POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Implicit Methods for the Eikonal Equation

Project for the course
"Advanced Programming for Scientific Computing"

Francesca Venturi 10671316, Francesco Fainello 10664609

Advisor: Prof. Luca Formaggia

Co-advisor: Prof. Anna Scotti
Martina Ciancarelli

Academic year: 2023-2024

1. Introduction

This project focuses on the development and implementation of efficient implicit methods for solving the Eikonal equation, a fundamental tool in modelling wave propagation and distance functions across various fields, including seismology, computer vision, and geospatial physics. The accurate solution of this equation is essential in contexts such as the simulation of subsurface geological structures, where understanding how waves propagate through different materials helps us make predictions about the Earth's subsurface and its dynamics. A key feature of this study is the shift from explicit methods, such as the fast marching method, to implicit numerical approaches based on variational formulations. These implicit techniques offer greater flexibility and robustness when dealing with complex boundary conditions and anisotropic media, making them more suitable for large-scale geophysical simulations. In particular, the variational formulation of the Eikonal equation enables the application of energy minimization principles, which can handle the nonlinearities and discontinuities often present in real-world geospatial models.

In the following sections, we will explore:

- Theoretical Background: A review of the Eikonal equation, its mathematical properties, and its variational formulation.
- Iterative Solutions: Methods used to solve the Eikonal equation iteratively, including the Laplacian iterations, the relaxation of this technique, and the Alternating Direction Method of Multipliers (ADMM).
- Implementation: A walkthrough of the key components of the code, including the mesh structure, initial conditions, and solver classes. Special attention will be given to the parallelization strategies employed to improve the computational efficiency of the solution.
- Results and Future Work: Presentation of the numerical results and potential future extensions of the project.

This study aims to demonstrate the effectiveness of implicit methods in solving the Eikonal equation and explore how such methods can be adapted to model complex physical phenomena in geospatial physics.

2. Theoretical Background

2.1. The Eikonal Equation

The eikonal equation is a fundamental first-order, nonlinear partial differential equation extensively applied in wave propagation problems across diverse areas, including geometrical optics, seismology, and physics. It offers a mathematical approach to compute distance functions within a domain, typically reflecting the time at which a wavefront arrives at a given point from a source. The equation establishes a connection between the gradient of the distance function, frequently referred to as the level set function, and the propagation speed of the wavefront through the medium. In its basic form, the eikonal equation is expressed as:

$$\|\nabla u(x)\| = F(x), \tag{1}$$

where $u(x)$ represents the distance function at a position x in the domain, $\nabla u(x)$ denotes the gradient of the function, and $F(x)$ corresponds to the speed of the wavefront at the point x.

In a broader context, the eikonal equation is utilized to model a variety of physical and computational phenomena, such as calculating shortest paths and minimal distances. The reason is that wave propagation speed is directly linked to the distance traversed, effectively converting the wavefront's arrival time into a distance function. As the wavefront moves through the medium, the distance from the source to any point in the domain can be determined, making the eikonal equation a powerful tool for constructing distance functions in numerous applications. Additionally, it proves to be particularly useful in modeling anisotropic propagation, where the wave speed depends on direction.

Let us consider a domain Ω in $\mathbb{R}^3$ and a function $u = u(\mathbf{x}), \mathbf{x} \in \Omega$. The eikonal equation is a special case of the nonlinear Hamilton–Jacobi partial differential equation (PDE) given by:

$$H(\mathbf{x}, u, \nabla u) = 1. \tag{2}$$

For the eikonal equation, the operator $H(\cdot, \cdot)$ simplifies to:

$$H(\mathbf{x}, u, \nabla u(\mathbf{x})) = \|\nabla u(\mathbf{x})\|_M = \sqrt{\nabla u(\mathbf{x})^T M(\mathbf{x}) \nabla u(\mathbf{x})}, \tag{3}$$

where $M(\mathbf{x})$ is a 3×3 symmetric positive-definite tensor, which encodes the speed information within the domain Ω.

From equation (2), we derive a nonlinear first-order PDE:

$$\begin{cases} \|\nabla u(\mathbf{x})\|_M = 1, & \mathbf{x} \in \Omega, \\ u(\mathbf{x}) = g(\mathbf{x}), & \mathbf{x} \in \Gamma \subset \partial\Omega, \end{cases} \tag{4}$$

where Γ represents a set of boundary conditions, and the function $g(\mathbf{x})$ is defined on Γ, representing the prescribed boundary data.

It is worth noting that equation (4) can be equivalently rewritten as equation (1) by taking $M = F^{-1}I$. Here, the term $F(\mathbf{x})$ represents a given positive speed function.

From a physical standpoint, the solution $u(\mathbf{x})$ to the eikonal equation can be interpreted as the time it takes for a wavefront, emitted from known sources inside the domain, to reach the point $\mathbf{x}$.

Moreover, the wavefronts can be interpreted as level sets of $u(\mathbf{x})$:

$$L_t = \{\mathbf{x} \in \Omega : u(\mathbf{x}) = t, t \in \mathbb{R}\}. \tag{5}$$

These level sets represent the location of the wavefront at "time" t.

One of the major challenges in solving this type of equation is that the solution might not be smooth, even with smooth boundary conditions. Due to the equation's nonlinear nature, the solution is often continuous but may lack differentiability.

2.2. Mathematical Details

After establishing the fundamental structure of the Eikonal equation, the next logical step is to delve into how this equation emerges from physical systems, particularly by interpreting it as a distance function or wavefront propagation. A deeper understanding naturally points to variational methods, which offer an energy-based perspective for solving such PDEs.

2.2.1 Basic Properties of Distance Functions

In practical applications, solutions to the eikonal equation facilitate efficient computation of distance functions, capturing the behavior of various physical phenomena as they propagate. These solutions are particularly crucial for understanding dynamic systems, especially in anisotropic or complex domains, such as those encountered in subsurface geological structures.

Let us assume that the boundary $\partial\Omega$ of the domain Ω is oriented by its inward normal $\mathbf{n}$. It is well known that the distance function $d(\mathbf{x})$ satisfies the eikonal equation:

$$\|\nabla d(\mathbf{x})\| = 1 \quad \text{in } \Omega. \tag{6}$$

Typically, the eikonal equation is used in combination with Dirichlet boundary conditions, which are expressed as:

$$\begin{cases} d = 0 \quad \text{on } \partial\Omega, \\ \dfrac{\partial d}{\partial n} = 1 \quad \text{on } \partial\Omega, \\ \dfrac{\partial^k d}{\partial n^k} = 0 \quad \text{on } \partial\Omega, \quad \forall k \geq 2. \end{cases} \tag{7}$$

It is also well known that the Laplacian of the distance function $d(\mathbf{x})$ is proportional to the mean curvature $H(\mathbf{x})$ of the level set $d = \text{const}$ passing through $\mathbf{x}$, which is expressed as:

$$\Delta d(\mathbf{x}) = (1 - m)H(\mathbf{x}) \tag{8}$$

where m is the number of dimensions, and it is assumed that the level set of $H(\mathbf{x})$ is smooth at the point $\mathbf{x}$. The mean curvature H is given by:

$$H = \frac{k_1 + \cdots + k_{m-1}}{m - 1} \tag{9}$$

where $k_1, \ldots, k_{m-1}$ represent the principal curvatures of the level set $d = \text{const}$. Since

$$(1 - m)H(\mathbf{x}) = \operatorname{div}(\mathbf{n}) \tag{10}$$

where $\mathbf{n}(\mathbf{x}) = \nabla d(\mathbf{x})$ is the unit inward normal to the level set of $d(\mathbf{x})$ at $\mathbf{x} \in \Omega$, we obtain the second-order nonlinear PDE:

$$\Delta d(\mathbf{x}) = \operatorname{div}\left(\frac{\nabla d(\mathbf{x})}{\|\nabla d(\mathbf{x})\|}\right) \tag{11}$$

which is equivalent to equation (8).

2.2.2 Euler-Lagrangian Equation for Energy Functional

In this section, we reframe the Eikonal equation in terms of energy minimization.

The Euler-Lagrange equation is a foundational tool in the calculus of variations, offering a powerful method to solve partial differential equations (PDEs) by minimizing a functional. Indeed, in many physical and engineering applications, PDEs often emerge from principles such as least action or minimum energy.

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$. In this context, the goal is often to find a function $u(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ that optimizes a functional, which is typically expressed as an integral:

$$E(u(\mathbf{x})) = \int_\Omega F(\mathbf{x}, u(\mathbf{x}), \nabla u(\mathbf{x}))\, d\mathbf{x}, \tag{12}$$

where $\nabla u(\mathbf{x})$ represents the gradient of $u(\mathbf{x})$ with respect to $\mathbf{x}$, and F is a given function of $\mathbf{x}, u(\mathbf{x})$, and $\nabla u(\mathbf{x})$. Specifically, if $E(u(\mathbf{x}))$ is to be optimized, the first variation of the functional must be zero. This leads to the Euler-Lagrange equation:

$$\frac{\partial E}{\partial u} = \frac{\partial F}{\partial u} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\left(\frac{\partial F}{\partial u_{x_i}}\right) = 0 \qquad \text{where } u_{x_i} = \frac{\partial u}{\partial x_i}. \tag{13}$$

The Euler-Lagrange equation provides a necessary condition for $u(\mathbf{x})$ to be an extremum of this functional of the "energy" functional $E(x(\mathbf{x}))$.

By reframing a PDE as the minimum of Lagrangian functional, variational methods become a natural approach to finding solutions. This can be particularly advantageous when directly solving the PDE is complex while minimizing the associated functional is more manageable. The Euler-Lagrange equation (13) not only provides a condition for minimization but also deepens our understanding of the underlying physical principles, such as conservation laws and symmetries.

This method is precisely how we approach solving the eikonal equation. Starting from the differential equation (11), we identify a functional that serves as the corresponding Euler-Lagrangian functional to optimize, leading to the following expression:

$$E(u(\mathbf{x})) = \int_\Omega (\|\nabla u(\mathbf{x})\| - 1)^2\, d\mathbf{x}. \tag{14}$$

In light of 14, it is immediate to introduce the Variational Formulation of the eikonal equation that is derived directly from the Euler-Lagrangian functional just obtained.

Assuming that the boundary condition is w = 0 on a measurable, non-null portion of the boundary, let $V = \{v \in H^1(\Omega), v|_{\partial\Omega} = 0\}$.

Then we just said that the solution of the PDE stems from the minimization problem:

$$w = \arg\min_{u \in V} E(u(\mathbf{x})) = \arg\min_{u \in V} \int_\Omega (\|\nabla u(\mathbf{x})\| - 1)^2\, d\mathbf{x}, \tag{15}$$

which can be easily generalized to the anisotropic case:

$$w = \arg\min_{u \in V} E(u(\mathbf{x})) = \arg\min_{u \in V} \int_{\Omega} \left(\|\nabla u(\mathbf{x})\|_M - 1 \right)^2 d\mathbf{x}, \tag{16}$$

where $M \in L^{\infty}(\Omega)$ is a symmetric positive definite tensor, independent of u. The anisotropy matrix M embodies the physical properties of the problem and the geometrical characteristics of the domain. In the following study, we will focus on diagonal geometry-independent frameworks, that is:

$$M = \begin{bmatrix} m_{11} & 0 & 0 \\ 0 & m_{22} & 0 \\ 0 & 0 & m_{33} \end{bmatrix}$$

with m_{11}, m_{22} and m_{33} potentially different.

The corresponding variational formulation to (16) is: Find $w \in V$ such that

$$(\nabla w, \nabla \varphi) - \left(\frac{1}{\|\nabla w\|_M} \nabla w, \nabla \varphi \right) = 0, \quad \forall \varphi \in V. \tag{17}$$

where $(\mathbf{a}, \mathbf{b}) = \int_{\Omega} \mathbf{a} \cdot \mathbf{b} \, d\Omega$.

Having covered how the variational approach enables more robust solutions for the eikonal equation, the next step would be discussing iterative methods.

3. Iterative Solutions of the Eikonal Equation

The variational formulation not only provides a solid mathematical foundation but also opens the door for advanced numerical techniques to solve the Eikonal equation. One of the most effective techniques for this purpose is the use of iterative solvers, which allow us to gradually approach the solution with improved precision at each step. In the following section, we focus on the details of iterative methods, specifically designed to solve variational problems like the Eikonal equation. The following theoretical findings are inspired by the main reference paper for this project: *On Variational and PDE-based Distance Function Approximations*, by A. G. Balyaev and P. Fayolle [1]

3.1. Method of Laplacian Iterations

To solve the second-order nonlinear PDE (11), a gradient normalization approach suggests a fixed-point iterative method for approximating the distance function:

$$\begin{cases} \Delta u_{k+1}(\mathbf{x}) = \operatorname{div}\left(\dfrac{\nabla u_k(\mathbf{x})}{\|\nabla u_k(\mathbf{x})\|}\right) & \text{in } \Omega \\ u_{k+1}(\mathbf{x}) = 0 & \text{on } \partial\Omega \end{cases} \tag{18}$$

This method can be understood as follows: given $u_k(\mathbf{x})$, the normalized gradient

$$n_k(\mathbf{x}) = \frac{\nabla u_k(\mathbf{x})}{\|\nabla u_k(\mathbf{x})\|} \tag{19}$$

aligns with the true normal $\nabla u(\mathbf{x})$ on $\partial\Omega$. On the other hand, $u_{k+1}(\mathbf{x})$ solves the minimization problem:

$$\int_\Omega \|\nabla u_{k+1}(\mathbf{x}) - n_k(\mathbf{x})\|^2 \, d\mathbf{x} \to \min, \tag{20}$$

which reduces the approximation error on $\partial\Omega$, while redistributing the remaining error throughout Ω.

3.2. Relaxation and Splitting of Lagrangian Iterations

Although iterative methods provide a powerful tool for solving the Eikonal equation, there are two key challenges associated with this approach:

- Difficulty in establishing rigorous convergence properties,
- In practice, a relatively slow convergence to the true distance function $d(\mathbf{x})$.

This is where the penalty formulation becomes useful, as it provides a way to maintain numerical stability and accelerate convergence. One effective approach to tackle these issues is to reformulate the energy functional as:

$$E(\mathbf{p}, u) = \int_\Omega \left(\frac{1}{2}(\|\mathbf{p}\| - 1)^2 + \frac{r}{2}(\mathbf{p} - \nabla u)^2\right) d\mathbf{x}, \tag{21}$$

where r is a positive constant and $\mathbf{p} = \nabla u$. This method introduces a penalty term that relaxes the strict constraint $\mathbf{p} = \nabla u$ by penalizing deviations from ∇u.

For a fixed ∇u, optimizing $E(\mathbf{p}, u)$ with respect to $\mathbf{p}$ yields $\mathbf{p}(\mathbf{x}) = c(\mathbf{x})\nabla u(\mathbf{x})$ for some scalar $c(\mathbf{x})$. A graphical illustration can be found in Fig. 1.

