^k, v) + (f^{k+1}, v)\right), \quad \forall v \in H_0^1(\Omega) \end{cases} \tag{3.7}$$

Chapter 4

Virtual Element Method

4.1 Virtual Element Space

The Virtual Element Method is an improvement to the Finite Element Method, and it is slightly more expensive than FEM, because basis functions are not known analytically and it is required to solve local problems in order to compute their local polynomial projections. Nevertheless, VEM shows the advantage of requiring a polygonal tessellation $\mathcal{T}_h$ of the domain $\Omega \subset \mathbb{R}^2$ with polygons characterized in general by a different number of edges from each other. As a consequence, the process of mesh generation is simplified and it requires a lower computational cost.

More specifically, in the VEM we approximate the solution in the VEM space, that will be defined later. These functions will not be computed exactly, nevertheless we are interested in knowing the exact values of the bilinear forms when one of their entries is a polynomial. Therefore, we will make extensive use of projections of VEM functions onto the space of piecewise polynomials of order k.

As described in da Veiga et al. [2014], the elements $E \in \mathcal{T}_h$ in the VEM have to satisfy the following conditions:

- $\forall E \in \mathcal{T}_h$, E has to be star-shaped with respect to a disk of radius $\rho_E h_E$, where h_E is the diameter of the element E, and it has to exist a $\rho_0 > 0$ such that $\rho_E \geq \rho_0 > 0$, $\forall E \in \mathcal{T}_h$

- $\forall E \in \mathcal{T}_h$, $\forall e \in \mathcal{E}_{h,E}$ has to satisfy $|e| \geq \rho_E h_E$, where $\mathcal{E}_{h,E}$ represents the set of edges of element E

Besides, the tessellation $\mathcal{T}_h$ of the DFN has to be a global conforming polygonal mesh, meaning that edges of elements have to match exactly in correspondence of traces (Berrone et al. [2019]).

On each element $E \in \mathcal{T}_h$ we define the local VEM space of order $k \in \mathbb{N}$ as

$$\begin{aligned}\mathcal{V}_h^E = \{ & v \in H^1(E) : \ \Delta v \in \mathbb{P}_k(E), \\ & v|_e \in \mathbb{P}_k(e) \quad \forall e \in \mathcal{E}_{h,E}, \\ & v \in C^0(\partial E), \\ & (v,p)_E = (\Pi^\nabla_{k,E}(v), p)_E \quad \forall p \in \mathbb{P}_k \backslash \mathbb{P}_{k-2} \}\end{aligned}$$

where $\Pi^\nabla_{k,E} : H^1(E) \to \mathbb{P}_k(E)$ is the $H^1(E)$ orthogonal projection operator, which satisfies:

$$(\nabla \Pi^\nabla_{k,E}(v), \nabla p)_E = (\nabla v, \nabla p)_E \qquad \forall p \in \mathbb{P}_k(E), \tag{4.1}$$

$$(\Pi^\nabla_{k,E}(v), 1)_{\partial E} = (v, 1)_{\partial E} \qquad \text{if } k = 1, \tag{4.2}$$

$$(\Pi^\nabla_{k,E}(v), 1)_E = (v, 1)_E \qquad \text{if } k > 1 \tag{4.3}$$

$\mathbb{P}_k(E)$ and $\mathbb{P}_k(e)$ are the space of polynomials of order k defined respectively on element E and on each edge e, and $\mathbb{P}_k \backslash \mathbb{P}_{k-2}(E)$ is the set of polynomials $p \in \mathbb{P}_k(E)$ that are $L^2(E)$ orthogonal to $\mathbb{P}_{k-2}(E)$.
It will also be useful the definition of the L^2-projection operator $\Pi^0_{k,E}$ on the space $\mathbb{P}_k(E)$. It acts on functions $v \in \mathcal{V}_h$ as follows:

$$(\Pi^0_{k,E} p, p)_E = (v, p)_E, \quad \forall p_k \in \mathbb{P}_k$$

The VEM space $\mathcal{V}_h^E$ is finite dimensional, so we can find a finite basis $\{\varphi_i\}$ for it and every function $v \in \mathcal{V}_h^E$ can be identified through its degrees of freedom $\{v_i\}$, that are the scalar values such that $v = \sum_i v_i \varphi_i$. Therefore, if we are looking for an approximation $u_h \in \mathcal{V}_h^E$ of a certain function $u \in H_0^1$, our objective is to find the scalar values $(u_h)_i$ that determine u_h as a function of $\mathcal{V}_h$.
In Section 5 we will describe an implementation of VEM of order $k = 1$. In this case the functions $v \in \mathcal{V}_h^E$ are identified by N^E degrees of freedom represented by the values of v at the N^E vertices of element E.
As basis functions of the space $\mathcal{V}_h^E$ of order $k = 1$ we can consider the Lagrangian basis $\{\varphi_i\}_{i=1}^{N^E}$, which satisfies $\varphi_i(x_j) = \delta_{ij}$, where $x_j, \quad j = 1, \dots, N^E$ denote the vertices of polygon E.
The global virtual element space is then defined as

$$\mathcal{V}_h = \{ v \in H_0^1(\Omega) : \ \ v|_E \in \mathcal{V}_h^E, \ \ \forall E \in \mathcal{T}_h \}$$

(da Veiga et al. [2014]).

4.2 Discrete Problem

To discretize the second order elliptic equation by the VEM, we restrict the problem on one element E of the tessellation $\mathcal{T}_h$ at a time. We will denote with $a^E(\cdot,\cdot)$, $b^E(\cdot,\cdot)$, $c^E(\cdot,\cdot)$ the restriction of the bilinear forms $a(\cdot,\cdot)$, $b(\cdot,\cdot)$, $c(\cdot,\cdot)$ defined in (3.2) - (3.4) to the generic element E.

The discretization requires the introduction of a stabilizing term S^E - a symmetric bilinear form on $\mathcal{V}_h^E \times \mathcal{V}_h^E$, such that

$$\alpha_* a^E(v_h, v_h) \leq S^E(v_h, v_h) \leq \alpha^* a^E(v_h, v_h), \qquad \forall v_h \in \mathcal{V}_h^E, \quad \Pi_k^\nabla v_h = 0$$

with α_*, α^* parameters independent of h and satisfying $0 < \alpha_* \leq \alpha^*$. In words, the symmetric bilinear form S^E scales like $a^E(\cdot,\cdot)$ on the kernel of Π_k^∇ on $\mathcal{V}_h^E$. This stabilization will guarantee the coercicity and continuity of $a_h^E(\cdot,\cdot)$.
The simplest choice for $S^E(u,v)$, that we will use in our implementation of VEM in C++ in 5, is the scalar product of the vectors of degrees of freedom of functions u and v. Thus, in the case $k = 1$ that we are considering, we define $S^E(u,v)$ as the scalar product between the values of u and v in the vertices of element E (Berrone and Borio [2020]).

Now $\forall u, v \in \mathcal{V}_h^E$ we can define the discretized bilinear forms and forcing term on element E as follows:

$$a_h^E(u,v) := \int_E \epsilon(\Pi^0_{k-1,E}\nabla u) \cdot (\Pi^0_{k-1,E}\nabla v)dx + S^E((I - \Pi^\nabla_{k,E})u, (I - \Pi^\nabla_{k,E})v) \tag{4.4}$$

$$b_h^E(u,v) := -\int_E (\Pi^0_{k-1,E}u)(\beta \cdot \Pi^0_{k-1,E}\nabla v)dx \tag{4.5}$$

$$c_h^E(u,v) := \int_E \sigma(\Pi^0_{k-1,E}u)(\Pi^0_{k-1,E}v)dx \tag{4.6}$$

$$(f_h, v)_E := \int_E f(\Pi^0_{k-1,E}v)dx \tag{4.7}$$

As we did in (3.1), we define

$$B_h^E(u,v) := a_h^E(u,v) + b_h^E(u,v) + c_h^E(u,v) \tag{4.8}$$

We can extend the discretized bilinear forms and forcing term to the whole domain Ω, by summing all the contributes on each element $E \in \mathcal{T}_h$:

$$a_h(u,v) := \sum_E a_h^E(u,v)$$
$$b_h(u,v) := \sum_E b_h^E(u,v)$$
$$c_h(u,v) := \sum_E c_h^E(u,v)$$
$$(f_h, v) := \sum_E (f_h, v)_E$$

and similarly

$$B_h(u,v) := a_h(u,v) + b_h(u,v) + c_h(u,v) = \sum_E B_h^E(u,v)$$

The bilinear forms b_h^E and c_h^E are well defined $\forall u, v \in H^1(E)$, as well as b_h and c_h are on $H_0^1(\Omega)$. However, this is not the case for a_h^E, because of the stabilizing term $S^E(\cdot,\cdot)$. Indeed, the latter is defined only on the space $\mathcal{V}_h^E$.

These definitions allow to define the discrete stationary problem as:

$$\begin{cases} \text{Find } u_h \in \mathcal{V}_h \text{ such that} \\ B_h(u_h, v_h) = (f_h, v_h) \quad \forall v_h \in \mathcal{V}_h \end{cases} \tag{4.9}$$

(da Veiga et al. [2014]).

In the evolutive case, as described in Section 3.2, we apply the Crank-Nicolson method. Using the previous discretized bilinear forms and forcing term, at each time step t_k, $k = 1, \dots, N$ we find the discrete version of (3.7):

$$\begin{cases} \text{Find } u^{k+1}_{\Delta t,h} \in \mathcal{V}_h \text{ such that} \\ (u^{k+1}_{\Delta t,h}, v_h) + \frac{\Delta t}{2} B(u^{k+1}_{\Delta t,h}, v_h) = \\ \quad = (u^k, v) - \frac{\Delta t}{2} B(u^k_{\Delta t,h}, v_h) + \frac{\Delta t}{2} \left((f^k, v_h) + (f^{k+1}, v_h) \right), \quad \forall v_h \in \mathcal{V}_h \end{cases} \tag{4.10}$$

4.3 Error Estimates

The VEM provides an approximate solution to the problem we are considering. It is important to understand how good is this approximation. A way to perform it, consists in finding an upper bound of the approximation error $\|u - u_h\|$ in a suitable norm $\|\cdot\|$ (being u and u_h the exact and numerical solutions respectively). This bound usually depends on the discretization parameters: h for the discretization in space (which could be defined as $h := \max_{E \in \mathcal{T}_h} h_E$, with $h_E := diam(E)$) and Δt when discretizing in time. In particular, we look for a relation of this kind:

$$\|u - u_h\| \leq C(h^a + \Delta t^b)$$

for constants $C > 0$ and a, b independent of the parameters h and Δt.

We will say that the method is of order a in space and b in time.

In the stationary problem, since we are dealing with sufficiently regular functions that lie in $H^1(\Omega)$, then a natural choice for the approximation error is the H^1-norm of the difference between exact and approximate solution:

$$\|u - u_h\|_1 = \left(\|u - u_h\|_0^2 + \|\nabla u - \nabla u_h\|_0^2 \right)^{\frac{1}{2}}$$

where $\|\cdot\|_0$ denotes the L^2-norm on Ω.
As proved in da Veiga et al. [2014], for h sufficiently small there exists a unique solution $u_h \in \mathcal{V}_h$ to the discrete problem 4.9, and if $u \in H^{s+1}(\Omega)$ and $f \in H^s(\Omega)$, u_h satisfies

$$\|u - u_h\|_1 \leq C h^s (\|u\|_{s+1} + |f|_s) \tag{4.11}$$

for $0 \leq s \leq k$, for a constant $C > 0$ independent of discretization parameter h, but in general depending on the parameters of the problem : ϵ, β, σ.

As before, k represents the order of the VEM space. If we let $k = 1$, then the error estimate (4.11) becomes:

$$\|u - u_h\|_1 \leq C h (\|u\|_2 + |f|_1) \tag{4.12}$$

When we are solving the evolutive problem, instead, we are looking for a function that approximates the exact solution $u \in L^2([0,T], H_0^1(\Omega))$. In this case the error is a combination of the errors due to the discretization in time and the discretization in space. If we are using the Crank-Nicolson method, which is second order in time, we have the following a priori error estimate:

$$\|u - u_{\Delta t,h}\|_{L^2([0,T],H_0^1(\Omega))} \leq C_1 \Delta t^2 + C_2 h \tag{4.13}$$

Chapter 5

Implementation in C++

To perform the VEM, we implemented a code in C++, following the object oriented programming principles.

5.1 Definition of the Problem

In the `main.cpp` file, after importing a DFN,

```
exitCode = DFNCustomImporter::Import(dfnTag, network);
```

we create a mesh of cells on it

```
DFNMinimalMesher mesher;
mesher.SetDFN(&network);
mesher.CreateMesh();
```

Then, we define and solve the primal problem that physically corresponds to the hydraulic head. It is characterized by the equation

$$-\nabla \cdot (K \nabla u) = f$$

Thus, it is required to import the data about the parameter of transmissivity K and the source term f on each fracture of the DFN. These are defined as vectors of `GenericPhysicalParameter*`. This class presents several subclasses, basing on the behavior of the parameters that we are dealing with:

- if the parameter is constant on the whole fracture, we define it as an object of the subclass `ConstantPhysicalParameter`;
- if its values are constant on each cell of the fracture, we can use the subclass `PiecewiseConstantPhysicalParameter`;

- if, instead, the parameter is defined as a function, we work with the subclass `VariablePhysicalParameter`, where we set the pointer to the function that defines the parameter on the fracture.

Similarly, for boundary conditions we created the class `GenericBoundaryCondition`, that admits the subclasses `ConstantBoundaryCondition` (for which we define a constant value of boundary condition on the whole border) and `VariableBoundaryCondition` (that requires the specification of a function pointer that expresses pointwise the value of the boundary condition). These classes can be used for all kinds of boundary conditions: basing on the value of a marker, we distinguish the case of Dirichlet and Neumann conditions. After defining the parameters, we declare the problem as an object of class `EigenDFNVemEllipticProblem`, and by means of setter functions we set DFN, parameters, source function and boundary conditions.

5.2 Resolution of the problem

In the function `EigenDFNVemEllipticProblem::Initialize()`, we initialize the problem. In particular, we define the discrete equation

```
EigenDiscreteEquation_IterativeSolver* discreteEquation = new
    EigenDiscreteEquation_IterativeSolver();
```

and the assembler that will build the system of linear equations

```
assembler = new EigenDFNAssembler();
```

For both objects we set a pointer to the global matrix and the right hand side that define the linear system associated to the problem, along with the matrix for the Dirichlet boundary conditions:

- the matrix is defined as an `Eigen::SparseMatrix<double>` and its size is $N_{dof} \times N_{dof}$ (where with N_{dof} we denote the number of degrees of freedom, meaning the number of nodes where no Dirichlet boundary conditions are imposed);
- the right hand side is a vector of `double` of size N_{dof}
- the Dirichlet matrix is an `Eigen::SparseMatrix<double>` of size $N_{dof} \times N_{dof}^D$, where N_{dof}^D denotes the number of nodes where Dirichlet conditions are specified.

The problem is solved in the function

```
Output::ExitCodes DFNVemEllipticProblem::Solve()
```

The resolution happens in two main steps. Firstly the system of linear equations is assembled through the assembler we defined before:

```
assembler->AssembleDiscreteSystem();
```

Secondly, the system of equations is solved iteratively:

```
system->Solve();
```

In particular, when the matrix of the system is symmetric - as in the hydraulic head problem - the iterative method that is used is the Conjugate Gradient method. Whereas, when more generally the system is non-symmetric - like in the Reaction-Convection-Diffusion problem - the Bi-Conjugate Gradient method is performed.
The solution of the linear system of equations will be a vector which components represent the approximated value of the unknown function in the nodes of the mesh.

