Master Calcul Scientifique et Mathématiques de l'Innovation

Development of algorithms for calculating distances to an interface in an HPC code

DEMUTH Axel

Supervisors : GILET Nicolas, GIRARDIN Mathieu

03/03/2025 - 22/08/2025

Academic Year 2024-2025

CEA DAM, Bruyères-le-Châtel, France

Abstract

The hydrodynamic code ARMEN is used at CEA to perform simulations with multiple materials. When those materials slip against one another, it is usefull to enforce a numerical model at the interface to improve the numerical results obtained. This numerical model requires to compute the distance to the interface between the two materials. Currently, the hydrodynamic code ARMEN uses the euclidean norm calculation on all mesh points relative to each point of the interface. This calculation is extremely time consuming especially in 3D as its complexity is $\mathcal{O}(N_{mesh} \times N_{interface})$, where N_{mesh} is the number of cells in the mesh, and $N_{interface}$ the number of points in the interface that is reconstructed on the mesh.
To reduce this computation time, during this internship, I studied the resolution of the eikonal equation, which is already used in other fields for calculating the smallest distance, such as seismology, tomography, and robotics. To solve it, we decided to focus on numerical methods called *Fast Methods*.

Résumé

Le code hydrodynamique ARMEN est utilisé au CEA pour effectuer des simulations avec plusieurs matériaux. Lorsque ces matériaux glissent les uns sur les autres. Il est alors utile d'utiliser un modèle numérique à l'interface pour améliorer les résultats numériques obtenus. Ce modèle numérique nécessite de calculer la distance à l'interface entre les deux matériaux. Actuellement, le code hydrodynamique ARMEN utilise la norme euclidienne sur tous les points du maillage par rapport à chaque point de l'interface. Ce calcul est extrêmement coûteux en temps de calcul, en particulier en 3D car sa complexité est de l'ordre : $\mathcal{O}(N_{mesh} \times N_{interface})$, où N_{mesh} est le nombre de cellules dans le maillage, et $N_{interface}$ le nombre de points dans l'interface reconstruite sur le maillage.
Pour réduire ce temps de calcul, j'ai pendant ce stage étudié la résolution de l'équation eikonale, qui est déjà utilisée dans d'autres domaines tels que la sismologie, la tomographie ou la robotique. Pour la résoudre, nous avons décidé de nous concentrer sur des méthodes numériques *Fast Methods*.

Contents

Chapter 1

Introduction

1.1 Resume

This internship took place at the conclusion of my Master's degree in Applied Mathematics and Computer Science at the University of Strasbourg. I have been doing an internship from March 3, 2025, to August 22, 2025, at CEA-DAM in Bruyères-le-Châtel under the supervision of GILET Nicolas and GIRARDIN Mathieu, both engineers in computer science. Especially, I joined the team in charge of the development of the hydrodynamic code ARMEN. The main goal of my work was to study and to implement an original method on ARMEN to compute distances from an interface between several materials. This distance-to-interface computation, currently based on the Euclidean norm, is highly time-consuming and strongly restricts some physical modeling, especially in 3-dimensions.
To optimize the computational time, I have successfully implemented on ARMEN a brand-new method based on the Eikonal equation solved by *Fast Methods* in 2-dimensions. To reach this goal, my first mission was to study some references on shortest-path problems modeled by Eikonal equation and solved by *Fast Marching* methods.
Then, I have implemented a toy model in sequential C++ code to familiarize myself with these methods and compare them with the Euclidean method in 2-dimensions. After analyzing the results obtained, I have extended two of these methods to a parallel environment, as discussed in Chapter 4. The internship concluded with the implementation of one of these methods into ARMEN.
report summarizes this work. It is organized as follows. First, the intership environment is described in Section 1.2. Second, we detail the Eikonal equation and the solving method in Chapter 2. Third, we present the *Fast Methods* and the toy model in Chapter 3. Then, we discuss the parallelized version in Chapter 4. Finally, we describe the result of my implementation on ARMEN in Chapter 5.

1.2 Présentation du CEA

1.2.1 Le CEA

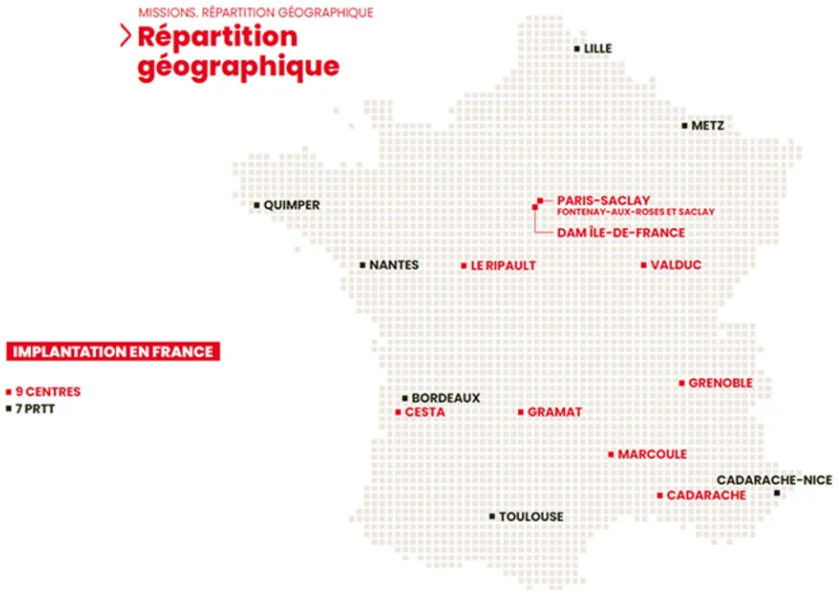
Dans ce chapitre, vous trouverez une présentation du CEA qui m'a été fournie par l'entreprise:

Créé en octobre 1945, afin de poursuivre « les recherches scientifiques et techniques en vue de l'utilisation de l'énergie atomique dans divers domaines de la science, de l'industrie et de la Défense nationale », le CEA, aujourd'hui Commissariat à l'énergie atomique et aux énergies alternatives, a pour mission de faire progresser la science et le savoir au service de la société, d'accompagner la réindustrialisation de notre pays et de contribuer à la souveraineté technologique française et européenne en relevant les grands défis de ce siècle : l'énergie et l'environnement, le climat, la santé, la défense et la sécurité, le numérique.

Acteur majeur de la recherche, du développement et de l'innovation, le Commissariat à l'énergie atomique et aux énergies alternatives intervient dans cadre de six missions :

- Défense et la sécurité ;
- Energies ;
- Transition numérique ;
- Technologies pour la santé ;
- Recherche fondamentale ;
- Assainissement-démantèlement.

S'appuyant sur une capacité d'expertise reconnue, le CEA participe à la mise en place de projets de collaboration avec de nombreux partenaires académiques et industriels.
Le CEA est implanté sur 9 centres et 7 plateformes régionales de transfert de technologie (PRTT).



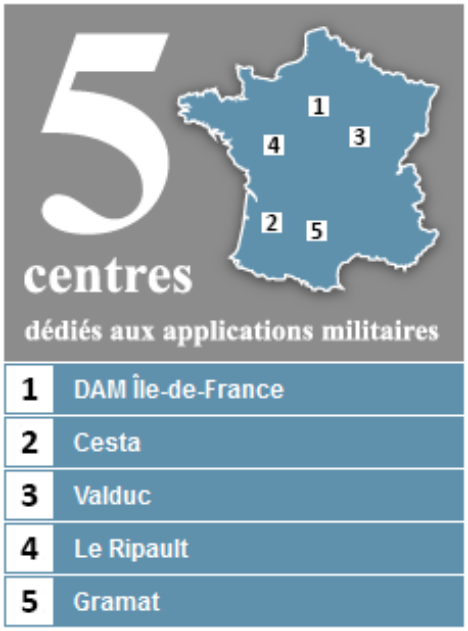
Reconnu comme un expert dans ses domaines de compétence, le CEA est pleinement inséré dans l'espace européen de la recherche et exerce une présence croissante au niveau international.

Le CEA compte plus de 21 000 technicien·ne·s, ingénieur·e·s, chercheur·e·s et collaborateur·trice·s pour un budget de 5,8 milliards d'euros (chiffres publiés fin 2022).

1.2.2 Le CEA/DAM

Une direction au service de la dissuasion

La Direction des applications militaires du CEA (DAM) a pour missions :



- de concevoir, fabriquer, maintenir en condition opérationnelle, puis démanteler les têtes nucléaires qui équipent les forces nucléaires aéroportée et océanique françaises ;
- de conduire les travaux de conception et réalisation des réacteurs nucléaires équipant les sous-marins et porte-avions de la Marine nationale ;
- de soutenir la Marine nationale pour le suivi en service et le maintien en condition opérationnelle de ces réacteurs ;
- d'approvisionner les matières nucléaires stratégiques pour les besoins de la dissuasion ;
- d'offrir un appui technique aux autorités nationales et internationales dans la lutte contre la prolifération et le terrorisme nucléaires;
- d'apporter son expertise au profit de la Défense dans le domaine de l'armement conventionnel.

Une direction ouverte à la recherche

Le partage national et international des connaissances (lorsqu'il est possible), la confrontation à l'évaluation scientifique extérieure, l'intégration à des réseaux de compétences constituent des gages de crédibilité scientifique.

Les équipes de la DAM réalisent chaque année environ 2000 publications et communications scientifiques. Cette ouverture de la DAM passe également par la mise à la disposition de la communauté des chercheurs de ses moyens expérimentaux et par la contribution de ses équipes à d'autres programmes de recherche.

Une direction actrice de la politique industrielle française

La DAM partage très largement son activité avec l'industrie française : c'est ainsi que le montant des achats, auprès de celle-ci, représente plus des deux tiers de son budget ; le dernier tiers se répartit entre les salaires des personnels (un cinquième) et les taxes.

La politique industrielle de la DAM est originale à plus d'un titre :

- d'abord parce que la DAM conserve la maîtrise d'œuvre d'ensemble de la grande majorité des systèmes dont elle a la responsabilité : elle veille ainsi au juste équilibre entre les grands groupes industriels de la Défense et les PME souvent innovantes, en contractualisant directement avec ces dernières, leur permettant ainsi de recevoir la juste rémunération de leur production ;
- ensuite, parce que la répartition de son budget est sous-tendue par une répartition des travaux : la DAM conduit la recherche dans ses laboratoires grâce à son personnel de haut niveau

scientifique et technologique. Une fois la définition d'un produit acquise, la DAM transfère la définition et les procédés vers les industriels qui en réalisent le développement, puis la production.

La DAM a également pour objectif que ses centres participent à la vie économique locale par leur implication dans les pôles de compétitivité. Hors de son propre champ d'utilisation, elle valorise ses recherches par le transfert de technologies vers l'industrie et le dépôt de nombreux brevets.

Les centres de la DAM

La DAM comprend cinq centres aux missions homogènes, dont les activités se répartissent entre la recherche de base, le développement et la fabrication :

- **DAM Ile-de-France (DIF)**, à Bruyères-le-Châtel, où sont menés les travaux de physique des armes, les activités de simulation numérique et de lutte contre la prolifération nucléaire. DIF est aussi le centre responsable de l'ingénierie à la DAM ; enfin, au centre DIF est rattachée l'INBS-Propulsion Nucléaire du centre CEA/Cadarache, en région Provence Alpes-Côte d'Azur, où sont implantées les installations d'essais à terre et une partie des fabrications de la propulsion nucléaire ;
- **Cesta**, en Aquitaine, consacré à l'architecture des armes, aux tests de tenue à l'environnement. Il met en œuvre le Laser Mégajoule, équipement majeur de la Simulation ;
- **Valduc**, en Bourgogne, dédié aux matériaux nucléaires et à l'installation expérimentale Epure du programme Simulation ;
- **Le Ripault**, en région Centre, dédié aux matériaux non nucléaires (explosifs chimiques...) ;
- **Gramat**, en Midi-Pyrénées, qui conduit au profit de la Défense des activités en vulnérabilité des systèmes et efficacité des armements.

1.2.3 Le centre DAM - Île-de-France

Le CEA/DAM - Île de France (DIF) est l'une des directions opérationnelles de la DAM.

Le site de la DIF compte environ 2000 salarié.e.s CEA et accueille quotidiennement environ 600 salarié.e.s d'entreprises extérieures. Il est situé à Bruyères-le-Châtel à environ 40 km au sud de Paris, dans l'Essonne.

Les missions de la DIF comprennent :

- **La conception et garantie des armes nucléaires, grâce au programme Simulation**. L'enjeu consiste à reproduire par le calcul les différentes phases du fonctionnement d'une arme nucléaire et à confronter ces résultats aux mesures des tirs nucléaires passés et aux résultats expérimentaux obtenus sur les installations actuelles (machine radiographique, lasers de puissance, accélérateurs de particules) ;
- **La lutte contre la prolifération et le terrorisme**, en contribuant notamment au programme de garantie du Traité de Non-Prolifération et en assurant l'expertise technique française pour la mise en œuvre du Traité d'Interdiction Complète des Essais Nucléaires (TICE) ;

- **L'expertise scientifique et technique**, dans le cadre de la construction et du démantèlement d'ouvrages complexes ainsi que pour la surveillance de l'environnement et les sciences de la terre ;

- **L'alerte des autorités**, mission opérationnelle assurée 24h sur 24, 365 jours par an, en cas d'essai nucléaire, de séisme en France ou à l'étranger, et de tsunami dans la zone Euro-méditerranéenne. La DIF fournit aux autorités les analyses et synthèses techniques associées.

1.2.4 TERATEC

Cette technopole, a été créée à l'initiative du CEA afin de développer et promouvoir la simulation numérique haute performance, à proximité du site du CEA, sur la commune de Bruyères-le-Châtel, dans le département de l'Essonne.

Le Campus TERATEC est l'une des deux composantes de la Technopole TERATEC, l'autre composante étant le « Très Grand Centre de Calcul du CEA » (TGCC). Le département DSSI est responsable de cette installation.

Le **CCRT** (Centre de calcul recherche et technologie) et Atos, leader international de la transformation digitale, collaborent pour mettre à disposition des utilisateurs industriels du CCRT un des simulateurs quantiques les plus performants au monde. La machine, construite par Atos, permettra à des partenaires comme EDF, Safran, l'IFPEN ou encore le CEA lui-même d'évaluer les potentialités des technologies quantiques pour leurs besoins.
Baptisé Atos Quantum Learning Machine (QLM), ce simulateur permet aux partenaires du CCRT d'expérimenter des technologies de rupture afin de mieux maitriser l'évolution de leurs applications et de relever les défis associés à la simulation numérique, le Big Data ou encore l'Intelligence artificielle et le « Machine Learning ».
Un centre de compétences en calcul quantique a été créé pour monter un écosystème dynamique, fédérant industriels utilisateurs, et centres de recherche, afin de monter rapidement en compétences et développer l'expertise dans le domaine de l'informatique quantique.

1.2.5 Les activités informatiques au CEA/DAM - Île de France

Le système d'information de la DAM est constitué de 4 sous-systèmes : entreprise, technique, industriel et scientifique, concernant respectivement :

- les aspects budget, ressources humaines, logistique, sécurité des personnes et des biens, patrimoine ...

- les activités « Programme » de la DAM : bureau d'études, méthodes, calcul, fabrication, qualité des produits fabriqués, gestion des matières ...

- les procédés industriels : acquisition mesures, système de supervision, surveillance, commande contrôle, contrôle des utilités, accès, autocom télécom ...

- le domaine scientifique associant la simulation avec le calcul haute performance.

Les activités déclinées dans ces systèmes d'information dans lesquelles s'inscrivent les postes proposés, sont les suivantes :

- étude, conception et réalisation du système d'information du CEA/DAM ;

- définition, conception, déploiement et exploitation des réseaux et des serveurs ;
- étude, conception, développement et maintenance de codes et outils de simulation numérique et d'environnement logiciel dans le domaine du calcul haute performance ;
- étude, conception, développement de systèmes informatiques dans le domaine du calcul haute performance ;

Auxquelles deux activités transverses sont ajoutées :

- expertise et activités opérationnelles dans le domaine de la sécurité informatique ;
- animation scientifique dans le domaine du calcul scientifique haute performance.

Chapter 2

Eikonal equation

2.1 A static Hamilton-Jacobi equation

General case

The Eikonal equation is an important static Hamilton-Jacobi (HJ) equation. HJ equations take the following form:

$$H(x, T, \nabla T, \nabla^2 T) = 0,\ x \in \Omega \subset \mathbb{R}^D \tag{2.1}$$

where the Hamiltonian H is a continuous scalar function defined on $\Omega \times \mathbb{R} \times \mathbb{R}^D \times \mathcal{S}_m(\mathbb{R})$. $\mathcal{S}_m(\mathbb{R})$ denotes the vector space of $m \times m$ symmetric matrices, ∇T is the gradient and $\nabla^2 T$ is the Hessian of T. The existence and the uniqueness of viscosity solution of HJ equations are shown in Rouy and Tourin (1992).
The Eikonal equation is derived from static HJ where $H(x, \nabla T) = F(x)|\nabla T| - 1$. Therefore, this is a non-linear, first-order, hyperbolic partial differential equation that can be written as:

$$\begin{cases} |\nabla T(x)|F(x) = 1 & \text{for } x \in \Omega \subset \mathbb{R}^D \\ T(x) = g(x) & \text{for } x \in X_s \subset \Omega \end{cases} \tag{2.2}$$

where Ω is an open subset of $\mathcal{R}^D$, $F : \Omega \to \mathbb{R}^+$ representing a positive local speed function and a known function $g : \Omega \to \mathbb{R}^+$ representing the initial condition of a set of points X_s in Ω, Once solved $T(x)$ represents the time it takes to go from the closest point from X_s to the point x, following the speed on $F(x)$. This equation is widely used to simulate the propagation of a wave-front, that is fundamental in many applications including seismology for instance.

For ARMEN purpose

To model the distance to an interface between several materials for ARMEN (see Chapter 5), we will focus on Eq. 2.2 assuming these following conditions:

- The dimension $D = 2$ (and 3 in Section 3.3.5)
- $\Omega \subset \mathbb{R}^D$ represents a rectangular cartesian grid composed by $N = n_x \times n_y$ cells (where n_x and n_y denoted the number of cells per dimension). We refer to grid points $x_{ij} = (x_j, y_i)$ corresponding to a cell (i, j) of the grid. We denote $T_{ij} = T(x_{ij})$ that represents an approximation to the real value of $T(x)$.
- X_s represents the set of points modeling the interface between materials.
- the initial condition is given by $g(x) = 0$ for all x in X_s.
- the positive speed function is defined as $F(x) = 1$ for all x in Ω.

Therefore, the Eikonal equation defined by Eq. 2.2 can be rewritten for our purpose as follows:

$$\begin{cases} |\nabla T(x)| = 1 & \text{for } x \in \Omega \subset \mathbb{R}^D \\ T(x) = 0 & \text{for } x \in X_s \subset \Omega \end{cases} \tag{2.3}$$

Note that the positive speed function $F = 1$ for each cells of the cartesian grid. Therefore, the solution $T(x)$ represents the minimal Euclidean distance from X_s to x. In the following, we will consider T as the minimal distance-to-interface function.
To validate our implementation, we will compare this solution computed by Eikonal equation with the one given by the Euclidean norm.