Substituting this expression into the energy functional and optimizing with respect to c leads to:

$$c = \frac{1 + r\|\nabla u\|}{(1 + r)\|\nabla u\|}, \quad \mathbf{p} = \frac{1 + r\|\nabla u\|}{(1 + r)\|\nabla u\|}\nabla u. \tag{22}$$

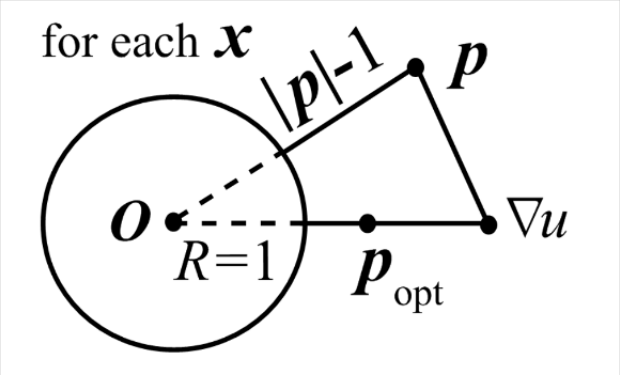

Figure 1: Geometrical intuition that suggests that the optimal $\mathbf{p}$ is proportional to ∇u. The figure is taken from [1].

Optimizing with respect to $u(\mathbf{x})$ (Appendix A) results in the equation:

$$\Delta u = \operatorname{div} \mathbf{p} \quad \text{in } \Omega, \tag{23}$$

This produces the following iterative scheme:

$$\begin{cases} \mathbf{p}_k = \dfrac{1 + r\|\nabla u_k\|}{(1+r)\|\nabla u_k\|} \nabla u_k, \\ \Delta u_{k+1} = \operatorname{div} \mathbf{p}_k. \end{cases} \tag{24}$$

This scheme reduces to the original Laplacian iteration (18) when $r = 0$ and can be viewed as a generalization of the method.

The convergence of this approach can be intuitively explained as follows:

- The first step, $\nabla u_k \to \mathbf{p}_k$, brings $\mathbf{p}_k$ closer to the unit sphere $\|\mathbf{p}\| = 1$ than ∇u_k,
- The second step, $\mathbf{p}_k \to \nabla u_{k+1}$, projects $\mathbf{p}_k$ orthogonally onto the space of gradients, moving ∇u_{k+1} closer to $\|\mathbf{p}\| = 1$.

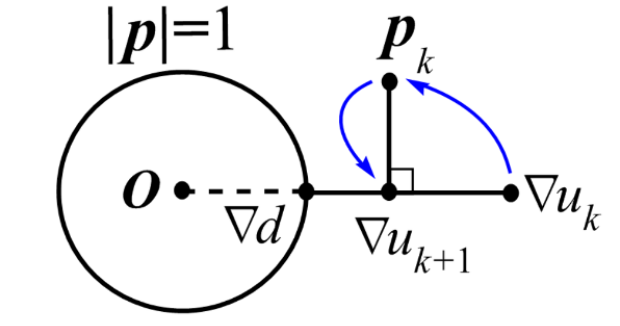

Figure 2: Geometrical interpretation of the iterative update 24. The figure is taken from [1].

3.3. Alternating Direction Method of Lagrange Multipliers (ADMM)

To more effectively handle the constraint $\mathbf{p} = \nabla u$ in the penalty method (21) while minimizing (14), the Alternating Direction Method of Multipliers (ADMM) can be applied. ADMM is a widely-used optimization technique that efficiently incorporates Lagrange multipliers into the minimization process. By introducing a Lagrange multiplier term, the penalty method leads to the following functional:

$$E(\mathbf{p}, u, \lambda) = \int_\Omega \left(\frac{1}{2}(\|\mathbf{p}\| - 1)^2 + \lambda(\mathbf{p} - \nabla u) + \frac{r}{2}(\mathbf{p} - \nabla u)^2 \right) d\mathbf{x}, \tag{25}$$

where $\lambda(\mathbf{x})$ is the Lagrange multiplier vector.

This expression can be reformulated as:

$$E(\mathbf{p}, u, \lambda) = \int_\Omega \left(\frac{1}{2}(\|\mathbf{p}\| - 1)^2 + \frac{r}{2}\left[\mathbf{p} - \left(\nabla u - \frac{\lambda}{r}\right)\right]^2 - \frac{\lambda^2}{2r} \right) d\mathbf{x}. \tag{26}$$

Thus, for fixed ∇u and λ, the optimal $\mathbf{p}$ is proportional to $\mathbf{q} = \nabla u - \frac{\lambda}{r}$, following the same reasoning as in Fig. 1. In other words,

$$\mathbf{p}(\mathbf{x}) = c(\mathbf{x})\mathbf{q}(\mathbf{x}) = c(\mathbf{x}) \left(\nabla u(\mathbf{x}) - \frac{\lambda(\mathbf{x})}{r} \right), \tag{27}$$

for some scalar $c(\mathbf{x})$. Substituting this into the functional and optimizing with respect to c yields:

$$c = \frac{1 + r\|\mathbf{q}\|}{(1 + r)\|\mathbf{q}\|}, \quad \mathbf{p} = \frac{1 + r\|\mathbf{q}\|}{(1 + r)\|\mathbf{q}\|}\mathbf{q}. \tag{28}$$

Optimizing with respect to $u(\mathbf{x})$ (similarly to Appendix A) leads to:

$$r\Delta u = r \operatorname{div} \mathbf{p} + \operatorname{div} \lambda. \tag{29}$$

Thus, we derive the iterative procedure:

$$\begin{cases} \mathbf{p}_k = \dfrac{1 + r\|\mathbf{q}_i\|}{(1 + r)\|\mathbf{q}_i\|}\mathbf{q}_i \quad \text{with } \mathbf{q}_i = \nabla u_k - \dfrac{\lambda_k}{r}, \quad \mathbf{q}_i = |q_i|, \\ \Delta u_{k+1} = \operatorname{div} \mathbf{p}_i + \dfrac{1}{r}\operatorname{div} \lambda_k \quad \text{in } \Omega, \quad u_{k+1} = 0 \quad \text{on } \partial\Omega, \\ \lambda_{k+1} = \lambda_k + r(\mathbf{p}_i - \nabla u_k). \end{cases} \tag{30}$$

4. Variational Formulations

Before presenting the discretization schemes, it is important to recall that all the following methods are derived from the variational formulation of the problem, as presented in (31). For clarity and completeness, we rewrite this formulation. Find $w \in V$ such that

$$(\nabla w, \nabla \varphi) - \left(\frac{1}{\|\nabla w\|_M} \nabla w, \nabla \varphi \right) = 0, \quad \forall \varphi \in V. \tag{31}$$

4.1. Method of Laplacian Iterations - Variational Formulation

The variational formulation of the eikonal equation leads to the first iterative scheme.

Given $w^{(0)} \in V$, with $\nabla w^{(0)} \neq 0$ in all of Ω, we approximate the solution by constructing the sequence:

$$(\nabla w^{(i+1)}, \nabla \varphi) = \left(\frac{1}{\|\nabla w^{(i)}\|_M} \nabla w^{(i)}, \nabla \varphi \right), \quad i = 0, 1, \ldots \tag{32}$$

This scheme, when reformulated incrementally, becomes:

$$\begin{cases} (\nabla z, \nabla \varphi) = \left(\dfrac{1 - \|\nabla w^{(i)}\|_M}{\|\nabla w^{(i)}\|_M + \gamma} \nabla w^{(i)}, \nabla \varphi \right), \\ w^{(i+1)} = w^{(i)} + z, \end{cases} \qquad i = 0, 1, \ldots \tag{33}$$

which is exactly the variational formulation of (18). Note that the division by very small values can be avoided by introducing a small $\gamma > 0$ as a stabilization strategy.

4.2. Relaxation - Variational Formulation

The iterative procedure (33) does not guarantee convergence in all cases. A more robust approach is derived from the penalized functional in (21), with $r > 0$ and $\mathbf{p} \in P = [L_2(\Omega)]^d$.

For fixed ∇u, the optimal $\mathbf{p}$ minimizing E is given by:

$$\mathbf{p} = \frac{r\|\nabla u\|_M + 1}{(1+r)\|\nabla u\|_M} \nabla u. \tag{34}$$

To discretize the relaxation method of the Laplacian iteration, we can show that

$$|\|\mathbf{p}\|_M - 1\| \leq |\|\nabla u\|_M - 1|,$$

which leads to the second iterative scheme. Given $w^{(0)} \in V$, with $\nabla w^{(0)} \neq 0$ in all of Ω, we perform:

$$\begin{cases} (\mathbf{p}^{(i+1)}, \mu) = \left(\dfrac{r\|\nabla w^{(i)}\|_M + 1}{(1+r)\|\nabla w^{(i)}\|_M} \nabla w^{(i)}, \mu \right), & \forall \mu \in P \\ (\nabla w^{(i+1)}, \nabla \varphi) = (p^{(i+1)}, \nabla \varphi), & \forall \varphi \in V, \end{cases} \qquad i = 0, 1, \ldots \tag{35}$$

Since $\nabla \varphi \in P$, we can eliminate $\mathbf{p}$ and write the iteration purely in terms of w:

$$(\nabla w^{(i+1)}, \nabla \varphi) = \left(\frac{r\|\nabla w^{(i)}\|_M + 1}{(1+r)\|\nabla w^{(i)}\|_M} \nabla w^{(i)}, \nabla \varphi \right), \quad \forall \varphi \in V. \tag{36}$$

It can be shown that this iteration converges in the sense that $\|\|\nabla w^{(i)}\|_M - 1\|$ decreases in $L^2(\Omega)$. The choice of r is crucial. This scheme can also be rewritten in incremental form:

$$\begin{cases} (\nabla z, \nabla \varphi) = \left(\dfrac{1 - \|\nabla w^{(i)}\|_M}{(1+r)\|\nabla w^{(i)}\|_M + \gamma} \nabla w^{(i)}, \nabla \varphi \right), & i = 0, 1, \dots \\ w^{(i+1)} = w^{(i)} + z, \end{cases} \tag{37}$$

4.3. ADMM - Variational Formulation

Finally, the augmented Lagrangian formulation introduces the Lagrange multipliers $\lambda \in P$, aiming to find the saddle point of the functional (25).

Given $w^{(0)} \in V$, with $\nabla w^{(0)} \neq 0$ on all Ω, and $\lambda^{(0)} \in P$, the third iterative scheme becomes:

$$\begin{cases} (\mathbf{p}^{(i)}, \mu) = \left(\dfrac{1 + r\|\mathbf{q}^{(i)}\|_M}{(1+r)\|\mathbf{q}^{(i)}\|_M} \mathbf{q}^{(i)}, \mu \right), & \forall \mu \in P \quad \text{where } \mathbf{q}^{(i)} = \nabla w^{(i)} - \dfrac{\lambda^{(i)}}{r} \\ (\nabla w^{(i+1)}, \nabla \varphi) = \left(\mathbf{p}^{(i)} + \frac{\lambda^{(i)}}{r}, \nabla \varphi \right), & \forall \varphi \in V, \quad i = 0, 1, \dots \\ (\lambda^{(i+1)}, \mu) = \left(\lambda^{(i)} + r(\mathbf{p}^{(i)} - \nabla w^{(i+1)}), \mu \right), & \forall \mu \in P \end{cases} \tag{38}$$

This formulation can be further simplified by eliminating $\mathbf{p}$. Setting

$$c^{(i)} = \frac{1 + r\|\mathbf{q}^{(i)}\|_M}{(1+r)\|\mathbf{q}^{(i)}\|_M}, \tag{39}$$

we obtain:

$$\begin{cases} (\nabla w^{(i+1)}, \nabla \varphi) = \left(c^{(i)} \nabla w^{(i)} + \dfrac{1 - c^{(i)}}{r} \lambda^{(i)}, \nabla \varphi \right), & \forall \varphi \in V, \quad i = 0, 1, \dots \\ (\lambda^{(i+1)}, \mu) = \left(\lambda^{(i)} + r\left(c^{(i)} \mathbf{q}^{(i)} - \nabla w^{(i+1)} \right), \mu \right), & \forall \mu \in P. \end{cases} \tag{40}$$

It can also be written in incremental form by setting $w^{(i+1)} = w^{(i)} + z$:

$$\begin{cases} (\nabla z, \nabla \varphi) = \left((c^{(i)} - 1) \nabla w^{(i)} + \dfrac{1 - c^{(i)}}{r} \lambda^{(i)}, \nabla \varphi \right), & \forall \varphi \in V \\ w^{(i+1)} = w^{(i)} + z, & \\ (\lambda^{(i+1)}, \mu) = \left(\lambda^{(i)} + r(c^{(i)} \mathbf{q}^{(i)} - \nabla w^{(i+1)}), \mu \right), & \forall \mu \in P, \end{cases} \qquad k = 1, 2, \dots \tag{41}$$

With some further manipulations, this becomes:

$$\begin{cases} (\nabla z, \nabla \varphi) = \left((c^{(i)} - 1) \nabla w^{(i)} + \dfrac{1 - c^{(i)}}{r} \lambda^{(i)}, \nabla \varphi \right), & \forall \varphi \in V \\ w^{(i+1)} = w^{(i)} + z, & \\ (\lambda^{(i+1)}, \mu) = \left((1 - c^{(i)}) \lambda^{(i)} - r c^{(i)} \nabla z, \mu \right), & \forall \mu \in P, \end{cases} \qquad k = 1, 2, \dots \tag{42}$$

4.4. Towards Discretization

The transition to the discrete setting is straightforward. Specifically, we set:

- V_h: the space of piecewise linear finite elements,
- P_h: the space of piecewise constant finite elements.

Note that with this choice, all operations involving test functions in P_h are, in fact, elemental operations. Instead, operations on the discrete w involve assembling the stiffness matrix.

5. Implementation

In this section, we will walk through the codebase developed for this project, detailing the key components and classes implemented to solve the eikonal equation.

The project is built around the `Eigen` library. `Eigen` is a high-performance `C++` library designed for linear algebra operations, including matrices, vectors and numerical solvers. This library can allocate algebraic objects in memory statically or dynamically. `Eigen` is widely used thanks to its flexibility: being a header-only library, it templates extensively, meaning developers can work with different scalar types like `int`, `float` and `double`, and even custom types. Moreover, in addition to the more standard management of dense matrices, it also provides specialized modules for handling sparse matrices, allowing efficient storage and operations for matrices with many zero elements. Indeed, `Eigen` minimizes unnecessary memory allocations and efficiently handles large-scale matrix operations, which is crucial for high-performance applications. To conclude, being header-only, it is cross-platform and does not require installation.

To manage the various data structures involved, we have implemented a dedicated traits file, which defines the types and containers needed for representing elements of the mesh, the nodes, and other key quantities. This traits system, built within Eigen's capabilities, allows for flexible handling of data structures, providing both efficiency and clarity when managing operations on nodes, values, and connectivity in different dimensional spaces. Here are some key components of data management:

- Physical and Intrinsic Dimensions: The file is designed to handle both the physical dimension (`PHDIM`) of the embedding space and the intrinsic dimension (`INTRINSIC_DIM`) of the manifold where the Eikonal equation is solved. The intrinsic dimension is less than or equal to the physical dimension, which accommodates scenarios where the equation is solved on a lower-dimensional manifold embedded in a higher-dimensional space (e.g., a 2D surface in 3D space). This scenario, however, does not represent a challenge faced in this project.

- Index Types:

```
1     using Index = long int;
2     using Indexes = std::vector<Index>;
```

  `Index` and `Indexes` are defined to handle the indexing of nodes and elements. This is essential when working with elements of a mesh or structures that require addressing nodes by index.