5.2.1 Discrete Linear System Assembler

The algorithm in the function `AssembleDiscreteSystem()` aims to build the discrete system of equations: the global matrix is built after the computation of

```
vector< Triplet<double> > tripletList;
```

as well as the Dirichlet matrix from

```
vector< Triplet<double> > tripletListDirichlet;
```

The values of these objects, along with the right-hand-side and the Dirichlet values in the Dirichlet nodes of the mesh, are computed locally by considering an element (cell) of the mesh at a time. In particular, we perform a loop on each fracture of the DFN, and for each fracture we run another loop on the cells of the mesh:

```
for(unsigned int fracPosition = 0; fracPosition < network.
   NumberDomains(); fracPosition++)
{
    const Fracture& fracture = dynamic_cast<const Fracture&>(
   network.DomainByPosition(fracPosition));
    const GenericMesh& mesh = fracture.Mesh();
    for(unsigned int e = 0; e < mesh.NumberOfCells(); e++)
    {
            const GenericCell& cell = *mesh.Cell(e);
```

```
...
```

On each element we define a local matrix, a local right-hand-side and a local vector of Dirichlet conditions:

```
MatrixXd cellStiffness;
VectorXd cellRightHandSide;
VectorXd cellDirichletTermValues;
```

If the cell is active, then we compute the values of these matrices and vectors through the function of the equation on the considered fracture:

```
equationPointer[fracPosition]->BuildLocalSystem(cell,
   cellStiffness, cellRightHandSide, cellDirichletTermValues)
   ;
```

After this computation, that will be described later, these local quantities are plugged into the global objects. We consider ordered pairs of vertices of the cell, and we took their global indices inside the mesh. We denote this two values as `globalDof_i` and `globalDof_j`.
To distinguish Dirichlet degrees of freedom, we decided to let the indices of Dirichlet nodes to be negative. So

```
if(globalDof_i >= 0)
```

and

```
if(globalDof_j >= 0)
```

we plug the computed value of the stiffness matrix into the `tripletList`:

```
tripletList.push_back(Triplet<double>(globalDof_i,
   globalDof_j, cellStiffness(i,j)));
```

Instead,

```
if(globalDof_i >= 0)
```

and

```
if(globalDof_j < 0)
```

then we are dealing with a Dirichlet condition in node `j`, therefore we let

```
tripletListDirichlet[posDirichlet++] = Triplet<double>(
   globalDof_i, -globalDof_j-1, cellStiffness(i,j));
```

and we set the Dirichlet value

```
solutionDirichlet[-globalDof_j-1] = cellDirichletTermValues(j
   );
```

The right-hand-side is determined as

```
rightHandSide[globalDof_i] += cellRightHandSide(i);
```

when `globalDof_i>=0`, that corresponds to the case when we have no Dirichlet condition on node `i`.

5.2.2 Computation of the Local System

As anticipated before, the local values are computed in the function

```
Output::ExitCodes VemEllipticEquation::BuildLocalSystem(
   const GenericCell& Gcell, MatrixXd& cellMatrix, VectorXd&
   cellRightHandSide, VectorXd& cellDirichletTermValues)
```

Here through the line of code

```
vemValues.ComputeVemProjectors(cell);
```

we compute the Vandermonde matrix and the matrices that perform the projections. Their computation follows the optimized implementation described in Berrone and Borio [2020].

Vandermonde Matrices

The Vandermonde matrix of element E for the VEM of order 1 is a matrix $V_1^E \in \mathbb{R}^{N^E \times 3}$. Its entries (i, j) are given by $m_j(x_i)$, $i = 1, \ldots, N^E$, $j = 1,2,3$, where m_j are the scaled monomials taken as basis for the local polynomial space $\mathbb{P}_1(E)$:

$$m_1(x, y) = 1 \tag{5.1}$$

$$m_2(x, y) = \frac{x - x_E}{h_E} \tag{5.2}$$

$$m_3(x, y) = \frac{y - y_E}{h_E} \tag{5.3}$$

Here we denoted by (x_E, y_E) the centroid of the element and by h_E its diameter.
We will also use the Vandermonde matrix of order 0, which has size $N^E \times 1$ and all its entries are 1 (indeed, as basis of $\mathbb{P}_0(E)$ we can take the set $\{m\}$, with $m(x, y) = 1$), and the Vandermonde matrices of derivatives of monomials $V_{1,x}^E, V_{1,y}^E \in \mathbb{R}^{N^E \times 3}$.
The latter are defined as

$$V_{1,x}^E = V_1^E D_{1,x}^E, \qquad V_{1,y}^E = V_1^E D_{1,y}^E$$

where

$$D_{1,x}^E = \begin{pmatrix} 0 & \frac{1}{h_E} & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \qquad D_{1,y}^E = \begin{pmatrix} 0 & 0 & \frac{1}{h_E} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Useful to our computations are also the projectors Π_1^∇, Π_0^0 and $\Pi_0^0\nabla$.

Projector Π_1^∇

The projector Π_1^∇ is the H^1-orthogonal projection defined by:

$$\begin{aligned} &(\nabla\Pi_1^\nabla(v), \nabla p)_E = (\nabla v, \nabla p)_E, && \forall p \in \mathbb{P}_1(E) \\ &(\Pi_1^\nabla(v),1)_{\partial E} = (v,1)_{\partial E} \end{aligned}$$

If we are considering the scaled monomials (5.1) - (5.3) as basis for $\mathbb{P}_1(E)$ and the Lagrangian basis for $\mathcal{V}_\delta^E$, then the vector $(\mathrm{dof}_{\mathbb{P}_k(E)}(\Pi_1^\nabla v))_j$ of degrees of freedom of $\Pi_1^\nabla v$ with respect to $\mathbb{P}_k(E)$ satisfies:

$$\sum_{j=1}^{3} (\nabla m_j, \nabla m_i)_E (\mathrm{dof}_{\mathbb{P}_k(E)}(\Pi_1^\nabla(v)))_j = (\nabla v, \nabla m_i)_E, \quad \forall j = 1, 2, 3$$

In matrix form, this is equivalent to

$$G_1^E \Pi_1^\nabla = B_1^E$$

where

- $\Pi_1^\nabla \in \mathbb{R}^{3 \times N^E}$;
- $G_1^E = \tilde{G_1^E} + \begin{pmatrix} (w^{\partial E})^T V_1^{\partial E} \\ 0 \end{pmatrix}$
  with $0 \in \mathbb{R}^{2\times 3}$ and
  $$(\tilde{G_1^E})_{ij} = (\nabla m_i, \nabla m_j)_E$$
  and it is performed matricially through
  $$\tilde{G_1^E} = (V_{1,x}^E)^T W^E V_{1,x}^E + (V_{1,y}^E)^T W^E V_{1,y}^E$$
  being W^E the diagonal matrix whose diagonal elements are the quadrature weights relative to the vertices of element E;

- $B_1^E = \tilde{B_1^E} + \begin{pmatrix} (w^{\partial E})^T \\ 0 \end{pmatrix}$
  with $0 \in \mathbb{R}^{2\times 3}$ and

$$(\tilde{B_1^E})_{ij} = (w^{\partial E})_j \frac{\partial m_i}{\partial \hat{n}}(x_j), \qquad j = 1, \dots, N^E$$

  given by

$$(\tilde{B_1^E})_{ij} = (V_{1,x}^{\partial E})^T W_x^{\partial E} + (V_{1,y}^{\partial E})^T W_y^{\partial E}$$

  with W_x and W_y defined as the diagonal matrices whose elements are the values of the quadrature weights along x and y, obtained by weighting them according to the x and y components of the vectors normal to the edges of element E.

In the code this is performed through

```
MatrixXd cellG = vanderInternalDerivatives[0].transpose()*
   cellInternalWeights.asDiagonal()*vanderInternalDerivatives
   [0] + vanderInternalDerivatives[1].transpose()*
   cellInternalWeights.asDiagonal()*vanderInternalDerivatives
   [1], cellB(numMonomials, numDofs);

cellB.leftCols(numBoundaryDofs) = vanderBoundaryDerivatives
   [0].transpose()*cellBoundaryWeightsNormal[0].asDiagonal()
   + vanderBoundaryDerivatives[1].transpose()*
   cellBoundaryWeightsNormal[1].asDiagonal();

MatrixXd cellD(numDofs, numMonomials);

cellD.topRows(numBoundaryDofs) = vanderBoundary;

cellB.row(0) = cellBoundaryWeights;

cellG.row(0) = vanderBoundary.transpose()*cellBoundaryWeights
   ;
```

Now we can compute Π_1^∇ as:

```
piNabla = cellG.partialPivLu().solve(cellB);
```

Projector Π_0^0

The $L^2(E)$-projection $\Pi_0^0 : \mathcal{V}_\delta \to \mathbb{P}_0(E)$ is defined through

$$(\Pi_0^0(v_\delta), p)_E = (v_\delta, p)_E, \quad \forall p \in \mathbb{P}_0(E), \forall v_\delta \in \mathcal{V}_\delta$$

This is matricially equivalent to

$$\Pi_0^0 = (H^* E_0)^{-1} C_0^E$$

where in general $(H_k^E)_{ij} = (m_i, m_j)_E, \quad i,j = 1,\dots,N_k$ ($N_0 = 1,\ N_1 = 3$) is the mass matrix of monomials and it is computed through $H_k^E = (V_k^E)^T W^E V_k^E$, whereas

$$C_0^E = ((H_1^E)_{1,\,1,2,3})\Pi_1^\nabla$$

This latter relation comes from the property of $\mathcal{V}_\delta$: $(\varphi_j,1)_E = (\Pi_1^\nabla(\varphi_j),1)_E$.
In C++ this is implemented in the following lines of code:

```
MatrixXd cellH = vanderInternal.transpose()*
   cellInternalWeights.asDiagonal()*vanderInternal;

cellC.bottomRows(n_km1-numInternalDofs) = cellH.block(
   numInternalDofs,0,n_km1-numInternalDofs,numMonomials)*
   piNabla;

const LLT<MatrixXd>& H_km1_LU =  cellH.topLeftCorner(n_km1,
   n_km1).llt();

projectors["pikm1_0"][0] = H_km1_LU.solve(cellC);
```

Note that, in the code, `km1` corresponds to 0.

Projector $\Pi_0^0\nabla$

Lastly, the projections of the derivatives $\Pi_0^0\frac{\partial}{\partial x},\ \Pi_0^0\frac{\partial}{\partial y} : \mathcal{V}_\delta \to \mathbb{P}_0(E)$ are such that

$$\left(\Pi_0^0\frac{\partial v_\delta}{\partial x}, p\right)_E = \left(\frac{\partial v_\delta}{\partial x}, p\right)_E, \qquad \forall p \in \mathbb{P}_0(E)$$
$$\left(\Pi_0^0\frac{\partial v_\delta}{\partial y}, p\right)_E = \left(\frac{\partial v_\delta}{\partial y}, p\right)_E, \qquad \forall p \in \mathbb{P}_0(E)$$

We will denote

$$\Pi_0^0\nabla v_\delta = \begin{pmatrix}\Pi_0^0\frac{\partial v_\delta}{\partial x}\\ \Pi_0^0\frac{\partial v_\delta}{\partial y}\end{pmatrix}$$

The matrices $\Pi_0^{0,x},\quad \Pi_0^{0,y} \in \mathbb{R}^{1\times N^E}$, corresponding to the projections of the x and y derivatives, are given by

$$\Pi_0^{0,x} = (H_0^E)^{-1} E_{0,x}^E, \qquad \Pi_0^{0,y} = (H_0^E)^{-1} E_{0,y}^E$$

where

$$E_{0,x}^E = (V_0^{\partial E})^T W_x^{\partial E} \qquad E_{0,y}^E = (V_0^{\partial E})^T W_y^{\partial E}$$

This can be implemented in C++ as

```
MatrixXd cellEx(n_km1,numDofs);

cellEx.leftCols(numBoundaryDofs) = vanderBoundary.leftCols(
   n_km1).transpose()*cellBoundaryWeightsNormal[0].asDiagonal
   ();


MatrixXd cellEy(n_km1,numDofs);

cellEy.leftCols(numBoundaryDofs) = vanderBoundary.leftCols(
   n_km1).transpose()*cellBoundaryWeightsNormal[1].asDiagonal
   ();


vector<MatrixXd>& piGradkm1xyz = projectors["piGradkm1xyz"];

piGradkm1xyz[0] = H_km1_LU.solve(cellEx);

piGradkm1xyz[1] = H_km1_LU.solve(cellEy);
```

In `BuildLocalSystem` we define

```
for(unsigned int i = 0; i < dimensionValue; i++)
     vanderpiGradkm1xyz[i] = Vander_km1*vemValues.
   PiGradkm1xyz(i);
```

and

```
MatrixXd vanderPikm1_0 = Vander_km1*vemValues.Pikm1_0();
```

Diffusion term

Now it is possible to compute the diffusion term, recalling (4.4):

$$a_h^E(u,v) := \int_E \epsilon(\Pi^0_{k-1}\nabla u)\cdot(\Pi^0_{k-1}\nabla v)dx + S^E((I-\Pi^\nabla_k)u,(I-\Pi^\nabla_k)v)$$

We first let

```
vector<VectorXd> weightsTimesDiffusion(diffusionTerm.size());
```

whose components are defined as the product between the weights of the vertices of E and the diffusion term evaluated in these points.
Afterwards, we add the component of the diffusion term to the local matrix of the cell:

```
cellMatrix += vanderpiGradkm1xyz[i].transpose()*
   weightsTimesDiffusion[0].asDiagonal()*vanderpiGradkm1xyz[i
   ];
```

To perform the stabilization $S^E((I - \Pi_k^\nabla)u, (I - \Pi_k^\nabla)v)$:

```
MatrixXd IminusPinabla = vemValues.PiNabla_vemdofs();

IminusPinabla.diagonal().array() -= 1.0;

cellMatrix += maxDiffTerm*pow(cell.Diameter(),dimensionValue
   -2)*IminusPinabla.transpose()*IminusPinabla;
```

In the previous code, `maxDiffTerm` denotes the maximum value of the diffusion term in the vertices of E. Besides, in the case we are considering, `dimensionValue` is 2.