2.2 First-order discretization of the Eikonal equation

In 2-dimension

The Eikonal equation is commonly discretized using an upwind-difference scheme to approximate the partial derivatives of T (Gómez et al., 2015; Detrixhe et al., 2013; Bak et al., 2010).
The partial derivatives of T using the upwind scheme can be approximated as:

$$\begin{array}{l} \frac{\partial T}{\partial x} \approx \max\left(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0\right) \\ \frac{\partial T}{\partial y} \approx \max\left(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0\right) \end{array} \tag{2.4}$$

where $D_{ij}^{\pm x}T$ and $D_{ij}^{\pm y}T$ represent the one-sided partial difference operator in direction $\pm x$ and $\pm y$ respectively and are defined as:

$$\begin{array}{l} D_{ij}^{\pm x}T = \frac{T_{i\pm1,j} - T_{i,j}}{\pm \Delta x} \\ D_{ij}^{\pm y}T = \frac{T_{i,j\pm1} - T_{i,j}}{\pm \Delta y} \end{array} \tag{2.5}$$

Here, Δx and Δy denote the grid size in the x and y directions, respectively. We detail the main steps to construct a numerical scheme for Eq. (2.3) proposed by Sethian (1996). First, we consider the solution:

$$\begin{aligned} |\nabla T(x)| &= \sqrt{\left(\frac{\partial T}{\partial x}\right)^2 + \left(\frac{\partial T}{\partial y}\right)^2} = 1 \\ &\approx \sqrt{\max\left(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0\right)^2 + \max\left(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0\right)^2} = 1 \end{aligned}$$

By elevating to the square, we obtain:

$$\max\left(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0\right)^2 + \max\left(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0\right)^2 = 1 \tag{2.6}$$

Eq. (2.6) can be rewritten as:

$$\max\left(\frac{T_{i-1,j} - T_{i,j}}{-\Delta x}, -\frac{T_{i+1,j} - T_{i,j}}{\Delta x}, 0\right)^2 + \max\left(\frac{T_{i,j-1} - T_{i,j}}{-\Delta y}, -\frac{T_{i,j+1} - T_{i,j}}{\Delta y}, 0\right)^2 = 1$$

Simplifying, we get :

$$\max\left(\frac{T_{i,j} - T_{i+1,j}}{\Delta x}, \frac{T_{i,j} - T_{i-1,j}}{\Delta x}, 0\right)^2 + \max\left(\frac{T_{i,j} - T_{i,j+1}}{\Delta y}, \frac{T_{i,j} - T_{i,j-1}}{\Delta y}, 0\right)^2 = 1$$

Which is equals to:

$$\max\left(\frac{T_{i,j}-\min(T_{i+1,j},T_{i-1,j})}{\Delta x},0\right)^2+\max\left(\frac{T_{i,j}-\min(T_{i,j+1},T_{i,j-1})}{\Delta y},0\right)^2=1$$

To simplify the notation, let us denote:

- $T=T_{i,j}$
- $T_x=\min(T_{i+1,j},T_{i-1,j})$
- $T_y=\min(T_{i,j+1},T_{i,j-1})$

With those notations, we obtain:

$$\max\left(\frac{T-T_x}{\Delta x},0\right)^2+\max\left(\frac{T-T_y}{\Delta y},0\right)^2=1 \tag{2.7}$$

Knowing that the distance will be computed from a source point from X_s and the computation is done for a positive speed ($F=1$), T is always greater than T_x and T_y. Thus, Eq. (2.7) can be simply rewritten as:

$$\left(\frac{T-T_x}{\Delta x}\right)^2+\left(\frac{T-T_y}{\Delta y}\right)^2=1 \tag{2.8}$$

By expanding, the scheme for the Eikonal equation (Eq. (2.3)) finally reads:

$$\frac{T^2-2T\times T_x+T_x^2}{\Delta x^2}+\frac{T^2-2T\times T_y+T_y^2}{\Delta y^2}=1 \tag{2.9}$$

Providing that T_x and T_y are known, to compute T, Eq. (2.9) can be written as a standard quadratic equation of the form $aT^2+bT+c=0$ with:

- $a=\Delta x^2+\Delta y^2$
- $b=-2(\Delta y^2T_x+\Delta x^2T_y)$
- $c=(\Delta xT_y)^2+(\Delta yT_x)^2-\Delta x^2\Delta y^2$

To simplify the description of algorithm, we will consider in Chapter 3 and 4 that the grid steping is equal for all direction namely $\Delta x=\Delta y=h$ this will no longer be the case in Chapter 5.
Therefore, Eq. (2.9) reads:

$$\frac{T^2-2T*T_x+T_x^2}{\Delta h^2}+\frac{T^2-2T*T_y+T_y^2}{\Delta h^2}=1$$

by multiplying by h^2, we get:

$$2T^2-2(T_x+T_y)\times T+(T_y^2+T_x^2)=h^2 \tag{2.10}$$

In n-dimension

The 2d first-order discretization of the Eikonal equation can be generalized for higher dimensions (see (Gómez et al., 2015)), with the grid size equal for all directions and T_d the generalization of T_x, T_y, the parameter of the quadratic equations reads:

- $a=D$
- $b=-2\sum_{d=1}^{D}T_d$
- $c=\sum_{d=1}^{D}T_d^2-h^2$

2.3 Solving the discrete Eikonal equation

We now present the algorithm designed to solve the Eikonal equation at a specific point using the method described in Section 2.2. In algorithm 1 we compute the distance on a point x_i,the approach involves retrieving the values of T in the neighbouring points of x_i. From these values, we determine the minima namely T_d along each axis and solve the resulting quadratic problem to find the distance for our point.

Algorithm 1 Solve Eikonal Equation (Gómez et al., 2015)

Input: x_i (point), T (distance of each point), h (grid step)
Initialize an empty list T_{values}
$size_{T_{value}} \leftarrow 0$
for $dim = 1$ to N **do**
    $min_t \leftarrow$ minimum T value from x_i neighbors along $dim-$axis
    if $min_t \neq \infty$ **and** $min_t < T(x_i)$ **then**
        $size_{T_{value}}$.push(min_t)
    end if
end for
if $size_{T_{value}} = 0$ **then**
    return ∞ ▷ happens if T has not been computed for any neighbor
end if
$T_{\text{values}} \leftarrow T_{\text{values}}$.sort() ▷ to compute the distance from the nearest point
if $size_{T_{value}} = 1$ **then**
    return $T_{values}[1] + h$ ▷ If we have only one neighbor, x_i is aligned with it, and the distance of x_i is the distance of the neighbor plus the step.
else
    for $dim = 1$ to a **do**
        $T_{\text{tilde}} \leftarrow \text{SolveQuadraticProblem}(x_i, T_{\text{values}}, h)$ ▷ Algorithm 2
        if $T_{\text{tilde}} < T_{\text{values}}[dim + 1]$ **then**
            break
        end if
    end for
end if
return T_{tilde}

The quadratic equation is computed with Algorithm 2.

Algorithm 2 Solve Quadratic Problem (Gómez et al., 2015)

Input: x_i (a point), dim (length of T_{values}), T_{values} (list of minimum if T for each direction), h (the grid size)
$sumT \leftarrow \sum_{d=1}^{T_{values_size}} T_{\text{values}}[d]$
$sumT^2 \leftarrow \sum_{d=1}^{T_{values_size}} T_{\text{values}}[d]^2$
$a \leftarrow T_{values_size}$
$b \leftarrow -2 \cdot sumT$
$c \leftarrow sumT^2 - \frac{h^2}{F[x_i]^2}$
$q \leftarrow b^2 - 4ac$
if $q < 0$ **then** ▷ Complex solution
    $t_{tilde} \leftarrow \infty$
else
    $t_{tilde} \leftarrow \frac{-b+\sqrt{q}}{2a}$
end if
if $t_{tilde} < max(T_{values})$ **then** ▷ Causality condition
    $t_{tilde} \leftarrow min(T_{values}) + h$
end if
return t_{tilde};

These two algorithms compute an approximate solution of the Eikonal equation on a point of a mesh x_{ij}. The challenge is that we can not freely browse the mesh. Indeed, to solve the equation at a point (i, j) we need the value T of at least one neighbor. Consequently, we need a method to browse the mesh in a relevent order to compute $T_{i,j}$ let us note that handling multiple sources presents a challenge. We need to determine a way to include these different sources in our traversal and propagate the information from all sources at the same speed to ensure that we do not recalculate a point multiple times. Three methods to deal with this problem are presented in the next chapter.

Chapter 3

Fast Methods grid exploration

3.1 Fast Methods

In this section, we detail methods to browse the mesh to efficiently compute T on the whole mesh from the source points X_s as described in Jeong and Whitaker (2008); Gómez et al. (2015); Zhao (2004). These methods are called *Fast Methods* and are very popular to compute distance maps (i.e. time-of-arrival) solving the Eikonal equation. Table 3.1 summarises some of the most common *Fast Methods*. Note that Gómez et al. (2015) have compared all of these methods for differents sources and mesh configurations. The main idea is to create an active band with all neighbors of the source points X_s, then spreading the information from the band to every neighbor of the band and to iterate until the whole mesh go through it. We briefly present three of these methods : Fast Marching Method (FMM, Section 3.1.1), Fast Iterative Method (FIM, Section 3.1.2) and Fast Sweeping Method (FSM, Section 3.1.3). We will compare the distance-to-interface computed by these three methods for some configurations.

Fast Methods	**References**
Fast Marching Method (FMM)	Sethian (1996)
Simplified FMM (SFMM)	Jones et al. (2006)
Untidy FMM (UFMM)	Yatziv et al. (2006)
Group Marching Method (GMM)	Kim (2001)
Fast Iterative Method (FIM)	(Jeong and Whitaker, 2008)
Fast Sweeping Method (FSM)	Zhao (2004)
Locking Sweeping Method (LSM)	Bak et al. (2010)
Double Dynamic Queue Method (DDQM)	Bak et al. (2010)

Table 3.1: Summary of some *Fast Methods*.

3.1.1 FMM — Fast Marching Method

The Fast Marching Method (FMM, (Sethian, 1999)) is the most commonly used solver for the Eikonal equation. It relies on partitioning the grid cells into three different sets:

- **Unknown**: Cells that have not been processed yet. These cells have no assigned value and still need to be evaluated.
- **Frozen**: Cells that have already been processed. Their values have been finalized and will not change.
- **Narrow Band**: Cells located on the interface between the Frozen and Unknown sets. These cells may already have values assigned, which can still be updated if a smaller value is found.

The efficiency of the FMM is highly dependent on how these three sets are implemented and managed. For the **Unknown** set, we need an efficient way to check if a point is in the set, and to remove it quickly when needed. For the **Frozen** set, we need an efficient way to check if a point is in the set, and simple insertion of new points. For the **Narrow Band**, more advanced operations are necessary: we need to support reordering, efficient insertion and removal (push/pop), retrieve the element with the minimal value(top), and modify the value of an existing element(increase). The algorithm of the Fast Marching Methods is detailed by Algorithm 3 and an illustration of the mesh exploration by the FMM is given in Figure 3.1.

Algorithm 3 Fast Marching Method (Gómez et al., 2015)

1: **Input:** X (set of points), X_s (set of sources), T (distance for each point), h (step size)
2: **Initialize:**
3: Unknown $\leftarrow X \setminus X_s$, Narrow $\leftarrow X_s$, Frozen $\leftarrow \emptyset$
4: Set all T values to ∞ except for points in X_s where $T = 0$
5: **while** Narrow is not empty **do**
6: $x_{\min} \leftarrow$ Narrow.top() ▷ Element with minimum distance
7: **for** each x_i neighbor of $x_{\min}$ **do**
8: **if** $x_i \notin$ Frozen **then**
9: $t_i \leftarrow \text{SolveEikonal}(x_i, T, F, n, h)$ ▷ Algorithm 1
10: **if** $t_i < T[x_i]$ **then**
11: **if** $x_i \in$ Narrow **then**
12: Narrow.increase(x_i, t_i)
13: **else**
14: Narrow.push(x_i)
15: Unknown.remove(x_i)
16: **end if**
17: **end if**
18: **end if**
19: **end for**
20: Frozen.add$(x_{\min})$
21: Narrow.pop$(x_{\min})$
22: **end while**
23: **return** T

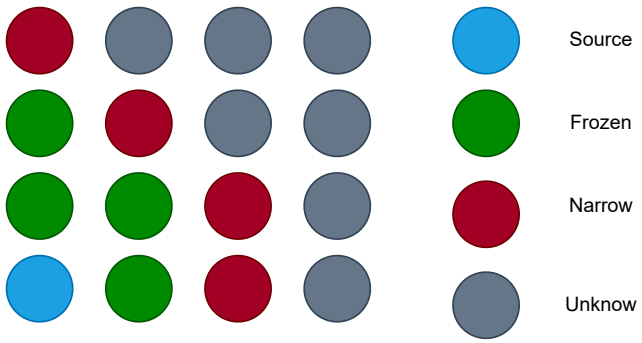


Figure 3.1: Illustration of distance function calculation from a source point (blue point). This figure is a snapshot of the calculation made by FMM with the three sets: Frozen (green points), Narrow (red points) and Unknown (gray points).

The grid is explored from source point X_s. Each time we process a point, it is marked as Frozen(line 3;20), and his Unknown neighbors are added to the Narrow (line 3;14). The algorithm

always selects the point with the smallest value in the Narrow band to continue. Since the Narrow band must stay ordered, each insertion or update requires sorting (line 12), leading to an overall complexity of $\mathcal{O}(Nlog(N))$.

3.1.2 FIM — Fast Iterative Method

One issue with the grid exploration of the FMM is how often update are required, which can make the algorithm quite slow in some cases. To adress this issue, we consider another method, the Fast Iterative Method (FIM) described in Jeong and Whitaker (2008); Cai et al. (2023); Gómez et al. (2015). This method iterates over the Narrow band of points as in 3.1. The narrow band is initiated with the neighbors of all sources X_s,then we solve the Eikonal equation on the Narrow point,then their Unknown neighbors are added to the end of the list,then we repeat the process for each new point add to the Narrow band. We also add a convergence criteria to know when to pop an element of the band and add it to the Frozen points, meaning we may need to itere multiple time on a point. In the worst case, the entire grid may end up in the Narrow band, leading to a time complexity of $\mathcal{O}(N)$, where N is the number of grid points. The algorithm of the Fast Iterative Method is detailed in algorithm 4 :

Algorithm 4 Fast Iterative Method (Gómez et al., 2015)

1: **Input:** X_s (set of sources), T (distance for each point), h (step size), ϵ (convergence parameter)
2: **Initialize:**
3: Frozen $\leftarrow X_s$
4: Narrow $\leftarrow X_s$ neighbor ▷ unordoned set
5: Set all T values to ∞ except for points in X_s where $T = 0$
6: **while** Narrow is not empty **do**
7: **for** x_i in narrow **do**
8: $T_i \leftarrow \text{SolveEikonal}(x_i, T, n, h)$
9: $\tilde{T}_i \leftarrow T[x_i]$
10: **if** $|T_i - \tilde{T}_i| < \epsilon$ **then**
11: **for** x_j neighbors of x_i **do**
12: **if** $x_j \notin$ frozen **then**
13: $p \leftarrow T[x_j]$
14: $q \leftarrow SolveEikonal(x_j, T, h)$ ▷ Algorithm 1
15: **if** $q < p$ **then**
16: narrow.push(x_j)
17: **end if**
18: **end if**
19: **end for**
20: narrow.delete(x_i)
21: frozen.add(x_i)
22: **end if**
23: **end for**
24: **end while**

3.1.3 FSM — Fast Sweeping Method

The last method we consider is the Fast Sweeping Method (FSM) described in (Zhao, 2004; Bak et al., 2010). This method is an iterative algorithm which solve the Eikonal equation by successively sweeping the whole grid in a specific order. The algorithm is based on Gauss-Seidel iterations in alternating directions. Unlike FMM or FIM, this method does not require any special data structures. We only need the dimension, the list of source points X_s, and the grid Ω. The method

relies on sweeping the entire grid in a set of predefined directions. For a given dimension, we typically consider:

- 2 directions in 1D (left to right, right to left),
- 4 directions in 2D (each cardinal direction: left to right, right to left, top to bottom, bottom to top),
- 8 directions in 3D (same as 2D but adding another axis,doubling the number)
- More directions (diagonals) can be added depending on the problem.

During each sweep, we iterate through the grid in a specific direction, updating the values by solving the Eikonal equation at each point. This process is repeated for all sweep directions no change are observed between two sweeps. The overall complexity is $\mathcal{O}(d \cdot N)$, where d is the number of sweep directions and N is the number of grid points. The Fast Sweeping Method is detailed by Algorithm 5

Algorithm 5 Fast sweeping Method (Zhao, 2004)

X (set of points), X_s (set of sources), T (distance for each point), h (step size), N (dimension)
$sweep\_dir \leftarrow [1 \dots 1]$ of len N
Initialize:
    Set all T values to ∞ except for points in X_s where $T = 0$
$stop \leftarrow false$
while !stop **do**
    $sweep\_dir \leftarrow getSweepDirection$ ▷ Algorithm 6
    $stop \leftarrow sweep(X, T, F, sweep\_dir, N, n, h)$ ▷ Algorithm 7
end while
return T

To compute the direction of the current sweep, we use Algorithm 6,

Algorithm 6 getSweepDirection (Zhao, 2004)

N (dimension) , $sweep\_dir$ (sweeping direction) ▷ sweep_dir is composed of 1 or -1
$i \leftarrow 0$
while $i < N$ **do**
    $sweep\_dir[i]$ += 2 ▷ if direction was -1 it become 1
    if $sweep\_dir[i]$ <= 1 **then**
        break
    else
        $sweep\_dir[i]$ = -1 ▷ if direction was 1 it become -1
    end if
    $i \leftarrow i + 1$
end while

And to perform the sweep, we use Algorithm 7.

Algorithm 7 Sweep (Zhao, 2004)

1: X (set of points), T (distance for each point), $SweepDirection$(sweeping direction), h (step size), N (dimension)
2: $stop \leftarrow true$
3: **for** $X_i \in X followingSweepDirection$ **do**
4: $\quad \tilde{T}_i \leftarrow \text{SolveEikonal}(x_i, T, F, n, h)$ ▷ Algorithm 1
5: $\quad$ **if** $T_i > \tilde{T}_i$ **then**
6: $\quad\quad T_i \leftarrow \tilde{T}_i$, $stop \leftarrow False$
7: $\quad$ **end if**
8: **end for**
9: return stop

Figure 3.2 shows the four sweep directions in 2D represented with arrows. We observe that each mesh point is visited by the second sweep in this configuration, and the third and fourth sweep serve the purposed of verifying the convergence of the distance.