- Node and Coordinate Representations:

```
1     using Node = Eigen::Matrix<double, PHDIM, 1, Eigen::ColMajor>;
2     using Nodes = Eigen::Matrix<double, PHDIM, Eigen::Dynamic, Eigen::ColMajor>;
```

  `Node` and `Nodes` use Eigen matrices to represent the coordinates of individual nodes and sets of nodes, respectively. Specifically, `Eigen::Matrix` is used for both fixed-size (`Node`) and dynamic-size (`Nodes`) matrices, with `Node` being a column vector representing the coordinates of a node in a space. `Nodes` is a dynamic matrix where each column corresponds to the coordinates of a node, allowing for flexible handling of a variable number of nodes.

- Values:

```
1     using Values = Eigen::Matrix<double, Eigen::Dynamic, 1>;
```

  `Values` is defined as a dynamic column vector, used to store scalar values (e.g., solution values or other quantities) at the nodes of the mesh.

- Anisotropy Matrix:

```
1     using AnisotropyM = Eigen::Matrix<double, PHDIM, PHDIM>;
```

`AnisotropyM` is a matrix used to store the anisotropy information, which is crucial when the Eikonal equation involves direction-dependent properties (e.g., wave propagation speed in different directions). It is represented as a fixed-size matrix of size `PHDIM x PHDIM`, which fits well with Eigen's efficient handling of small, fixed-size matrices.

The following sections will present, one by one, the classes developed for the project:

- *Mesh Class.* This class forms the foundation for discretizing the domain and solving the Eikonal equation. We will explore the methods dealing with mesh generation, connectivity, and manipulation.
- *Initial Conditions Class.* A class dedicated to setting up the initial conditions for the problem, ensuring that the solution begins with the correct data.
- *Eikonal Solver Class.* In this class, the solution to the Eikonal equation is implemented. We will cover the key algorithms, iterative methods, and the handling of anisotropy and boundary conditions.

Additionally, one subsection will focus on the parallelization strategies we adopted to improve the performance and scalability of the solver. Finally, we will conclude with a discussion of the structure and functionality of the `main.cpp`, which ties all the components together and drives the overall execution of the program.

5.1. Mesh reading, problem initialization and saving of the solutions

Many preliminary operations and functions are handled in the `MeshData.hpp` file. This defines a class `MeshData` which is deputed primarily to reading the data for the mesh contained in `.vtk` files and provides functionality for manipulating mesh-related information, such as nodes, elements, and boundary conditions. Filling of the global and local matrices for the problem is handled by the class, as well as saving of the solutions to `.vtk` files after convergence has been obtained by the iterative schemes.

Public members of the class include:

- This member stores the gradient coefficient matrices for each element in the mesh. The matrix size is determined by `PHDIM` (the physical dimension of the mesh). The vector holds one matrix per element, making it crucial for local calculations

```
std::vector<Eigen::Matrix<double, PHDIM, PHDIM>> gradientCoeff
```

- This member stores the local stiffness matrices for each element. Each matrix is of size `(PHDIM+1) x (PHDIM+1)` and represents the local contribution to the global stiffness matrix, which arises from the discretization of the governing equations over each finite element.

```
std::vector<Eigen::Matrix<double, PHDIM+1, PHDIM+1>> localStiffnessMatrices
```

- To compute the term $(\psi, \nabla\phi)$, where $\psi \in P_h$ and $\phi \in V_h$ it was necessary to create what we refer to as "reaction matrices." Despite the potentially misleading name, these are actually tensors with dimensions `n x n x PHDIM`, where n refers to the number of nodes of the mesh. Unlike stiffness and reaction matrices, which result from the integration of elements with the same dimensionality ((ϕ_i, ϕ_j) and $(\nabla\phi_i, \nabla\phi_j)$), this does not occur in the case of the reaction matrix, which is why a tensor is required. Given the limitation of not being able to define tensors with the Eigen library, the strategy used was to define a vector of vectors of vectors (specifically of `Eigen::Matrix<double, PHDIM, 1>`).
  To this, we add an outer layer of vectors that contains one of these tensors for each element of the mesh.

```
std::vector<std::vector<std::vector<Eigen::Matrix<double, PHDIM, 1>>>>
    localReactionMatrices
```

These variables are crucial in assembling the right hand side of all equations solved in the iterative schemes at the element level, and are made public for simpler broadcasting between ranks.

Private members of the class include:

- This member stores the coordinates of all the nodes in the mesh. Each column in the matrix represents the coordinates of a node.

```
        Nodes nodes
```

- This member represents the connectivity of the elements in the mesh. Each column of this matrix contains the indices of the nodes that make up an element, defining how nodes are connected to form elements in the mesh.

```
        Elements connectivity
```

- This member stores the total number of elements in the mesh.

```
        int numElements
```

- This member stores the total number of nodes in the mesh.

```
        int numNodes
```

- This member stores the indices of nodes that are on the boundary of the mesh.

```
        Indexes boundaryNodes
```

In the following we describe the methods of the class.

5.1.1 readMesh

```
    void readMesh(const std::string& filename) {
        // Read mesh from file
        std::ifstream file(filename);
        if (!file) {
          std::cerr << "Failed to open file: " << filename << std::endl;
          return;
        }

        // Read element connectivity and nodes
        std::string line;
        std::istringstream ss;
        while (std::getline(file, line)) {
            if (line.find("CELLS") != std::string::npos) {
                // extract the substring starting from the first digit
                std::string numElementsStr =
                    line.substr(line.find_first_of("0123456789"));
                // put the string in a stringstream
                ss.str(numElementsStr);
                ss.seekg(0, std::ios::beg);
                // read the number of elements
                ss >> numElements;
                connectivity.resize(4, numElements);
                for (int i = 0; i < numElements; i++) {
                int dummy;
                std::getline(file, line);
                ss.str(line);
                ss.seekg(0, std::ios::beg);
                ss >> dummy;
                for (int j = 0; j < 4; j++) {
                    ss >> connectivity(j, i);
                }
                }
            } else if (line.find("POINTS") != std::string::npos) {
                std::string numNodesStr = line.substr(line.find_first_of("0123456789"));
                ss.str(numNodesStr);
                ss.seekg(0, std::ios::beg);
```

```
                ss >> numNodes;
                nodes.resize(3, numNodes);
                for (int i = 0; i < numNodes; i++) {
                std::getline(file, line);
                ss.str(line);
                ss.seekg(0, std::ios::beg);
                ss >> nodes(0, i) >> nodes(1, i) >> nodes(2, i);
                }
            }
        }

        // Reserve memory for vectors
        gradientCoeff.reserve(numElements);
        localStiffnessMatrices.reserve(numElements);
        localReactionMatrices.reserve(numElements);
    }
```

The `readMesh` method is designed to initialize the mesh by reading data from a file containing the node coordinates and the connectivity of elements. The method starts by opening the file specified by `fileName` and proceeds to read the data. The first part reads the coordinates of each node and stores them in the `nodes` matrix. The second part of the function handles the connectivity of the elements and stores the information in the `connectivity` matrix. Finally, the method updates the total number of nodes `numNodes` and elements `numElements` in the mesh to match the data read from the file. This method directly modifies internal data members such as `nodes`, `connectivity`, `numNodes`, and `numElements`.

5.1.2 fillGlobalVariables

```
    void fillGlobalVariables(Eigen::SparseMatrix<double>& stiffnessMatrix,
                        std::vector<std::vector<Eigen::Matrix<double, PHDIM, 1>>>&
                            reactionMatrix,
                        Values& globalIntegrals) {

        Nodes localNodes(PHDIM, PHDIM+1);
        Indexes globalNodeNumbers(PHDIM+1);

        apsc::LinearFiniteElement<PHDIM> linearFiniteElement;

        for (int k = 0; k < getNumElements(); ++k) {
            for (int i = 0; i < PHDIM+1; ++i) {  // Node numbering
                globalNodeNumbers[i] = getConnectivity()(i, k);
                for (int j = 0; j < PHDIM; ++j) {  // Local node coordinates
                    localNodes(j, i) = getNodes()(j, getConnectivity()(i, k));
                }
            }

            // Compute local nodes and update global node numbers
            linearFiniteElement.update(localNodes);
            linearFiniteElement.updateGlobalNodeNumbers(globalNodeNumbers);

            // Compute integrals and update global matrices
            linearFiniteElement.computeLocalIntegral();
            linearFiniteElement.updateGlobalIntegrals(globalIntegrals);

            // Compute stiffness and update global stiffness matrix
            linearFiniteElement.computeLocalStiffness();
            linearFiniteElement.updateGlobalStiffnessMatrix(stiffnessMatrix);

            // Compute the local reaction matrix and update the global reaction matrix
            linearFiniteElement.computeLocalReaction();
            linearFiniteElement.updateGlobalReactionMatrix(reactionMatrix);

            // Compute gradient coefficients
```

```
            gradientCoeff.push_back(linearFiniteElement.computeGradientCoeff());

            // Compute local matrices
            localStiffnessMatrices.push_back(linearFiniteElement.computeLocalStiffness());
            localReactionMatrices.push_back(linearFiniteElement.computeLocalReaction());
        }

        // Impose Dirichlet boundary conditions on the stiffness matrix
        linearFiniteElement.updateMatrixWithDirichletBoundary(stiffnessMatrix,
            getBoundaryNodes());
    }
```

The `fillGlobalVariables` method is responsible for setting up all global variables and data structures required for the finite element computation. This method ensures that all auxiliary variables that depend on the mesh configuration are computed and stored. These variables include various matrices and vectors needed for the numerical solution process, such as global and local stiffness matrices, gradient coefficient matrices, global and local reaction matrices.

The method begins by iterating over each element in the mesh. It computes the gradient coefficient matrices `gradientCoeff` for each element. This involves assembling local derivatives of shape functions with respect to the global coordinate system. Next, it computes the local stiffness matrices `localStiffnessMatrices` for each element. The stiffness matrix entries are computed by integrating shape functions over each element. It also calculates the local reaction matrices `localReactionMatrices`.

The method modifies internal data structures such as `gradientCoeff`, `localStiffnessMatrices`, and `localReactionMatrices`, while other data structures such as `stiffnessMatrix`, `reactionMatrix` and `globalIntegrals` are passed by reference and updated.

5.1.3 addScalarField

```
    void addScalarField(const Eigen::Matrix<double, Eigen::Dynamic, 1>& values,
                        const std::string& inputFilePath,
                        const std::string& outputFilePath) {

        if (values.size() != numNodes) {
            throw std::invalid_argument("The size of values must match the number of nodes
                in the mesh.");
        }

        // std::string outputFilePath = inputFilePath.substr(0, inputFilePath.find_last_of
            ('.')) + "_" + bc + "_" + ic + iterativeMethod + ".vtk";

        // Open the input file and create the output file
        std::ifstream inputFile(inputFilePath);
        std::ofstream outputFile(outputFilePath);

        if (!inputFile.is_open()) {
            std::cerr << "Error: Could not open input file " << inputFilePath << std::endl;
            return;
        }

        if (!outputFile.is_open()) {
            outputFile.open(outputFilePath);
            std::cerr << "Error: Could not open output file " << outputFilePath << std::
                endl;
            return;
        }

        // Copy the contents of the input file to the output file
        std::string line;
```

```
        while (std::getline(inputFile, line)) {
            outputFile << line << std::endl;
        }

        // Append the scalar field data to the new file
        outputFile << "POINT_DATA " << numNodes << std::endl;
        outputFile << "SCALARS sol float" << std::endl;
        outputFile << "LOOKUP_TABLE default" << std::endl;
        for (int i = 0; i < values.size(); ++i) {
            outputFile << values(i) << std::endl;
        }

        inputFile.close();
        outputFile.close();
    }
```

The `addScalarField` method is used to introduce a new scalar field into the mesh, which represents a physical quantity. This method extends the class's functionality by allowing multiple fields to be added to the mesh, providing a versatile framework for multi-physics simulations.

The scalar field values are stored in a vector that matches the `Values` data type, which is defined as an `Eigen::Matrix<double, Dynamic, 1>`, representing a column vector where each entry corresponds to a node in the mesh. The method then writes the data appending to the `.vtk` file which represented the information about the mesh. This is done to easily represent the field using `Paraview`.

5.1.4 updateBoundaryNodes

```
    void updateBoundaryNodes(const int& boundaryCondition) {
        std::map<std::set<int>, int> faceCount;
        for (int i = 0; i < numElements; ++i) {
            // Each element has 4 faces
            for (int j = 0; j < 4; ++j) {
                std::set<int> face;
                for (int k = 0; k < 4; ++k) {
                    if (k != j) {
                        face.insert(connectivity(k, i)); // Assuming connectivity is a 2D
                            array or similar structure
                    }
                }
                faceCount[face]++;
            }
        }

        std::set<int> boundaryNodesSet;
        for (const auto& entry : faceCount) {
            if (entry.second == 1) { // Boundary face
                for (int node : entry.first) {
                    boundaryNodesSet.insert(node);
                }
            }
        }

        boundaryNodes.clear();
        switch (boundaryCondition) {
            case 1:
                // Include only nodes with z=0
                for (int node : boundaryNodesSet) {
                    if (nodes(2, node) == 0) { // Assuming node position z is at index 2 in
                         nodes
                        boundaryNodes.push_back(node);
                    }
                }
```

```
                break;
            case 2:
                // Include only the center node of the face with z=0
                {
                    std::vector<int> z0Nodes;
                    for (int node : boundaryNodesSet) {
                        if (nodes(2, node) == 0) { // Check for z=0 nodes
                            z0Nodes.push_back(node);
                        }
                    }

                    if (!z0Nodes.empty()) {
                        // Calculate the geometric center by averaging coordinates
                        double centerX = 0.0, centerY = 0.0;
                        for (int node : z0Nodes) {
                            centerX += nodes(0, node); // X coordinate
                            centerY += nodes(1, node); // Y coordinate
                        }
                        centerX /= z0Nodes.size();
                        centerY /= z0Nodes.size();

                        // Find the node closest to the calculated center
                        int centerNode = z0Nodes[0];
                        double minDistance = std::numeric_limits<double>::max();
                        for (int node : z0Nodes) {
                            double distance = std::sqrt(std::pow(nodes(0, node) - centerX,
                                2) + std::pow(nodes(1, node) - centerY, 2));
                            if (distance < minDistance) {
                                minDistance = distance;
                                centerNode = node;
                            }
                        }
                        boundaryNodes.push_back(centerNode);
                    }
                }
                break;
            case 3:
                // Include only the vertex at the origin (0, 0, 0)
                for (int node : boundaryNodesSet) {
                    if (nodes(0, node) == 0 && nodes(1, node) == 0 && nodes(2, node) == 0)
                        {
                        boundaryNodes.push_back(node);
                        break; // No need to search further, there is only one origin
                    }
                }
                break;
            case 4:
                // Include all boundary nodes
                boundaryNodes.assign(boundaryNodesSet.begin(), boundaryNodesSet.end());
                break;
            default:
                std::cerr << "Invalid option. Please run the program again with a valid
                    choice.\n";
                return; // Exit with an error code
            }
    }
```

The `updateBoundaryNodes` method updates the list of boundary nodes for the mesh depending on the specific boundary conditions that have to be applied. This method assumes that the domain is cubic, with one vertex in the origin and a face on the plane $z = 0$. The condition considered is passed as an `int` among different choices prompted to the user.