Right-hand-side

To compute the right-hand-side, we have to evaluate the forcing term in the vertices of the element, and then we use the projection Π_0^0, recalling (4.7):

```
VectorXd forcingTermValues;

vemValues.EvaluatePhysicalParameterInQuadraturePoints(*
   forcingTerm, forcingTermValues);

cellRightHandSide = vanderPikm1_0.transpose()*
   cellInternalWeights.asDiagonal()*forcingTermValues;
```

Boundary Conditions

Dirichlet conditions are defined in

```
vemValues.ComputeDirichlet(cell, *dirichletConditions,
   cellDirichletTermValues);
```

where the Dirichlet conditions, when present, are evaluated in the vertices of element E. Neumann boundary conditions, instead, provide a contribute to the right-hand-side. In fact, they are originated after the integration by part in the variational formulation, and they bring to the term

$$\int_{\Gamma_N} g_N \gamma_{\Gamma_N}(v) d\Gamma$$

as discussed in Chapter 3.
In the code, this is performed in

```
VectorXd neumannTermValues;

if(vemValues.ComputeNeumann(cell, *neumannConditions,
   neumannTermValues))

        cellRightHandSide.segment(0,numBoundaryDofs) +=
   neumannTermValues;
```

Robin boundary conditions are executed instead in

```
MatrixXd robinMatrix;

if(vemValues.ComputeRobin(cell, *robinConditions, robinMatrix
   ))

    cellMatrix.block(0,0,numBoundaryDofs,numBoundaryDofs)
   += robinMatrix;
```

5.2.3 Reaction-Convection-Diffusion problem

Through the previous algorithm it is possible to solve the diffusive problem for the hydraulic head. From its solution u_H we can define the Darcy velocity of the flow as

$$\beta = -K\nabla u_H \tag{5.4}$$

To compute the Darcy velocity β, we created a specific class, `DarcyVelocity`, in which by means of the projectors we perform

```
const MatrixXd Vander0 = MatrixXd::Ones(points.size(), 1);

MatrixXd VanderPiGrad0 = Vander0*(*PiGrad0);

results = -transmiss(0)*VanderPiGrad0*cellDofs;
```

In the Reaction-Convection-Diffusion problem, the Darcy velocity represents the convective term. The other two terms of reaction and diffusion are defined through the definition of the parameters of diffusion ϵ and reaction σ. Also in this case, as for the hydraulic head problem, they can be defined as `GenericPhysicalParameter` and can be constant, piecewise constant or variable.
Thus, the Reaction-Convection-Diffusion problem is defined in the `main.cpp` file as the problem for the hydraulic head:

```
EigenDFNVemEllipticProblem problem;
```

and the parameters are set in the same way as for the transmissivity terms in the primal problem.
Similarly, the problem is initialized through

```
problem.Initialize();
```

where the equations and the assembler are defined, and it is solved in

```
problem.Solve();
```

In addition to the previous algorithm, here we have to compute also the contribution of the transport and the reaction terms, that are defined by (4.5) and (4.6).

and their contribution to the local matrix is implemented in `VemEllipticEquation::BuildLocalSystem` by

```
for(unsigned int i=0; i<dimensionValue; i++)
{
        cellMatrix += vanderPikm1_0.transpose()*
   cellInternalWeights.cwiseProduct(transportTermValues.col(i
   )).asDiagonal()*vanderpiGradkm1xyz[i];
}
```

and

```
MatrixXd reactionMatrix = vanderPikm1_0.transpose()*(
   cellInternalWeights.cwiseProduct(reactionTermValues)).
   asDiagonal()*vanderPikm1_0;

cellMatrix += reactionMatrix;
```

where `transportTermValues` and `reactionTermValues` contain respectively the evaluation of the Darcy velocity and the reaction term in the vertices of the cell.

Evolutive Problem

The evolutive Reaction-Convection-Diffusion problem is implemented as a modification of the stationary problem. It is defined through a specific class (`EvolutiveProblemParameters`) where, among the others, we have to initialize the same parameters as in the stationary case. We point out that the parameters, the source term and the boundary conditions are defined as before as objects of class `GenericPhysicalParameter` and `GenericBoundaryCondition`, but in order to specify the time dependance of these terms, we defined the subclasses `TimeDependentPhysicalParameter` and `TimeDependentBoundaryCondition`.

In addition to these parameters, we have to specify the number of time steps (`numTimeSteps`), the size of the discretization in time Δt (`timeStepSize`) and the initial conditions.

In our implementation we are performing the Crank-Nicolson method. Starting with a given initial condition u_0, we can compute iteratively the approximate values of the solution u^{k+1} in the degrees of freedom at time step t_{k+1}, $k = 0, \dots,$`numTimeSteps - 1` as the solution of a linear system where the right-hand-side depends on the known forcing term and on the approximate solution at the previous time step k, as described in (4.10).

The bilinear form $B(\cdot, \cdot)$ is built similarly as before and it brings to the definition of the `RCDMatrix`.

If we suppose that the parameters are independent of time, then the `RCDMatrix` that we build is the same $\forall k = 0, \dots,$`numTimeSteps - 1`.

In addition to the code for the stationary problem, here we have to implement the L^2-scalar product

$$(u^{k+1}_{\Delta t,h}, v_h)$$

that corresponds to the application of a `massMatrix` (that before we denoted as H^E_1) to the vectors of degrees of freedom at time $k+1$. Also this matrix does not depend on time. Therefore, in the function

```
\lstinline{EigenDFNVemEvolutiveProblem::Solve()}
```

we are going to solve a linear system of equations characterized always by the same global matrix

```
globalMatrix = massMatrix + timeStepSize/2 * (RCDMatrix);
```

All the components of `globalMatrix` are computed in function

```
EigenDFNAssemblerEvolutiveProblem::AssembleMassRCDMatrices();
```

created on purpose for the evolutive problem. Specifically, in this function the problem is divided in local problems on each cell of the mesh, exactly as in the stationary case, and in function

```
EigenDFNVemEvolutiveEquation::BuildMassRCDMatrices(const
    GenericCell& Gcell, MatrixXd& cellMassMatrix, MatrixXd&
    cellRCDMatrix)
```

we determine the `cellRCDMatrix` in the same way as for the `cellMatrix` in function `VemEllipticEquation::BuildLocalSystem`.
`cellMassMatrix` is set as

```
cellMassMatrix = vanderPikm1_0.transpose()*
    cellInternalWeights.asDiagonal()*vanderPikm1_0;
```

and it also needs to be stabilized:

```
MatrixXd IminusPinabla = vemValues.PiNabla_vemdofs();

IminusPinabla.diagonal().array() -= 1.0;

cellMassMatrix += pow(cell.Diameter(),dimensionValue)*
    IminusPinabla.transpose()*IminusPinabla;
```

Whereas these matrices are independent of the time step, at each t_{k+1} we have to determine the right-hand-side of the evolutive problem. This happens in `EigenDFNAssemblerEvolutiveProblem::AssembleRightHandSide()`. To compute it, we need the solution at previous time step t_k (`eigenPreviousSolution`), the forcing term evaluated in the degrees of freedom at time step t_k (`eigenPreviousForcingTerm`), and the vectors of Dirichlet conditions both at t_k and t_{k+1} (`eigenCurrentSolutionDirichlet` and `eigenPreviousSolutionDirichlet`). The terms corresponding to the previous time step have already been computed and are saved as instances of the object of class `DFNAssembler`. The "current" terms, instead, must be computed. As for the computation of the other matrices and vectors, we set a local problem on each cell of the mesh. In

```
Output::ExitCodes EigenDFNVemEvolutiveEquation::
    BuildForcingTerm(const GenericCell& Gcell, VectorXd&
    cellForcingTerm, VectorXd& cellDirichletTermValues, const
    double& currentTime)
```

we compute the forcing term as

```
MatrixXd vanderPikm1_0 = Vander_km1*vemValues.Pikm1_0();

vemValues.EvaluatePhysicalParameterInQuadraturePoints(*
    forcingTerm, forcingTermValues);
```

```
cellForcingTerm = vanderPikm1_0.transpose()*
    cellInternalWeights.asDiagonal()*forcingTermValues;
```

and the boundary conditions, where defined:

```
vemValues.ComputeDirichlet(cell, *dirichletConditions,
    cellDirichletTermValues);

if(vemValues.ComputeNeumann(cell, *neumannConditions,
    neumannTermValues))
            cellForcingTerm.segment(0,numBoundaryDofs) +=
    neumannTermValues;
```

The `GenericPhysicalParameter` and `GenericBoundaryCondition` are in this case of subclasses `TimeDependentPhysicalParameter` and `TimeDependentBoundaryCondition` respectively. Therefore, we have to set the time t_k through the setter function `SetCurrentTime(currentTime);`.
After determining all these terms, we can set the right-hand-side in `EigenDFNAssemblerEvolutiveProblem::AssembleRightHandSide()` as

```
eigenRightHandSide = (*globalMassMatrixPointer - timeStepSize
    /(2.0) * (*globalRCDMatrixPointer))*eigenPreviousSolution;
...
eigenRightHandSide += timeStepSize/(2.0) * (
    eigenCurrentForcingTerm + eigenPreviousForcingTerm);
...
eigenCompleteRightHandSide -= (*dirichletMassMatrixPointer)*(
    eigenCurrentSolutionDirichlet -
    eigenPreviousSolutionDirichlet);

eigenCompleteRightHandSide -= (*dirichletRCDMatrixPointer) *
    timeStepSize/(2.0) * (eigenCurrentSolutionDirichlet +
    eigenPreviousSolutionDirichlet);
```

Finally, the problem at time step t_{k+1} is solved in `EigenDFNVemEvolutiveProblem::Solve()` through

```
system->Solve();
```

After running this algorithm, we get the vectors of values of the approximate solution u^k in the nodes of the mesh at each time step t_k.

Chapter 6

Stabilization Methods

6.1 SUPG

When we try to solve a Reaction-Convection-Diffusion problem by the VEM in presence of a diffusion term $-\nabla \cdot (\epsilon \nabla u)$ of several orders of magnitude smaller than the convective term $\nabla \cdot (\beta u)$, then oscillations of the numerical solutions may occur. These oscillations do not have a physical meaning and their source is of a numerical nature, rather than being associated to an error in the model.

The occurrence of this anomalous behavior can be related to the Péclet number of the elements $E \in \mathcal{T}_h$, that in the VEM of order $k = 1$ is defined as:

$$Pe_E := \frac{\beta_E h_E}{6\epsilon_E} \tag{6.1}$$

where

$$\begin{aligned} \beta_E &= \sup_{x \in E} \|\beta(x)\|_{\mathbb{R}^2} \\ \epsilon_E &= \inf_{x \in E} \epsilon(x) \\ h_E &= diam(E) \end{aligned}$$

Spurious oscillations occur for high values of Pe_E and to prevent them we can use the Streamline Upwind Petrov-Galerkin (SUPG) stabilization method.

In the context of VEM of order $k = 1$, SUPG is performed by defining the bilinear form

$$B_{SUPG} := a'(\cdot, \cdot) + b(\cdot, \cdot) + c(\cdot, \cdot) \tag{6.2}$$

where $b(\cdot, \cdot)$ and $c(\cdot, \cdot)$ are defined as in (3.3) and (3.4), and $a'(\cdot, \cdot)$ is given by:

$$a'(u, v) := a(u, v) + \sum_{E \in \mathcal{T}_h} \tau_E \int_E (\beta \cdot \nabla u)(\beta \cdot \nabla v) dx$$

with $a(\cdot, \cdot)$ defined as in (3.2) and $\tau_E := \frac{h_E}{2\beta_E} \min\{Pe_E, 1\}$.

Besides, we define

$$F_{SUPG}(v) := (f, v) + \sum_{E \in \mathcal{T}_h} \tau_E \int_E f(\beta \cdot \nabla v) dx \tag{6.3}$$

The discrete counterpart of (6.2) and (6.3) are the discrete bilinear form $B_{SUPG,h} : \mathcal{V}_h \times \mathcal{V}_h \to \mathbb{R}$ and the discrete right-hand-side $F_{SUPG,h} : \mathcal{V}_h \to \mathbb{R}$ defined as:

$$B_{SUPG,h}(u, v) := a_h(u, v) + b_h(u, v) + c_h(u, v)$$

where

$$\begin{aligned} a_h^E(u, v) := &\int_E \epsilon(\Pi_0^0 \nabla u) \cdot (\Pi_0^0 \nabla v) dx + \\ &(\epsilon'_E + \tau_E \beta_E^2) S^E((I - \Pi_1^\nabla)u, (I - \Pi_1^\nabla)v) + \\ &\tau_E \int_E (\beta \cdot \Pi_0^0 \nabla u)(\beta \cdot \Pi_0^0 \nabla v) dx \end{aligned}$$

with $\epsilon'_E = \sup_{x \in E} \epsilon(x)$,

$$\begin{aligned} b_h^E(u, v) &:= -\int_E (\Pi_0^0 u)(\beta \cdot \Pi_0^0 \nabla v) dx \\ c_h^E(u, v) &:= \int_E \sigma(\Pi_0^0 u)(\Pi_0^0 v) dx \\ a_h(u, v) &:= \sum_E a_h^E(u, v) \\ b_h(u, v) &:= \sum_E b_h^E(u, v) \\ c_h(u, v) &:= \sum_E c_h^E(u, v) \end{aligned}$$

and

$$\begin{aligned} F_{SUPG,h}^E(v) &:= \int_E f \Pi_0^0 v dx + \tau_E \int_E f(\beta \cdot \Pi_0^0 \nabla v) \\ F_{SUPG,h}(v) &:= \sum_E F_{SUPG,h}(v) \end{aligned}$$

In Benedetto et al. [2016] it is proven that the order of convergence of VEM is preserved also when introducing the SUPG stabilization. Indeed, assuming sufficient regularity of the data functions, the following relation holds:

$$|||u - u_h||| \leq Ch(\|u\|_2 + \|f\|1)$$

for a constant $C > 0$ depending on the parameters of the problem ϵ and β and independent of the meshsize h, and the norm $|||\cdot|||$ defined as:

$$|||v||| = \left\{ \|\sqrt{\epsilon} \nabla v\|^2 + \sum_{E \in \mathcal{T}_h} \tau_E \|\beta \cdot \nabla v\|_E^2 \right\}^{\frac{1}{2}} \quad \forall v \in H_0^1(\Omega)$$

This norm is equivalent to the H^1-seminorm $|\cdot|_1$, which is a norm on the subspace $H_0^1(\Omega) \subset H^1(\Omega)$. Therefore, the order of convergence of VEM of order $k = 1$ with the introduction of SUPG stabilization remains the same as the non-stabilized method. In particular the error is linear with respect to the meshsize h.

6.1.1 Implementation in C++

SUPG stabilization can be implemented adding the stabilization terms to the `cellMatrix` computed in

```
Output::ExitCodes VemEllipticEquation::BuildLocalSystem(const
    GenericCell& Gcell, MatrixXd& cellMatrix, VectorXd&
   cellRightHandSide, VectorXd& cellDirichletTermValues)
```

in the case of stationary problem, and to `cellRCDMatrix` in

```
Output::ExitCodes EigenDFNVemEvolutiveEquation::
   BuildMassRCDMatrices(const GenericCell& Gcell, MatrixXd&
   cellMassMatrix, MatrixXd& cellRCDMatrix)
```

in the evolutive case.

In both situations we have to compute the cell Péclet number and determine the value of τ_E:

```
Pe = mk*supTransportTerm*cell.Diameter()/(2.0*
   infDiffusionTerm);

if(Pe<1)

        tau = mk*cell.Measure()/(4.0*infDiffusionTerm);

else

      tau = cell.Diameter()/(2.0*supTransportTerm);
```

being `mk` $= \frac{1}{3}$, `infDiffusionTerm` the minimum value of the diffusion parameter ϵ and `supTransportTerm` the maximum value of the Darcy Velocity in the vertices of the element.
Both to `cellMatrix` for the stationary and to `cellRCDMatrix` for the evolutive problem, we add

```
tau*vanderpiGradkm1xyz[i].transpose()*cellInternalWeights.
   cwiseProduct(transportTermValues.col(i)).cwiseProduct(
   transportTermValues.col(j)).asDiagonal()*
   vanderpiGradkm1xyz[j];
```

Besides, we add a new term in the stabilization matrix

```
MatrixXd IminusPinabla = vemValues.PiNabla_vemdofs();
```

```
IminusPinabla.diagonal().array() -= 1.0;

cellMatrix += tau*pow(supTransportTerm,2)*IminusPinabla.
   transpose()*IminusPinabla;
```

6.2 Mass Lumping

Similarly, oscillations can occur in the numerical solution of the partial differential equation when the reactive term $\sigma(x)u$ is dominant with respect to the diffusion term $-\nabla\cdot(\epsilon\nabla u)$. In this case, a stabilization method that can be performed is the Mass Lumping technique. It consists in the diagonalization of the mass matrix. Indeed, the reaction term depends only on the punctual behavior of the solution, and it is independent of the reciprocal position of the points of the mesh. That is why non-diagonal terms, which are introduced through the discretization of the variational problem, have no sense.

