Adjoint-based PDE-constrained optimization using HPC techniques

Oscar Francisco Peredo Andrade

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

A thesis submitted for the degree of
Master in Information Technology
with specialization in the professional area of
Supercomputing

September, 2013

Director:	José María Cela	Director, CASE department, Barcelona Supercomputing Center
Tutor:	Mariano Vázquez	Team leader, High Performance Computational Mechanics group, CASE department, Barcelona Supercomputing Center

Contents

Agradecimientos

En primer lugar, agradezco a las personas que confiaron en mí para realizar este proyecto. José María, Mariano y Guillaume, pues sin el financiamiento y las herramientas que me entregaron, no hubiera podido conocer esta hermosa ciudad, este excelente centro de supercomputación y este apasionante tema de investigación. Espero que sigamos conectados en el futuro y que el nexo creado no se pierda.

También quiero agradecer a las personas que nos recibieron en su hogar sin condiciones durante nuestro primer año en Barcelona, Alberto y Olga. La generosidad y simpatía con la que nos acogieron y nos integraron en su grupo de amistades sin duda siempre será recordada. Agradezco la buena onda y cariño de todas las nuevas y viejas amistades que disfrutaron con nosotros en esta ciudad, siempre recordaremos los buenos momentos que pasamos, y esperamos con ansias volver a encontrarnos con todos ustedes.

Agradezco el apoyo que nos dió mi suegro Ángel, quién a su manera nos demostraba su cariño, y me entregaba su confianza en esta aventura que iniciamos junto a mi esposa Natalia. Agradezco también a mi suegra Sara quién nos acompañó diariamente a través del teléfono o del computador, entregandonos su apoyo y cariños. Agradezco a mi madre Flora por su apoyo, que a pesar de las dificultades que le tocó enfrentar estos años, siempre tenía tiempo para conversar y entregar cariño. También agradezco a mi hermana Paulina, quien nos acompaño durante el segundo año, con quien compartimos y profundizamos nuestra amistad como hermanos. Y agradezco a mi hermana chica, Valentina, por su cariño a la distancia y el apoyo que prestó a mi madre y abuelos en el año en que sólo estuvo ella en Santiago. Sin ese apoyo, nuestra familia hubiera sufrido aún más por nuestra ausencia.

Finalmente quiero agradecer a mi esposa Natalia, por su apoyo incondicional en todo este proceso, el cual ha tenido momentos difíciles en los que siempre ella me ha acompañado. Estos años han sido de mucho esfuerzo, en términos académicos y emocionales, y sin ese apoyo todo sería más difícil. En el futuro recordaremos con nostalgia estos años en Barcelona, pues fueron nuestros primeros años de matrimonio y donde formamos nuestro primer hogar. Sin embargo, cuando nos invada la pena debido a la añoranza de tiempos pasados, siempre podemos recurrir a los incontables hermosos momentos y divertidas anécdotas que han ocurrido en éstas tierras. Nuevas aventuras están por venir, nuevos lugares y nuevas personas. Todos los cambios son difíciles, pero si estamos juntos, es seguro que avanzaremos y seremos felices donde sea. Te amo.

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Technological advances in High Performance Computing (HPC) has forced the application experts to adapt themselves to the new way of design computers and to develop parallel versions of their algorithms using specific well-proven programming models.

Focusing our work in the utilization of current HPC technologies, our aim is to explore mathematical optimization algorithms in which a partial differential equation (PDE) acts as main constraint. The main motivation to explore this kind of algorithms is the current need in the CASE department of Barcelona Supercomputing Center (BSC) to have available an optimization solver on top of an in-house developed parallel PDE simulator. This need comes from the fact that resolution of PDE-constrained optimization (PDECO) problems have become central in many fields such as shape optimization in computational fluid dynamics (CFD), material inversion in geophysics, data assimilation in weather prediction modeling, structural optimization of stressed systems and control of chemical processes.

New scientific fields and business ventures can be explored using this kind of tool, and our aim is to make a breakthrough in this area providing algorithms and services to BSC's researchers for ongoing and future projects.