First of all a `set` containing the indices of all boundary nodes of the cube is filled. Next, depending on

the input choice, the vector `boundaryNodes` is filled by extracting the necessary nodes from the set. The different options are:

1. Boundary conditions apply only to the bottom face ($z = 0$)
2. The center of the bottom face is the only node where the boundary conditions apply
3. The vertex in the origin is the only node where the boundary conditions apply
4. Boundary conditions apply to all boundary nodes

5.2. Initial solution computation

```
class InitialConditions {
    public:
        using Traits = Eikonal::Eikonal_traits<PHDIM, INTRINSIC_DIM>;
        using Values = typename Traits::Values;
        using Indexes = typename Traits::Indexes;

        InitialConditions() = default;

        Values HeatEquation(const Eigen::SparseMatrix<double>& stiffnessMatrix,
                            const Values& Integrals,
                            const Values& forcingTerm,
                            const Indexes& boundaryIndices) const {

            // Right-hand side vector of the equation, all ones
            Values rhs = forcingTerm * Integrals;

            // Apply null Dirichlet boundary conditions
            for (int idx : boundaryIndices) {
                rhs[idx] = 0.0 * 1e40;
            }

            // Compute the solution
            Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;

            solver.compute(stiffnessMatrix);
            Values solution = solver.solve(rhs);

            // Return the computed values
            return solution;
        }

    private:

    };
```

Computation of the initial solution for the iterative methods is handled by `InitialConditions.hpp`. Specifically, it computes the solution to the heat equation to initialize the solution close enough to the desired solution of the eikonal equation. The primary method of the class is in fact `HeatEquation`, solving using a sparse linear solver `Eigen::SparseLU`. The method takes the global stiffness matrix, integrals at each node for the forcing term, a forcing term which is generally made up of constant values, and a vector representing the nodes where Dirichlet boundary conditions are applied as inputs.

The function constructs a right-hand side vector for the linear system based on the forcing term and integrals provided. It then applies Dirichlet boundary conditions by zeroing out corresponding entries in the right-hand side vector, effectively enforcing the boundary conditions where necessary. The LU decomposition is performed on the sparse stiffness matrix to solve for the nodal values that represent the solution to the heat equation.

5.3. Implicit Methods for the Eikonal Equation

In this paragraph we present the class which is intended to manage the actual solution of the problem. The class structure is designed with an abstract base class (`EikonalEquation`) that defines the common framework and methods required for solving the eikonal equation using linear finite elements. This abstract class serves as a foundation for specialized derived classes that implement the specific details of the three methods introduced in Sections 3 and 4. These methods are implemented in the derived classes (`StandardEikonal`, `PenaltyEikonal`, and `LagrangianEikonal`), each corresponding to a specific variational formulation of the implicit iterative schemes in their incremental form. Indeed, this design decision arises from the variational formulations of the implicit iterative schemes discussed in the theoretical section. In particular, the structure reflects equations (33) for `StandardEikonal`, (37) for `PenaltyEikonal`, and (42) for `LagrangianEikonal`. By using inheritance, we ensure code reusability and clarity, while also allowing each method to implement its own version of the stiffness term, reaction term, and solution update process based on its formulation.

5.3.1 Eikonal Equation - Parent Class

Protected Section

In the `EikonalEquation` class, the attributes are placed in the `protected` section to ensure that derived classes have access to the core variables and can customize or extend their behaviour. Key attributes are:

- `mesh` defines the domain and connectivity (please refer to Section 5.1 for further details).

```
    MeshData<PHDIM> mesh;
```

- The global right-hand side (`rhs`) accumulates contributions from all elements to solve the linear problem, the vector `w` which stores the solution of the equation at each iteration, for each node of the mesh, and the incremental update `z` that adjusts the solution in each iteration.

```
    Values rhs; // Global RHS
    Values& w; // Solution
    Values z; // Incremental update
```

- Additional attributes, such as the `stiffnessMatrix`, the sparse `solver`, and the convergence tolerance (`tol`) provide essential components for finite element computations.

```
    const Eigen::SparseMatrix<double>& stiffnessMatrix;
    const Eigen::SparseLU<Eigen::SparseMatrix<double>>& solver;
    double tol; // Convergence tolerance
```

- Local variables, like `local_w`, `localStiffness`, and `local_rhs`, manage element-wise data during iterations and are overwritten to save memory during computations, together with the current `linearFiniteElement`.

```
    Values local_w;
    Eigen::SparseMatrix<double> localStiffness;
    Values local_rhs;
    Eigen::Matrix<double, Eigen::Dynamic, PHDIM> grad_w;
    apsc::LinearFiniteElement<PHDIM> linearFiniteElement;
```

  On the other hand, the `grad_w` matrix stores gradients for each element as they are needed for plotting final results.

- The anisotropy matrix is stored for the anisotropic norm ($\|\cdot\|_M$) computation.

```
    AnisotropyM M;
```

Moreover, in the `protected` section of the parent class `EikonalEquation`, there are virtual methods to be overridden and the method serving for the computation of the anisotropic norm ($\|\cdot\|_M$). This method is in the `protected` section as it is explicitly called by derived classes.

```
    // Virtual functions to be implemented by derived classes
    virtual Values computeStiffnessTerm(int i) = 0;
    virtual Values computeReactionTerm(int i) = 0;
    virtual void updateLagrangians(const Values& z) = 0;

    // Computes the anisotropic norm of a vector using the matrix M
    double anisotropicNorm(const Eigen::Matrix<double, 1, PHDIM>& u) const {
        return std::sqrt((u * M * u.transpose()).value());
    }
```

Finally, since the base class is abstract, it is not meant to be initialized if not via one of the derived classes. For this reason, we can put the constructor in the `protected` section.

```
    // Constructor initializing mesh data, values, and solver
    EikonalEquation(const MeshData<PHDIM>& mesh,
                    Values& w,
                    const Eigen::SparseMatrix<double>& stiffnessMatrix,
                    const Eigen::SparseLU<Eigen::SparseMatrix<double>>& solver,
                    double gamma = 1e-3,
                    double tol = 1e-3)
        : mesh(mesh),
          w(w),
          stiffnessMatrix(stiffnessMatrix),
          solver(solver),
          gamma(gamma),
          tol(tol),

          z(mesh.getNumNodes()),
          local_w(PHDIM+1),
          localStiffness(PHDIM+1, PHDIM+1),
          local_rhs(PHDIM+1),
          grad_w(this->mesh.getNumElements(), PHDIM),
          rhs(Values::Zero(mesh.getNumNodes())),
          linearFiniteElement(),
          localNodes(PHDIM, PHDIM+1),
          globalNodeNumbers(PHDIM+1),
          M(1.0 * Eigen::Matrix<double, PHDIM, PHDIM>::Identity())
          {}
```

Private Section

Hereafter, you can find `private` methods of the class: they are used only within the class, and they represent part of the core methods for the solution of the eikonal equation.

- The `reset_attributes` method is responsible for resetting key variables at the beginning of each update cycle. This ensures that the local variables used for computations, such as `local_w`, `localStiffness`, and `local_rhs`, are set to zero. Additionally, it resets `rhs` and `grad_w` to zero as well, preparing them to accumulate new values. The method also initializes a local finite element object and resets the local nodes and global node numbers to their initial states. This reset is crucial to prevent any leftover data from previous iterations from affecting the current update.

```
    // Resets attributes before each update cycle
    void reset_attributes() {

        local_w.setZero();
        localStiffness.setZero();
        local_rhs.setZero();

        grad_w.setZero();
        rhs.setZero();

        apsc::LinearFiniteElement<PHDIM> linearFiniteElement;
```

```
12
13          localNodes.setZero();
14          globalNodeNumbers.resize(globalNodeNumbers.size(), 0);
15      }
```

- The `computeLocalRhs` method calculates the local right-hand side for a given element in the mesh. It begins by collecting the local values of the solution vector (`w`) associated with the nodes of the element in `local_w`. These local values are necessary to compute the gradient of the solution within the element, which is done by leveraging the gradient coefficients associated with the mesh. Once the local gradient is calculated, the method computes the stiffness term and the reaction term, which represent the contributions of the finite element model to the governing equations. Refer to the specific derived class for the implementation of the methods `computeStiffnessTerm` and `computeReactionTerm`. The sum of these terms is then stored in the `local_rhs`, which will later be used to update the global right-hand side.

```
1       // Computes the local right-hand side (rhs) for the i-th element
2       void computeLocalRhs(int i) {
3
4           // Collect local w values
5           for (int j = 0; j < PHDIM+1; j++) {
6               local_w(j) = w(mesh.getConnectivity().col(i)(j));
7           }
8
9           // Compute local gradient
10          grad_w.row(i) = apsc::computeGradient<PHDIM>(this->mesh.getGradientCoeff()[i],
                local_w);
11
12          // Compute stiffness term and reaction term
13          Values stiffnessTerm = computeStiffnessTerm(i);
14          Values reactionTerm = computeReactionTerm(i);
15
16          // Compute local rhs
17          local_rhs = stiffnessTerm + reactionTerm;
18      }
```

- The `updateGlobalRhs` method takes the local right-hand side (`local_rhs`) computed for an element and assembles its values into `rhs`.

```
1       // Updates the global right-hand side with local rhs values
2       void updateGlobalRhs(int i) {
3           for (int j = 0; j < PHDIM+1; ++j) {
4               // Accumulate local rhs into global rhs
5               rhs(mesh.getConnectivity().col(i)(j)) += local_rhs(j);
6           }
7       }
```

Public Section

The `public` section of this class provides the primary interface for interacting with the solver. It contains two key methods: `updateSolution` and `getMetrics`, that are used in the `main` function.

- The `updateSolution` is the core function of the project. It is responsible for assembling the right-hand side (RHS) of the linear system, solving that system, and updating the solution. After resetting the necessary internal attributes `reset_attributes`, it proceeds to loop over the elements of the mesh, computing the local contributions to the rhs via the `computeLocalRhs` and `updateGlobalRhs` functions. These two methods ensure the system is ready to be solved. The method then handles boundary conditions, particularly Dirichlet conditions, by modifying the entries in the rhs corresponding to boundary nodes. Next, the method proceeds to solve the linear system using the preconfigured solver. The result, `z`, is the incremental update to the solution vector `w`. A diagnostic message is printed to the console, showing the norm of `z`, which gives an indication of convergence. As a stopping criterion, if the norm of `z` falls below a given tolerance (`tol`), it signifies that the system is

indeed convergence. Finally, the method updates any associated Lagrangian multipliers through the `updateLagrangians`, if the derived method used accounts for it.

```
    bool updateSolution() {

        reset_attributes();
        for(int i = 0; i < mesh.getNumElements(); i++) {
            computeLocalRhs(i);
            updateGlobalRhs(i);
        }

        // Update for Dirichlet BC. If == 0, set with value*TGV, where TGV=1e40
        for (int idx : mesh.getBoundaryNodes()) {
            rhs(idx) = 0.0 * 1e40;
        }

        // Solve the lienar system
        Values z = solver.solve(rhs);

        // Update the solution
        w += z;
        std::cout << "z : " << z.norm() << std::endl;

        updateLagrangians(z);

        return (z.norm() < tol);
    }
```

As far as parallelization is concerned, this method undertakes a few adjustments. Further details in Section 5.4.

- `getMetrics` offers a convenient way to retrieve important values, for later visualization purposes. add

5.3.2 StandardEikonal - Derived Class

The `StandardEikonal` class extends the base `EikonalEquation` class, implementing the Method of Laplacian Iterations for the eikonal equation. This class is designed to inherit the attributes and methods described in the parent class and provide implementations of the key virtual methods.

- The constructor takes full advantage of the base class constructor.

```
    // The StandardEikonal uses the EikonalEquation constructor
    StandardEikonal(MeshData<PHDIM>& mesh,
                    Values& w,
                    const Eigen::SparseMatrix<double>& stiffnessMatrix,
                    const Eigen::SparseLU<Eigen::SparseMatrix<double>>& solver)
        : EikonalEquation<PHDIM>(mesh, w, stiffnessMatrix, solver) {}
```

- The first method, `computeStiffnessTerm`, is responsible for calculating the stiffness term for a given element in the mesh. The implementation of the stiffness term follows exactly the iterative scheme (33).

```
    // Compute the stiffness term for the i-th element
    Values computeStiffnessTerm(int i) override {
        Eigen::Matrix<double, 1, PHDIM> grad = this->grad_w.row(i);
        double stiffnessCoeff = (1.0 - this->anisotropicNorm(grad)) / (this->
            anisotropicNorm(grad) + this->gamma);
        return stiffnessCoeff * (this->mesh.getLocalStiffnessMatrices()[i] * this->
            local_w);
    }
```

- The `computeReactionTerm` method returns a zero vector in this specific implementation, as there is no reaction term in this formulation. The same holds for `updateLagrangians`.

```
    // Not used in the StandardEikonal
    Values computeReactionTerm(int i) override {
        return Values::Zero(PHDIM+1);
    }

    // Not used in the StandardEikonal
    void updateLagrangians(const Values& z) override { return; }
```

5.3.3 PenaltyEikonal - Derived Class

The `PenaltyEikonal` implements a relaxed version of the Method of Laplacian Iterations, following closely the iterative scheme (37). This approach introduces a penalty parameter, `r`, which adjusts how the solution evolves over iterations and it is stored in the `private` section of the class.

- The constructor of the `PenaltyEikonal` class initializes the base class with the mesh, solution vector w, stiffness matrix, and solver. It also takes an additional parameter, r, which represents the penalty factor that modifies how the stiffness term is computed.

```
    // Constructor for Penalty Eikonal
    PenaltyEikonal(MeshData<PHDIM>& mesh,
                   Values& w,
                   const Eigen::SparseMatrix<double>& stiffnessMatrix,
                   const Eigen::SparseLU<Eigen::SparseMatrix<double>>& solver,
                   double r)
        : EikonalEquation<PHDIM>(mesh, w, stiffnessMatrix, solver), r(r) {
            std::cout << "r_penalty: " << r << std::endl;
        }
```

- The `computeStiffnessTerm` method is responsible for calculating the stiffness term for a given element in the mesh. The implementation of the stiffness term follows exactly the iterative scheme 37.

```
    // Compute the stiffness term for the i-th element
    Values computeStiffnessTerm(int i) override {
        Eigen::Matrix<double, 1, PHDIM> grad = this->grad_w.row(i);
        double stiffnessCoeff = (1.0 - this->anisotropicNorm(grad)) / ((1.0 + this->r)
             * this->anisotropicNorm(grad) + this->gamma);
        return stiffnessCoeff * (this->mesh.getLocalStiffnessMatrices()[i] * this->
            local_w);
    }
```

- Similarly to `StandardEikonal` The `computeReactionTerm` method returns a zero vector in this specific implementation, as there is no reaction term in this formulation. The same holds for `updateLagrangians`.

```
    // Not used in the StandardEikonal
    Values computeReactionTerm(int i) override {
        return Values::Zero(PHDIM+1);
    }

    // Not used in the StandardEikonal
    void updateLagrangians(const Values& z) override { return; }
```

5.3.4 LagrangianEikonal - Derived Class

The `LagrangianEikonal` implements the more advanced approach of Alternating Direction Method of Lagrange Multipliers (ADMM).