In particular, in the Mass Lumping stabilization the mass matrix is substituted with the diagonal matrix, whose elements are the sum of the elements of the respective row in the original mass matrix.

6.2.1 Implementation in C++

Mass Lumping stabilization is performed in

```
Output::ExitCodes VemEllipticEquation::BuildLocalSystem(const
    GenericCell& Gcell, MatrixXd& cellMatrix, VectorXd&
   cellRightHandSide, VectorXd& cellDirichletTermValues)
```

and in

```
Output::ExitCodes EigenDFNVemEvolutiveEquation::
   BuildMassRCDMatrices(const GenericCell& Gcell, MatrixXd&
   cellMassMatrix, MatrixXd& cellRCDMatrix)
```

as

```
reactionMatrix.diagonal().array() = reactionMatrix.rowwise().
   sum();

VectorXd tmp = reactionMatrix.diagonal();

reactionMatrix = tmp.asDiagonal();
```

in the case of very high values of the inverse of the Karlowitz number

```
double invKa = supReactionTerm*cell.Measure()/(6.0*
   infDiffusionTerm);
```

In the code we applied Mass Lumping stabilization when `invKa > 8.0`.

Chapter 7

Numerical Simulation

7.1 Description of the Problem

To test the previous mathematical methods, we consider a specific physical problem: we simulate a single phase flow in a porous medium, as a rock matrix, in presence of an embedded network of fractures, the so-called Discrete Fracture Network (DFN). This problem regards many practical applications, such as geological storage of pollutants (e.g. CO_2), aquifers monitoring, nuclear waste disposal and other geothermal or environmental issues.

The DFN is composed by fractures, which are regions of the rock that present a drastic change in the properties of the porous medium, and they are characterized by one of the three dimensions (the thickness) of several orders of magnitude smaller than the other two dimensions. Therefore, fractures are geometrically represented as two-dimensional objects.

In realistic applications, DFNs could comprise a huge number of fractures, with sizes ranging from small to very large scales. Since the setting of the problem is affected by a large uncertainty in both the hydrogeological parameters and the geometrical configuration, usually DFNs are generated stochastically, and as a consequence a multitude of numerical simulations is required.

In particular, we focus the attention on impervious rock matrices, assuming the absence of longitudinal flows in the traces, which are the segments generated by the intersection of two fractures. Moreover, we consider highly conductive fractures, which have a crucial influence on the fluid behavior, since they can induce preferential flow paths.

More generally, the evolutive problem on the DFN is described by the following partial differential equation in time and spatial domain:

$$e_i \rho c \frac{\partial u^*}{\partial t^*} - e_i \nabla^* \cdot (\lambda \nabla^* u^*) + e_i \rho c \beta^* \cdot \nabla^* u^* + \hbar u^* = \hbar u_r^*$$

We can get a dimensionless equation through the following change of variables

- $u^* = Uu$
- $x^* = Lx, \quad y^* = Ly, \quad z^* = Lz$

- $t^* = Tt = \frac{L}{B}t$
- $h^* = Hh$
- $\beta^* = B\beta \quad \beta^* = -\frac{K_i \nabla H}{e_i}$

where U, L, B, T, H are the characteristic units of temperature, length, velocity, time and hydraulic head respectively.

The dimensionless problem takes the form:

$$\frac{\partial u}{\partial t} - \nabla \cdot (\epsilon \nabla u) + \beta \cdot \nabla u + \sigma u = \sigma u_r$$

In a realistic application $U \sim 1K$, $L \sim 10^2 m$, $H \sim 10^3 m$, and the parameters of this model represent

- $e_i \sim 2 \cdot 10^{-3} \sqrt[12]{A_i} m$: thickness of the fracture, being i the index of the fracture and A_i (in m^2) the fracture area
- $\rho \sim 10^3 \frac{kg}{m^3}$: water density
- $c = 4186 \frac{J}{kg \cdot K}$: specific heat of water
- $\lambda = 0.6 \frac{W}{m \cdot K}$: thermal conductivity of water
- $K_i \sim 10^{-\alpha} \sqrt[4]{A_i} \frac{kg}{m \cdot s}$: fracture transmissivity, with the parameter α being $\alpha = 7$ in our application
- β^*: average Darcy velocity in the fracture section. Since $\beta^* = B\beta$ and $\beta^* = -\frac{K_i \nabla H}{e_i}$, then $B \sim \frac{10^{-\alpha} \sqrt[4]{L^2} H}{eL} = 10^{5-\alpha}$. As a consequence, $T = \frac{L}{B} \sim 10^{\alpha - 3}$
- $\hbar \sim 10 \frac{W}{m^2 K}$: heat transfer coefficient, which is en empiric coefficient such that the flux of heat ϕ entering the fracture is $\phi = \hbar(u_{rock} - u_{fracture})$. We suppose the rock as having an infinite thermal capacity, so that its temperature is considered constant and therefore it is a heat sink.
- $\epsilon = \frac{\lambda}{\rho c L B} \sim \frac{0.6}{4186} 10^{\alpha - 10}$ the diffusivity in the fracture
- $\sigma = \frac{L\hbar}{B \rho c e_i} \sim \frac{10^{\alpha - 2}}{2 \cdot 4186 \sqrt[12]{A_i}}$ the reaction coefficient

It is clear that in such flows in DFNs the characteristic unit of Darcy velocity field B may result of many orders of magnitude higher than the diffusivity ϵ, which is reflected in high Péclet numbers. In this framework, we say that the problem is convection-dominated, and when we try to solve it numerically through VEM, we could get a numerical solution which may present oscillations that do not have a physical source, therefore they do not represent the solution.

Similarly, oscillations may occur when the problem is reaction-dominated, meaning that the reaction phenomena governed by the parameter σ, dominate over the diffusion effects regulated through ϵ (this is the case of very low Karlowitz numbers).

In order to avoid these numerical instabilities, we apply stabilization methods to VEM.

7.2 Simulation and Error Analysis on a simple DFN

7.2.1 Stationary Problem

The numerical simulations that follow aim to solve the Reaction-Convection-Diffusion equation on a bounded convex polygonal domain $\Omega \subset \mathbb{R}^2$ with boundary $\partial\Omega$. The strong formulation of the stationary problems given by:

$$-\epsilon\Delta u + \beta \cdot \nabla u + \sigma u = f, \qquad \text{in } \Omega$$

with the addition of boundary conditions on $\partial\Omega$, which could be Dirichlet, Neumann or Robin conditions as discussed previously.

In the physical context of DFNs, the unknown function u represents a scalar field such as the temperature or the concentration of a pollutant on each fracture.

In these problems, the diffusivity $\epsilon : \Omega \to \mathbb{R}$ and the reaction term $\sigma : \Omega \to \mathbb{R}$ are a smooth functions with $\epsilon(x) \geq \epsilon_0 > 0, \quad \forall x \in \Omega$, and they model respectively the diffusivity and the reaction term in the considered fracture.

The field $\beta : \Omega \to \mathbb{R}^2$ is a smooth vector-valued function and represents the Darcy velocity, i.e. the convective field on the fracture. It is defined after the solution of the primal problem of the hydraulic head $u : \Omega \to \mathbb{R}$. In particular, $\beta = -K\nabla u$, where K is the transmissivity of the fracture.

The DFN where we run the numerical simulation is composed by two intersecting rectangular fractures, as shown in Figure 7.1.

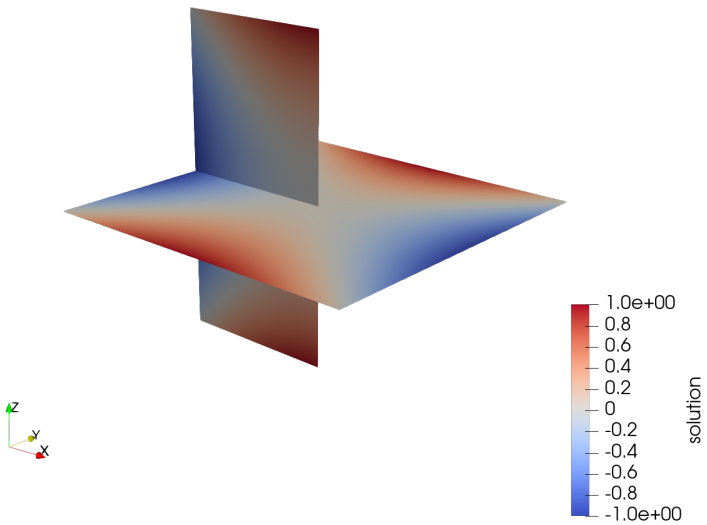


Figure 7.1: Discrete Fracture Network

We can map each fracture on $\mathbb{R}^2$ so that the vertices are:

- (0,1), (−1,1), (−1, −1), (0, −1) for the first fracture
- (1,1), (−1,1), (−1, −1), (1, −1) for the second fracture

The primal problem on the hydraulic head u is defined as:

$$\begin{cases} -\nabla \cdot (K\nabla u) = 0 & \text{in } \Omega_i \\ u = u_{D_i} & \text{on } \partial\Omega_i \end{cases}$$

where $K = 1$ and Ω_i represents the first fracture for $i = 1$, and the second one for $i = 2$, and u_D is the homogeneous Dirichlet boundary condition.

On both fractures, the parameters are set so that the solution of the hydraulic head u is the function

$$u(x, y) = (y^2 - 1) - (x^2 - 1)$$

defined respectively on Ω_1 and Ω_2.

As a consequence, homogeneous Dirichlet $u_{D_1} = u|_{\partial\Omega_1}$ and $u_{D_2} = u|_{\partial\Omega_2}$ conditions are set respectively on the boundaries $\partial\Omega_1$ and $\partial\Omega_2$.

The function u satisfies the requirement of no longitudinal flow in the trace. Indeed, in both fractures the trace is parametrized as $(x,0), x \in [-1,0]$ and $\frac{\partial u}{\partial y}\Big|_{y=0} = 2y\Big|_{y=0} = 0$.

For our numerical simulation, we set the forcing term $f : \Omega \to \mathbb{R}$ so that the solution on both domains Ω_1 and Ω_2 is the same solution of the primal problem for the hydraulic head. Therefore, $f = -\epsilon\Delta u + \beta\nabla u + \sigma u$. Since u is a polynomial function of degree 2, then $\Delta u = 0$; from the definition $\beta = -K\nabla u$, we get

$$\beta\nabla u = -K\nabla u \cdot \nabla u = -K\|\nabla u\|_2^2$$

In conclusion, $f = -K\|\nabla u\|_2^2 + \sigma u$.
Moreover, in order to have the same solution u also for the problem of our numerical simulation, we set the same u_D defined above as boundary conditions.

From the definition of Darcy velocity $\beta = -K\nabla u$ and letting $K = 1$, we get $\beta = \left(-\frac{\partial u}{\partial x}, -\frac{\partial u}{\partial y}\right) = (2x, -2y)$.

The purpose of the simulation is to understand when it is appropriate to use the stabilizations SUPG and Mass Lumping. For this purpose, we set ϵ and σ as constant functions and we varied their values in order to isolate the cases of convection- and reaction-domination.

Numerical Results

As discussed in Section 6, the need for a SUPG stabilization can be related to the Péclet number, which is defined on each element E as $Pe = \frac{\beta_{sup} h}{6\epsilon_{inf}}$, where $\beta_{sup} = \sup_{x\in E} \|\beta(x)\|_2$ and $\epsilon_{inf} = \inf_{x\in E} \epsilon(x)$. Similarly, Mass Lumping stabilization is linked to the inverse of the Karlowitz number: $Ka^{-1} = \frac{\sigma_{sup} h^2}{6\epsilon_{inf}}$, where $\sigma_{sup} = \sup_{x\in E} \sigma(x)$.

We ran the simulations on the same mesh, characterized by rectangular elements. The behavior of Péclet numbers and inverse of Karlowitz numbers for the different values of ϵ and σ can be derived from the following tables.

Table 7.1: **Péclet numbers statistics**

ϵ	1	10^{-3}	10^{-6}	10^{-9}
min Pe	1.570360e-03	1.570360e+00	1.570360e+03	1570360
avg Pe	1.993662e-02	1.993662e+01	1.993662e+04	1.993662e+07
max Pe	4.153510e-02	4.153510e+01	4.153510e+04	41535100

Table 7.2: **Inverse of Karlowitz numbers statistics -** $\sigma = 10^3$

ϵ	1	10^{-3}	10^{-6}	10^{-9}
min Ka^{-1}	3.255210e-01	3.255210e+02	325521	325521000
avg Ka^{-1}	4.330881e-01	4.330881e+02	4.330881e+05	4.330881e+08
max Ka^{-1}	6.510420e-01	6.510420e+ 02	651042	651042000

Table 7.3: **Inverse of Karlowitz numbers statistics -** $\sigma = 1$

ϵ	1	10^{-3}	10^{-6}	10^{-9}
min Ka^{-1}	3.255210e-04	3.255210e-01	3.255210e+02	325521
avg Ka^{-1}	4.330881e-04	4.330881e-01	4.330881e+02	4.330881e+05
max Ka^{-1}	6.510420e-04	6.510420e-01	6.510420e+02	651042

Table 7.4: **Inverse of Karlowitz numbers statistics -** $\sigma = 10^{-3}$

ϵ	1	10^{-3}	10^{-6}	10^{-9}
min Ka^{-1}	3.255210e-07	3.255210e-04	3.255210e-01	3.255210e+02
avg Ka^{-1}	4.330881e-07	4.330881e-04	4.330881e-01	4.330881e+02
max Ka^{-1}	6.510420e-07	6.510420e-04	6.510420e-01	6.510420e+02

Table 7.5: **Inverse of Karlowitz numbers statistics -** $\sigma = 10^{-6}$

ϵ	1	10^{-3}	10^{-6}	10^{-9}
$\min Ka^{-1}$	3.255210e-10	3.255210e-07	3.255210e-04	3.255210e-01
avg Ka^{-1}	4.330881e-10	4.330881e-07	4.330881e-04	4.330881e-01
$\max Ka^{-1}$	6.510420e-10	6.510420e-07	6.510420e-04	6.510420e-01

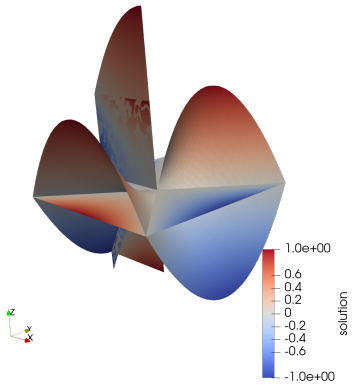
When we let the reaction coefficient $\sigma = 0$, then by definition of Karlowitz number $Ka^{-1} = 0$

SUPG stabilization

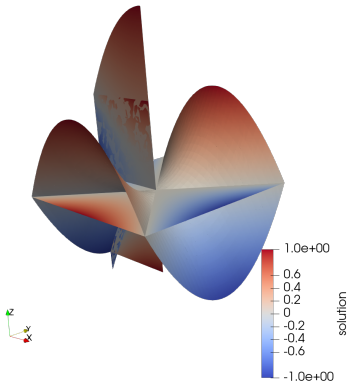
To isolate the convection-dominated case, we firstly considered $\sigma = 0$ and we ran simulations for values $\epsilon = 1, 10^{-3}, 10^{-6}, 10^{-9}$, both using the standard VEM, and the VEM with the introduction of the stabilization SUPG term. The results show that the considered stabilization is needed starting from Péclet numbers of order 10^3, which correspond to values $\epsilon \lesssim 10^{-6}$. Indeed, without a stabilization the numerical solution shows oscillations that are not present in the exact solution. These oscillations are prevented when the stabilized method is applied.

The following figures show the graphs of the solutions on each of the two fractures of the DFN, corresponding to problems with different parameters ϵ and σ.