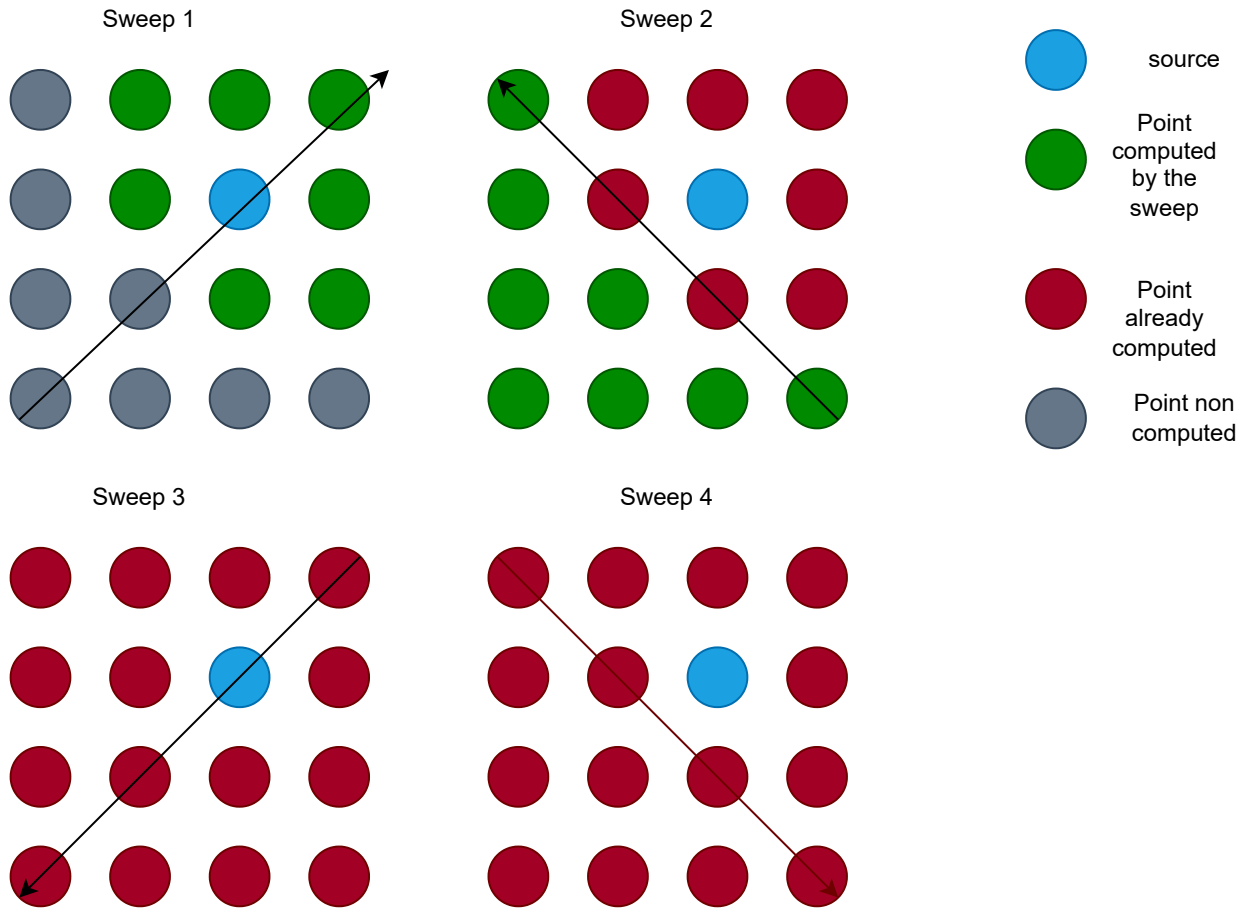


Figure 3.2: FSM sweep direction in 2D.

3.2 *Fast Methods* implementation

I have implemented the three methods presented in Section 3.1 in a 10.000 lines sequential C++ code for both 2D and 3D *Fast Methods*, alongside a Python code for visualization purpose, using gitlab to keep version of the project, a public repository with final version is available on my github.

In the following section, I present the results obtained from my sequential code along with performance comparaisons. All execution were performed on the Inti supercomputer and the -O2 optimization was used with the g++ compiler.

3.3 Numerical Results

In this section, we present the numerical result obtained by my implementation. In particular, we compare the three *Fast Methods* (FMM, FIM and FSM) in terms of error and performance. For that, we use the Euclidean norm ($T_{eucli_{ij}}$) as a reference solution and compute an error named $error_{ij}$ for each cell grid $x_{ij} = (i, j)$ by the following formula:

$$error_{ij} = |T_{eucli_{ij}} - T_{ij}| \tag{3.1}$$

3.3.1 Simple configuration with random sources - numerical result

First, we consider a simple configuration of a regular cartesian grid $\Omega = [0, 10] \times [0, 10]$ composed by $N = n_x \times n_y$ points with $n_x = n_y = 400$. The set of source points X_s is defined by $n_s = 5$ points that are randomly located in Ω.

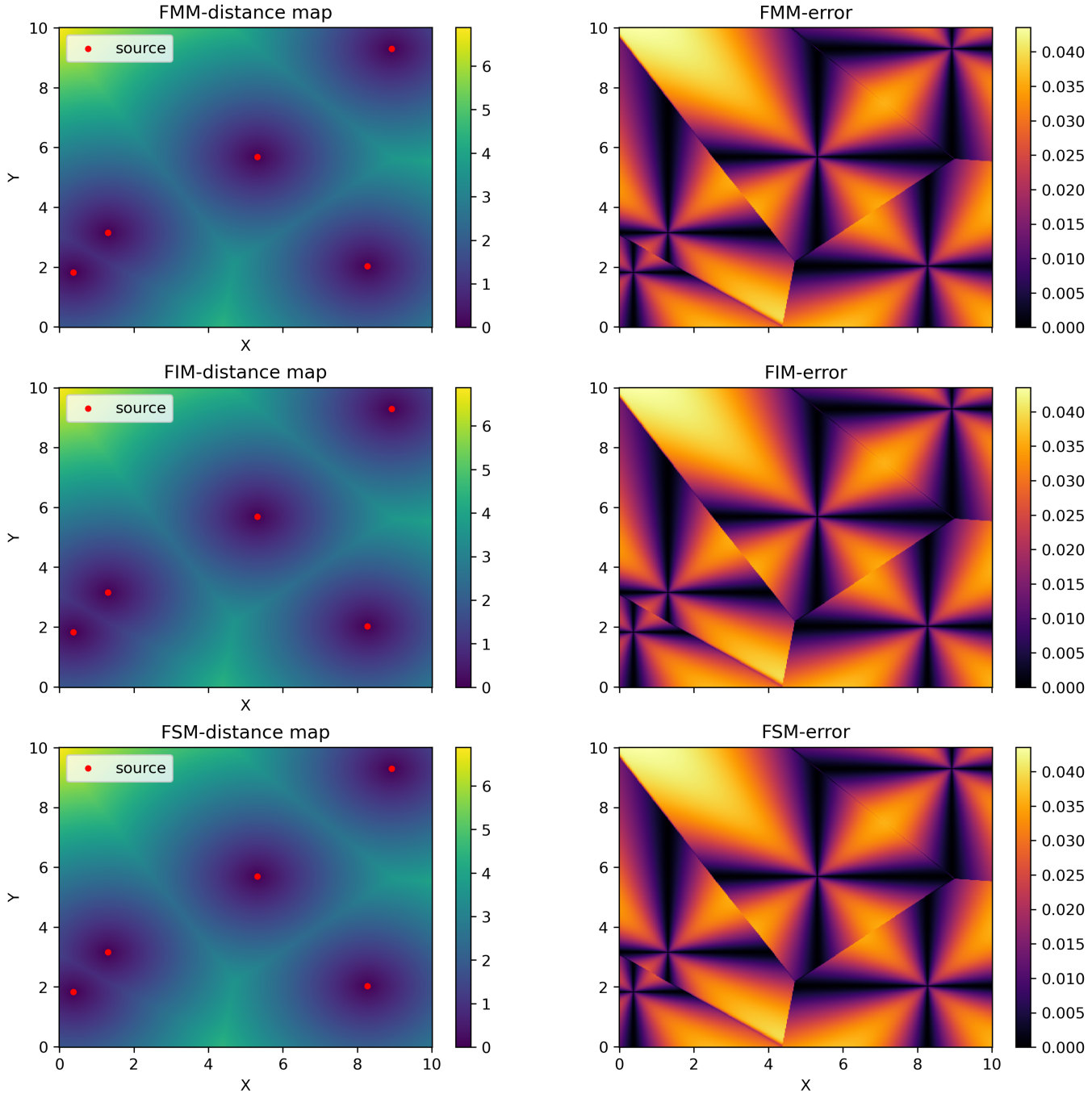


Figure 3.3: Distance map (left column) and error map (right column) for the FMM/FIM/FSM with $n_s = 5$ source points (red points) randomly located in $\Omega = [0, 10] \times [0, 10]$ composed by $N = 400 \times 400$ points. The colorbar shows the minimal distance to the source points (left column) and the error at each grid point compared to the Euclidean norm (right column).

In Figure 3.3, we observe that the three *Fast Methods* (FMM/FIM/FSM) produce a similar distance-to-interface map (left column) as expected. As explained in the introduction of this section, we have computed the error made by the three *Fast Methods* at each grid point of Ω using Eq. (3.1). We see that the three *Fast Methods* computed an exact distance in the four cardinal directions (shown in black color). Then, we observe that the error is higher in the diagonal direction (up to 1%) of all source points. This can be explained by the first-order discretization and the upwind scheme defined to solve the Eikonal equation (Section 2.2). Algorithm 2 is used to compute the distance. For the grid points aligned with a source point, the minimal distance is only the step size multiplied by the number of cells between the given point and the closest source point. As Ω is a regular cartesian grid, the minimal distance corresponds to the Euclidean distance.
However, for the grid points located in the diagonal direction of a point source, the distance is

computed with a quadratic equation (Eq. (2.9)), which lead to an approximation of the Euclidean distance.
Then, in Figure 3.4, we plot the tag of the closest source for each grid point (left column) computed by the three *Fast Methods* (FSM/FIM/FMM). We have modified our implementation during the grid exploration by propagating the tag by the closest neighbor in order to have access to this information. This tag could be helpful for ARMEN purpose.
We have also computed the error compared to the tag given by the Euclidean method (right column). The error is equals to 0 (black) if the tag is similar and to 1 (yellow) if the tag is different We observe that the propagation of the information is accurate except at the edge of each cluster, where equidistance issues may arise.

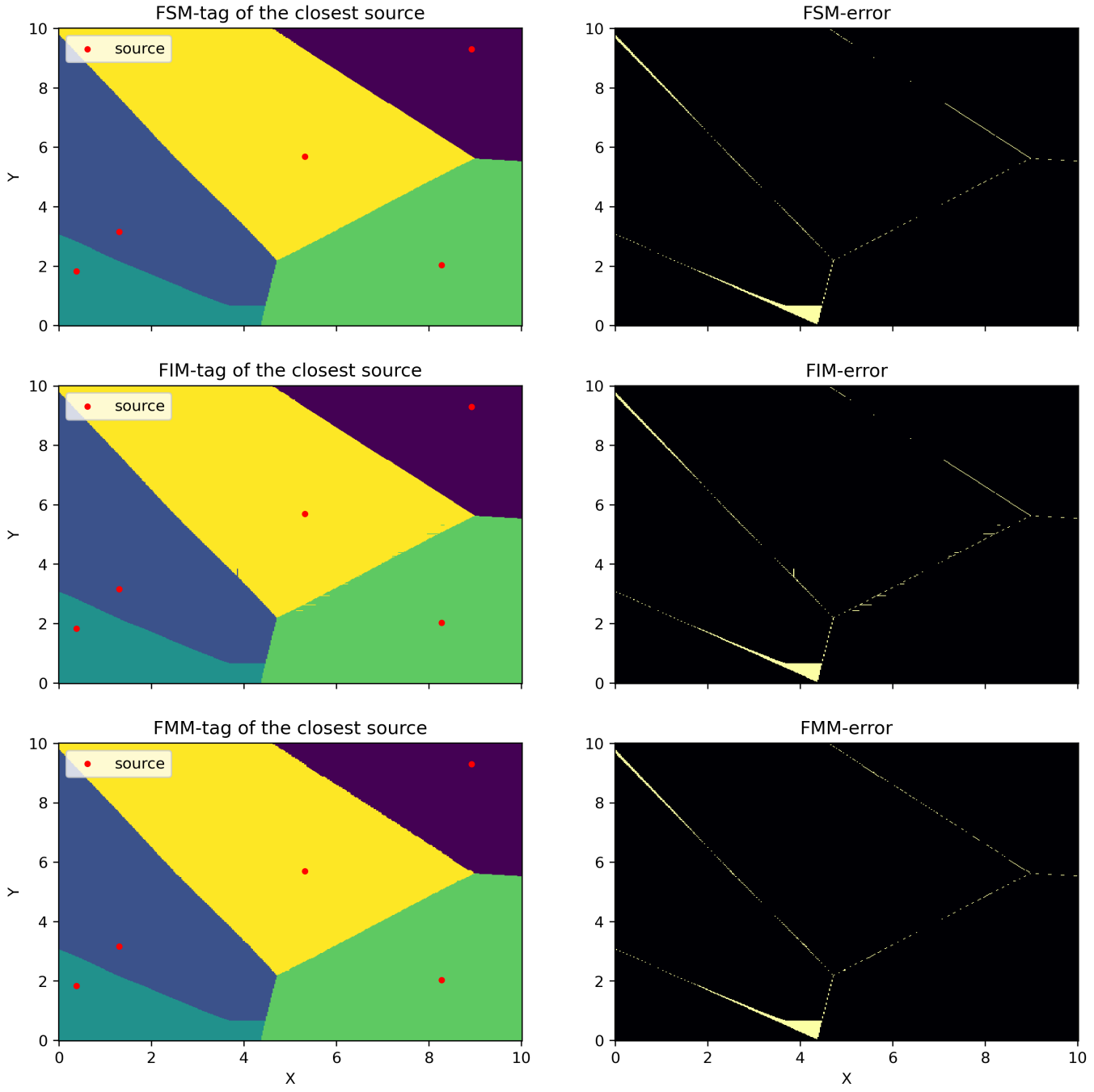


Figure 3.4: Tag of the closest source (left column) and associated error (right column) with source points (red points) for a mesh of $N = 400 \times 400$ and $n_s = 5$. Each color is associated to a tag of a point source. For the error, black color correspond to points where correct tags are computed.

3.3.2 Simple configuration with random sources - performance evaluation

Now, we take a look at the performance of the three *Fast Methods* (FMM/FIM/FSM) and Euclidean method in terms of CPU time and the influence of two critical parameters of the modeling: (i) the mesh size $N = n_x \times n_y$ (i.e. cells number) and, (ii) the number n_s of source points that modeled the interface X_s. For this study, we consider $n_x = n_y$. Therefore, the mesh size N is equals to n_x^2.

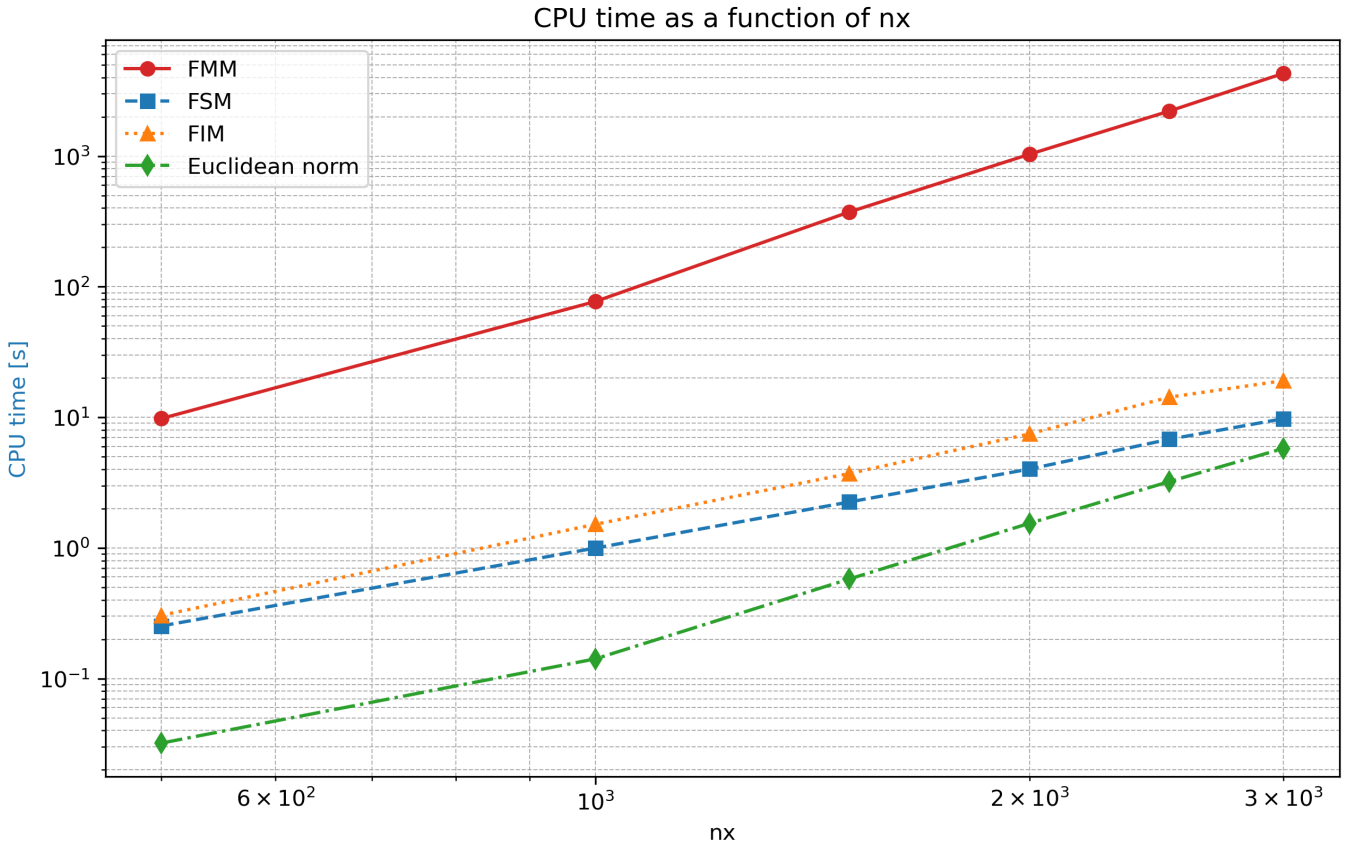


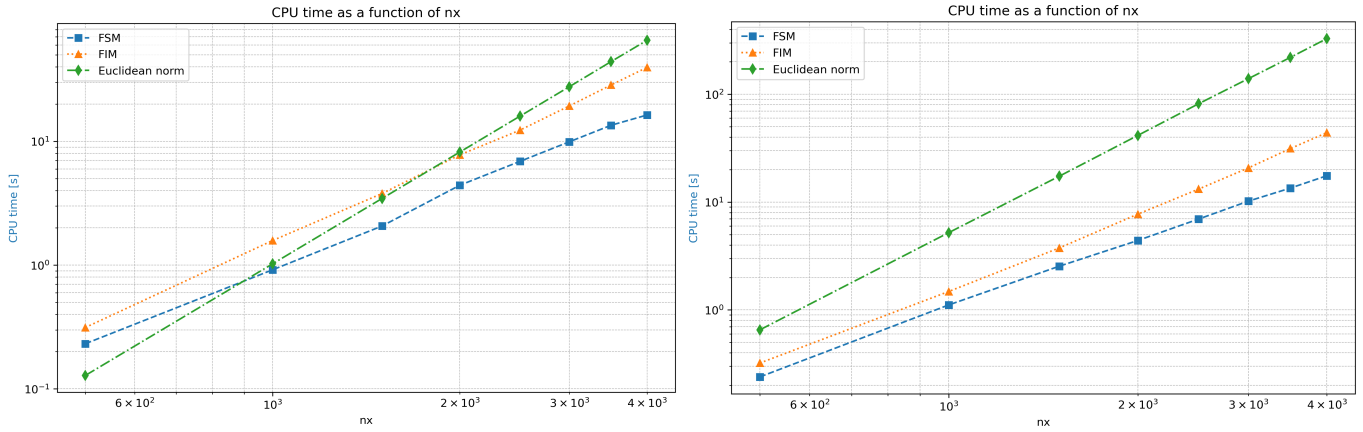
Figure 3.5: CPU time of FMM (red curve), FSM (blue curve), FIM (orange curve) and Euclidean method (green curve) in logarithmic scale for $\Omega = [0, 10] \times [0, 10]$ composed by $N = n_x \times n_x$ with n_x varying from 5.10^2 to 3.10^3 with a source number $n_s = 50$.