- In the constructor, the class takes an additional matrix of Lagrangian multipliers (`lagrangians`), along with thealready introduced standard inputs. Once again, `r` and `lagrangians` are stored in the `private` section of the class.

```
    // Constructor for Lagrangian Eikonal
    LagrangianEikonal(MeshData<PHDIM>& mesh,
                Values& w,
                const Eigen::SparseMatrix<double>& stiffnessMatrix,
                const Eigen::SparseLU<Eigen::SparseMatrix<double>>& solver,
                double r,
                Eigen::Matrix<double, Eigen::Dynamic, PHDIM>& lagrangians
                )
        : EikonalEquation<PHDIM>(mesh, w, stiffnessMatrix, solver), r(r), lagrangians(
            lagrangians) {}
```

- The `computeStiffnessTerm` method is designed to mimic the first contribution in the iterative scheme (42), namely $\left((c^{(i)}-1)\nabla w^{(i)}, \nabla\varphi\right)$. qui forse ha senso definite prima c poi scrivere il termine come (c-1)*grad... vedi sotto

```
    // Compute the stiffness term for the i-th element
    Values computeStiffnessTerm(int i) override{
        Eigen::Matrix<double, 1, PHDIM> grad = this->grad_w.row(i);
        Eigen::Matrix<double, 1, PHDIM> localLagrangian = lagrangians.row(i);
        double q_norm = this->anisotropicNorm(grad - localLagrangian/this->r);
        double stiffnessCoeff = (1.0 - q_norm) / ((1.0 + this->r)*q_norm + this->gamma
            );
        return stiffnessCoeff * (this->mesh.getLocalStiffnessMatrices()[i] * this->
            local_w);
    }
```

- The `computeReactionTerm` method refers to the computation of the second term in (42), that is $\left(\frac{1-c^{(i)}}{r}\lambda^{(i)}, \nabla\varphi\right)$. Notice that lagrangians are defined as piecewise constant finite elements, leading to the necessity to employ a reaction matrix mimicking the behaviour of $(\psi, \nabla\phi)$, where ψ are shape functions for piecewise constant finite elements P_h, while ϕ are those for linear finite elements V_h.

```
    // Compute the reaction term for the i-th element
    Values computeReactionTerm(int i) override {
        Eigen::Matrix<double, 1, PHDIM> grad = this->grad_w.row(i);
        Eigen::Matrix<double, 1, PHDIM> localLagrangian = lagrangians.row(i);
        double q_norm = this->anisotropicNorm(grad - localLagrangian/r);
        double reactionCoeff = (q_norm - 1.0) / ((r * (1.0+r) * q_norm) + this->gamma)
            ;

        Values uu(PHDIM+1);
        for (int j = 0; j < PHDIM+1; j++) {
            for (int k = 0; k < PHDIM+1; k++) {
                uu(j) += this->mesh.getLocalReactionMatrices()[i][j][k].dot(
                    localLagrangian.transpose());
            }
        }
        return reactionCoeff * uu;
    }
```

- The `updateLagrangians` method is where the true power of this class comes into play. After each iteration, the Lagrangian multipliers are updated based on the current state of the solution and the gradients. This update allows the solution to progressively satisfy constraints enforced by the Lagrangians. The first step in the function involves computing the norms of the difference between the current gradient of the solution and the scaled Lagrangian multipliers. The implementation follows the fact that the linear system can be solved "directly" (no need for a linear solver), as the mass matrix for piecewise constant finite elements is diagonal, hence easily invertible. The implementation follows accordingly.

```
    // Update the Lagrangian multipliers for each element
    void updateLagrangians(const Values& z) override {
        Eigen::Matrix<double, Eigen::Dynamic, 1> q_norm(this->grad_w.rows());

```

```
        // Compute the norm of the difference between the gradient and the Lagrangian
        for (int i = 0; i < this->grad_w.rows(); ++i) {
            Eigen::Matrix<double, 1, PHDIM> diff = this->grad_w.row(i) - lagrangians.
                row(i) / this->r;
            q_norm(i) = this->anisotropicNorm(diff);
        }

        // Compute coefficients for Lagrangian and gradient updates
        Eigen::Matrix<double, Eigen::Dynamic, 1> c = (1.0 + this->r * q_norm.array())
            / ((1 + this->r) * q_norm.array());
        Eigen::Matrix<double, Eigen::Dynamic, 1> lagrangianCoeff = 1 - c.array();
        Eigen::Matrix<double, Eigen::Dynamic, 1> gradzCoeff = this->r * c;

        // Local variables for each element
        Values local_z = Values::Zero(this->mesh.getConnectivity().rows());
        Eigen::Matrix<double, Eigen::Dynamic, PHDIM> grad_z(this->mesh.getNumElements
            (), PHDIM);

        // Update the Lagrangian multipliers for each element
        for(int i = 0; i < this->mesh.getGradientCoeff().size(); i++) {
            // Collect local z values
            for (int j = 0; j < PHDIM+1; j++) {
                local_z(j) = z(this->mesh.getConnectivity().col(i)(j));
            }

            // Compute the gradient of z
            grad_z.row(i) = apsc::computeGradient<PHDIM>(this->mesh.getGradientCoeff()
                [i], local_z);

            // Update the Lagrangian multipliers
            lagrangians.row(i) = lagrangianCoeff(i) * lagrangians.row(i) + gradzCoeff(
                i) * grad_z.row(i);
        }
    }
```

5.4. Parallelization

Since the assembly of the right-hand side of the iterations for each method is done element by element, we decide to parallelize this process. Specifically, the elements of the mesh are split between the available cores and each of the processes computes the portion of the right-hand side vector relative to the subset of elements.

`MPI` is initialized in the main script

```
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

5.4.1 Parallelized iterations

The fundamental parallelization of the computations is done in the method `updateSolution` within `EikonalEquation.hpp`, which was already described in Section 5.3

```
    bool updateSolution(const int rank, const int size) {

        reset_attributes();
        int n = this->mesh.getGradientCoeff().size();

```

```
        int local_n = n / size;
        int start = rank * local_n;
        int end = (rank == size - 1) ? n : start + local_n; // Last process handles the
            remainder

        for (int i = start; i < end; i++) {
            computeLocalRhs(i);
            updateGlobalRhs(i);
        }

        Values global_rhs(rhs.size());
        MPI_Reduce(rhs.data(), global_rhs.data(), rhs.size(), MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);

        Eigen::Matrix<double, Eigen::Dynamic, PHDIM> global_grad_w(grad_w.size() / PHDIM,
            PHDIM);
        MPI_Reduce(grad_w.data(), global_grad_w.data(), grad_w.size(), MPI_DOUBLE, MPI_SUM,
             0, MPI_COMM_WORLD);

        bool localConverged = true; // Initialize to true for ranks other than 0

        if (rank == 0) { // Only the master process handles the solution update
            rhs = global_rhs;
            grad_w = global_grad_w;
            // Continue with the rest of the updateSolution as before
            for (int idx : mesh.getBoundaryNodes()) {
                rhs(idx) = 0.0 * 1e40;
            }

            z = solver.solve(rhs);
            w += z;
            std::cout << "z : " << z.norm() << std::endl;
            updateLagrangians(z);

            localConverged = (z.norm() < tol);
        }

        // Broadcast the updated value of w from rank 0 to all other ranks
        MPI_Bcast(w.data(), w.size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
        // std::cout << "Rank:" << rank << " Maximum value of w: " << w.maxCoeff() << std::
            endl;
        return localConverged;
    }
```

Each process is assigned a subset of the elements of the mesh. For each of the elements, each process calls the methods `computeLocalRhs` and `updateGlobalRhs` in the class to perform the computations necessary to fill the right-hand side at the specific locations. Next all updates are reduced through sum to `global_rhs` at rank 0. The same is done for the computations that happened in the executions of the methods updating the gradients of the current solution, reduced to `global_grad_w`. The master process then handles the update of the solution and broadcasts the updated solution `w` to all processes. This is done since the current solution is needed in computations during the following iteration. Finally all processes return a `bool` representing if the convergence criterion has been satisfied or not. In reality only the master process checks the condition, while all others return `True`. Then all ourputed results are gathered in the main

```
    bool localConverged = eikonal->updateSolution(rank, size);
    MPI_Allreduce(&localConverged, &converged, 1, MPI_C_BOOL, MPI_LAND, MPI_COMM_WORLD);
```

5.4.2 Input handling

Handling of the settings in input is done by the master process and then data is broadcasted to all other processes so that each can initialize the mesh, boundary nodes and eikonal solution method. See Section

5.5 for more details on how this is done in more detail.

5.4.3 Broadcasting of variables

To avoid repeating computations which can be done once only across all ranks, most preliminary operations are executed by the master process only. This includes calculating the initial solution for the iterative method (details in Section 5.2) and filling the local and global matrices (details in Section 5.1.2). All resulting variables must be broadcasted to all other ranks for the following computations which will be performed in a parallel. This includes broadcasting

- The initial solution

```
    // Broadcast the size of w
    MPI_Bcast(&w_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Resize w on all other ranks
    if (rank != 0) {
        w.resize(w_size);
    }
    // Broadcast the content of w
    MPI_Bcast(w.data(), w_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

  Broadcasting of `w` is straightforward since it suffices to know the size and value type of the matrix to broadcast.
- The global stiffness matrix

```
    // Rank 0: flatten and send
    if (rank == 0) {
        std::vector<Eigen::Triplet<double>> tripletList;
        for (int k = 0; k < stiffnessMatrix.outerSize(); ++k) {
            for (Eigen::SparseMatrix<double>::InnerIterator it(stiffnessMatrix, k); it
                ; ++it) {
                tripletList.push_back(Eigen::Triplet<double>(it.row(), it.col(), it.
                    value()));
            }
        }
        int tripletSize = tripletList.size();
        MPI_Bcast(&tripletSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

        std::vector<int> rows(tripletSize), cols(tripletSize);
        std::vector<double> values(tripletSize);
        for (int i = 0; i < tripletSize; ++i) {
            rows[i] = tripletList[i].row();
            cols[i] = tripletList[i].col();
            values[i] = tripletList[i].value();
        }
        MPI_Bcast(rows.data(), tripletSize, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(cols.data(), tripletSize, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(values.data(), tripletSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    } else {
        int tripletSize;
        MPI_Bcast(&tripletSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

        std::vector<int> rows(tripletSize), cols(tripletSize);
        std::vector<double> values(tripletSize);
        MPI_Bcast(rows.data(), tripletSize, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(cols.data(), tripletSize, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(values.data(), tripletSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        // Reconstruct the sparse matrix on other ranks
        Eigen::SparseMatrix<double> stiffnessMatrix(mesh.getNumNodes(), mesh.
            getNumNodes());
        std::vector<Eigen::Triplet<double>> tripletList(tripletSize);
```

```
        for (int i = 0; i < tripletSize; ++i) {
            tripletList[i] = Eigen::Triplet<double>(rows[i], cols[i], values[i]);
        }
        stiffnessMatrix.setFromTriplets(tripletList.begin(), tripletList.end());
    }
```

Broadcasting the global matrices is more complex, since these are vectors of sparse matrices. Hence not only do we need to flatten the vectors in order to broadcast them, but we also need to make use of `Eigen::Triplets` to broadcasts positions and values of all entries. Once the vector has been flattened by the master process, all information is broadcasted to all other processes, which proceed to reassemble the vector of matrices.

- The global reaction matrix

```
    // Flatten reactionMatrix
    if (rank == 0) {
        std::vector<double> flatReactionMatrix;
        for (const auto& outerVec : reactionMatrix) {
            for (const auto& innerVec : outerVec) {
                for (int i = 0; i < PHDIM; ++i) {
                    flatReactionMatrix.push_back(innerVec(i));
                }
            }
        }
        int flatSize = flatReactionMatrix.size();
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(flatReactionMatrix.data(), flatSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    } else {
        int flatSize;
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

        std::vector<double> flatReactionMatrix(flatSize);
        MPI_Bcast(flatReactionMatrix.data(), flatSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        // Reconstruct reactionMatrix on other ranks
        int idx = 0;
        for (int i = 0; i < mesh.getNumNodes(); ++i) {
            for (int j = 0; j < mesh.getNumNodes(); ++j) {
                for (int k = 0; k < PHDIM; ++k) {
                    reactionMatrix[i][j](k) = flatReactionMatrix[idx++];
                }
            }
        }
    }
```

Broadcasting of the global reaction matrix is analogous to what is described in the previous bullet point.

- `mesh.gradientCoeff`

```
    // Flatten gradientCoeff
    if (rank == 0) {
        std::vector<double> flatGradientCoeff;
        for (const auto& mat : mesh.gradientCoeff) {
            for (int i = 0; i < mat.size(); ++i) {
                flatGradientCoeff.push_back(mat(i));
            }
        }
        int flatSize = flatGradientCoeff.size();
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(flatGradientCoeff.data(), flatSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    } else {
        int flatSize;
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

```

```
        std::vector<double> flatGradientCoeff(flatSize);
        MPI_Bcast(flatGradientCoeff.data(), flatSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        // Reconstruct gradientCoeff on other ranks
        int idx = 0;

        for (int i = 0; i < mesh.getNumElements(); ++i) {
            Eigen::Matrix<double, PHDIM, PHDIM> tempMatrix;  // Temporary matrix
            for (int j = 0; j < PHDIM * PHDIM; ++j) {
                tempMatrix(j % PHDIM, j / PHDIM) = flatGradientCoeff[idx++];
            }
            mesh.gradientCoeff.push_back(tempMatrix);
        }
    }
```

If the current process is rank 0, the method begins by flattening each matrix in the `gradientCoeff` vector into a linear array. The flattened data is broadcasted to all other ranks together with its size. After receiving the flattened data, each process reconstructs the vector by iterating over the flattened array and filling in temporary matrices `tempMatrix`. The reconstructed matrices are then stored back in the `gradientCoeff` vector of the mesh.

- `mesh.localStiffnessMatrices`

```
    // Flatten localStiffnessMatrices (std::vector<Eigen::Matrix<double, PHDIM+1,
        PHDIM+1>>)
    if (rank == 0) {
        std::vector<double> flatLocalStiffnessMatrices;
        for (const auto& matrix : mesh.localStiffnessMatrices) {
            for (int i = 0; i < (PHDIM + 1) * (PHDIM + 1); ++i) {
                flatLocalStiffnessMatrices.push_back(matrix(i));
            }
        }
        int flatSize = flatLocalStiffnessMatrices.size();
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(flatLocalStiffnessMatrices.data(), flatSize, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
    } else {
        int flatSize;
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

        std::vector<double> flatLocalStiffnessMatrices(flatSize);
        MPI_Bcast(flatLocalStiffnessMatrices.data(), flatSize, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);

        // Reconstruct localStiffnessMatrices on other ranks
        int idx = 0;
        for (int i = 0; i < mesh.getNumElements(); ++i) {
            Eigen::Matrix<double, PHDIM+1, PHDIM+1> tempMatrix;  // Temporary matrix
            for (int j = 0; j < (PHDIM + 1) * (PHDIM + 1); ++j) {
                tempMatrix(j % (PHDIM + 1), j / (PHDIM + 1)) =
                    flatLocalStiffnessMatrices[idx++];
            }
            mesh.localStiffnessMatrices.push_back(tempMatrix);
        }
    }
```

Once again rank 0 proceeds by flattening each matrix in the vector into a linear array and broadcasting data and size. Each process reconstructs the vector by iterating over the flattened array and filling in temporary matrices `tempMatrix`. The reconstructed matrices are then stored back in the vector of the mesh.