Figure 7.2: Solution of the stationary problem on Fracture 1 - $\epsilon = 1, \sigma = 0, Pe \sim 10^{-3}$

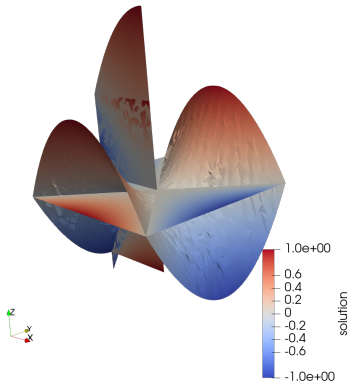


(a) No Stabilization

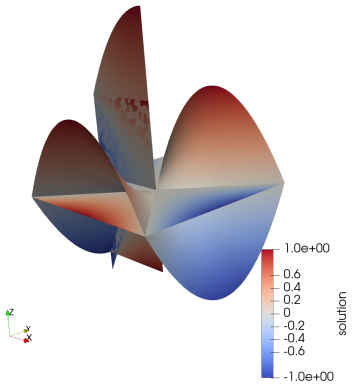


(b) SUPG

Figure 7.3: Solution of the stationary problem on Fracture 1 - $\epsilon = 10^{-3}, \sigma = 0, Pe \sim 1$

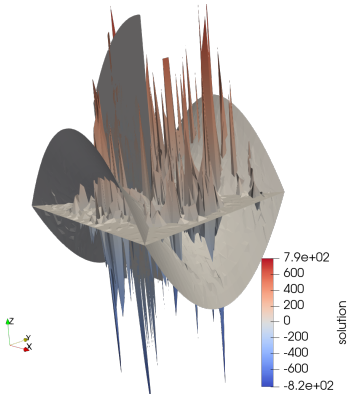


(a) No Stabilization

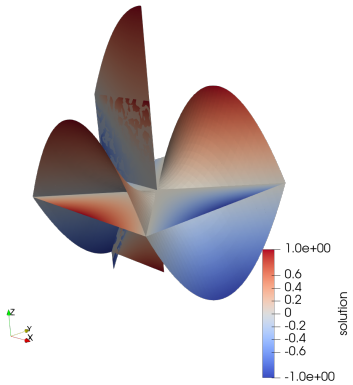


(b) SUPG

Figure 7.4: Solution of the stationary problem on Fracture 1 - $\epsilon = 10^{-6}, \sigma = 0, Pe \sim 10^3$

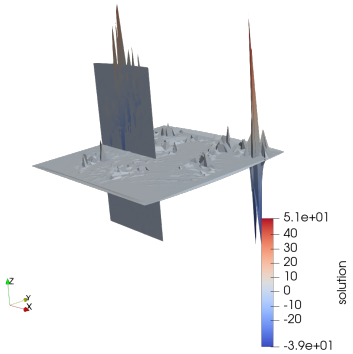


(a) No Stabilization

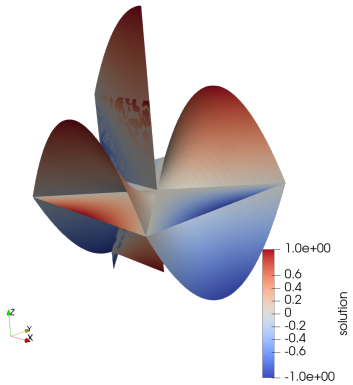


(b) SUPG

Figure 7.5: Solution of the stationary problem on Fracture 1 - $\epsilon = 10^{-9}, \sigma = 0, Pe \sim 10^6$

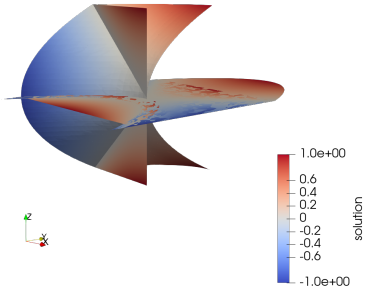


(a) No Stabilization

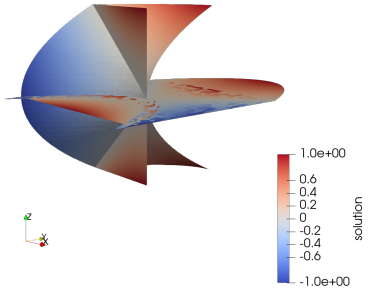


(b) SUPG

Figure 7.6: Solution of the stationary problem on Fracture 2 - $\epsilon = 1, \sigma = 0, Pe \sim 10^{-3}$

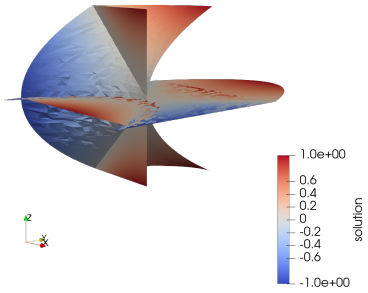


(a) No Stabilization

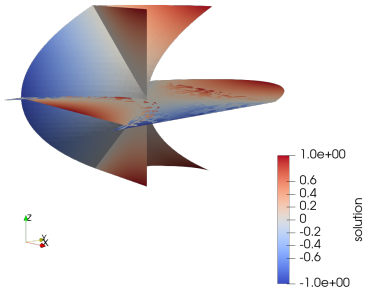


(b) SUPG

Figure 7.7: Solution of the stationary problem on Fracture 2 - $\epsilon = 10^{-3}, \sigma = 0, Pe \sim 1$

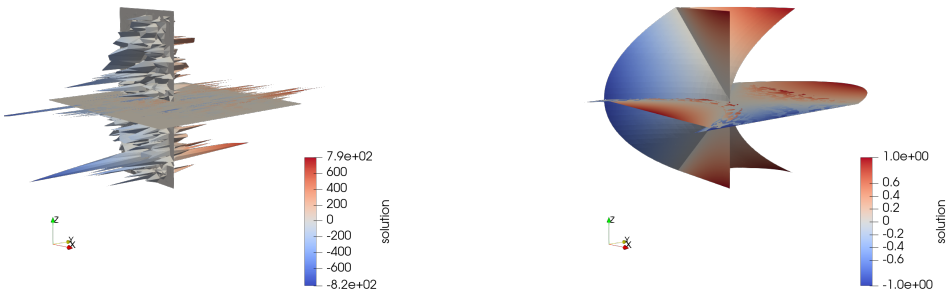


(a) No Stabilization



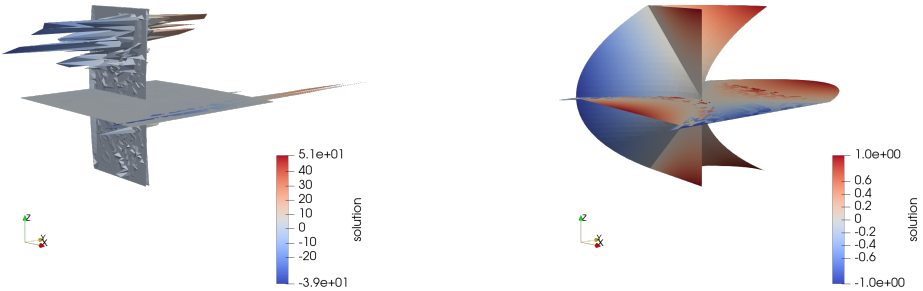
(b) SUPG

Figure 7.8: Solution of the stationary problem on Fracture 2 - $\epsilon = 10^{-6}, \sigma = 0, Pe \sim 10^3$



(a) No Stabilization (b) SUPG

Figure 7.9: Solution of the stationary problem on Fracture 2 - $\epsilon = 10^{-9}, \sigma = 0, Pe \sim 10^6$



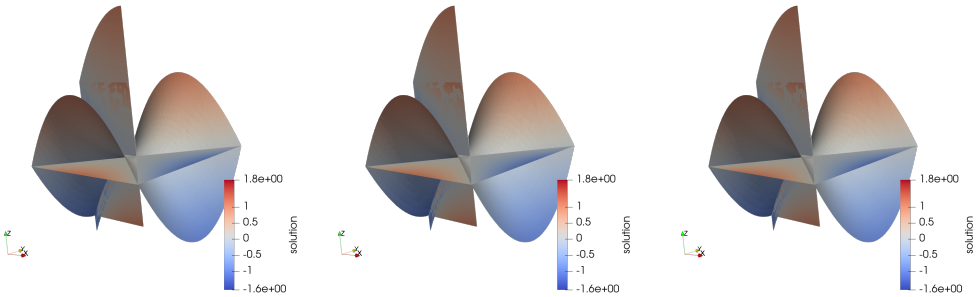
(a) No Stabilization (b) SUPG

Mass Lumping stabilization

Mass Lumping stabilization is required for reaction-dominated problems, which are characterized by high values of the inverse of Karlowitz number. In our simulations we considered values of $\sigma = 10^{-6}, 10^{-3}, 1, 10^{3}$. We applied non-stabilized VEM, VEM with SUPG stabilization and VEM with SUPG and Mass Lumping stabilizations. It resulted that Mass Lumping has no effective influence on the numerical solution when considering values $\sigma \leq 1$, indeed, the numerical solution with SUPG and Mass Lumping has the same behavior as the solution with only SUPG stabilization involved. Instead, when considering $\sigma = 10^{3}$, SUPG is no longer sufficient to get a satisfactory numerical solution: for values $\epsilon \leq 1$, oscillations occur. If we apply Mass Lumping stabilization, the numerical solution is not particularly smooth, but the mentioned oscillations are significatively reduced.

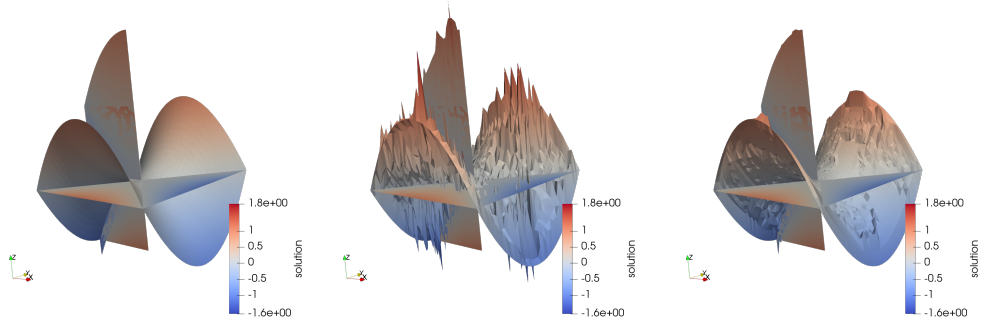
Besides, it is worth noticing that for values $\epsilon \leq 10^{-6}$, the standard VEM is not even able to find a solution, as we get NaN values.

Figure 7.10: Solution of the stationary problem on Fracture 1 - $\epsilon = 1, \sigma = 10^3, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-1}$



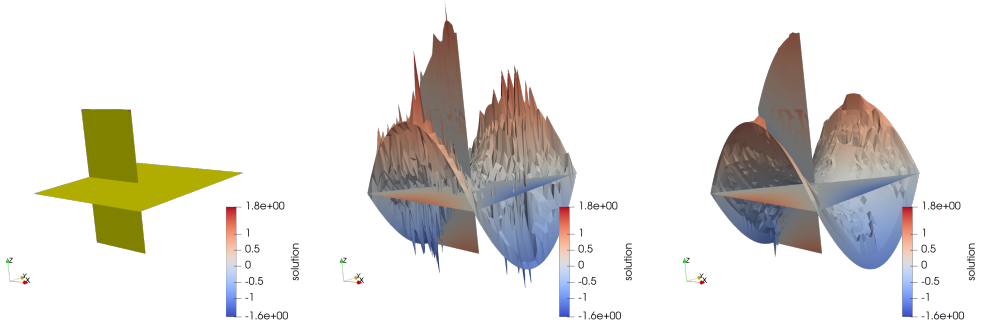
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.11: Solution of the stationary problem on Fracture 1 - $\epsilon = 10^{-3}, \sigma = 10^3, Pe \sim 10, Ka^{-1} \sim 10^2$



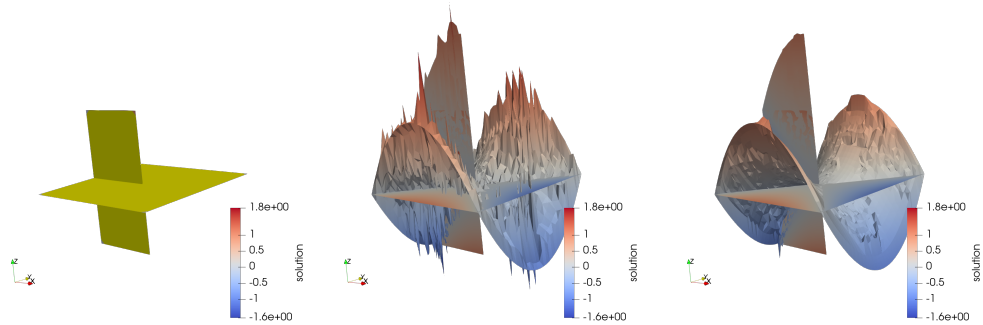
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.12: Solution of the stationary problem on Fracture 1 - $\epsilon = 10^{-6}, \sigma = 10^{3}, Pe \sim 10^{4}, Ka^{-1} \sim 10^{5}$



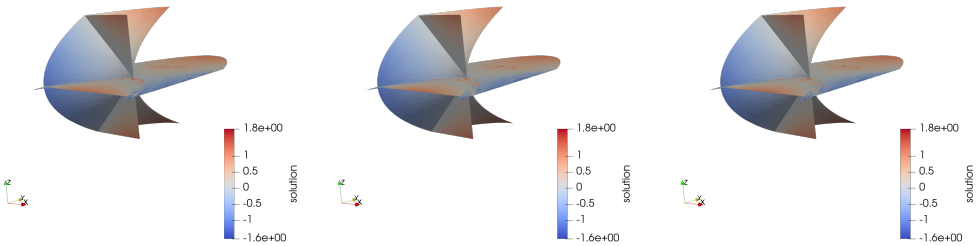
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.13: Solution of the stationary problem on Fracture 1 - $\epsilon = 10^{-9}, \sigma = 10^{3}, Pe \sim 10^{7}, Ka^{-1} \sim 10^{8}$



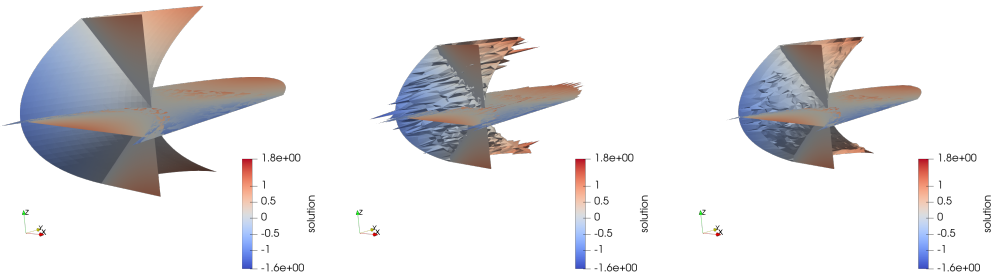
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.14: Solution of the stationary problem on Fracture 2 - $\epsilon = 1, \sigma = 10^3, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-1}$