First, Figure 3.5 shows, in logarithmic scale, the CPU time of FMM, FSM, FIM and Euclidean method for a mesh size N varying from $5.10^2 \times 5.10^2$ to $3.10^3 \times 3.10^3$, and a fixed amount of 50 sources points. For this configuration, the FMM is the slowest method by a factor greater than 100 for each mesh size. The Euclidean method is faster than the *Fast Methods* by a factor greater than 10 on mesh of size $5.10^2 \times 5.10^2$. However, as the mesh size increases, the factor is reduce to 4-5. To conclude, for $n_s = 50$ source points, the Euclidean method is faster than the *Fast Methods* for each mesh size. Then, for the rest of this report, as we saw in Figure 3.5, we have excluded the FMM because this method is less efficient in terms of CPU time compared to the two others *Fast Methods*.

Second, we consider the CPU time of FIM/FSM methods and the Euclidean method for higher number of sources n_s. The result are shown in Figure 3.6. In Figure 3.6, we observe the influence of the mesh size $N = n_x^2$ on CPU time for $n_s = 200$ and $n_s = 1000$ source points. We notice in Figure 3.6 (a) that with an higher number of sources, the CPU time for FSM and FIM is almost the same (± 1 seconds) for $n_s = 50$ (see Figure 3.5). This is in contrast to the Euclidean method, where the CPU time for every mesh size $N = n_x^2$ increases by a factor 5. We also observe that FSM and FIM are faster than the Euclidean method for mesh size bigger than $10^3 \times 10^3$ for FSM and $1.5.10^3 \times 1.5.10^3$ for FIM.

In Figure 3.6 (b), we notice that FIM and FSM have the same CPU times as in Figure 3.6 (a). Therefore, both method seems to be independent of the number of sources n_s. For the Euclidean

method, we observe an increase of a factor 5. In this case, both FIM and FSM are faster than Euclidean for all mesh size.



(a) $n_s = 200$ (b) $n_s = 1000$

Figure 3.6: CPU time of FSM (blue curve), FIM (orange curve) and Euclidean method (green curve) depending on the mesh size (from 500x500 to 4000x4000) and different number of sources: (a) $n_s = 200$, (b) $n_s = 1000$

To check the independence of the FSM and FIM to the number of sources, we have fixed the mesh size ($N = 3.10^3 \times 3.10^3$) and we have varied the number of sources n_s from 500 to 3000. The result is shown in Figure 3.7.

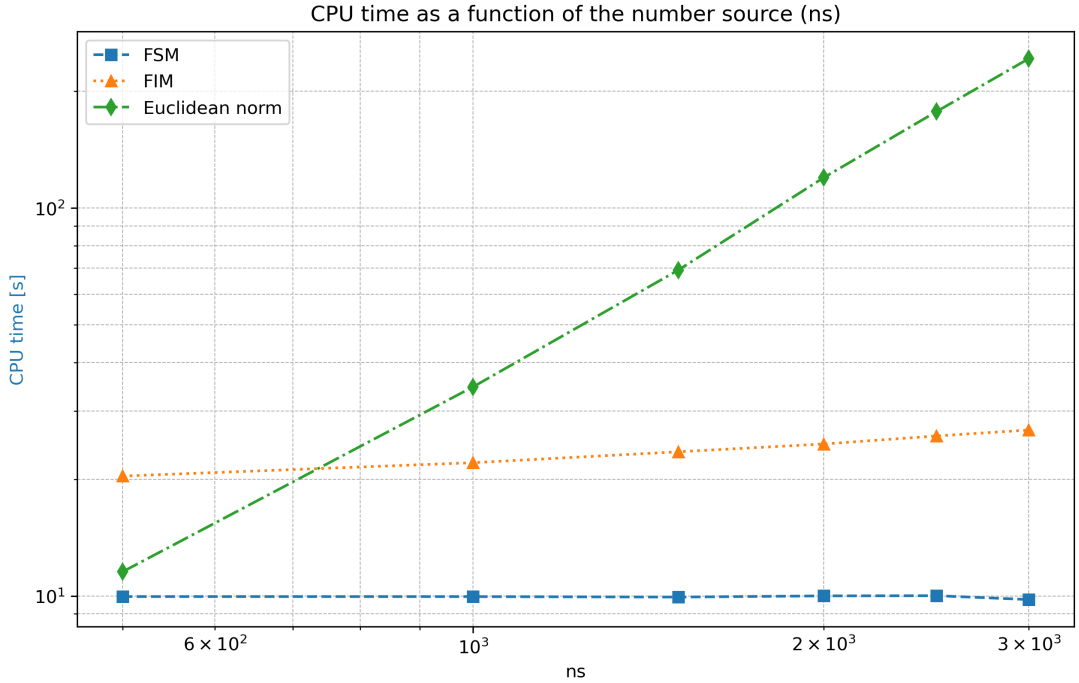


Figure 3.7: CPU time of FSM (blue curve), FIM (orange curve) and Euclidean method (green curve) depending on the number of source X_s (from 500 to 3000).

In Figure 3.7, we observe that a constant CPU times $\pm 0.1s$ for FSM, a small increase of less than a second for FIM, and a linear increase for Euclidean method. This result shows the complexity of FIM and FSM respectively $\mathcal{O}(N)$ and $\mathcal{O}(d \cdot N)$ are independent of number of sources, when the Euclidean norm complexity $\mathcal{O}(n_s \cdot N)$ is linear with n_s.

To conclude, for this simple configuration, we observe that the Fast Marching Method is too slow

and will not be considered for the rest of the study due to the necessity of sorting the list at each iteration. On the other hand, the FSM and FIM are more efficient in terms of CPU time than the Euclidean norm for the cases where the mesh size becomes larger and more source points are considered.

3.3.3 Configuration with an interface

We consider a second configuration with an interface instead of random source points.
Ω is still a rectangular cartesian grid composed by $N = n_x \times n_y$ cells on $[0, 10] \times [0, 10]$.

The interface is built using the following method:

1. Start with a random point at coordinate $(i, 0)$.
2. Iterate until reaching (i, n_y).
3. At each step, randomly choose among the following points: $(i, j+1)$, $(i-1, j+1)$, or $(i+1, j+1)$.

This configuration is closed to the real interface between materials defined in ARMEN code. Note that by construction, the number of source points is linked to the grid size : $n_s = n_y$.

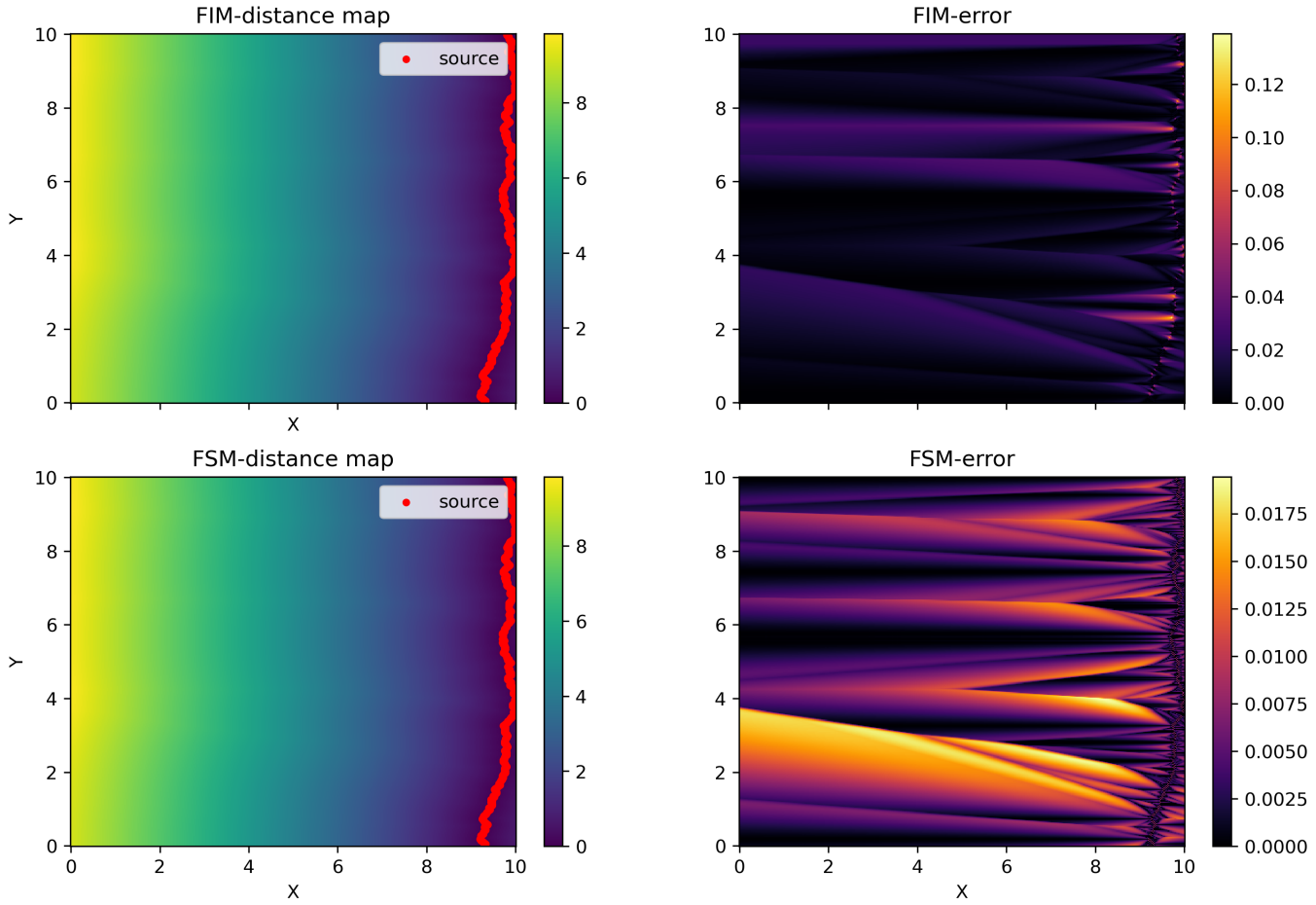


Figure 3.8: Distance map (left column) and error map (right column) for FIM/FSM with source points built as an interface (red point) for $N = 400 \times 400$ and $n_s = 400$.

First, we observe in Figure 3.8 similar results between FIM and FSM for this configuration. FSM exhibits the same error patterns as in Section 4.3, with an exact value when the closest point from the interface is aligned with the axis. We notice an higher error near the interface for FIM compare to the rest of the mesh. We notice an error near the interface for the FIM, but when we look at the

mean of the error equals to $1.2e^{-3}$ for FIM and $8e^{-4}$ for FSM. We realize both method have similar result.

Now, we want to compare the perfomance in CPU time for these interfaces for mesh size varied from 1000×1000 to 4000×4000. Note that by construction, the number of source points n_s is linked to the grid size : $n_s = n_y$.

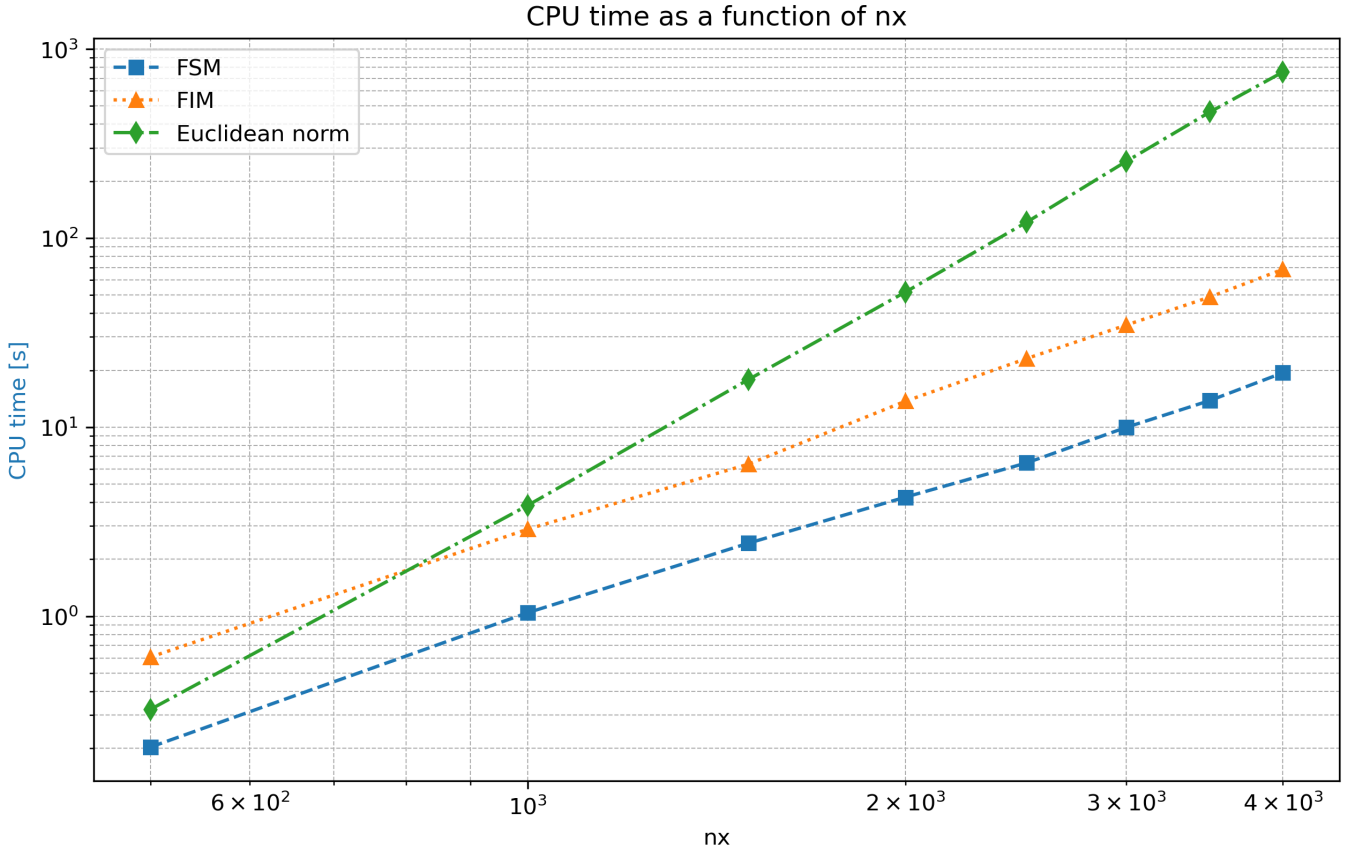


Figure 3.9: CPU time of FSM (blue curve), FIM (orange curve) and euclidean method (green curve) depending on the mesh size (from 1000×1000 to 4000×4000) ,and $n_s = n_y$

In Figure 3.9, we compare the CPU times for FIM, FSM, and the Euclidean method. We observe that FSM being faster than FIM and the Euclidean method for all mesh size, from a factor 4 on smallest mesh and 15 to the biggest. FIM has the same CPU time around 3s for the smallest mesh as Euclidean method, but as the mesh size increases, the time grow faster for Euclidean method than FIM 30s to 15s for 2000×2000, 40s to 150s for 3000×3000 and 90 to 300 for 4000×4000.

We compare to the case in Figure 3.6 a increase for the CPU time for FIM for the same size of mesh but a different number of sources when FSM still has the same times. This could be explain by the form of the interface with multiple deeps, where we can compute some false values on some points, making the algorithm compute multiple times some points to correct the result.

3.3.4 Introduction of a threshold for the grid exploration

The numerical model we want to improve, the slipping of two materials against each other for our target application is only only active near the interface, we only need to compute the distance to the interface near the interface. To that end we propose to introduce a threshold on the distance only compute the distance when the distance is lower than this threshold. The *Fast methods* unlike the Euclidean norm allow us to stop the propagation of the information by adding a condition in our algorithme when the distance computed reach the threshold, while the euclidean norms require to compute the distance on each points, for each sources points, then check if the points is within the threshold.

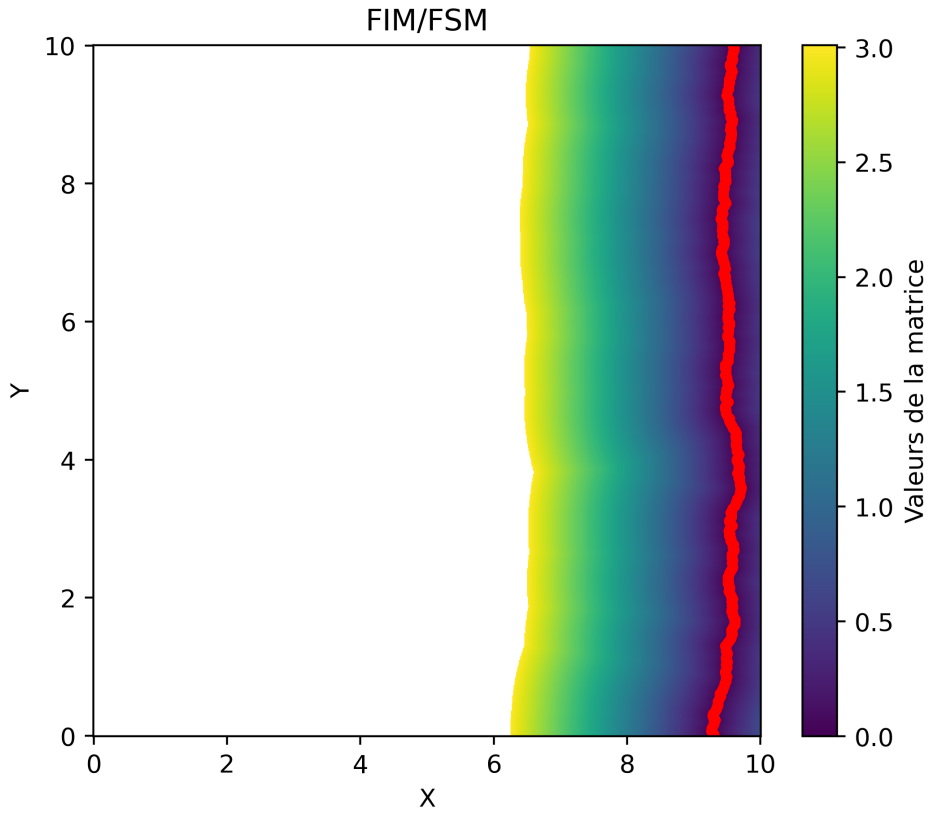


Figure 3.10: Exemple of result with FIM and FSM for the same configuration as in Figure 3.8 with a threshold equal to $100 * h$

We observe on figure 3.10 the same result as in figure 3.8, but we stoped to compute the distance when the distance values are biggest biggest than $100h$. In figure 3.11 we observe the impact of the threshold on the CPU times for FIM, FSM and the euclidean norm with a grid size varying from 500×500 to $4.10^3 \times 4.10^3$ and $n_s = n_y$.