- `mesh.localReactionMatrices`

```
    // Flatten localReactionMatrices (std::vector<std::vector<std::vector<Eigen::
        Matrix<double, PHDIM, 1>>>>)
```

```
    if (rank == 0) {
        std::vector<double> flatLocalReactionMatrices;
        for (const auto& outerVec : mesh.localReactionMatrices) {
            for (const auto& middleVec : outerVec) {
                for (const auto& matrix : middleVec) {
                    for (int i = 0; i < PHDIM; ++i) {
                        flatLocalReactionMatrices.push_back(matrix(i));
                    }
                }
            }
        }
        int flatSize = flatLocalReactionMatrices.size();
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(flatLocalReactionMatrices.data(), flatSize, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);

    } else {
        int flatSize;
        MPI_Bcast(&flatSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

        std::vector<double> flatLocalReactionMatrices(flatSize);
        MPI_Bcast(flatLocalReactionMatrices.data(), flatSize, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);

        // Reconstruct localReactionMatrices on other ranks
        int idx = 0;
        for (int i = 0; i < mesh.getNumElements(); ++i) {
            std::vector<std::vector<Eigen::Matrix<double, PHDIM, 1>>> tempTensor;

            for (int j = 0; j < (PHDIM + 1); ++j) {
                std::vector<Eigen::Matrix<double, PHDIM, 1>> tempIntermediate;

                for (int k = 0; k < (PHDIM + 1); ++k) {
                    Eigen::Matrix<double, PHDIM, 1> tempVec;  // Temporary matrix
                    for (int l = 0; l < PHDIM; ++l) {
                        tempVec(l) = flatLocalReactionMatrices[idx++];
                    }
                    tempIntermediate.push_back(tempVec);
                }
                tempTensor.push_back(tempIntermediate);
            }
            mesh.localReactionMatrices.push_back(tempTensor);
        }
    }
```

Also for `localReactionMatrices`, which is a nested vector, the data is flattened by iterating through all layers and pushing back each entry of the inner matrix. After receiving the flattened data, each rank reconstructs the vector. This involves: iterating over the flattened data to fill in a temporary tensor structure `tempTensor`, creating temporary intermediate vectors and matrices `tempIntermediate`/`tempVec`, and pushing the reconstructed matrices into the `localReactionMatrices` vector.

5.5. Main

The `main.cpp` file sets up and runs the simulation for solving the Eikonal equation over a given mesh. In the following we describe the main sections of how the code is run by the main function, without considering elements needed for parallelization which are described in Section 5.4. All functions called and classes initialized have been previously described.

5.5.1 Reading and Initializing the Mesh

```
    // Read the mesh file and create the mesh object
    MeshData<PHDIM> mesh;
    std::string mesh_file;
    std::cout << "\nEnter the mesh file name (default: mesh_uniform_3.vtk): ";
    std::getline(std::cin, mesh_file);

    // Append directory to filename if not empty, otherwise use default file
    if (!mesh_file.empty()) {
        mesh_file = "vtk_files/" + mesh_file;
    } else {
        mesh_file = "vtk_files/mesh_uniform_3.vtk";
    }
    // Check if the file can be opened
    std::ifstream file(mesh_file);
    if (!file) {
        std::cerr << "Error: File '" << mesh_file << "' not found. Please check the
            filename and try again.\n";
        return 1; // Exit with error code 1
    } else {
        std::cout << "Using mesh file: " << mesh_file << std::endl;
    }

    mesh.readMesh(mesh_file);
```

A `MeshData` object is instantiated to represent the computational mesh. The program prompts the user for the name of the mesh file and the mesh data is read using `mesh.readMesh`, which populates the mesh object with nodes, elements, and other relevant geometric data.

5.5.2 Setting Boundary Conditions

```
    int choice = 1; // Default value
    std::string input;

    std::cout << "\nEnter the Dirichlet Boundary Conditions:\n";
    std::cout << "1. Dirichlet Null BCs on the bottom face of the domain (z=0)\n";
    std::cout << "2. Dirichlet Null BCs on the central point of the bottom face (z=0, x
        =0.5, y=0.5)\n";
    std::cout << "3. Dirichlet Null BCs on one vertex of the domain (x=0, y=0, z=0)\n";
    std::cout << "4. Dirichlet Null BCs on the whole boundary of the domain\n";
    std::cout << "Please choose an option (1-4, default 1): ";

    std::getline(std::cin, input); // Read the whole line as string

    // Attempt to parse the input as an integer
    std::stringstream ss(input);
    if (!(ss >> choice) || choice < 1 || choice > 4) {
        choice = 1; // Default to 1 if input is invalid or not within the range
        std::cout << "Defaulting to option 1." << std::endl;
    }

    std::cout << "Selected option: " << choice << std::endl;
        std::string choiceString = "";
    if (choice == 1) {
            choiceString = "face";
        } else if (choice == 1) {
            choiceString = "center";
        } else if (choice == 2) {
            choiceString = "vertex";
        } else if (choice == 3) {
            choiceString = "all";
        }

    mesh.updateBoundaryNodes(choice);
```

The user is prompted to select from different options for Dirichlet boundary conditions, and the selected option is then applied to the mesh using the `mesh.updateBoundaryNodes(choice)` method.

5.5.3 Preparing Global Matrices

```
Eigen::SparseMatrix<double> stiffnessMatrix(mesh.getNumNodes(), mesh.getNumNodes());
std::vector<std::vector<Eigen::Matrix<double, PHDIM, 1>>> reactionMatrix(mesh.
    getNumNodes(), std::vector<Eigen::Matrix<double, PHDIM, 1>>(mesh.getNumNodes()));

Values globalIntegrals = Values::Constant(mesh.getNumNodes(), 0.0);

mesh.fillGlobalVariables(stiffnessMatrix, massMatrix, reactionMatrix, globalIntegrals);
```

The program prepares the global matrices required for the FEM simulation. The stiffness matrix is created as a sparse matrix using `Eigen`'s `SparseMatrix` class. A reaction matrix is created as a vector of vectors of `Eigen::Matrix` objects. A `Values` object is initialized to store integral values over the mesh, set to zero initially. These matrices are filled with appropriate values by calling `mesh.fillGlobalVariables(...)`.

5.5.4 Solving the Heat Equation for Initial Conditions

```
Values forcingTerm = Values::Constant(mesh.getNumNodes(), 1.0);
InitialConditions<PHDIM> initialConditions;
Values initial_conditions = initialConditions.HeatEquation(stiffnessMatrix,
    globalIntegrals, forcingTerm, mesh.getBoundaryNodes());

mesh.addScalarField(initial_conditions, mesh_file, choiceString, "", "heat");
```

The program sets up a forcing term (a vector initialized to ones) and uses the `InitialConditions` class to solve the heat equation, which provides the initial conditions for the Eikonal equation. The heat equation is solved by calling `initialConditions.HeatEquation(...)`, which takes the stiffness matrix, integral values, forcing term, and boundary nodes as arguments. The computed initial conditions are added to the mesh as a scalar field using `mesh.addScalarField(...)` to generate a `.vtk` file to plot the initial solution.

5.5.5 Iterative Solution of the Eikonal Equation

```
Values w = initial_conditions;

bool converged = false;
int maxIterations = 10000;

Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
solver.compute(stiffnessMatrix); // Decompose the stiffness matrix with a direct solver

int methodChoice = 1; // Default value

std::cout << "\nSelect the eikonal method to use:\n";
std::cout << "1. Standard Eikonal\n";
std::cout << "2. Penalty Eikonal\n";
std::cout << "3. Lagrangian Eikonal\n";
std::cout << "Enter your choice (1-3, default: 1): ";

std::getline(std::cin, input); // Read the whole line as string

// Attempt to parse the input as an integer
std::stringstream sss(input);
if (!(sss >> methodChoice) || methodChoice < 1 || methodChoice > 3) {
    methodChoice = 1; // Default to 1 if input is invalid or not within the range
```

```
        std::cout << "Defaulting to option 1." << std::endl;
    }
    std::cout << "Selected option: " << methodChoice << std::endl;
    std::string itMethString = "";
    if (methodChoice == 0) {
            itMethString = "";
        } else if (methodChoice == 1) {
            itMethString = "_standard";
        } else if (methodChoice == 2) {
            itMethString = "_penalty";
        } else if (methodChoice == 3) {
            itMethString = "_lagrangian";
        }

    std::unique_ptr<EikonalEquation<PHDIM>> eikonal = nullptr;
    if (methodChoice == 1) {
        eikonal = std::make_unique<StandardEikonal<PHDIM>>(mesh, w, stiffnessMatrix, solver
            );
        std::cout << "Standard Eikonal selected." << std::endl;
    } else if (methodChoice == 2) {
        double r_penalty = 0.1;
        eikonal = std::make_unique<PenaltyEikonal<PHDIM>>(mesh, w, stiffnessMatrix, solver,
             r_penalty);
        std::cout << "Penalty Eikonal selected." << std::endl;
        // std::cout << eikonal << std::endl;
    } else if (methodChoice == 3) {
        double r_lagrangian = 5;
        Eigen::Matrix<double, Eigen::Dynamic, PHDIM> lagrangians = Eigen::Matrix<double,
            Eigen::Dynamic, PHDIM>::Constant(mesh.getNumElements(), PHDIM, 0.0);
        eikonal = std::make_unique<LagrangianEikonal<PHDIM>>(mesh, w, stiffnessMatrix,
            solver, r_lagrangian, lagrangians);
        std::cout << "Lagrangian Eikonal selected." << std::endl;
        // std::cout << eikonal << std::endl;
    }
    else {
        std::cerr << "Invalid choice. Exiting..." << std::endl;
        return 1;
    }
```

The program proceeds to solve the Eikonal equation iteratively. It initializes a vector w with the previously computed initial conditions. The stiffness matrix is decomposed with a sparse direct solver `Eigen::SparseLU`, preparing it for efficient repeated solving.

The user is prompted to choose one of the three methods for solving the Eikonal equation and, based on the user's choice, a corresponding solver object is instantiated using `std::make_unique`. The usage of smart pointers allows for the inheritance to happen.

5.5.6 Iterative Solution Loop

```
    for (int iter = 0; iter < maxIterations && !converged; ++iter) {
        std::cout << "-------------- Iteration " << iter + 1 << " --------------" << std::
            endl;

        converged = eikonal->updateSolution();

        if (converged) {
            std::cout << "Solution converged after " << iter + 1 << " iterations." << std::
                endl;
            mesh.addScalarField(w, mesh_file, choiceString, itMethString, "heat");
            // std::cout << w << std::endl;
        }
    }
```

```
12
13      if (!converged) {
14          std::cout << "Solution did not converge within the maximum number of iterations."
                << std::endl;
15      }
16
17      return 0;
```

A loop runs updating the solution at each step. In each iteration, the `updateSolution()` method of the chosen Eikonal solver is called to update the solution. If the solution converges, the program breaks out of the loop, prints a convergence message, and saves the current solution. If the solution does not converge after the maximum number of iterations, the program prints a non-convergence message.

6. Results

6.1. Solutions

Figures 3, 4 and 5 display the initial conditions (top) and solutions for the Eikonal equation with three different Dirichlet boundary conditions imposed. The initial solution is always obtained as the solution of the Laplace equation with analogous boundary conditions to the actual problem. The original formulations do not allow us to start with $w^{(0)} = 0$ everywhere, since we have a division by zero and null gradients would result in no updates. Thus we start with the solution of the problem:

$$-\Delta w^{(0)} = 1,$$

whose weak form is clearly:

$$(\nabla w^{(0)}, \nabla \varphi) = \int_{\Omega} \varphi \, d\Omega, \quad \forall \varphi \in V.$$

We choose this option since the solution of this problem is "close" to the physical solution we want to obtain. This characteristic is crucial to obtain a physical solution instead of other discrete fields which might display non-physical features like oscillations.

Moreover, we show results for null Dirichlet conditions applied to the whole bottom face, to the center of the bottom face and to the vertex in the origin. All solutions displayed in these plots are relative to an isotropic medium, hence the solution would be analogous for changes in the face/vertex considered. From the contour plots (bottom of all figures) it is evident how the solution represents the times of arrival of a wave propagating from the Dirichlet boundary nodes within an isotropic medium. The times are the same for all heights in Figure 3, while the contour lines are not planes in the remaining two cases.

We also display how the solution changes if the medium is anisotropic. Specifically, Figure 6 displays the solution for a medium with an anisotropy matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

meaning that the time to propagate in the y direction will be shorter. As we can see from the image, the solution is different from the isotropic counterpart in Figure 5, showing clear difference in the propagation of the wave in the x and y directions. We don't experiment further with different matrices since this is beyond the scope of our work.