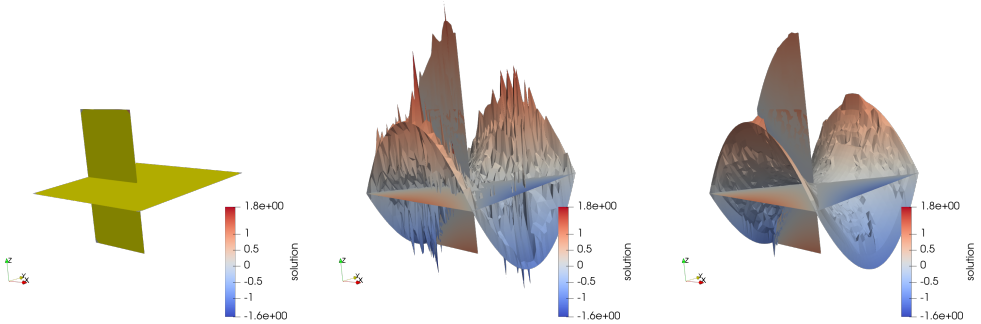
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.15: Solution of the stationary problem on Fracture 2 - $\epsilon = 10^{-3}, \sigma = 10^3, Pe \sim 10, Ka^{-1} \sim 10^2$



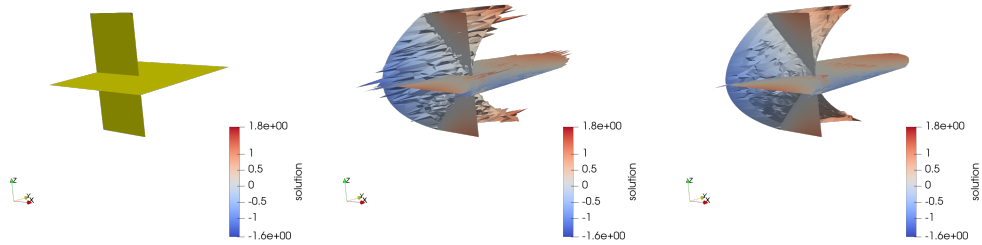
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.16: Solution of the stationary problem on Fracture 2 - $\epsilon = 10^{-6}, \sigma = 10^{3}, Pe \sim 10^{4}, Ka^{-1} \sim 10^{5}$



(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.17: Solution of the stationary problem on Fracture 2 - $\epsilon = 10^{-9}, \sigma = 10^{3}, Pe \sim 10^{7}, Ka^{-1} \sim 10^{8}$



(a) No Stabilization (b) SUPG (c) Mass Lumping

Error Analysis

As discussed in Section 4.3, an $H^1(\Omega)$ a priori error estimate for the VEM of order $k = 1$ is given by

$$\|u - u_h\|_1 \leq Ch(\|u\|_2 + |f|_1) \tag{7.1}$$

To check if this rate of convergence is preserved also when applying the stabilization methods SUPG and Mass Lumping, we refined the mesh multiple times (thus we increased the number of degrees of freedom N_{dof}) and we plotted the H^1-errors versus N_{dof}. As the following figures show, it is preserved the rate of convergence of VEM of $\frac{1}{2}$ with respect to the number of degrees of freedom, that scales as $\frac{1}{h^2}$.

From the figures it is also evident that for small values of ϵ, the stabilized methods (yellow and blue curves) are required to compute a satisfactory approximation of the solution of the partial differential problem. Indeed, the non stabilized VEM (red curve) leads to significant errors. For example, in Figure 7.19 we see that the error is even increasing when doubling the order of magnitude of N_{dof}. The a priori error estimate for non-stabilized VEM is supposed to be evident for very high values of N_{dof}.

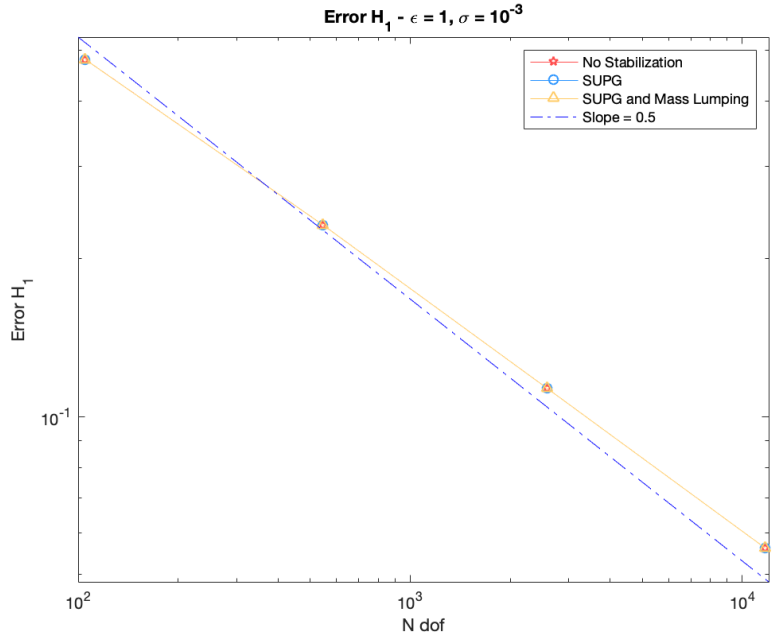


Figure 7.18: Stationary problem H^1-error versus N_{dof} - $\epsilon = 1, \sigma = 10^{-3}$

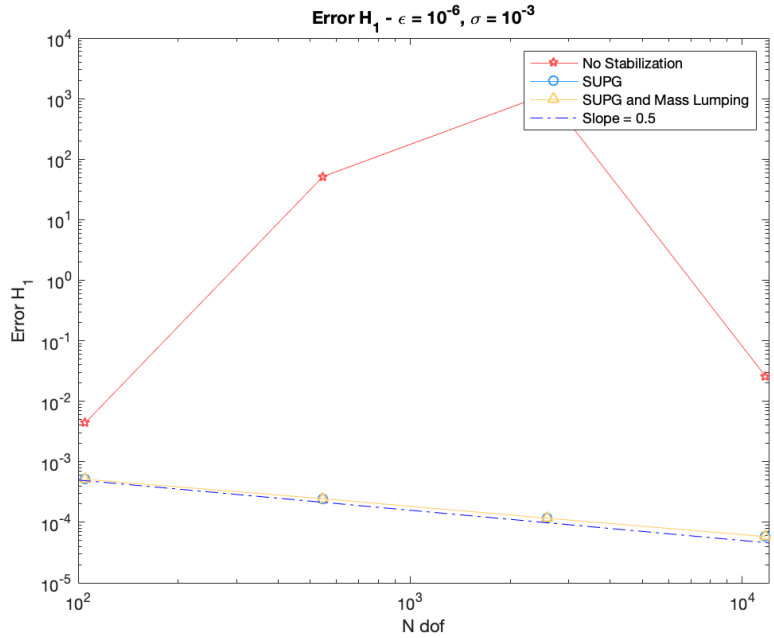


Figure 7.19: Stationary problem H^1-error versus N_{dof} - $\epsilon = 10^{-6}, \sigma = 10^{-3}$

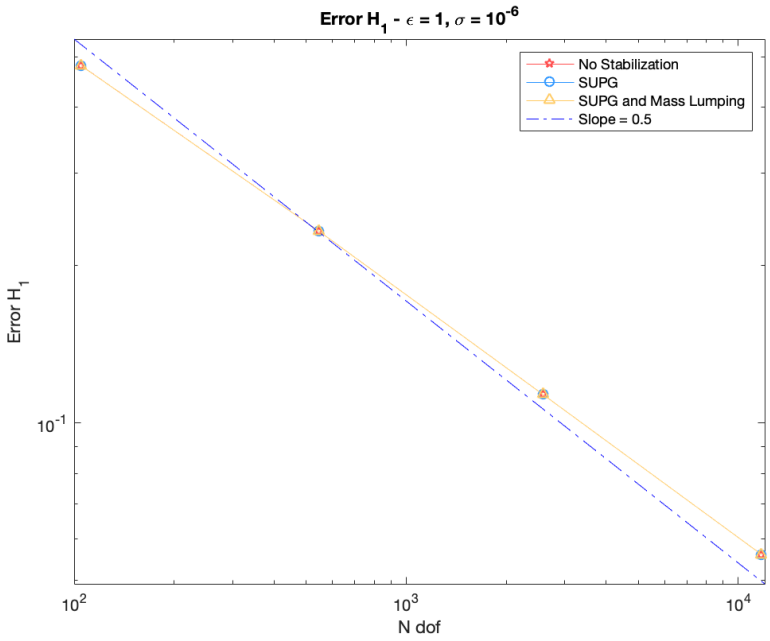


Figure 7.20: Stationary problem H^1-error versus N_{dof} - $\epsilon = 1, \sigma = 10^{-6}$

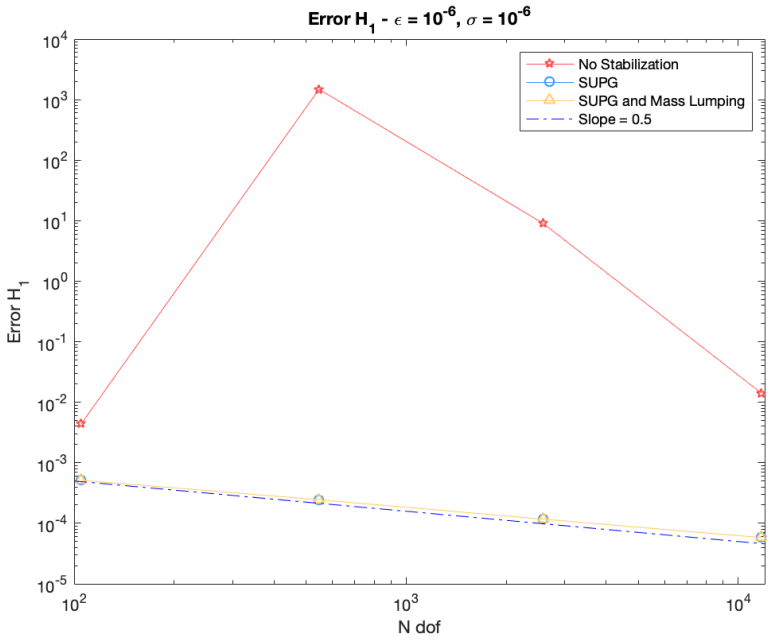


Figure 7.21: Stationary problem H^1-error versus N_{dof} - $\epsilon = 10^{-6}, \sigma = 10^{-6}$

7.2.2 Evolutive Problem

The evolutive problem or Reaction-Convection-Diffusion is governed by the equation

$$\frac{\partial u}{\partial t} - \epsilon \Delta u + \beta \cdot \nabla u + \sigma u = f, \quad [0, T] \times \Omega$$

We ran simulations on the same DFN considered in the stationary problem. We set the parameters so that the exact solution of the problem is

$$u(t, u) = t^3 u(x), \quad (t, x) \in [0, T] \times \Omega$$

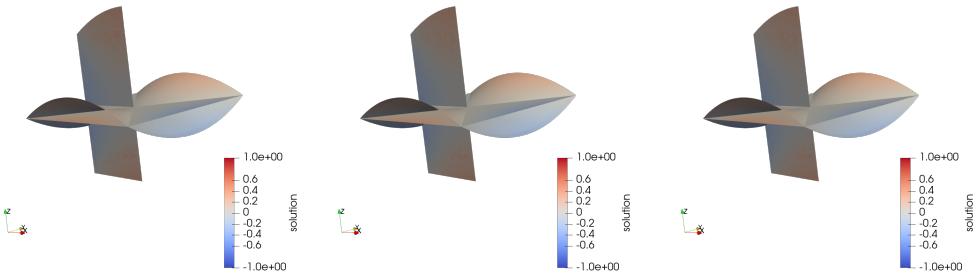
where $u(x)$ is the solution of the stationary problem. Therefore, we set $f = 3t^2 u + t^3 f_s$, being $f_s = -\epsilon \Delta u + \beta \nabla u + \sigma u$ the forcing term of the stationary problem.

Numerical Results

We ran several simulations, considering different constant values for the parameters of diffusivity ϵ and reaction term σ, in order to isolate convection-dominant and reaction-dominant cases, and thus to understand when it is convenient to apply stabilization techniques.

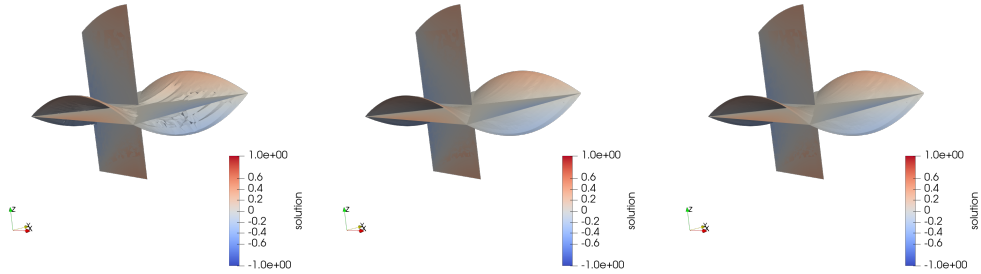
We ran three simulations with the different stabilizations considered, where we set a time step size of $\Delta t = 0.01$ and $N = 100$ time steps. Thus, we simulated the scalar field in the DFN from $t = 0$ to $t = 1$. From the following figures, it is clear that when the problem is convection-dominated (for example for values $\epsilon = 10^{-6}$) the method with SUPG provides a numerical solution which is smoother than the solution obtained with the non stabilized VEM, so it is worth to apply SUPG. Instead, when $\epsilon = 1$ the problem is not convection-dominated and SUPG does not seem to provide a better numerical solution with respect to the application of the standard VEM.

Figure 7.22: Solution of the evolutive problem on Fracture 1 - $t_k = 67, \epsilon = 1, \sigma = 1, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-4}$



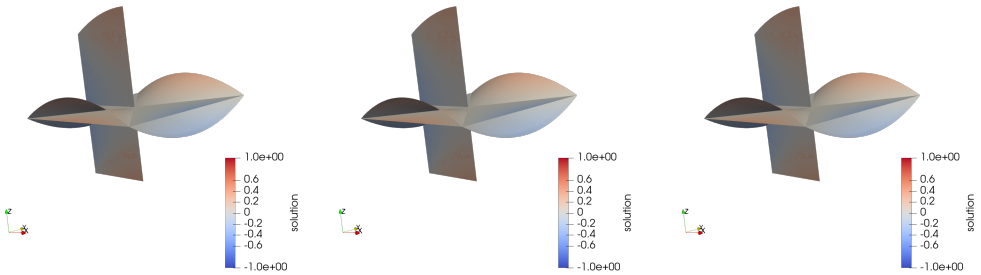
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.23: Solution of the evolutive problem on Fracture 1 - $t_k = 67, \epsilon = 10^{-6}, \sigma = 1, Pe \sim 10^{4}, Ka^{-1} \sim 10^{2}$



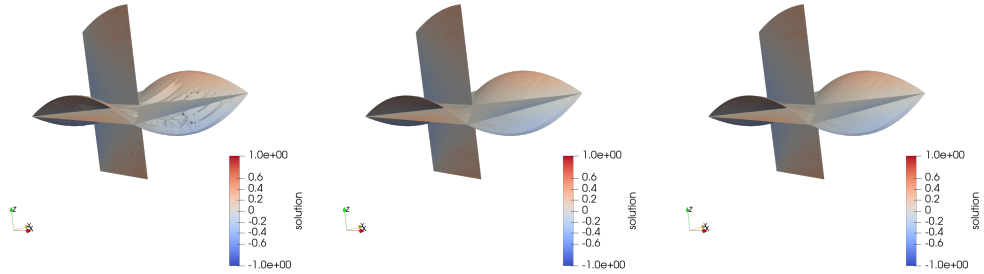
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.24: Solution of the evolutive problem on Fracture 1 - $t_k = 67, \epsilon = 1, \sigma = 10^{-6}, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-10}$