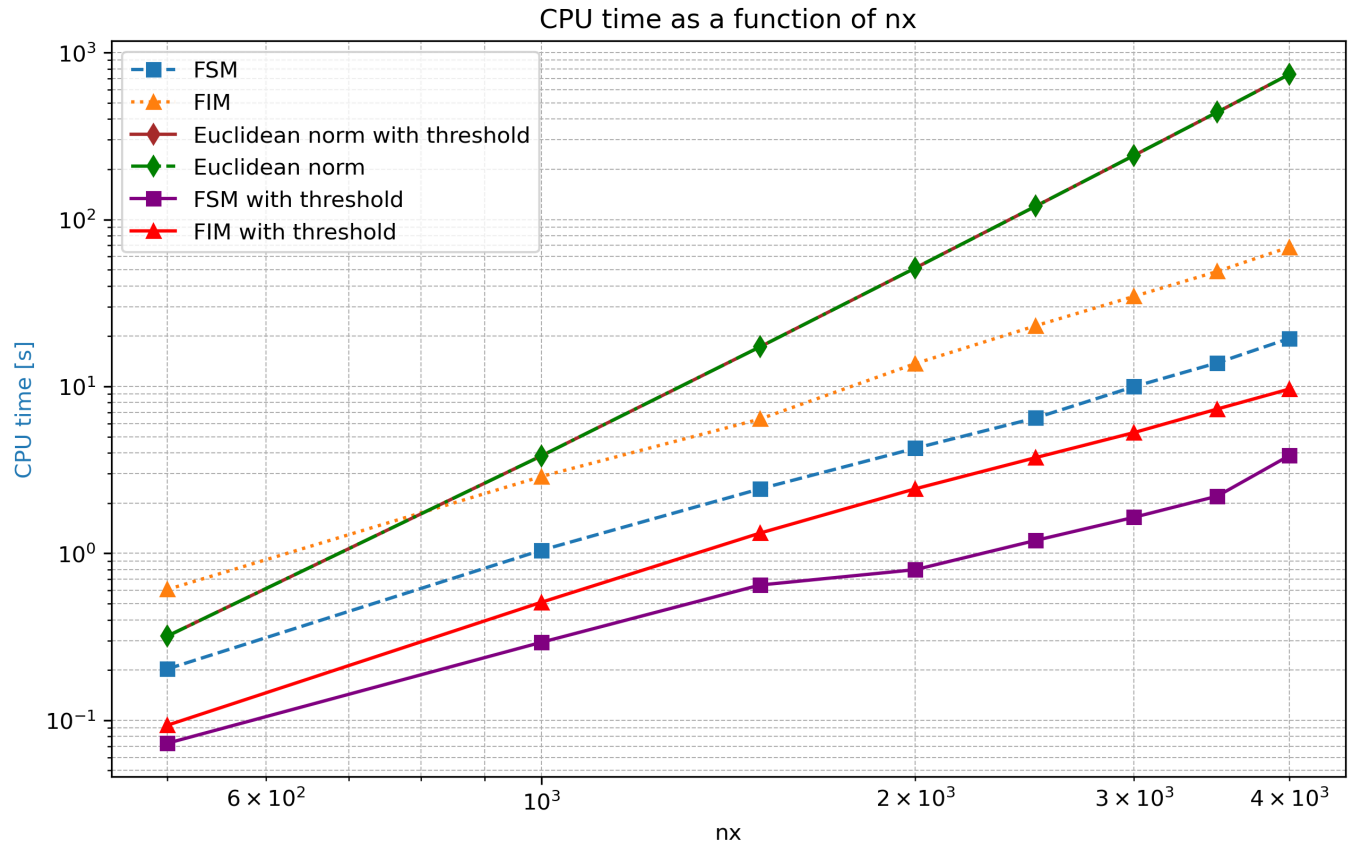


Figure 3.11: Comparaison of CPU times for FIM/FSM/euclidean method on an interface with (respectively orange, blue, green curve) and without a threshold (respectively red, purple, brown curve)

In Figure 3.11 we notice the FIM and FSM being faster up to a factor 10 in term of CPU time compare to their CPU time when computing on the whole mesh, as FIM and FSM were already 10-20 times faster than euclidean norm without the threshold, it is now up to 100 times faster than euclidean norm. Let us note euclidean curve being the same with and without the threshold.

3.3.5 3D extension

One strong point of *Fast Methods* is how easy it is to adapt the implementation for higher dimension problems. The Eikonal solver algorithm 1 is already shown in D-dimension. For FIM, we simply add two neighbors for each point along the z axis, and for FSM, we push in eight directions instead of four. The development of the numerical scheme is also available in the appendix 7 . While the main goal of the internship remains the implementation in 2D in the hydrodynamic code ARMEN, I implemented the algorithm in 3D to compare the different methode.

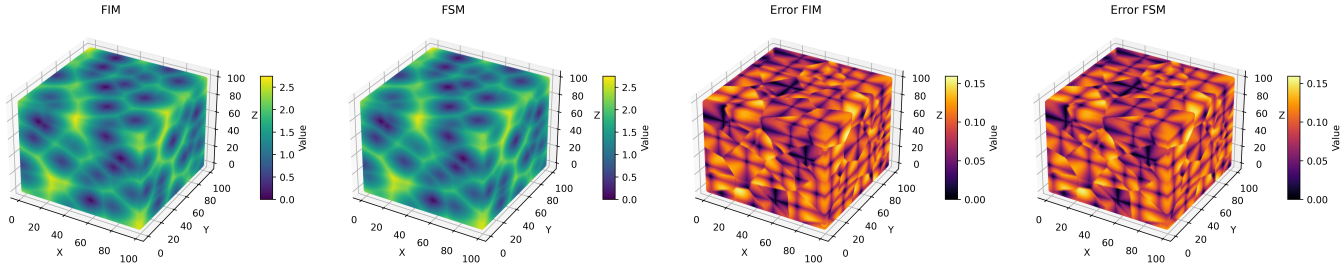


Figure 3.12: Distance map (left) and error (right) for the FIM/FSM , for $N = 50 \times 50 \times 50$ and $n_s = 200$.

In Figure 3.12, we observe a 3D cube composed of $N = 100 \times 100 \times 100$ points, with $n_s = 200$. As in 2D, we have the color bar on the left column representing the smallest distance to an interface, and on the right column, the error computed with Eq. 3.1. We notice the same error as in 2D, with exact precision in the cardinal directions and an error in the diagonals of up to 5%. The CPU time, in 3D become too great in sequential therefore we do not do a performance comparaison on this configuration.

3.4 First conclusion on the *Fast Methods*

In this part, we compared three *Fast Methods* and the Euclidean norm. These preliminary results show that the three *Fast Methods* offer precise computation of the Eikonal equation. Although the FMM ends up being significantly slower than the others, it laid the groundwork for the development of the FIM, which is a faster alternative. Therefore, we put the FMM aside for the rest of the study.

In terms of performances, we observe FIM and FSM are faster in term of CPU time for larger meshes or larger number of source points compared to the euclidean methods. FIM and FSM are both well suited to the introduction of a threshold for the grid exploration thats makes them even more appealing in terms of CPU Times. The next phase of the internship is to enhance those method through parallelization.

Chapter 4

Parallel *Fast Methods*

In this chapter, we only consider the FSM and FIM methods and we explore the possibility of parallelization of those methods. We focus here on the parallelization offered by the multi-threading library **OpenMP** (van der Pas et al., 2017) in C++. The parallelization for distributed memory with the library **MPI** (Gropp et al., 2014) will be consider later during the implementation in ARMEN.

4.1 Parallel FSM

A parallel method for the FSM can be found in some references e.g. (Zhao, 2007; Bak et al., 2010; Dang et al., 2013). The idea is to process each sweep simultaneously on copies of the mesh, then to perform a reduction operation to recover the minimum value found for each point. This lead to in a complexity of $O(N)$ and the cost of a reduction. The parallel FSM algorithm is described in algorithm 8:

Algorithm 8 Parallel FSM (Zhao, 2007; Bak et al., 2010; Detrixhe et al., 2013)

X (set of points), T (distance for each point), X_s (interface), h (step size), dim (dimension)
  Set all T values to ∞ except for points in X_s where $T = 0$
direction $\leftarrow$ all predefined direction
while all point did not converge **do**
  $u_1, u_2, u_3....u_n$ one for each direction $\leftarrow$ copie of T
  each thread get one u_i
  each thread get one direction
  each thread do sweep(X,T,direction,dim) for their direction storing their result in u_i ▷ algorithm 7
  $T \leftarrow min(u_i)$ for each point
  convergence $\leftarrow$ abs $(old_T[x_i] - new_T[x_i]) <$ seuil
end while
return T

4.2 Parallel FIM

For the FIM, the propagation of information depends on the number of sources and their placement. My initial approach to implementing FIM in parallel was to use OpenMP tasks on the narrow band. OpenMP tasks allow for dynamic assignment of work to threads: when a thread finishes computing a point, it can automatically be assigned a new point from the narrow band. However, due to the dynamic nature of the narrow band, this led to many data race issues. As a result, many critical sections were implemented, where only one thread could write to the band or update a value, significantly slowing down the algorithm. This problem is also described in references e.g. (Detrixhe et al., 2013; Dang et al., 2013).

On the second attempt, I retained the idea of using tasks, since the size of the narrow band is not fixed at any time. However, instead of parallelizing over the narrow band, we parallelized over the sources. This involved associating each thread with a specific set of sources, allowing each thread to explore the mesh from its assigned sources and solve the Eikonal equation independently. Each thread maintains a local narrow band and updates the global matrix with its results. With this approach, information propagates simultaneously from all sources, and each thread stops spreading when it encounters a region already processed by another thread. Even if one thread over-extends its propagation, neighboring threads correct the results during their own processing. The parallel FIM algorithm is detailed in algorithm 9.

Algorithm 9 Parallel FIM (Gómez et al., 2015; Cai et al., 2023)

Input: X_s (set of sources), T (distance for each point), h (step size), ϵ (convergence parameter)
Initialize:
  Set all T values to ∞ except for points in X_s where $T = 0$
  Spread equal number of source from X_s to each threads
  Narrow $\leftarrow$ X_s neighbors ▷ Narrow unordoned set
while Narrow is not empty **do**
  $Narrow_{copie} \leftarrow$ Narrow
  for x_i in $Narrow_{copie}$ **do** ▷ we add element to narrow to avoid data race
    $T_i \leftarrow \text{SolveEikonal}(x_i, T, h)$
    $\tilde{T}_i \leftarrow T[x_i]$
    if $|T_i - \tilde{T}_i| < \epsilon$ **then**
      for x_j neighbor of x_i **do**
        $p \leftarrow T[x_j]$
        $q \leftarrow SolveEikonal(x_j, T, h)$
        if $q < p$ **then**
          narrow.push(x_j)
        end if
      end for
      narrow.delete(x_i)
    end if
  end for
end while

4.3 Numerical results in parallel

4.3.1 Simple case with random sources

First, we compare the result obtained by our sequential algorithm of FIM and FSM (algorithm 4 and 5) and parallel algorithm of FSM and FIM (algorithm 8 and 9) on the configuration presented in fig 3.3 of a regular cartesian grid $\Omega = [0, 10] \times [0, 10]$ composed by $N = n_x \times n_y$ points with $n_x = n_y = 400$. The set of source points X_s is defined by $n_s = 5$ points that are randomly located in Ω.

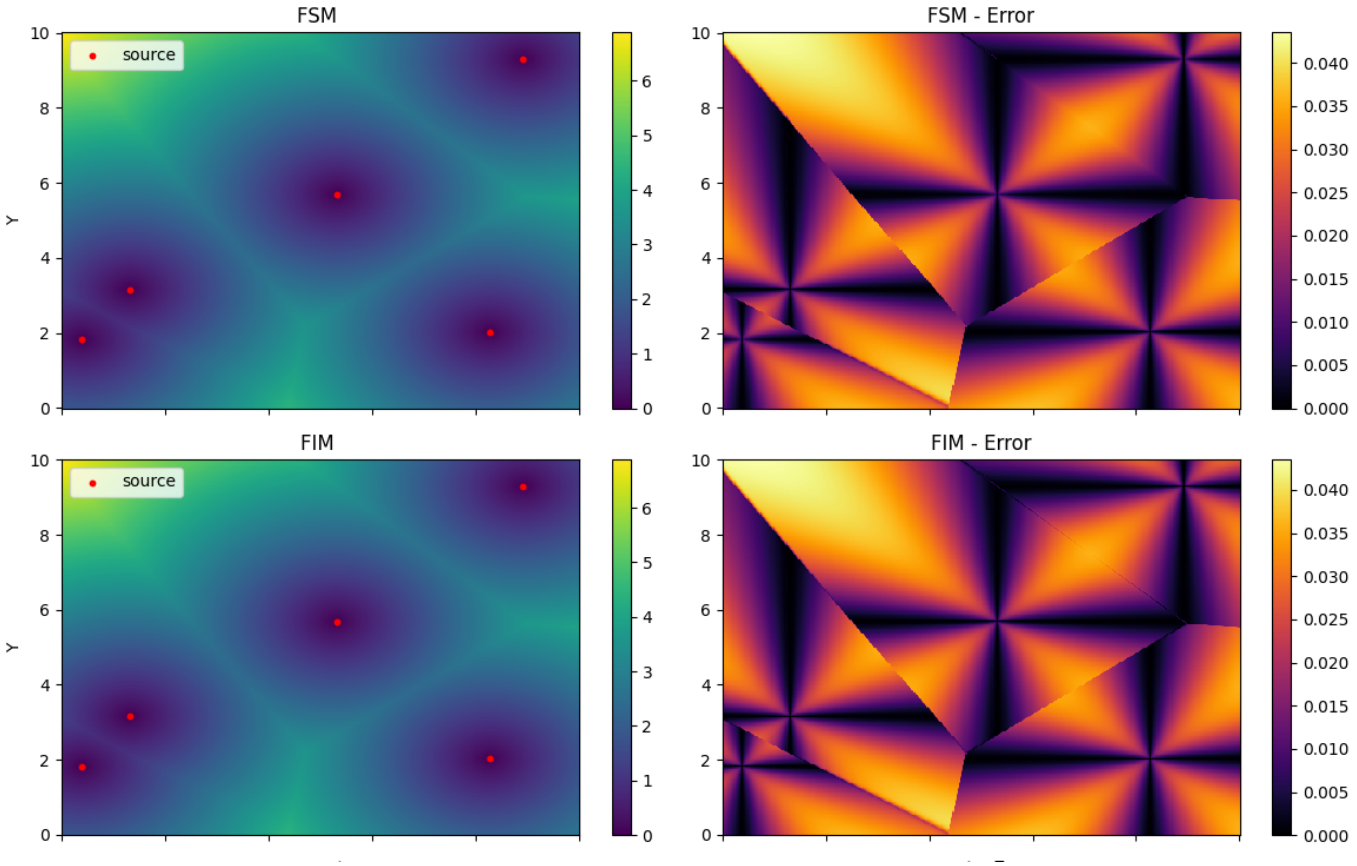


Figure 4.1: Distance map (left column) and error map (right column) for the parallel FIM/FSM with 4 threads, $n_s = 5$ source points (red points) randomly located in $\Omega = [0, 10] \times [0, 10]$ composed by $N = 400 \times 400$ points. The colorbar shows the minimal distance to the source points (left column) and the error at each grid point compared to the Euclidean norm (right column).

In Figure 4.1, we observe that both the parallel implementations of FSM and FIM produce the same distance-to-interface map as their sequential counterparts, as shown in Figure 3.3. As before, we computed the error for FSM and FIM at each grid point in Ω using Eq. (3.1). The error distribution remains identical to the sequential case, with exact distances computed in the four cardinal directions (shown in black) and a higher error in the diagonal directions (up to 1%) for all source points. Next, we examine the CPU time savings achieved with the parallel implementation, as presented in Table 4.1. The domain Ω consists of $N = 1000 \times 1000$ points, and $n_s = 50$ sources.

Number of OMP Threads	**1**	**2**	**4**	**8**	**12**
CPU TIME FIM [s]	4.65	2.3	0.965	0.345	0.315
CPU TIME FSM [s]	0.46	0.36	0.225	0.258	0.297

Table 4.1: CPU times for FIM and FSM as a function of the number of OpenMP threads.

Table 4.1 demonstrates a reduction in CPU time for both methods with the introduction of OpenMP parallelization. However, for FSM, we observe that beyond 4 threads, the CPU time increases due to the limitation of 4 sweeps in 2D.

4.4 Introduction of a threshold in parallel

As discussed in Section 3.3.4, for the application to ARMEN, we introduce a threshold around the interface. We compare the CPU time performance of all methods in this scenario in Figure 4.3, using the interface configuration described in Section 3.3.3 with a threshold value of $100 \times h$. Recall

that $n_s = n_y$. In Figure 4.2, we illustrate the distance-to-interface map obtained with the threshold applied for both FIM and FSM in their parallel implementations which is the same as in sequential.