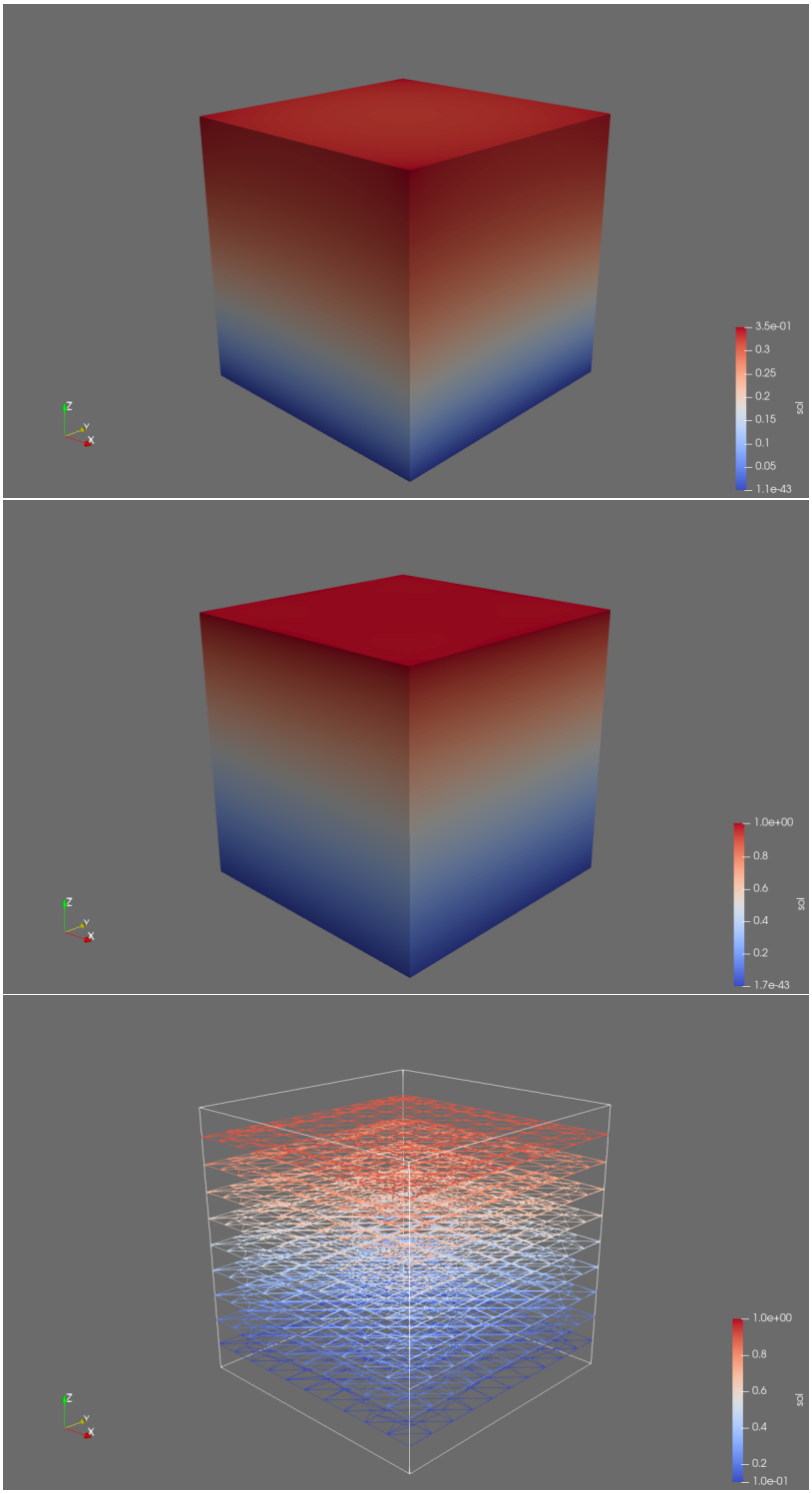

Figure 3: Solution of the eikonal equation on a cubic domain with null Dirichlet boundary conditions imposed on the bottom face: initial solution (top), solution (middle), contours (bottom)

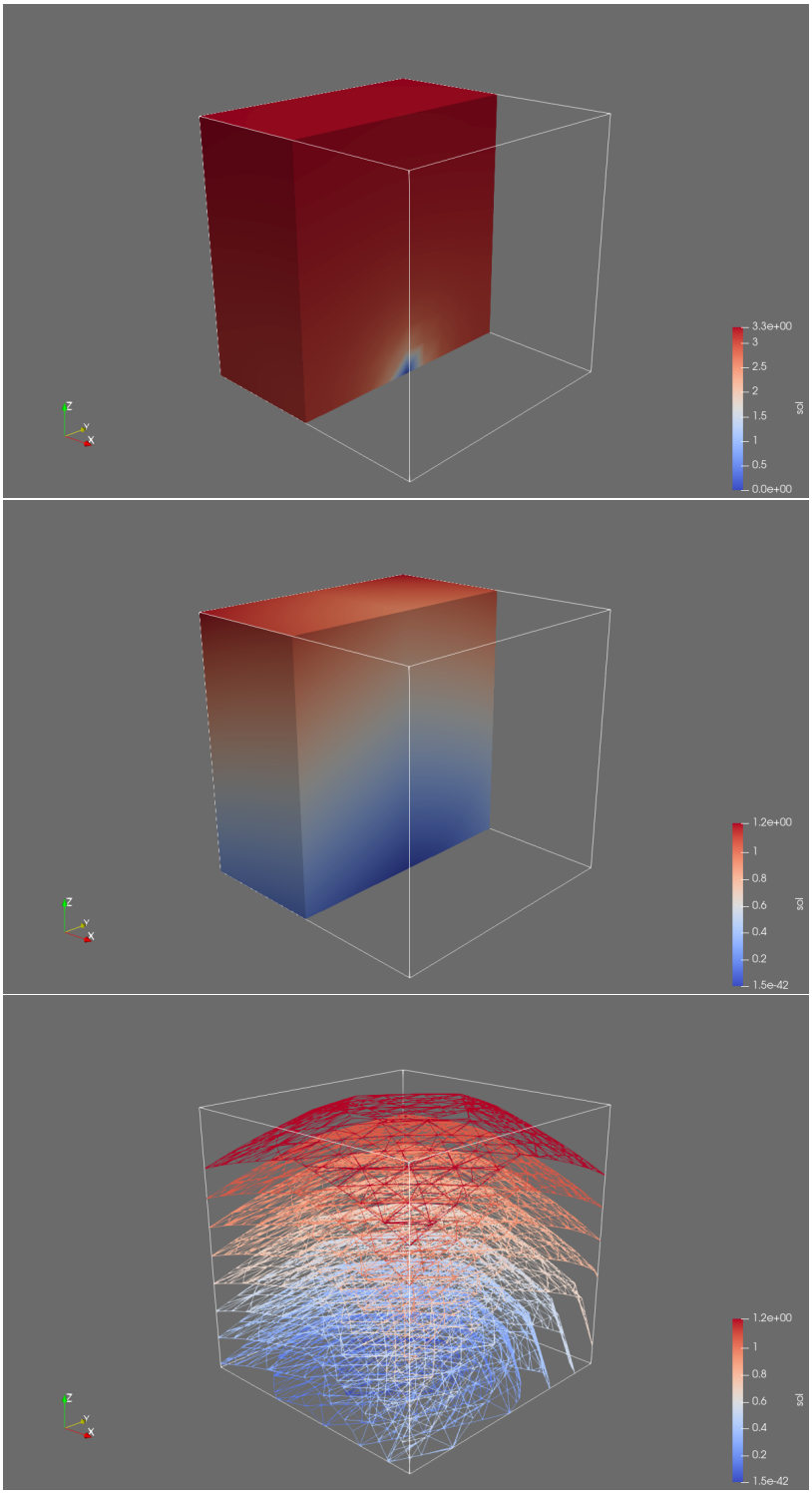

Figure 4: Solution of the eikonal equation on a cubic domain with null Dirichlet boundary conditions imposed on the central node of the bottom face: initial solution (top), solution (middle), contours (bottom)

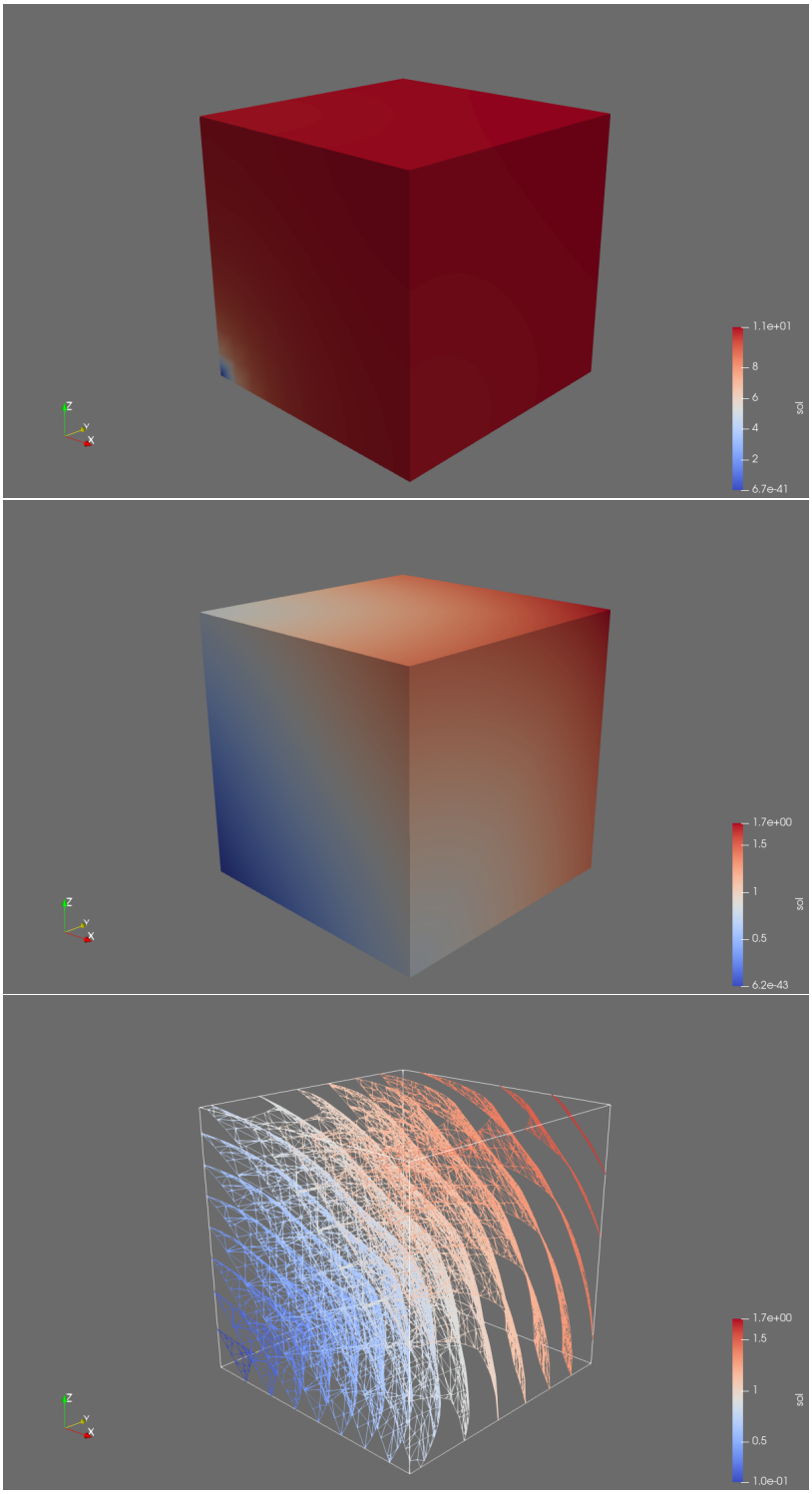

Figure 5: Solution of the eikonal equation on a cubic domain with null Dirichlet boundary conditions imposed on the vertex in the origin: initial solution (top), solution (middle), contours (bottom)

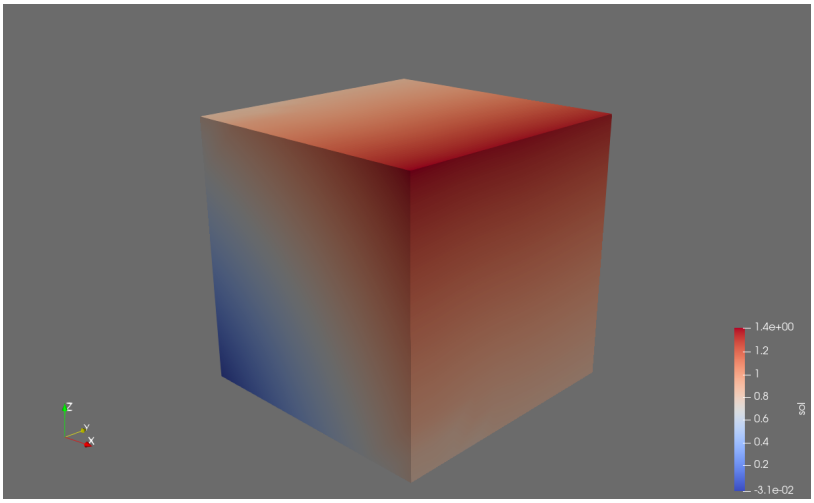

Figure 6: Solution of the eikonal equation on a cubic domain with null Dirichlet boundary conditions imposed on the vertex in the origin and diagonal anisotropic matrix with values $(1, 5, 1)$

6.2. Convergence of the implicit methods

We show plots displaying the convergence of two key quantities for all three methods in the standard case of isotropic medium and boundary conditions applied to one face of the domain. We report:

- The norm of the incremental solution z computed when solving the linear system at each iteration. z is not only exploited to update the solution, but also to check for convergence of iterations and thus to end the iteration loop. Hence we check the convergence of this value towards 0
- The minimum and maximum value of the norm of the gradient of the solution across all elements. Since the solution of the Eikonal equation would impose unitary gradient across the domain, we check that convergence of the iterative method actually yields a solution which approximates the real solution by monitoring the gradients. We expect the values to converge to 1

All values are displayed for all iterations during the solving process. We can clearly see from Figures 7 and **??** that both the incremental solution and the gradients evolve as expected.

Moreover, we display the iterations required for convergence of the methods depending on the refinement of the mesh in Figure 8. The refinements considered are for a cubic mesh with 5, 10 and 20 cells per edge, and thus 216, 1331 and 9261 nodes in total respectively.

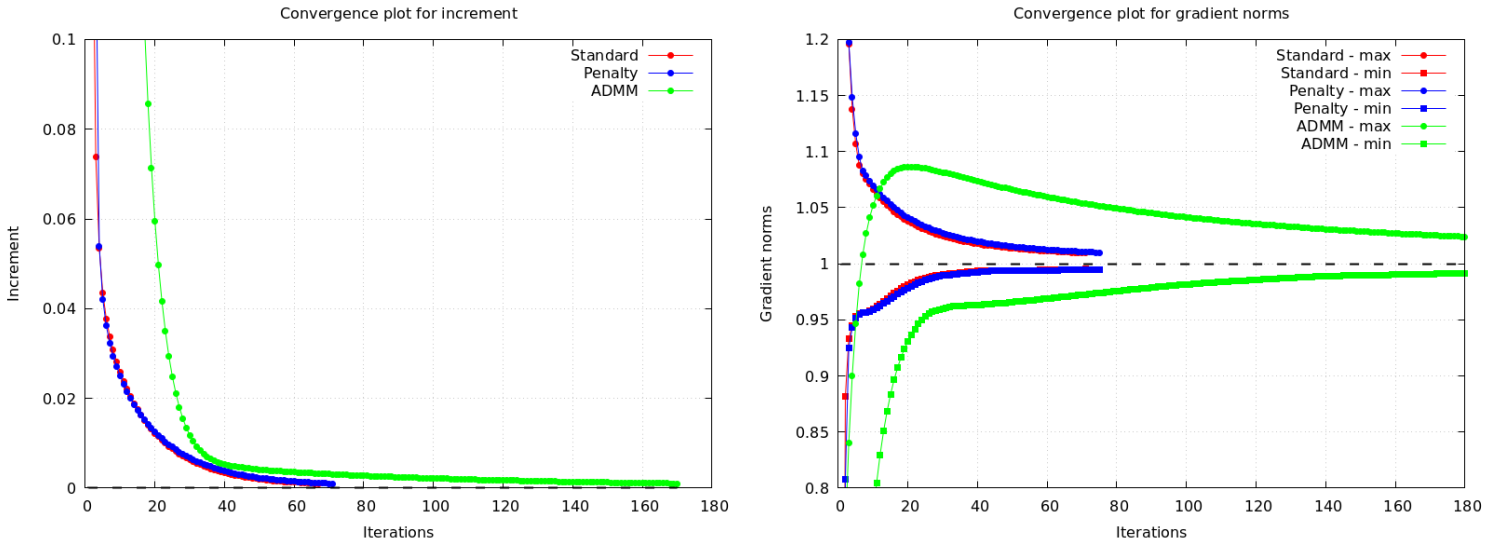

Figure 7: Convergence of the incremental solution and of the gradients

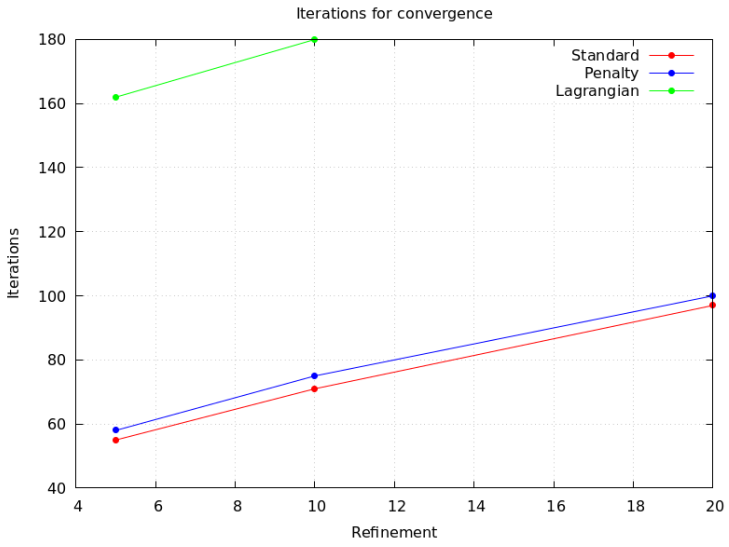

Figure 8: Iterations needed for convergence of the methods for different refinement levels

6.3. Dependence on the initial solution

When solving the Eikonal equation as described above, we have no guarantee of the uniqueness of the seeked solution. However, we consider that a physical solution is unique and is the field we want to obtain. We argue that reaching the physical solution at the end of the iterative method depends strongly on the initial solution exploited for the first iteration, meaning that if the initial condition is too far from the desired result, the method will either not converge or converge to an undesired solution. For this reason in all results displayed above we exploited the solution of the Laplace equation as the starting point for our methods, and this has proven a good choice since we obtain physical solutions which don't display any artifacts such as oscillations. However, we experiment with different initial solutions to evaluate this claim and verify if the more advanced methods offer better robustness to far initial solutions.