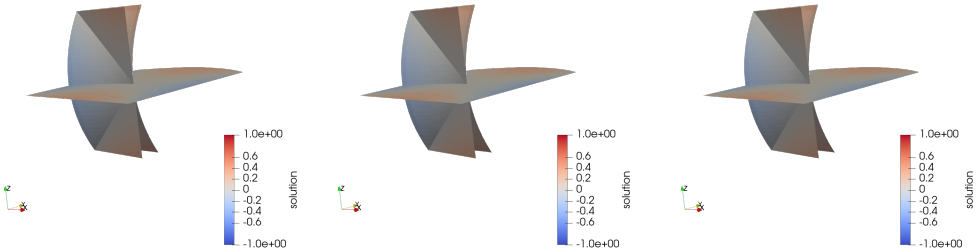
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.25: Solution of the evolutive problem on Fracture 1 - $t_k = 67, \epsilon = 10^{-6}, \sigma = 10^{-6}, Pe \sim 10^{4}, Ka^{-1} \sim 10^{-4}$



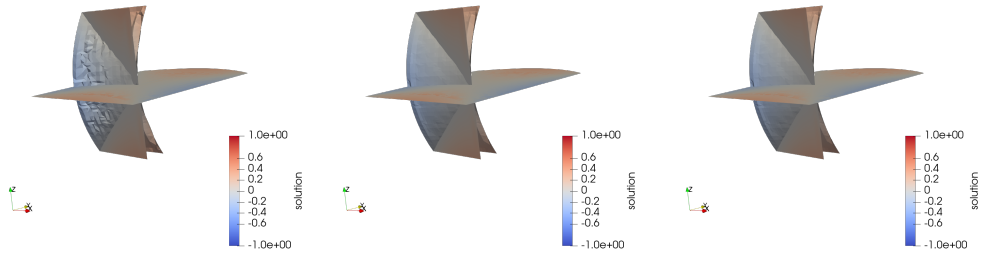
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.26: Solution of the evolutive problem on Fracture 2 - $t_k = 67, \epsilon = 1, \sigma = 1, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-4}$



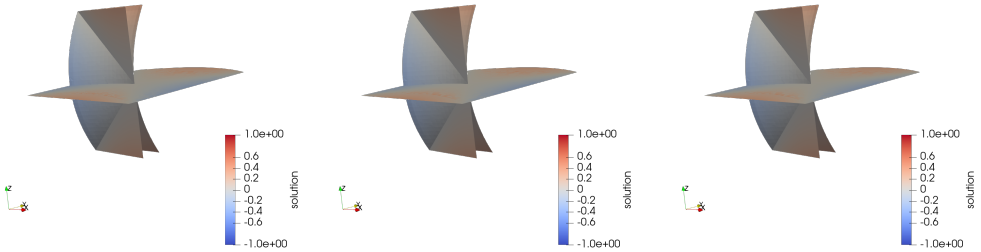
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.27: Solution of the evolutive problem on Fracture 2 - $t_k = 67, \epsilon = 10^{-6}, \sigma = 1, Pe \sim 10^{4}, Ka^{-1} \sim 10^{2}$



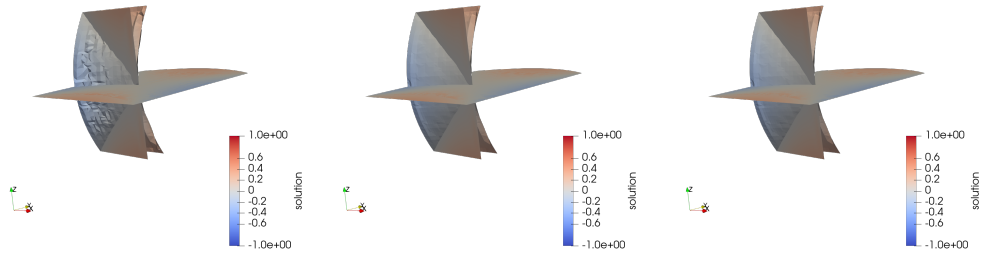
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.28: Solution of the evolutive problem on Fracture 2 - $t_k = 67, \epsilon = 1, \sigma = 10^{-6}, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-10}$



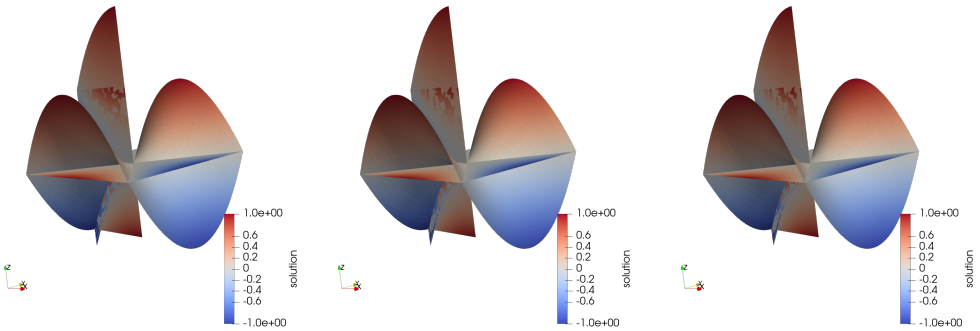
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.29: Solution of the evolutive problem on Fracture 2 - $t_k = 67, \epsilon = 10^{-6}, \sigma = 10^{-6}, Pe \sim 10^{4}, Ka^{-1} \sim 10^{-4}$



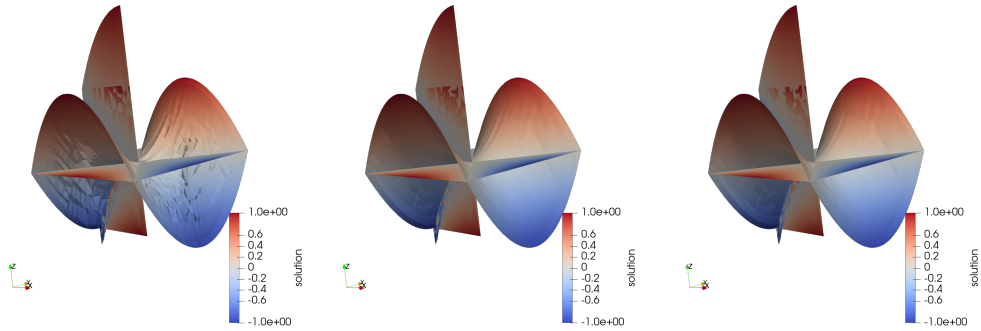
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.30: Solution of the evolutive problem on Fracture 1 - $t_k = 100, \epsilon = 1, \sigma = 1, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-4}$



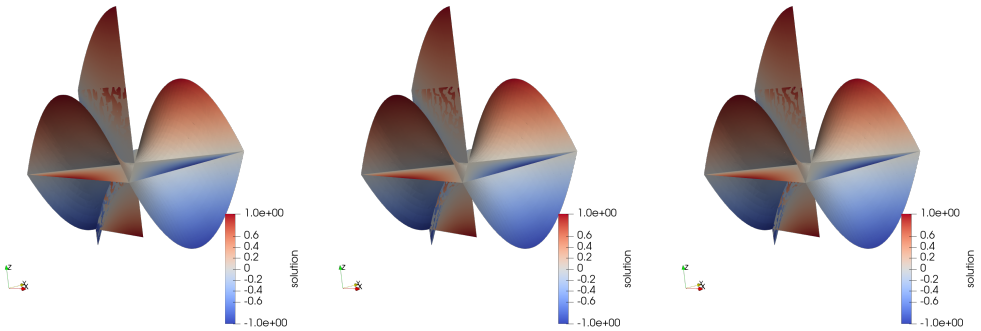
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.31: Solution of the evolutive problem on Fracture 1 - $t_k = 100, \epsilon = 10^{-6}, \sigma = 1, Pe \sim 10^{4}, Ka^{-1} \sim 10^{2}$



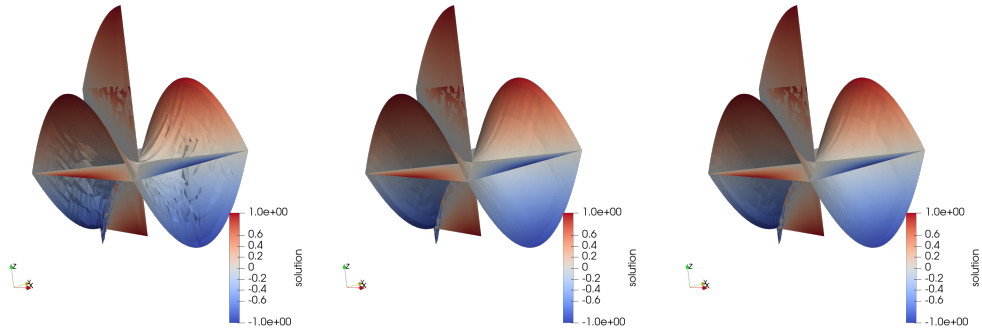
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.32: Solution of the evolutive problem on Fracture 1 - $t_k = 100, \epsilon = 1, \sigma = 10^{-6}, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-10}$



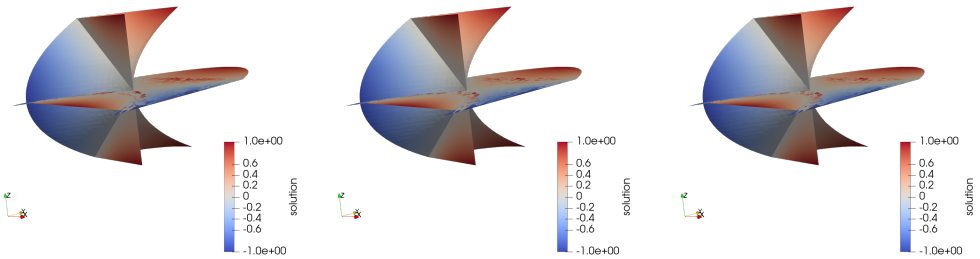
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.33: Solution of the evolutive problem on Fracture 1 - $t_k = 100, \epsilon = 10^{-6}, \sigma = 10^{-6}, Pe \sim 10^{4}, Ka^{-1} \sim 10^{-4}$



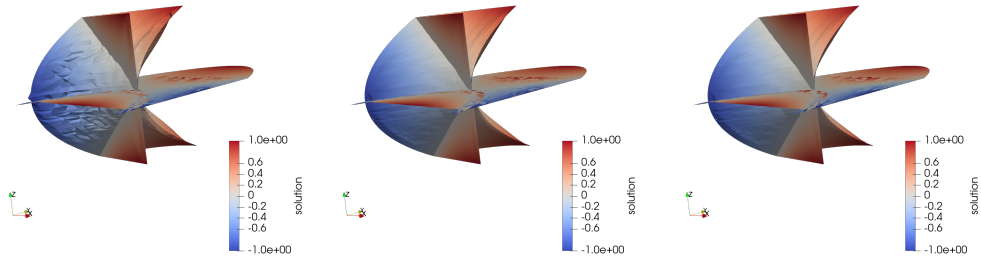
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.34: Solution of the evolutive problem on Fracture 2 - $t_k = 100, \epsilon = 1, \sigma = 1, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-4}$



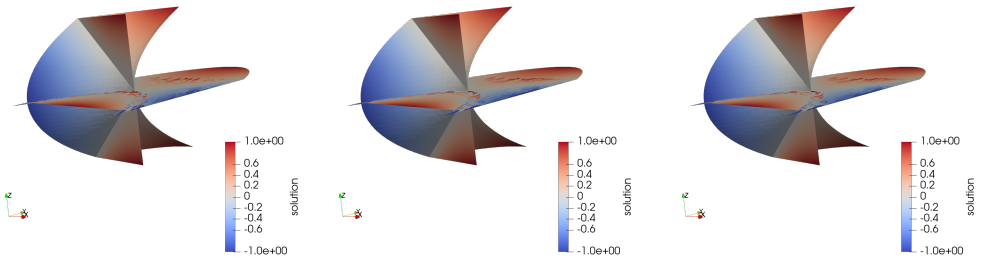
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.35: Solution of the evolutive problem on Fracture 2 - $t_k = 100, \epsilon = 10^{-6}, \sigma = 1, Pe \sim 10^{4}, Ka^{-1} \sim 10^{2}$



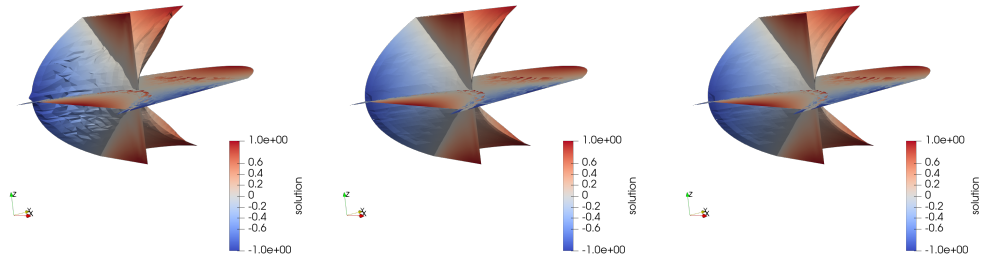
(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.36: Solution of the evolutive problem on Fracture 2 - $t_k = 100, \epsilon = 1, \sigma = 10^{-6}, Pe \sim 10^{-2}, Ka^{-1} \sim 10^{-10}$



(a) No Stabilization (b) SUPG (c) Mass Lumping

Figure 7.37: Solution of the evolutive problem on Fracture 2 - $t_k = 100, \epsilon = 10^{-6}, \sigma = 10^{-6}, Pe \sim 10^{4}, Ka^{-1} \sim 10^{-4}$



(a) No Stabilization (b) SUPG (c) Mass Lumping

Error Analysis

As for the stationary problem, we analyzed the behavior of the H^1-norm of the difference between the numerical and the exact solution of the evolutive problem. Whereas in the stationary case we only varied the number of degrees of freedom of the mesh, here we also changed the discretization time step size, and related the H^1-error to it.

For the discretization in time, we considered values of $\Delta t = 0.05, 0.1, 0.2, 0.4$, that correspond to numbers of time steps $N = 200, 100, 50, 25$ respectively, if we want to compute the solution up to a time value $t = 1$.

To discretize the spatial domain, we refined the mesh multiple times and we solved the numerical problem for values of numbers of degrees of freedom $N_{dof} = 105, 546, 2576, 11738$.

We solved the problem for the different combinations of the two types of discretization, and computed the H^1-error of the exact and numerical solution at the last time step, that in each case corresponds to a value $t = 1$. At this value of t, all numerical problems aim to approximate the same exact solution

$$u(t = 1, x, y) = (y^2 - 1) - (x^2 - 1)$$

These errors are plotted in the following figures. Each figure corresponds to a particular method (VEM, VEM with SUPG, VEM with SUPG and Mass Lumping).

In the error-plots relative to a value of the diffusion term $\epsilon = 1$ (Figures 7.38, 7.40 and 7.42), we observe that when the mesh is coarse, i.e. for $N_{dof} = 105$, the error due to the discretization in space dominates and we cannot detect the convergence in time. Indeed, the error does not seem to decrease, even for high values of number of time steps. Likewise, when we set $\Delta t = 0.4$, corresponding to $N = 25$, the numerical solution does not converge when increasing N_{dof}, because the discretization in time prevails in the error.

On the contrary, for high values of N_{dof}, for example $N_{dof} = 11738$, the error converges in time, and, in the same way, when the number of time steps is high ($N = 200$), the error in space seems to converge.

When we decrease the order of magnitude of the diffusion term ϵ, letting $\epsilon = 10^{-3}$, we find out that a stabilization technique is needed in order to get a satisfactory numerical solution. Indeed, as Figure 7.39 shows, the non-stabilized version of the VEM does not show a convergence behavior for the values of N_{dof} and number of time steps N that we considered. In fact, we should refine further the mesh and the discretization of the time interval in order to perceive the convergence of the method.