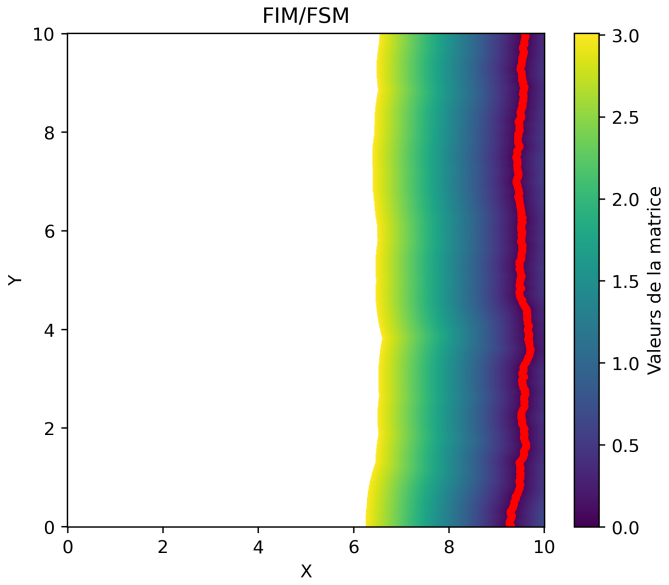


Figure 4.2: Distance map for the FSM and FIM in parallel for a configuration with a threshold as in figure 3.10, with $N = 400 \times 400$

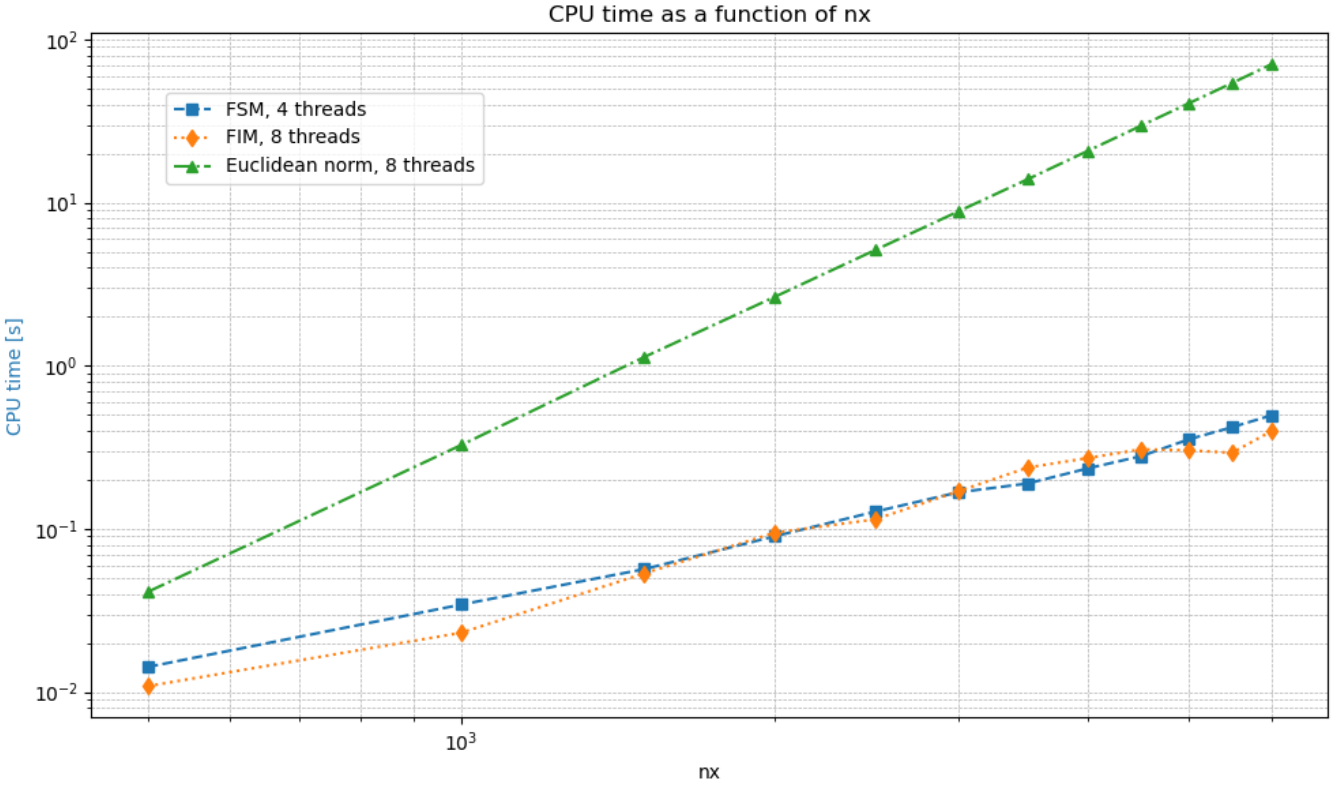


Figure 4.3: CPU TIMES of FIM (orange curve), FSM (blue curve), euclidean(green curve) for an interface with a threshold of 100 steps in each direction around the interfaces

In Figure 4.3, we compare the perfomance in CPU time for these interfaces for mesh size varied from 500×500 to 6000×6000. We observe that both FIM and FSM compute the distance within the threshold in less than one second for all mesh size, while euclidean norm CPU times grow from few second on mesh of size (1500×1500 , 2000×2000) to 70s on the biggest mesh (6000×6000).

4.4.1 Scalability of *Fast Methods*

In parallel computing, one important metric to evaluate the performance of an algorithm is its scalability. Scalability indicates how well the algorithm improves in terms of execution time as the number of threads increases.

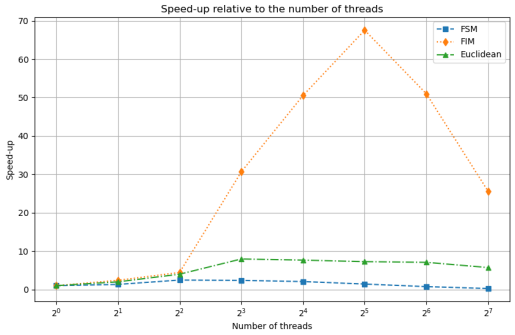
The speed-up of a parallel algorithm is calculated using the following formula:

$$\text{Speed-up} = \frac{T_{n0}}{T_{nk}}$$

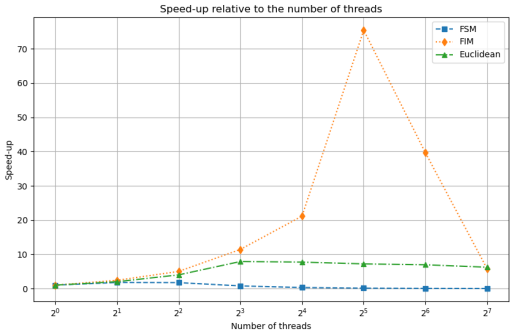
where:

- T_{n0} is the execution time of the algorithm using a single thread.
- T_{nk} is the execution time of the algorithm using k threads.

This formula measures how much a parallel algorithm speeds up the computation compared to a single-threaded execution.



(a) Speed-up of FSM(blue curve), FIM (orange curve), euclidean norm (green curve) on simple configuration with randomly placed source as in subsection 4.3.1, with $N = 4000 \times 4000$ and $n_s = 1000$



(b) Speed-up of FSM(blue curve), FIM (orange curve), euclidean norm (green curve) with a threshold as in figure 4.3, with $N = 4000 \times 4000$ and $n_s = 4000$

Figure 4.4: Speed up of FSM, FIM, and euclidean norm in different configuration

As shown in Figure 4.4, FIM achieves a speed-up of up to 70 for 64 threads; however, the speed-up decreases with additional threads. For FSM, the optimal speed-up is observed at 4 threads, as we only work with 4 directions, leaving additional threads with no tasks. Euclidean distance computation shows a speed-up in the tens at 8 threads, which slightly reduces with more threads.

4.4.2 Note on FSM

For the FSM we can look at the number of iteration of four sweeps necessary to converge (Zhao, 2007). Result are show in table 4.2. As we initialized our distance with an infinite value, we do not calcul the variations on the first iteration.

Size of mesh	400×400	1000×1000	2000×2000	3000×3000
Second iter	6.5×10^{-4}	2×10^{-4}	1×10^{-4}	1×10^{-5}
Third iter	3×10^{-7}	2×10^{-9}	1×10^{-10}	9×10^{-10}
Fourth iter	2×10^{-8}	8×10^{-10}	1×10^{-10}	2×10^{-10}

Table 4.2: Mean of variations for differents iterations of sweeps and mesh

Each iteration is equal to one sweep in each direction executed in parallel. To calculate the variations, we use the following formula for each point (i, j):

$$\text{mean} = \frac{1}{N} \sum_{i,j} | \left(T_{ij}^{\text{iter}} - T_{ij}^{\text{iter}+1} \right) |$$

We observe in Table 4.2 after the second iteration the variations are negligible. We can consider to do only two iteration of the four sweep for the FSM.

4.5 Conclusion

In conclusion, we successfully optimize our method in CPU times using OpenMP multithreading, with FIM and FSM being faster than Euclidean methods in the cases we need for ARMEN. As shown in Section 4.3 FSM has faster CPU times on each cases, but require an implementation of the mesh that allow us to perform our sweep easyly, when FIM is more generalizable. Furthermore in the literature, 3D cases are handle better with FIM than FSM. For those reason, we decide to implement FIM in CEA hydrodynamic code ARMEN. We detail this implementation and the comparison with the Euclidean norm in the next chapter.

Chapter 5

Implementation in ARMEN

5.1 Armen

After studying the *Fast Methods* for some simple cases, my task for the remainder of the internship is to implement the Fast Iterative Method (FIM) in an hydrodynamic code named ARMEN that is developed at the CEA. ARMEN is a C++ code oriented toward research and partnership in numerical analysis and high performance computing. The code runs on the supercomputer INTI that is located at the CEA. The hydrodynamic code ARMEN features several numerical solvers to solve the gas dynamics equations that reads

$$\begin{cases} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \\ \frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u}) + \nabla p = \mathbf{0} \\ \frac{\partial E}{\partial t} + \nabla \cdot (\mathbf{u}(E + p)) = 0 \end{cases}$$

where:

- ρ is the density,
- $\mathbf{u}$ is the velocity vector,
- p is the pressure,
- E is the total energy per unit volume, $E = \rho e + \frac{1}{2}\rho \|\mathbf{u}\|^2$, with e being the internal energy per unit mass.

There is also the possibility to perform simulation for solids to some extent using the Wilkins hypoelastic model (Wilkins, 1963).
The hydrodynamic code ARMEN is an Eulerian code so that the mesh remains fixed over time, providing a robust implementation but at a higher computational and memory cost when compared to Lagrangian code for which the nodes of the mesh move along with the speed of the material. Let us note that in the Eulerian approach, a cell may contain several materials with their respective volume fractions, and a specific treatment is required to handle such cells.
To alleviate the computational and memory cost of the Eulerian approach, an Adaptive Mesh Refinement (AMR) strategy is used. It is a computational technique that dynamically increases mesh resolution in regions requiring higher accuracy to improve solution efficiency and precision. This is illustrated in Figure 5.1 that displays a Richtmyer-Meshkov instabilities, the mesh resolution is much finer at the interface between the blue and green fluids. Several criterion to control the Adaptive Mesh Refinement are implemented in Armen.

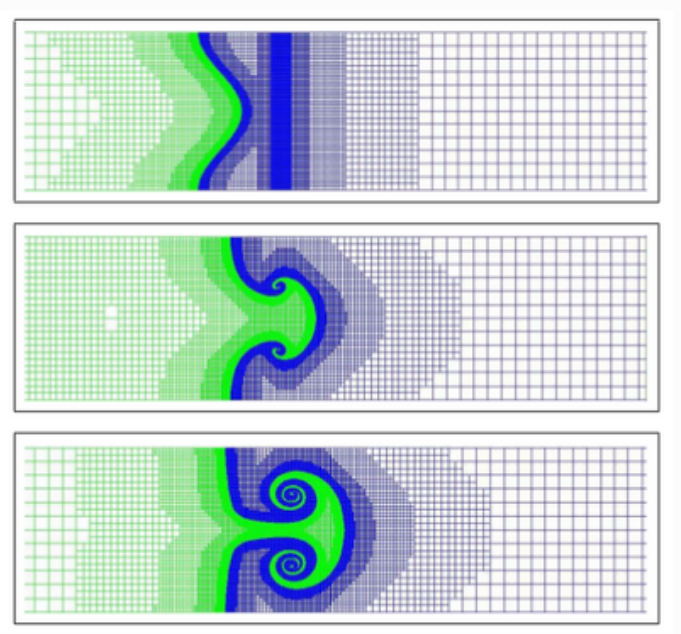


Figure 5.1: Example of Adaptive Mesh Refinement (AMR) over time for a Richtmyer-Meshkov instability.

The hydrodynamic code offers the possibility to use both MPI and OpenMP parallelization. Since we are using Adaptive Mesh Refinement (AMR), the number of cells in a given region of space changes over time. Therefore, specific load-balancing algorithms are required to redistribute the cells among MPI subdomains, in order to avoid large discrepancies in the number of cells between different computational domains.

5.1.1 Numerical slip model

We focus here on the numerical slip model that allows to emulate the slip between two materials. First an interface is reconstructed using their respective volume fractions on the Eulerian mesh. Then, a specific treatment is applied locally near the interface. To do so, the distance to the interface d is computed, then an attenuation function $f(d)$ is used. This function allows a smooth transition between the area where the model is applied near the interface ($f(d) = 0$) and the area where it is no longer used ($f(d) = 1$), namely when d reaches a given threshold. Let us note that as the interface evolves over time, the distance needs to be computed several times during a simulation.

Currently, the Euclidian norm is used to compute the distance to the interface in ARMEN. This method requires to compute the distance from every cell of the mesh to every point of the interface to determine whether the distance is within the threshold or not. This leads to a significant amount of computational time spent on points that are above the threshold.

Those issues are why we aim to incorporate *Fast Methods* into ARMEN.

5.2 Fast Iterative Method in ARMEN

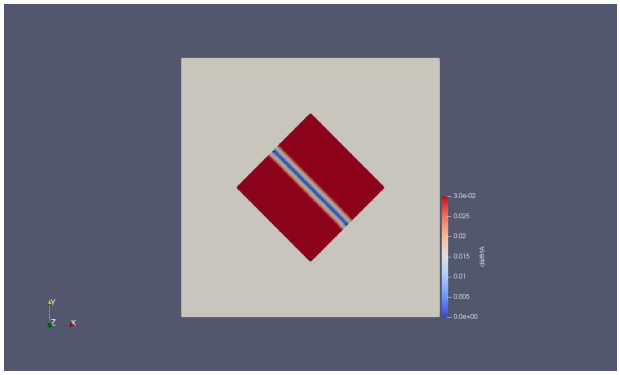
Now we integrate the Fast Iterative Method into ARMEN, a task that presented several challenges. The first difficulty was the massive size of the ARMEN code, which is hundreds of thousands of lines long, making adaptation particularly complex. Next, it was necessary to familiarize myself with the data format used in this code. Unlike my toy model, where I indexed points using matrices and indices (i, j), I had to adopt a new method of data storage. This involved modifying my approach and adding various checks to handle the different cases provided. Additionally, the presence of macros sometimes complicated the understanding of certain environments. Despite these difficulties, I successfully implemented the Fast Iterative Method. In the following section, I will present results

obtained using my implementation of the Fast Iterative Method in Armen and compare it to the Euclidian norm that was already implemented.

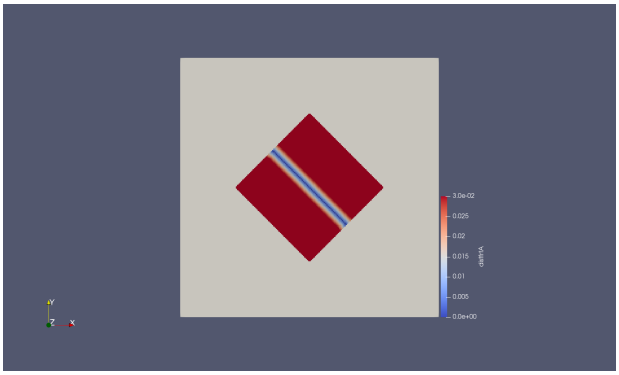
5.3 Numerical results

5.3.1 A slip between two materials

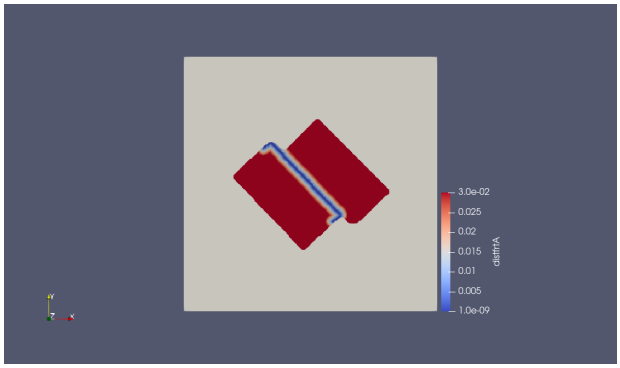
First, We consider a configuration with two materials sliding on each other with a 45-degree inclination with respect to the mesh. Each material is composed of a rectangle of length $l = 0.2$, and width $L = 0.4$. Their speed has a magnitude of 1, is tangential to the interface, and points in opposite directions. The mesh grid is of $h = 0.005$ and no AMR is used. The numerical slip model is used with a threshold at $6 * h = 0.030$ as in Section 3.3.4.
Initially, the interface is the edge of contact of the two rectangles and is composed of 80 cells. However this number will evolve during the simulation as the reconstructed interface between the two materials evolves.



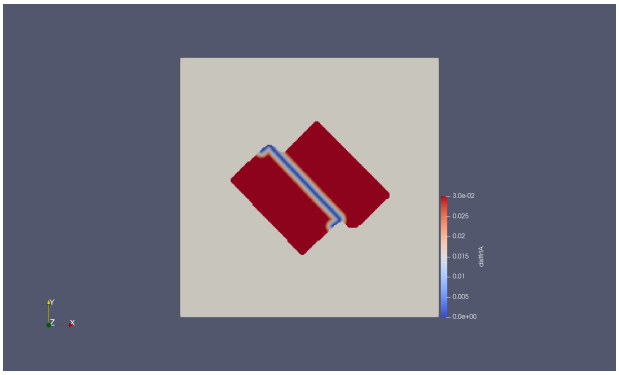
(a) FIM - Initial condition



(b) Euclidian norm - Initial condition



(c) FIM - results after 10 time steps

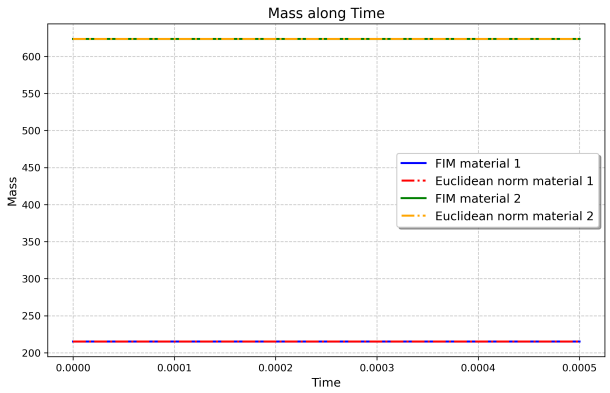


(d) Euclidian norm - results after 10 time steps

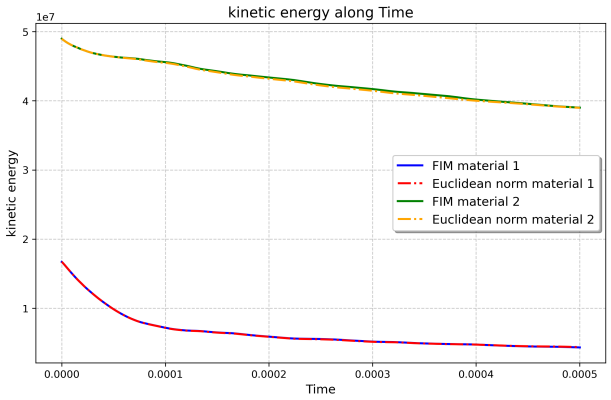
Figure 5.2: Distance map to the reconstructed interface for the Fast Iterative Method and Euclidian norm, at two different time steps 0 and 10, for the slip between two materials test case.

In Figure 5.2, we observe that the Fast Iterative Method and Euclidian norm appear to give similar results after 10 iterations. Let us note that due to numerical diffusion, there is a thin layer of cells containing both materials that form as they slip against each other. Those cells are used to reconstruct an interface between the two materials and the distance to the interface is computed for them as well.

We also compared the evolution over time of the mass and kinetic energy of each material for both methods. In Figure 5.3a and Figure 5.3b, we observe that the mass is conserved as expected for both the Fast Iterative Method and Euclidian norm, while similar results are obtained for the kinetic energy with an error between 0.0003 and 0.0004, which is less than 1%.