Many far fetched initial solutions, such as random initialization, don't yield any convergence for any of the methods. We experiment with a slight alteration of the approach used thus far. We define an initial solution which is constant and equal to 1 in all of the domain, except for the nodes where the Dirichlet boundary condition is applied which are set to 0 in accordance with the boundary condition. For this experiment we apply the Dirichlet condition on one face only (see top image of Figure 10) and run the standard and penalty iterative methods. From the results displayed in Figure 10 we can clearly see that the standard method fails, converging after a couple of iterations to a solution approximately equal to the initial. The method based on the penalty formulation, on the other hand, converges to a solution much closer to the desired result seen before (3), thus displaying better robustness to the initial solution given. Figures 9 and **??** display the convergence of the method, which in this case is not as regular as for the previously employed initial condition and takes a larger number of iterations.

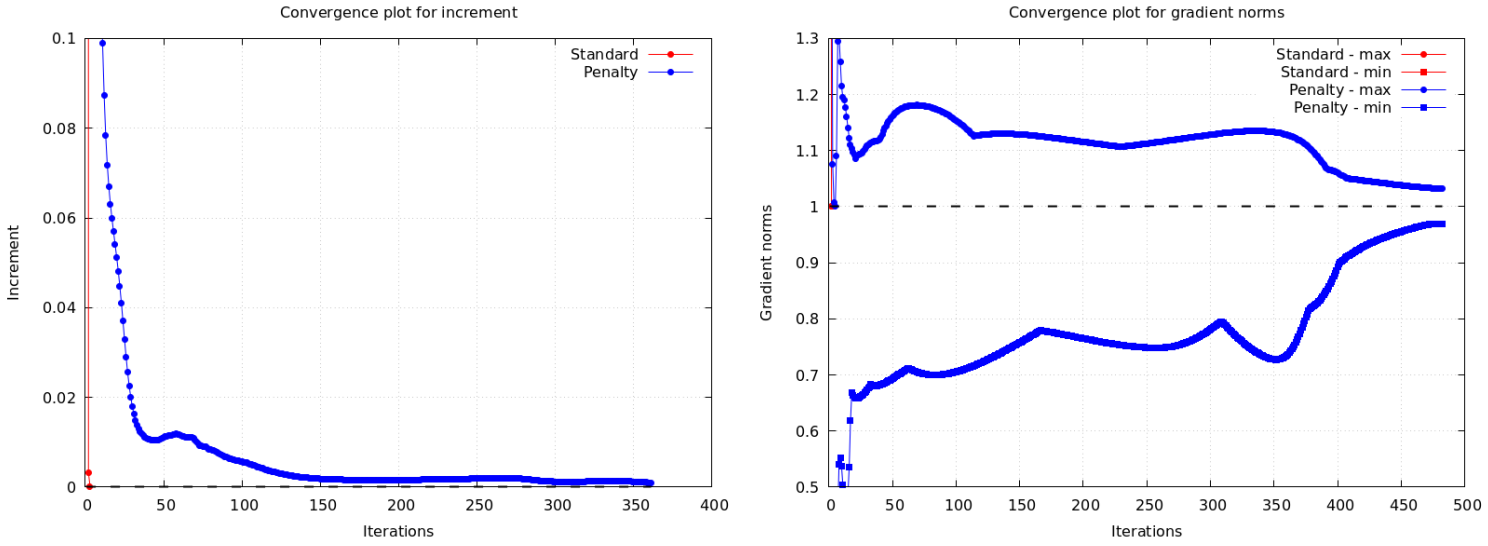

Figure 9: Convergence of the incremental solution and of the gradients for the non-physical initial condition

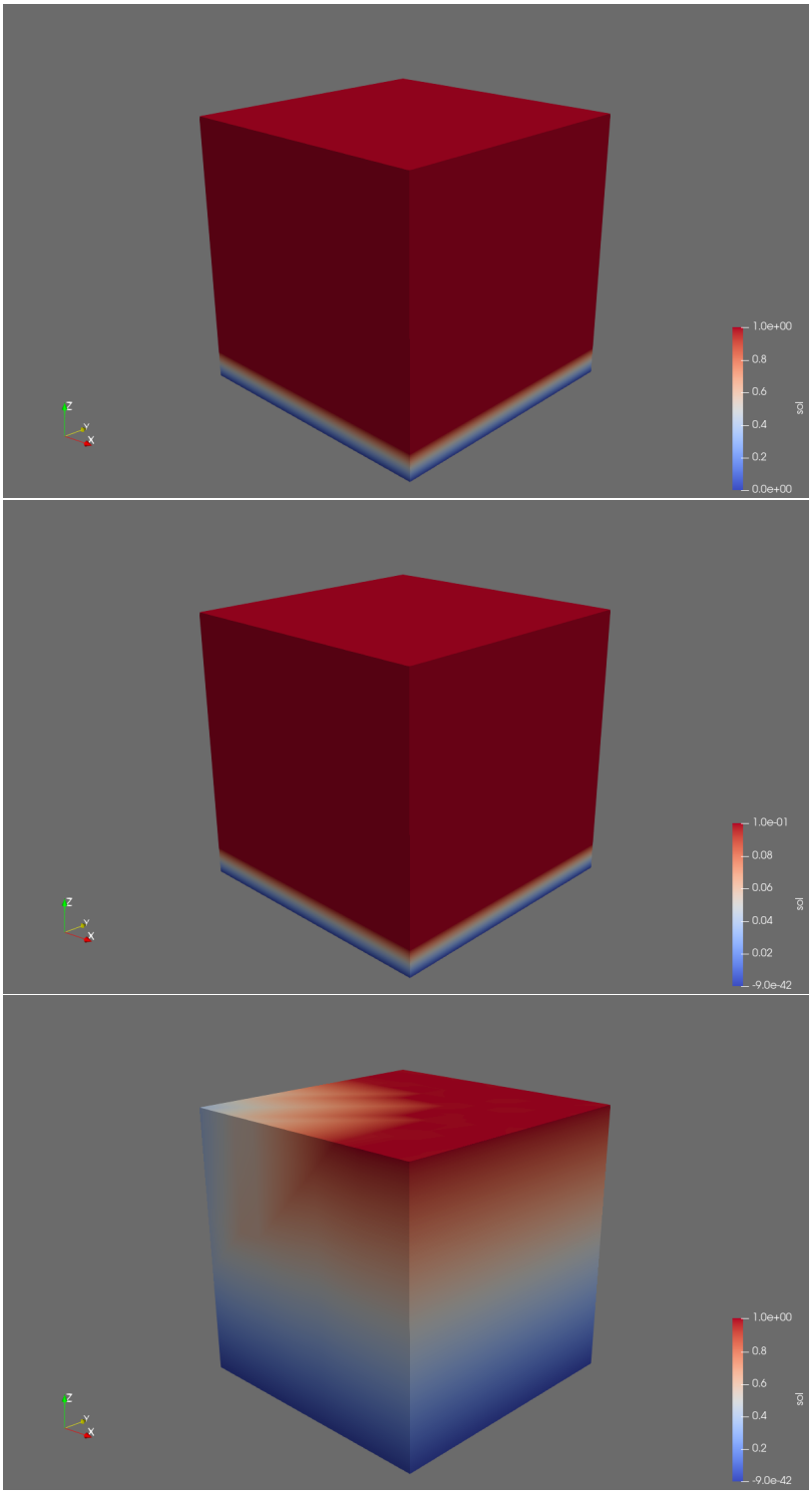

Figure 10: Solution of the eikonal equation on a cubic domain with null Dirichlet boundary conditions imposed on the bottom face and **"delta" initial solution**: initial solution (top), standard method solution (middle), penalty method solution (bottom)

6.4. Parallelization

Finally we study the effect of the parallelization on the computation times for our problem. As a reminder, the process which is parallelized is the assembly of the right-hand side of the system solved at each iteration for all methods. The complexity of this operation of course scales with the number of elements in the mesh considered, and can represent a high cost in the overall computation. To evaluate the time savings with this approach, we solve the equation with the standard iterative method for three different refinements of the mesh, exploiting 1,2 and 4 cores. The refinements of the mesh considered have 5, 10 and 20 elements per edge of the cubic domain, hence respectively 750, 6000 and 48000 elements in total. The quantity we monitor is the time taken for the computation and assembly of the right-hand side at each iteration. Figure 11 displays the results averaged across all iterations, with the time expressed in seconds, whereas Table 1 also reports the standard deviations of the measurement. The improvement when exploiting multiple processes is evident. It is important to notice that the time gained is for each iteration, and thus the contribute of the parallelization is great when evaluated for the whole solution process.

# Elements	1 Core	2 Cores	4 Cores
750	0.0086 ± 0.0002	0.0046 ± 0.0004	0.0025 ± 0.0003
6000	0.0861 ± 0.0342	0.0439 ± 0.0178	0.0292 ± 0.0111
48000	0.6944 ± 0.2517	0.4268 ± 0.1597	-

Table 1: Times [s] for computation of the right-hand side of the linear system

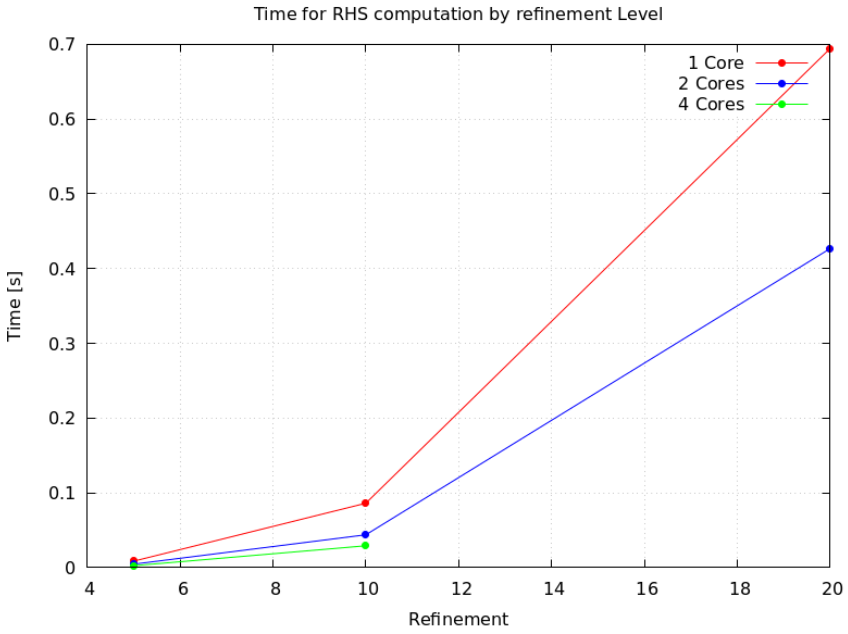

Figure 11: Time for the computation of the right-hand side of the linear system to solve at each iteration of the iterative method

References

[1] Alexander G. Belyaev and Pierre-Alain Fayolle. On variational and pde-based distance function approximations. *Computer Graphics Forum*, 34(8):104–118, 2015.

A. Functional Derivative of $E(\mathbf{p}, u)$ w.r.t. u

To find how the functional

$$E(\mathbf{p}, u) = \int_\Omega \left\{ (|\mathbf{p}| - 1)^2 + \frac{r}{2} (\mathbf{p} - \nabla u)^2 \right\} dx$$

changes with respect to small changes in u, we consider a small perturbation δu in u. The goal is to compute:

$$\frac{\delta E}{\delta u} = \lim_{\epsilon \to 0} \frac{E(\mathbf{p}, u + \epsilon \delta u) - E(\mathbf{p}, u)}{\epsilon}$$

Let's expand $E(\mathbf{p}, u + \epsilon \delta u)$:

$$E(\mathbf{p}, u + \epsilon \delta u) = \int_\Omega \left\{ (|\mathbf{p}| - 1)^2 + \frac{r}{2} (\mathbf{p} - \nabla (u + \epsilon \delta u))^2 \right\} dx$$

Expanding the term $(\mathbf{p} - \nabla (u + \epsilon \delta u))^2$:

$$(\mathbf{p} - \nabla u - \epsilon \nabla \delta u)^2 = (\mathbf{p} - \nabla u)^2 - 2\epsilon (\mathbf{p} - \nabla u) \cdot \nabla \delta u + \epsilon^2 (\nabla \delta u)^2$$

As $\epsilon \to 0$, the ϵ^2 term vanishes and the first order term in ϵ dominates. Thus, the first variation of E w.r.t. u becomes:

$$\frac{\delta E}{\delta u} = - \int_\Omega r[(\mathbf{p} - \nabla u) \cdot \nabla (\delta u)] \, d\mathbf{x}$$

Integration by parts (and suitable boundary conditions such as $\delta u = 0$ on $\partial \Omega$ to eliminate boundary terms) gives:

$$\frac{\delta E}{\delta u} = - \int_\Omega r (\mathbf{p} - \nabla u) \cdot \nabla (\delta u) \, dx = - \int_\Omega r \, \mathrm{div}(\mathbf{p} - \nabla u)(\delta u) \, dx$$

Since this must be true for any arbitrary variation δu, the integrand itself must vanish:

$$r \, \mathrm{div}(\mathbf{p} - \nabla u) = 0 \quad \text{or} \quad \mathrm{div}(\mathbf{p} - \nabla u) = 0$$

This simplifies further to:

$$\mathrm{div} \, \nabla u = \mathrm{div} \, \mathbf{p} \quad \text{or} \quad \Delta u = \mathrm{div} \, \mathbf{p}$$