Instead, if we apply SUPG stabilization or both SUPG and Mass Lumping, we can see that the H^1-error of the solution at time $t = 1$ converges when increasing N_{dof} and the number N of time steps. Nevertheless, the the meshes we used for the simulations are still too coarse to detect the polynomial convergence in time of the numerical solution.

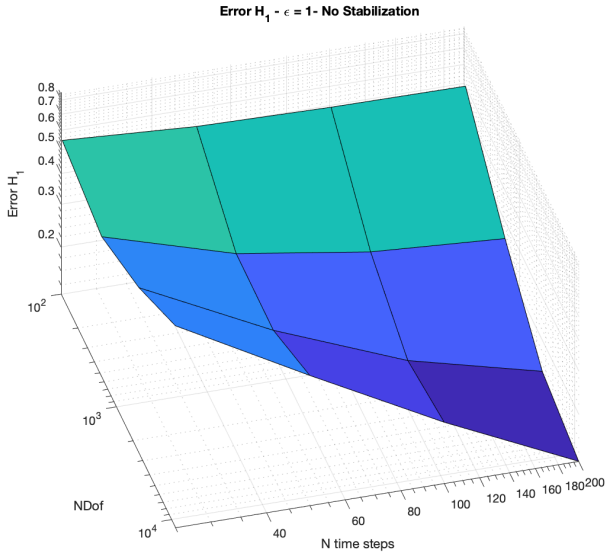


Figure 7.38: Evolutive problem H^1-error of solution at $t = 1$ versus N_{dof} and N time steps - No Stabilization - $\epsilon = 1, \sigma = 1$

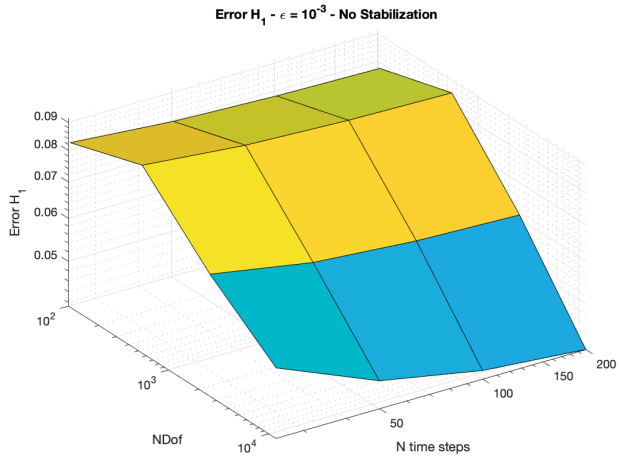


Figure 7.39: Evolutive problem H^1-error of solution at $t = 1$ versus N_{dof} and N time steps - No Stabilization - $\epsilon = 10^{-3}, \sigma = 1$

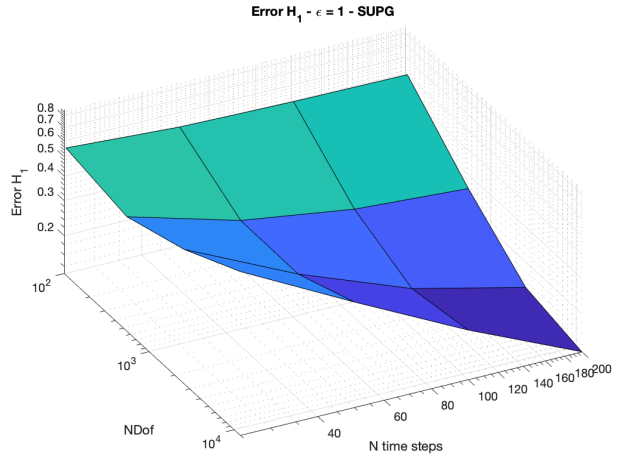


Figure 7.40: Evolutive problem H^1-error of solution at $t = 1$ versus N_{dof} and N time steps - SUPG - $\epsilon = 1, \sigma = 1$

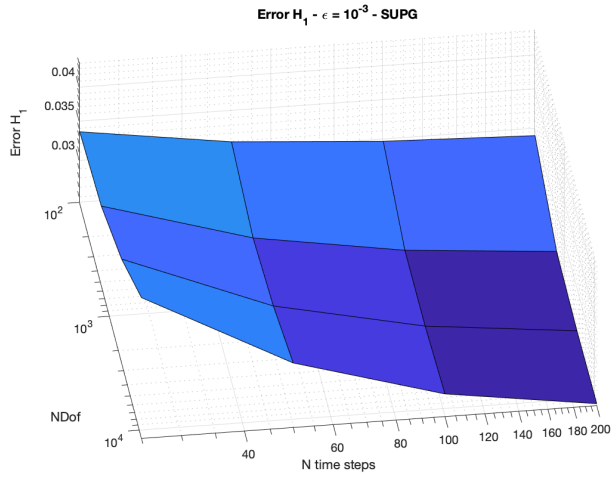


Figure 7.41: Evolutive problem H^1-error of solution at $t = 1$ versus N_{dof} and N time steps - SUPG - $\epsilon = 10^{-3}, \sigma = 1$

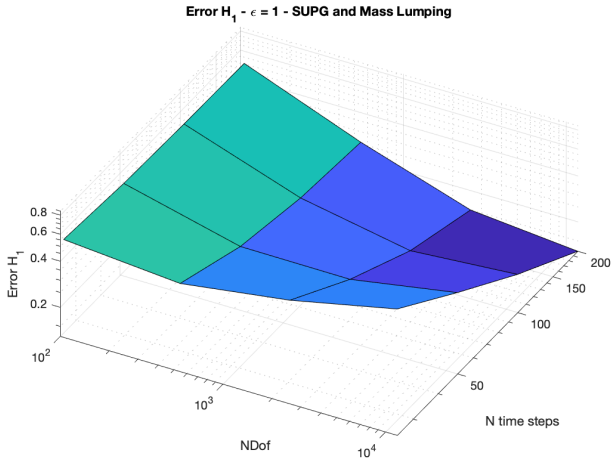


Figure 7.42: Evolutive problem H^1-error of solution at $t = 1$ versus N_{dof} and N time steps - SUPG and Mass Lumping - $\epsilon = 1, \sigma = 1$

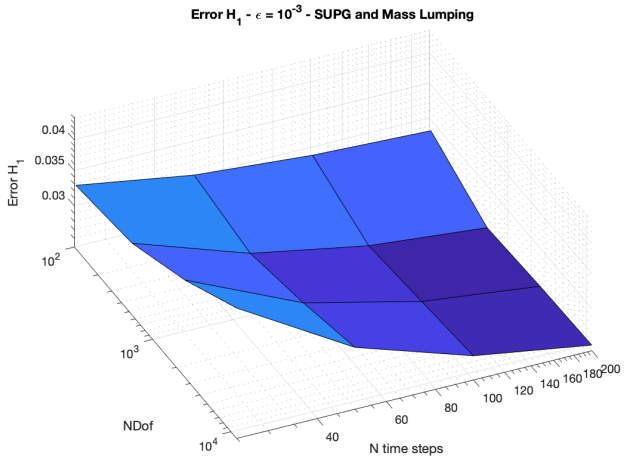


Figure 7.43: Evolutive problem H^1-error of solution at $t = 1$ versus N_{dof} and N time steps - SUPG and Mass Lumping - $\epsilon = 10^{-3}, \sigma = 1$

7.3 Realistic Geophysical Simulation

Stabilization methods become particularly important when dealing with realistic problems, which are usually characterized by mesh with high numbers of degrees of freedom. Besides, in problems like the computation of flows in fractured porous media, we could first need to solve a primal problem on a physical quantity such as the hydraulic head, and from the solution of this first problem, we could determine the convective field (the so-called Darcy velocity). Since in general we do not know the hydraulic head field (it is indeed the unknown function of the primal problem), then we find uncertainty on the convective field, which could dominate on the diffusivity of the media. In order to prevent numerical instabilities due to a convection-domination, we could apply the SUPG stabilization to the VEM.

To show the applicability of SUPG, we ran a simulation on a realistic DFN, which is generated stochastically. It represents the fracture network in an underground rock matrix, and two wells.

We firstly solved the primal problem to find the hydraulic head field in the DFN. This stationary problem is defined by the equation

$$-\nabla \cdot (K \nabla u) = 0$$

with Dirichlet conditions on the endings of the two wells: $u = 0.13$ on the first and $u = 0$ on the second.

In the simulation for the evolution of the temperature field in the fracture network, we considered an initial condition where all the rock matrix is at the same temperature ($T = 50°C$). The evolutive problem is defined by

$$\frac{\partial u}{\partial t} - \epsilon \Delta u + \beta \cdot \nabla u + \sigma u = f$$

Here the diffusivity has a constant value $\epsilon = \frac{30}{4.186} 10^{-5}$.

β is Darcy velocity, defined as usual as $\beta = -K \nabla u_H$, with u_H the solution of the hydraulic head.

The reaction coefficient σ is $\sigma = 10^{-3}$ in the wells, and $\frac{1}{5 \cdot 4.186} c$ elsewhere, being c the convective transfer coefficient (in our simulations $c = 0.1$).

Finally, we considered a source term f given by $f = \sigma \cdot T_0$, where $T_0 = 50°C$ is the initial temperature of the rock matrix.

We ran a simulation for a nondimensionalized time interval of length 1. Recalling that $t^* = Tt$, with $T \sim 10^{\alpha - 3}$ and $\alpha = 7$ in our simulations, then this time interval corresponds to a realistic value of the order of magnitude of the hour, that is a reasonable time scale for geophysical phenomena.

In this time interval, we pump cold water in a well, in order to cool the underground rock. Mathematically, we set a Dirichlet boundary condition on the edge where we pump water with a temperature value of $T = 15°C$, and homogeneous Neumann border conditions on all other edges of the DFN. The simulation shows that gradually the rock matrix is cooled, starting from regions close to the well where we are pumping cold water. The following figures show the results of this simulation at different time steps t_k.

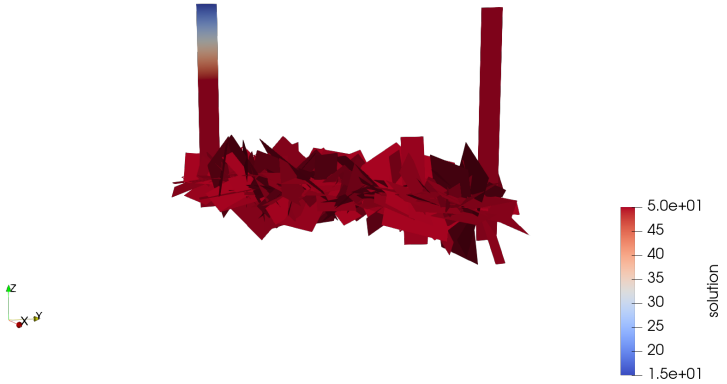


Figure 7.44: Realistic simulation - $t_k = 0$

Figure 7.45: Realistic simulation - $t_k = 1$

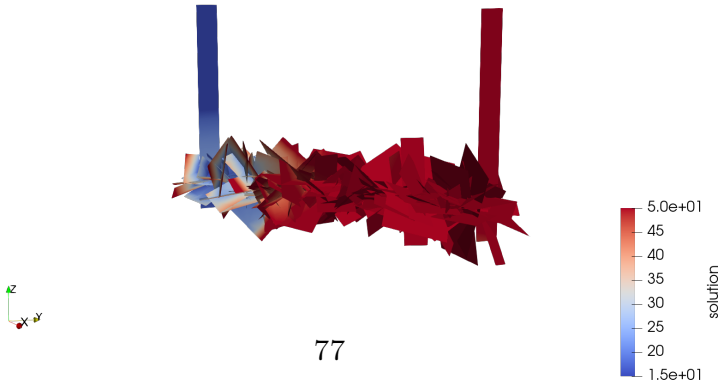


Figure 7.46: Realistic simulation - $t_k = 7$

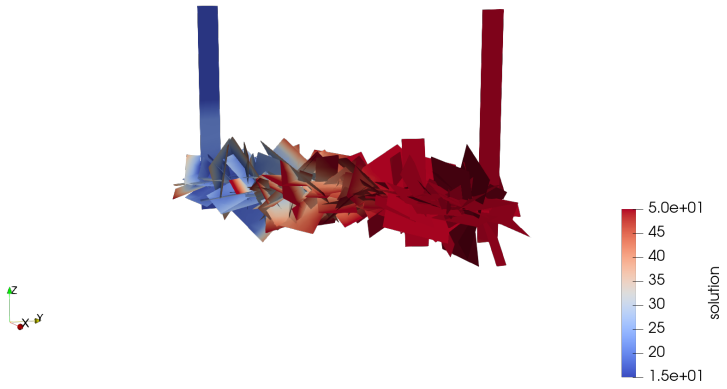


Figure 7.47: Realistic simulation - $t_k = 13$

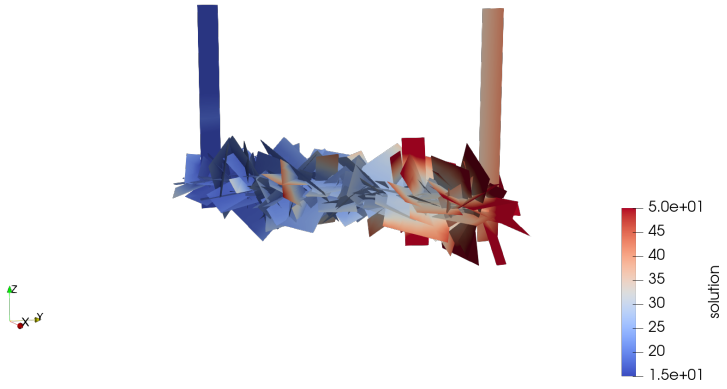


Figure 7.48: Realistic simulation - $t_k = 41$

Chapter 8

Conclusions and Future Developments

In this thesis we analyzed the second order elliptic problem of Reaction-Convection-Diffusion, considering both the stationary and the evolutive versions. We derived the variational formulation of this problem and applied the Virtual Element Method to solve numerically the partial differential equation.

We also described the stabilization methods of Streamline Upwind Petrov Galerkin and Mass Lumping, respectively in the cases of convection-dominated and reaction-dominated problems. We also discussed the convergence of the methods with respect to the discretization parameters.

Then we described the implementation of this method in C++. The code was useful to analyze the behavior of VEM, in its stabilized and non-stabilized versions. In particular, we assessed that the convergence of the non-stabilized VEM is preserved also when applying the stabilization techniques.

Finally, we performed a simulation on a realistic geophysical problem, to show the applicability of the VEM to problems characterized by a complex geometry.

These results lay out the groundwork for future developments. In particular, also the VEM of order $k > 1$ and in dimensions greater than 2 can be analyzed and implemented in C++, along with the study of the behavior of SUPG and Mass Lumping stabilizations.

Bibliography

Matías Benedetto, Stefano Berrone, Andrea Borio, Sandra Pieraccini, and Stefano Scialo. Order preserving SUPG stabilization for the virtual element formulation of advection-diffusion problems. *Computer Methods in Applied Mechanics and Engineering*, 311, 08 2016. doi: 10.1016/j.cma.2016.07.043.

S. Berrone and A. Borio. An optimized implementation of the Virtual Element Method for second order PDEs, 2020.

Stefano Berrone, Andrea Borio, and Alessandro D'Auria. Refinement strategies for polygonal meshes applied to adaptive VEM discretization, 12 2019.

L. Beirão da Veiga, F. Brezzi, L. D. Marini, and A. Russo. Virtual Element Methods for general second order elliptic problems on polygonal meshes, 2014.