(a) Mass along time of each material with distance at interface computed by FIM and euclidean norm



(b) Kinetic energy along time of each material with distance at interface computed by FIM and euclidean norm

5.3.2 A slip between two materials - Performance

We take a look at the performance in term of CPU time for FIM and euclidean norm. We keep the configuration of Section 5.3.1.
We focus here on the CPU time spent in the methods that compute the distance to the interface for a whole simulation. The total CPU time for computing the distance at the interface throughout the entire simulation is determined by summing the CPU times for all iterations.
The relative importance of the CPU time spent computing the distance to the interface, compared to the total simulation time, will also be discussed.

grid size	**0.1**	**0.05**	**0.01**	**0.005**	**0.0025**	**0.00125**
n_s **at the start**	8	15	60	172	253	485
CPU TIME FIM [s]	1.39e-01	4.24e-01	9.8e+00	4.26e+01	1.839e+02	7.32e+02
CPU TIME Euclidean norm [s]	1.36e-01	4.55e-01	9.69e+00	4.70e+01	1.95e+02	9.47e+02

Table 5.1: Table of CPU times spent computing the distance to the interface for FIM and euclidean norm for grid sizes ranging from 0.1 to 0.0025.

We observe similar result as in Section 3.3.2, with Euclidean method being faster or having the same CPU time for the three first grid step, but as we reduce the grid step, the size of the interface increase consequently FIM become faster than euclidean norm in sequential by 23% on the case with a grid step of 0.00125. For this grid size, the execution time of the simulation is about 3 hours and 20 minutes, so that the computation of the distance to the interface represents about 5-6% of it. We also compare the CPU time of both methods in parallel with openmp parallelization with 8 threads in the next table :

grid size	**0.1**	**0.05**	**0.01**	**0.005**	**0.0025**	**0.00125**
n_s **at the start**	8	15	60	172	253	485
CPU time - FIM [s]	1.16e-01	1.99e-01	3.26e+00	1.49e+01	7.55e+01	5.46e+02
CPU time - Euclidean norm [s]	9.25e-02	1.64e-01	2.76e+00	1.28e+01	6.75e+01	5.72e+02

Table 5.2: Table of CPU times spent computing the distance to the interface for FIM and euclidean norm for grid sizes ranging from 0.1 to 0.0025, with 8 threads.

We observe similar result as in subsection 4.4, euclidean norm is faster than FIM on the coarser mesh, but on the finest mesh with a grid step of 0.00125, FIM is faster by 6%.

We expect the difference of CPU Time between the two methods to increase for finer meshes. We didn't push further than a grid size of 0.00125 as this simulation took 3 hours with 640 000 cells.

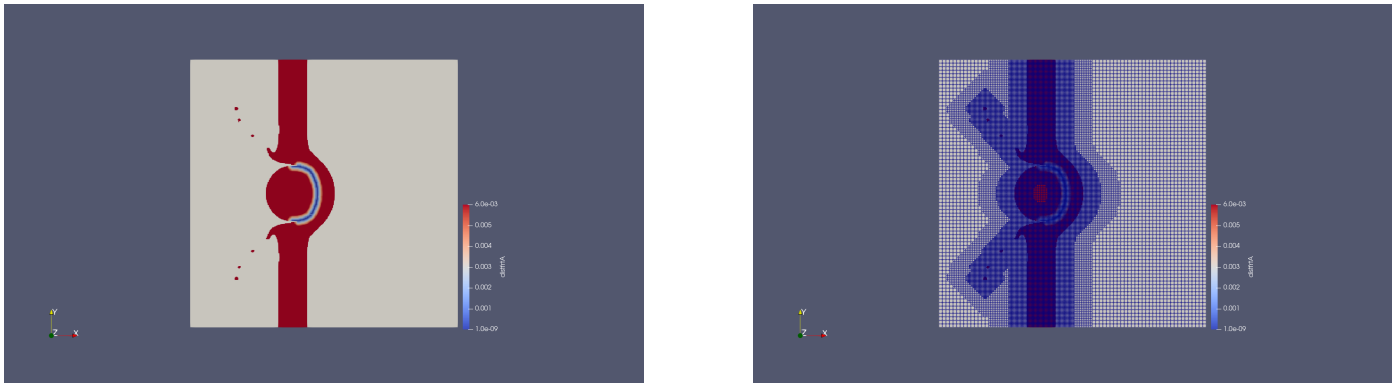
To reduce the total execution time of the simulation and use finer meshes, we could adapt our FIM algorithm to MPI parallelization, this will be discussed in the perspectives. An other way is to use Adaptive Mesh Refinement (AMR) to reduce the number of cells, this is the choice we made for the end of the internship that allowed us to use a more complex configuration showed in the next section.

5.3.3 Sphere impacting a plate with Adaptive Mesh Refinement

We want to ensure that the Fast Iterative Method implementation is compatible with Adaptive Mesh Refinement. Due to the lack of time, we could not adapt the grid exploration algorithm used in the FIM to AMR level jumps. However, its implementation is working if all cells where we compute the distance to the interface are at the same AMR level.

We consider the 2D configuration of a sphere impacting a plate. This test case features a complex interface between the sphere and the plate over time. Using a numerical slip model is crucial to recover the expected sphere velocities especially for low initial speed. An AMR strategy is used, where the finer grid size is applied only at the interface between the sphere and the plate, with a layer of fine cells wide enough to ensure that all cells within a given distance threshold from the interface are at the finest level.

We consider an initial speed of the sphere of 1000 and use a mesh size of 0.001 on the finest AMR level, leading to a number of cells between 43261 and 54853 during the simulation. We use 8 threads OpenMP. This configuration is illustrated on figure 5.4, where we plot the distance to the interface computed by the FIM and the AMR mesh used at a time step when the sphere has impacted the plate.

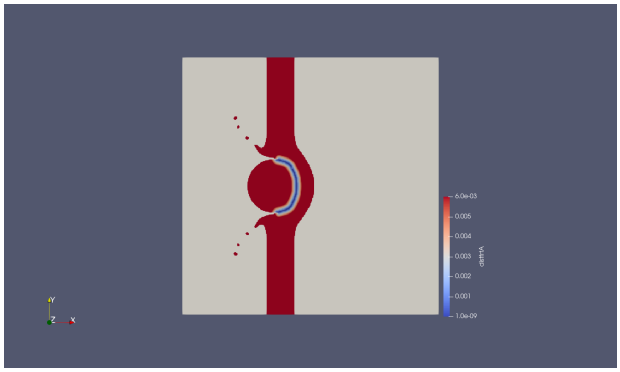


(a) FIM - Distance to the interface

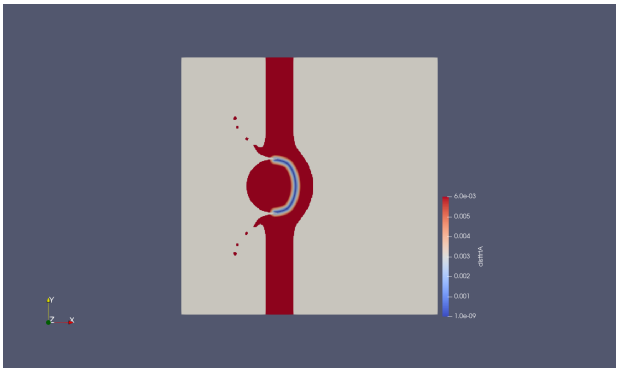
(b) AMR mesh

Figure 5.4: Illustration of the sphere impacting a plate configuration at a time step when the sphere has impacted the plate.

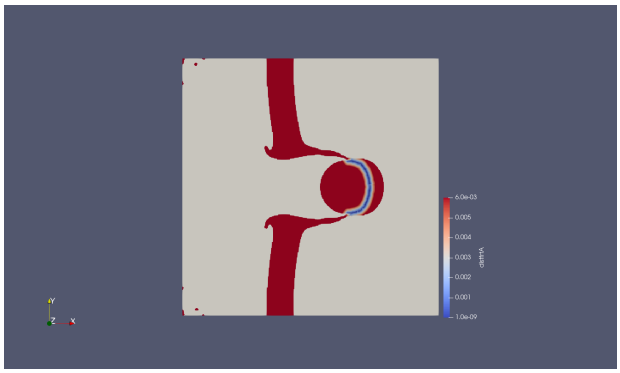
We observe in Figure 5.4a that the distance to the interface is well computed by the FIM on the AMR mesh shown in Figure 5.4b with different AMR levels but with the same level near the reconstructed interface.
We also want to compare the results obtained with FIM and the euclidean norm. We observe in Figure 5.5, that both method appears to give similar results, to ensure that we will compare other metrics such as the evolution over time of the mass of each materials and the speed of the Sphere.



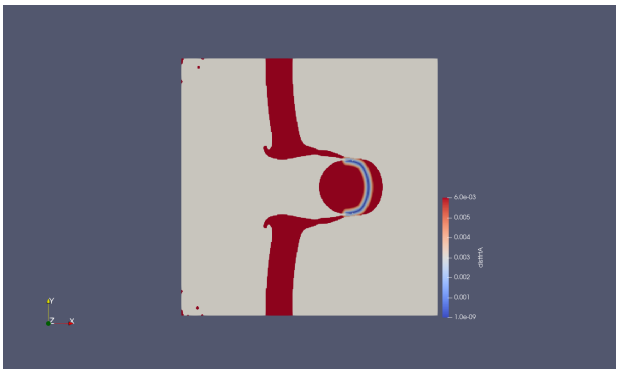
(a) FIM - results after 10 time steps



(b) Euclidian norm - results after 10 time steps

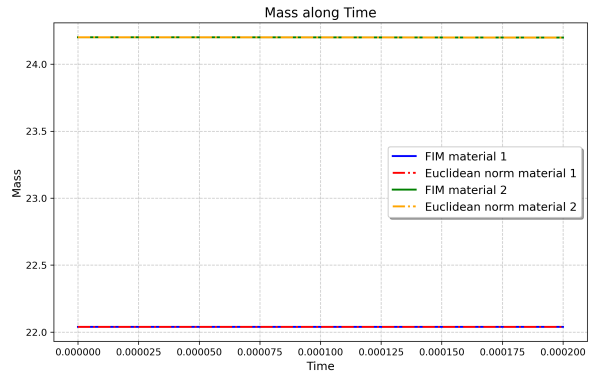


(c) FIM - results after 30 time steps

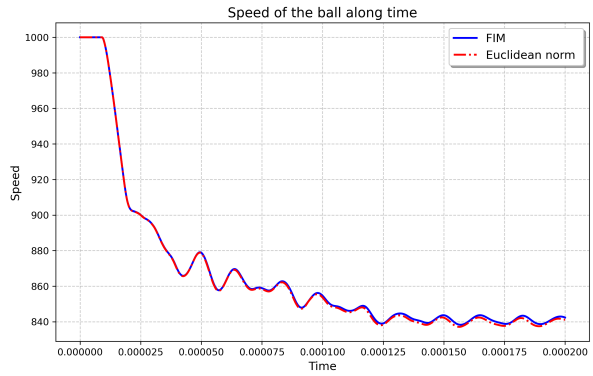


(d) Euclidian norm - results after 30 time steps

Figure 5.5: Distance map to the reconstructed interface for the Fast Iterative Method and Euclidian norm, at two different time steps 10 and 30, for the sphere impacting a plate test case.



(a) Mass along time of each material with distance at interface computed by FIM and euclidean norm



(b) Speed along time of the sphere with distance at interface computed by FIM and euclidean norm

Figure 5.6: Comparaison of metric obtain with FIM and euclidean norm

In Figure 5.6, we observe once again that FIM and the euclidean norm gives similar results with an error inferior to 1%. The CPU Time of those simulations is about 140 s and the CPU time to compute the distance to the interface for FIM and euclidian method is about 16 s, so that it represents $11,4\%$ of the computation time.

We now run simulations with the FIM and euclidian norm for initial speed of the sphere ranging from 200 to 1000. We use 8 threads OpenMP. The final simulation time is set such that, if the plate were not present, the sphere would have completely crossed the region where the plate is located. The final speed of the sphere is shown in Table 5.3. The discrepancy in the final speed of the sphere, when computed using FIM versus the Euclidean norm, is below 1

Initial speed	**1000**	**800**	**600**	**400**	**200**
Final speed - FIM	842.5	655.3	455.1	197.6	94.9
Final speed - euclidean norm	841.1	652.9	452.1	198.5	94.8

Table 5.3: Final speed of the sphere obtained with FIM and euclidean norm.

The numerical results shown above on the slip between two materials and sphere impacting a plate test cases validate the implementation of the Fast Iterative Method in Armen as it gives results close to those obtained using the euclidean norm.

We were not able to do performance comparisons on this configuration due to the CPU Time limitations we already mentioned in section 5.3.1. We didn't had enough time to implement the extension of the FIM to MPI parallelization but we give some ideas on how to perform it in the appendix.

Chapter 6

Conclusion & Perspectives

6.1 Conclusion

To conclude on this internship, I successfully understood the numerical resolution of the eikonal equation and the use of *Fast Methods* for the grid exploration. I implemented the algorithm to solve the eikonal equation along with three *Fast Methods* presented in this report. We focused our study on the FSM and FIM as the FMM was too slow in term of CPU times. The results obtained with the FSM and FIM confirmed the independence of the method from the size of interface, unlike the euclidean norm which became slower as the interface grew in size. Another interesting feature of the FIM and FSM is that we can easily limit the grid exploration to cells for which the distance is under a given threshold.

After that we successfully extend the FSM and FIM implementation to a parallel environment using OpenMP. The results obtained showed the same trend as the sequential results, with *Fast Methods* being more efficient with great mesh and interface, especially with the threshold.

To finish we implemented the openmp version of FIM in the hydrodynamic code ARMEN, and validated it by comparison to the Euclidian norm on two test cases (the slip between two material and the Sphere impacting a plate). A partial performance comparison was performed and confirmed what was observed before between the FIM and euclidian norm.

The perspectives of this work are the extensions of the FIM algorithm to MPI domain decomposition, 3D and AMR level jumps. Some ideas on the first two points are given in appendix.

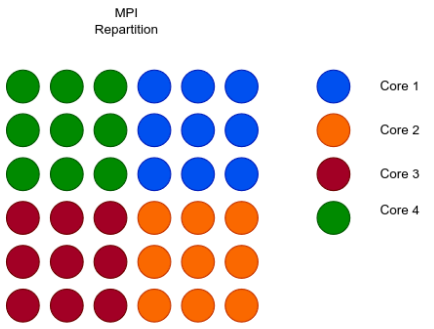
From a personal standpoint, I learned a lot during this internship, particularly about scientific methodology, the writing and reading of scientific reports, as well as the best practices for managing a project over several months. I would like to thank Agnes Grenouille, Alain Pascal for the management of the informatics environments, Guénolé Harel for the configuration of ARMEN on the Inti computer, my supervisor Nicolas Gilet and Mathieu Girardin and the CEA once again for their help and the positive environment they provided during the internship.

6.2 Perspectives

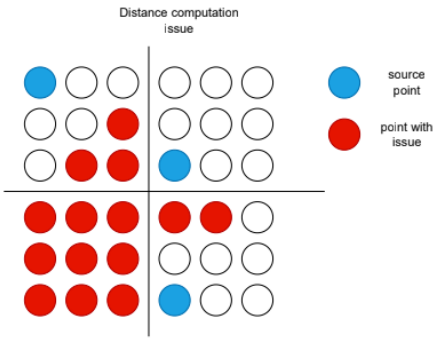
6.2.1 MPI extension of the FIM in ARMEN

MPI for C++ is a library that enables developers to write parallel algorithms by facilitating communication between processes, thereby allowing the workload to be distributed across different cores.Unlike OpenMP, where threads within a single process share memory, MPI requires explicit communication between processes, as each core is unaware of the others' states. This necessitates the implementation of communication protocols to share results and synchronize the different cores.

We can find in some references (e.g. Detrixhe et al. (2013)), the idea of splitting the mesh, with each core working on a part of the mesh, but with this idea come a first difficulties represented in the illustration 6.1:



(a) Repartition of the mesh along the MPI core.



(b) Representation of all point with an issue with the MPI repartition

Figure 6.1: illustration of the MPI repartition and the difficulties

In Figure 6.1a, we see the repartion of the mesh along four cores, each one represented by a color. In Figure 6.1b is the illustration of the difficulties that comes with the repartition, as said before each core has no information about what is computed in the other, so all the area of core 3 cannot computed the distance as there is no source point in it. And the sources in core 1 area create issue on the core 2 and 3 area, has some point in those are are closer to the sources of area 1, than the source within their area. With this description we understand the difficulties coming with the mpi implementation would be how to implement a correction of those false result.

In Detrixhe et al. (2013), we also find the idea of creating "ghost cells" at the limits of each area. Those ghost cells are area computed by each neighboring cores, then we can effectue a reduction operation in those area to get the correct value. Then the question is how to propagate the corect value to the rest of the core area.

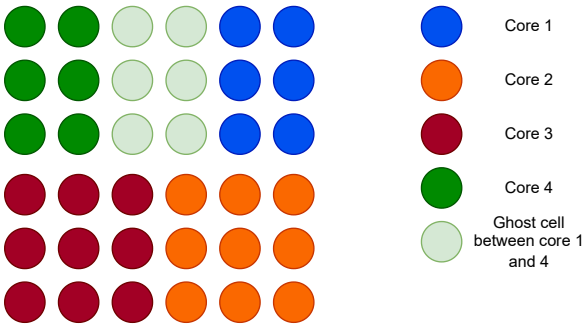


Figure 6.2: Representation of ghost cells between core 1 and core 4

In Figure 6.2, we represent only the ghost cells between core 1 and core 4 for simplification of the visualization. The idea is to compute the distance within core 1 and core 4 on those ghost cells, then apply a reduction operation on those cells to get the minimum distance for each cell. Depending on which cells are modified, we need to apply a correction within each core.Then, we can consider the modified points as sources and redo a FIM iteration within the core, but without initializing the source point at a distance value of 0. The correction applied with the ghost cells of core 1 and core 4 is shown in Figure 6.3.

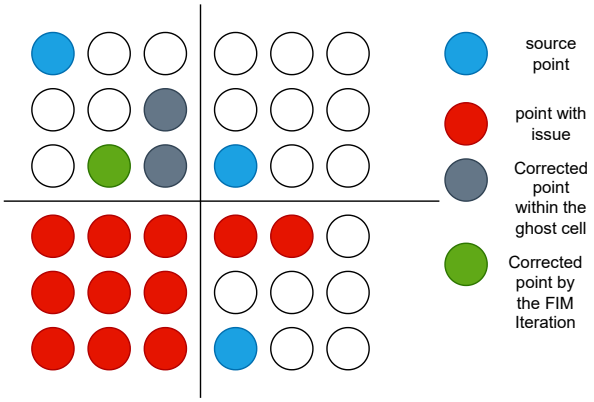


Figure 6.3: Representation of the correction from the ghost cells between core 1 and core 4.

In Figure 6.3, all red cells shown in Figure 6.1b in core 1 are corrected by either the reduction operation within the ghost cells or the FIM iteration starting from the modified point.

6.2.2 3D extension of the FIM in ARMEN

Another mission would be to implement the 3D version of the FIM in ARMEN, as we saw in Section 5.3.1, in 2D the computation of the distance is only about 5-6% of the simulation time. But in 3D, this percentage expected to be greater. To gain a better understanding of the difference in computation time between 2D and 3D, we consider the configuration defined in Section 3.3.4, let us not n_{seuil} the number of cells within the threshold.

In this configuration, we have $n_s = n_y$. If we extend the interface to three dimensions while keeping it uniform along the z-axis, the interface size becomes $n_y \times n_z$.

For the two-dimensional case:

- The number of calculations for the Fast Iterative Method (FIM) is n_{seuil}.
- For the Euclidean norm, it is $n_x \times n_y \times n_s$.

For the three-dimensional case:

- The number of calculations for FIM becomes $n_{\text{seuil}} \times n_z$.
- For the Euclidean norm, it becomes $n_x \times n_y \times n_z \times n_y \times n_z$, since the interface is $n_y \times n_z$.

Assuming we simplify the calculations by setting $n_x = n_y = n_z$:

- The complexity for FIM in 3D is $n_{\text{seuil}} \times n_z$, where n_{seuil} in the worst case is $n_x \times n_y$. Thus, the worst-case complexity is approximately n_x^3 operations.
- For the Euclidean norm, the complexity becomes n_x^5.

In this configuration for three dimensions, the euclidean norm becomes computationally infeasible with the complexity reaching $\mathcal{O}(n_x^5)$, making the Fast Iterative Method (FIM) a far superior choice with the complexity degree being $\mathcal{O}(n_x^3)$. For that reason I have already implemented the FIM in 3D in my toy model, and gave the details of the numerical scheme in 3D in the appendix.

Chapter 7

Appendix

7.1 Eikonal equation 3D numerical scheme

The Numerical scheme presented in chapter 2 can be written in 3D as :

$$|\nabla T(x)| = \sqrt{\left(\frac{\partial T}{\partial x}\right)^2 + \left(\frac{\partial T}{\partial y}\right)^2 + \left(\frac{\partial T}{\partial z}\right)^2} = 1$$
$$\approx \sqrt{\max\left(D_{ijk}^{-x}T, -D_{ijk}^{+x}T, 0\right)^2 + \max\left(D_{ijk}^{-y}T, -D_{ijk}^{+y}T, 0\right)^2 + \max\left(D_{ijk}^{-z}T, -D_{ijk}^{+z}T, 0\right)^2} = 1$$

With the one-sided partial difference operator in direction $\pm z$ define as:

$$T_z(x_{ijk}) \approx D_{ijk}^{\pm z} = \frac{T_{i,j,k\pm1} - T_{i,j,k}}{\pm\Delta z} \tag{7.1}$$

As in 2D, by elevating to the square we obtain :

$$\max(-D_{ijk}^{+x}, D_{ijk}^{-x}, 0)^2 + \max(-D_{ijk}^{+y}, D_{ijk}^{-y}, 0)^2 + \max(-D_{ijk}^{+z}, D_{ijk}^{-z}, 0)^2 + = 1 \tag{7.2}$$

Eq. (7.2) can be rewritten as:

$$\max\left(-\frac{T_{i+1,j,k} - T_{i,j,k}}{\Delta x}, \frac{T_{i-1,j,k} - T_{i,j,k}}{-\Delta x}, 0\right)^2 + \max\left(-\frac{T_{i,j+1,k} - T_{i,j,k}}{\Delta y}, \frac{T_{i,j-1,k} - T_{i,j,k}}{-\Delta y}, 0\right)^2$$
$$+ \max\left(-\frac{T_{i,j,k+1} - T_{i,j,k}}{\Delta z}, \frac{T_{i,j,k-1} - T_{i,j,k}}{-\Delta z}, 0\right)^2 = 1$$

Which is equals to :

$$\max\left(\frac{T_{i,j,k} - \min(T_{i+1,j,k}, T_{i-1,j,k})}{\Delta x}, 0\right)^2 + \max\left(\frac{T_{i,j,k} - \min(T_{i,j+1,k}, T_{i,j-1,k})}{\Delta y}, 0\right)^2$$
$$+ \max\left(\frac{T_{i,j,k} - \min(T_{i,j,k+1}, T_{i,j,k-1})}{\Delta z}, 0\right)^2 = 1$$

By keeping the same notation as in 2.7, with $T_z = \min(T_{i,j,k+1}, T_{i,j,k-1})$ and the same supposition about T being greater than T_x, T_y, T_z, we obtain :

$$\left(\frac{T - T_x}{\Delta x}\right)^2 + \left(\frac{T - T_y}{\Delta y}\right)^2 + \left(\frac{T - T_z}{\Delta z}\right)^2 = 1 \tag{7.3}$$

Then we got the same type of quadratic equation as in 2D with a, b, c being :

- $a = \Delta x^2 + \Delta y^2 + \Delta z^2$
- $b = -2(\Delta y^2 \Delta z^2 T_x + \Delta x^2 \Delta z^2 T_y + \Delta x^2 \Delta y^2 T_z)$
- $c = \Delta y^2 \Delta z^2 T_x^2 + \Delta x^2 \Delta z^2 T_y^2 + \Delta x^2 \Delta y^2 T_z^2 - \Delta x^2 \Delta y^2 \Delta z^2 * 1$

with $\Delta y = \Delta z = \Delta x = h$ and dividing by h^4
we obtain the same generalization as in 2D

- $a = N$
- $b = -2 \sum_{d=1}^{3} T_d$
- $c = \sum_{d=1}^{3} T_d^2 - h^2$

7.2 Documentation of the Toy model

Here I describe the main function of my implementation and how to execute the code. To execute C++ codes, the following libraries are necessary:

- Standard Template Library for vectors and tuples.
- Classic C++ file writing libraries: os, fstream.
- Numpy and matplotlib.pyplot for visualization.
- OpenMP for parallel computing in C++.

```
1  double FMM(std::vector<std::vector<double>> &T,int n,double lenght,const
       std::vector<std::pair<int,int>> &Xs, std::vector<std::vector<double>> &
       Closest)
2  /*
3  * @Brief Execute the Fast Marching Methods on a mesh of size n*n, with Xs
       as source, all distance are stored in T. Return the execution time.
4  * -T: Matrix that stores the distances
5  * -n: number of points along x and y
6  * -length: length of the x and y axes
7  * -Xs: vector containing the index of the sources
8  * -Closest: Matrix containing the tag of the nearest source
9  */
10
11 double FIM2D(std::vector<std::vector<double>> &T,int n,double lenght,const
       std::vector<std::pair<int,int>> &Xs,std::vector<std::vector<double>> &
       Closest )
12 /*
13 * @Brief Execute the Fast Iterative Methods on a mesh of size n*n, with Xs
       as source, all distance are stored in T. Return the execution time.
14 * -T: Matrix that stores the distances
15 * -n: number of points along x and y
16 * -length: length of the x and y axes
17 * -Xs: vector containing the index of the sources
18 * -Closest: Matrix containing the tag of the nearest source
19 */
20
21 double FSM2D(std::vector<std::vector<double>> &T,int n,double lenght,const
       std::vector<std::pair<int,int>> &Xs,std::vector<std::vector<double>> &
       Closest )
```

```
/*
* @Brief Execute the Fast Sweeping Methods on a mesh of size n*n, with Xs as source, all distance are stored in T. Return the execution time.
* -T: Matrix that stores the distances
* -n: number of points along x and y
* -length: length of the x and y axes
* -Xs: vector containing the index of the sources
* -Closest: Matrix containing the tag of the nearest source
*/

void generate_source2d(std::vector<std::pair<int,int>> &Xs,int n,int n_source, unsigned int seed = 0 , bool give_seed = false)
/*
* @Brief generate randomly n_source point, to compose Xs, all point are chosen between 0 and n for x and 0 and n for y as the index of the source point.
* - Xs: Vector that will store the index of the interfaces
* - n: Number of points along the x and y axes
* - n_source: Number of interfaces to generate
* - seed: Generation seed
* - give_seed: Specifies whether a seed is provided or not
*/

void generate_source_sinus2d(std::vector<std::pair<int,int>> &Xs,int n, unsigned int seed = 0 , bool give_seed = false )
/* @Brief generate an interface along the y axis in Xs
* - Xs: Vector that will store the index of the interfaces
* - n: Number of points along the x and y axes
* - seed: Generation seed
* - give_seed: Specifies whether a seed is provided or not
*/

double euclidien2d( std::vector<std::vector<double>> &T_exact,std::vector <std::pair<int,int>> Xs ,int n,double lenght,std::vector<std::vector<double>> &Closest_exact  )
/*
* @Brief Execute the Euclidean norm from all point in Xs on a mesh of size n*n, all distance are stored in T. Return the execution time.
* -T: Matrix that stores the distances
* -n: number of points along x and y
* -length: length of the x and y axes
* -Xs: vector containing the index of the sources
* -Closest: Matrix containing the tag of the nearest source
*/

int write_files(const  std::vector<std::vector<double>> &T ,std::string file)
/*
* @Brief write the distance matrix T in the file.txt
* -T: Matrix that stores the distances
* -file : Name of the file without .txt
*/
```

There are other versions of FSM and FIM functions, as FIM2D_seuil, and FSM2D_seuil with the need to add the threshold value at the end of the arguments list, and their parallel version , with the number of thread placed at the end of the arguments list.

How to Execute the Code from the GitHub repository

The repository contains three directories:

- `eikonal_d`,
- `Eikonal_nouveau_maillage` (same functionality as `eikonal_d`, but source points are not centered in cells to avoid equidistance issues),
- `eikonal_parallel`.

To run the code, navigate to one of these directories.

1. `eikonal_d` **Directory**
This directory provides three main executables:

Executable	Command-Line Syntax
`main2d.cxx`	`./<executable> <number of source points> <number of points on x,y axis>`
`main3d.cxx`	`./<executable> <number of source points> <number of points on x,y,z axis>`
`main2d_compare.cxx`	`./<executable> <number of source points> <number of points on x,y axis> <n_point1> [<n_point2> ... <n_pointn>]`

2. `Eikonal_nouveau_maillage` **Directory**
This directory provides two main executables:

Executable	Command-Line Syntax
`main2d.cxx`	`./<executable> <number of source points> <number of points on x,y axis>`
`main2d_compare.cxx`	`./<executable> <number of source points> <number of points on x,y axis> <n_point1> [<n_point2> ... <n_pointn>]`

3. `eikonal_parallel` **Directory**
This directory provides four main executables:

Executable	Command-Line Syntax
`main2d.cxx`	`./<executable> <number of source points> <number of points on x,y axis> <number of threads>`
`main3d.cxx`	`./<executable> <number of source points> <number of points on x,y axis> <number of threads>`
`main2d_compare.cxx`	`./<executable> <number of source points> <number of points on x,y axis> <n_point1> [<n_point2> ... <n_pointn>]`
`main_comparaison _thread.cxx`	`./<executable> <number of sources> <n> <n_thread1> [<n_thread2> ... <n_threadk>]`

Note: Each executable writes the results to a file. You must manually change the output filename in the main file before execution.

Visualizing Results

Each directory contains a `visu` subdirectory with Python scripts for visualizing the results.

Script	Description
`visu_T.py`	Visualizes the results from a provided file.
`visu_result _erreur.py`	Visualizes results, computes the error, and plots it if a file with the Euclidean norm is available.
`comparaison.py`	Visualizes execution times for different grid sizes.
`visu_thread.py`	Visualizes execution times for different numbers of threads.

7.2.1 Example of Code Execution

Below is an example of executing the code for a sequential 2D run of FIM, FSM, and the Euclidean norm:

1. Navigate to the `eikonal_d` directory.
2. Compile the code using:

```
g++ -o main2d main2d.cxx -O2
```

3. Execute the program with:

```
./main2d 10 400
```

Output:

```
Valeur de n_source et n  : 10 400  # n_source and n_x given
nombre de sweep : 10        # number of sweep during FSM
Temps d'exécution FSM : 0.145772 secondes.   #CPU time FSM
Temps d'exécution FIM : 0.212645 secondes.  #CPU time FIM
Temps d'exécution euclidien : 0.00411472 secondes. #CPU time euclidean norm
La matrice a été écrite dans : results/length_10_n_400_n_source_10_FIM.txt
#result files of FIM, names composed of input parameter
La matrice a été écrite dans : results/length_10_n_400_n_source_10_FSM.txt
#result files of FSM, names composed of input parameter
La matrice a été écrite dans : results/length_10_n_400_n_source_10_exact.txt
#result files of euclidean norm, names composed of input parameter
Coordonnées sauvegardées dans coords_mesh #files storing source points
```

Visualization:

1. Place yourself at the root of the directory.
2. In the visualization script you want to use, change the lines:
   - `tags = []` to `tags = ["FIM","FSM"]`
   - `filename = old_filename` to `filename = f"results/length_10_n_400_n_source_10_{tag}.t`
   - `plot_filename = "old_image_name.png"` to `plot_filename = "images/new_image_name.png"`

- `points_files = "old_source_files.txt"` to `points_files = "new_source_files.txt"`

The exact result files will be used to compute the error in the Python files.

3. Run the visualization script:

```
python visu/visu_result_erreur.py
```

The image and output below are generated by the `visu_result_erreur.py` script:

Output:

```
FIM : length = 10, n = 400      #tag and  input parameter
Fichier des sources trouvé      #coords_mesh.txt find
norme euclidienne : Fichier trouvé   #exact solution find
erreur =  0.018374373186250004          #mean error
FSM : length = 10, n = 400           # tag and input parameter
Fichier des sources trouvé           #coords_mesh.txt find
norme euclidienne : Fichier trouvé   #exact solution find
erreur =  0.018374206936249998          #mean error
Graphique sauvegardé sous plot_fsm_fim_interface.png    #image file
```

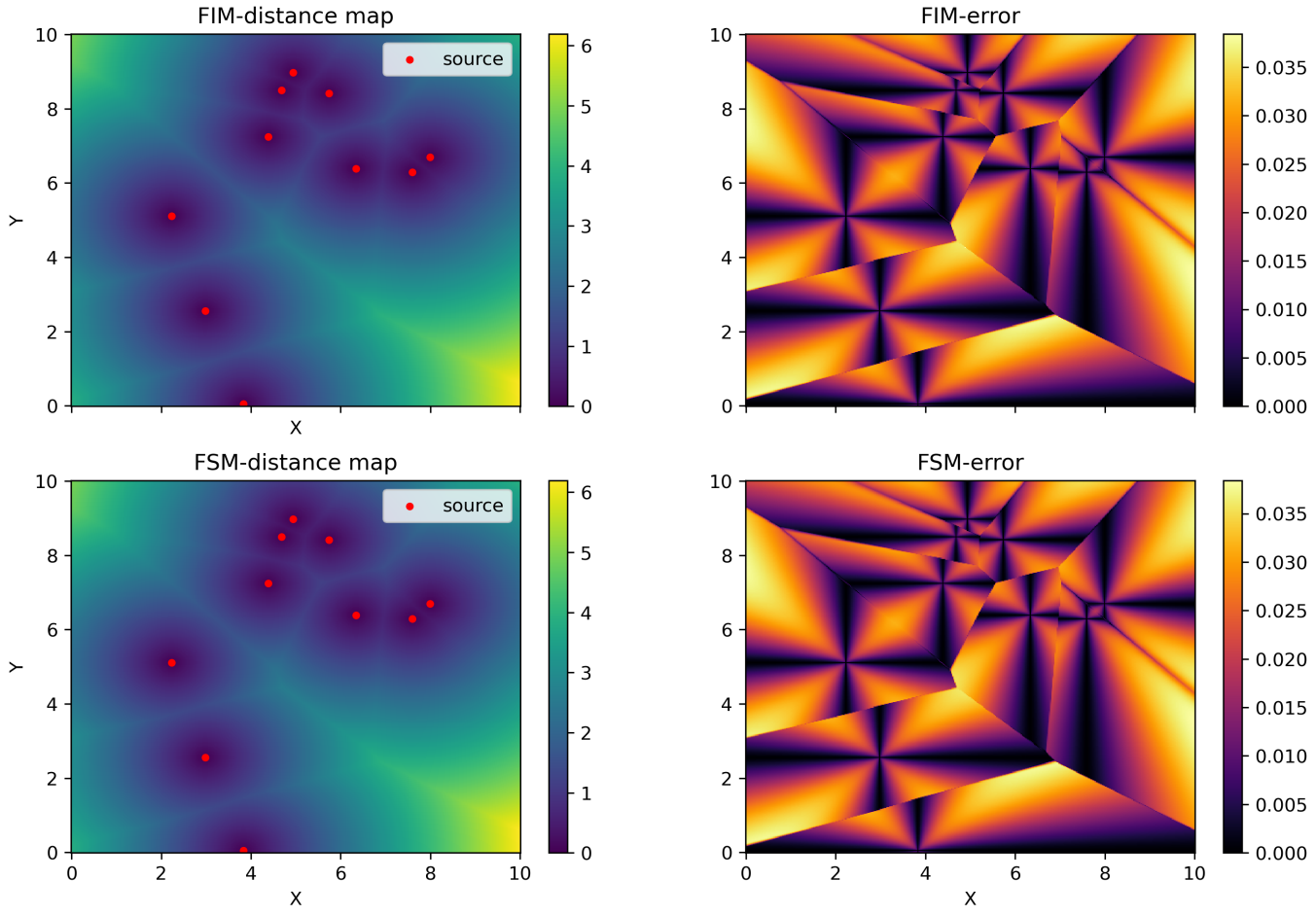


Figure 7.1: Distance map (left column) and error map (right column) for FIM/FSM with source points (red point) for $N = 400 \times 400$ and $n_s = 10$.

Bibliography

Bak, S., McLaughlin, J. R., and Renzi, D. (2010). Some improvements for the fast sweeping method. *SIAM J. Sci. Comput.*, 32:2853–2874.

Cai, W., Zhu, P., and Li, G. (2023). Improved fast iterative method for higher calculation accuracy of traveltimes. *Computers and Geosciences*, 174:105331.

Dang, F., Emad, N., and Fender, A. (2013). A fine-grained parallel model for the fast iterative method in solving eikonal equations. In *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 152–157.

Detrixhe, M., Gibou, F., and Min, C. (2013). A parallel fast sweeping method for the eikonal equation. *Journal of Computational Physics*, 237:46–55.

Gómez, J. V., Álvarez, D., Garrido, S., and Moreno, L. E. (2015). Fast methods for eikonal equations: An experimental survey. *IEEE Access*, 7:39005–39029.

Gropp, W., Lusk, E., and Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.

Jeong, W.-K. and Whitaker, R. (2008). A fast iterative method for eikonal equations. *SIAM J. Scientific Computing*, 30:2512–2534.

Jones, M., Bærentzen, J., and Sramek, M. (2006). 3d distance fields: A survey of techniques and applications. *Transactions on Visualization and Computer Graphics*, 12(4).

Kim, S. (2001). An ϰ(\) level set method for eikonal equations. *Siam Journal on Scientific Computing*, 22.

Rouy, E. and Tourin, A. (1992). A viscosity solutions approach to shape-from-shading. *SIAM J. Numer. Anal.*, 29(3):867–884.

Sethian, J. A. (1996). A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595.

Sethian, J. A. (1999). *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge Monograph on Applied and Computational Mathematics. Cambridge Univeristy Press, second edition.

van der Pas, R., Stotzer, E., and Terboven, C. (2017). *Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD*. The MIT Press.

Wilkins, M. L. (1963). Calculation of elastic-plastic flow. Technical report, California. Univ., Livermore. Lawrence Radiation Lab.

Yatziv, L., Bartesaghi, A., and Sapiro, G. (2006). O(n) implementation of the fast marching algorithm. *Journal of Computational Physics*, 212(2):393–399.

Zhao, H. (2004). A fast sweeping method for eikonal equations. *Math. Comput.*, 74:603–627.

Zhao, H. (2007). Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429.