1.2 State of the field

Formally, a continuous PDECO problem can be formulated as:

$$\begin{array}{lll} \underset{(u,d)\in\mathcal{U}\times\mathcal{D}}{\text{minimize}} & \mathcal{J}(u,d) & \\ \text{subject to} & \mathcal{R}(u,d) & = \quad 0 \end{array} \tag{1.2.1}$$

with $\mathcal{U}$ and $\mathcal{D}$ assumed to be appropriate functional spaces, $\mathcal{J} : \mathcal{U} \times \mathcal{D} \to \mathbb{R}$ the cost functional and $\mathcal{R} : \mathcal{U} \times \mathcal{D} \to \mathcal{U}$ the constraint operator, which is a PDE with boundary conditions defined in a domain $\Omega \subset \mathbb{R}^n$ with $n \in \{1, 2, 3\}$. The function $u \in \mathcal{U}$ can be viewed as the solution of the PDE and $d \in \mathcal{D}$ as the design parameter function which acts as an input on the differential equation $\mathcal{R}(u, d) = 0$.

Historically, shape optimization in CFD has arguably made the largest contributions in the field. In this kind of problems, the cost function measures the simulated pressure, velocity or other physical magnitude, inside a sub-domain (for example a box below a wing or a sphere in the back of an aircraft) and the design variables are related to a shape embedded in a flow (for example a wing embedded in air at high velocity). First attempts were credited to Pironneau [42] and Jameson [33] applying adjoint formulations for minimum drag and shape optimization to calculate sensitivities in Stokes, incompressible Navier-Stokes and Euler flow equations. Since those earlier works, numerous results have been published on shape optimization, including optimization of three dimensional wings and large-scale aerodynamic models (planes, helicopters, cars or space shuttles, among others).

PDECO:pde-constrained problem

In the geophysical community, inverse problems have been studied for many years as well. The inverse wave propagation [50] and magnetotelluric/electromagnetic inversion [9, 15] are amongst the most important problems in this field, and both can be formulated as a PDECO problem. In this kind of problems, typically the cost functional is an L^2-norm between simulated and observed data (previously collected through expensive offshore/onshore data acquisition surveys) and the design variables are related to material properties in the domain (acoustic velocity distribution or electric conductivity). Data-assimilation problems [20] are very related to the previous problems, the main difference is in the design variable choice, which can be an initial boundary condition (in time or space domain) or source parameters included in the right-hand side of the PDE.

Concerning parallel implementations, in these two communities there are few successful documented experiences in the usage of parallel PDE simulators in the context of PDECO [11, 12, 13, 40]. The primary reason for this slow progress is that parallel PDE simulation code development requires considerable more effort than the sequential counterpart. A significant understanding of computer architecture and good programming skills using the standard libraries MPI [48] and OpenMP [17] (and more recently CUDA [35]) are necessary to develop a useful and scalable code. A secondary reason is related with the tight coupling between the PDE simulator and the optimization routines used on top of it. Typically, PDE simulation codes involve a large amount of *know-how* specific to the application area of the research group who has developed it, with its own software frameworks and programming language choice and style. This situation makes even harder to build a generic PDECO solver capable of use different PDE simulators for different application areas.

Is in this aspect in which BSC has an opportunity, because their in-house parallel PDE simulator was designed from scratch to be able to solve different physical equations, even multi-physics coupled systems, using an integrated framework based in mesh partitioning

techniques and parallel matrix-vector operations. If PDECO methods are implemented on top of this framework, several physics can be used combined with different cost functionals, each one defined by the final user of the optimization solver. Similar experiences have been documented concerning the integration of PDECO algorithms and parallel PDE multi-physics simulators [51, 49]. In these experiences, a considerable effort in code development and software engineering has been made, in order to keep the PDE simulation and optimization codes as decoupled as possible, using object-oriented frameworks and libraries for both codes. In our case, we must adapt our development to our PDE simulator, with its strengths and weaknesses, which will be described in further chapters.

Concerning general PDECO algorithms, a detailed description can be reviewed in [7] and [51]. In this algorithms, the key step is related with the first-order necessary conditions, or Karush-Kuhn-Tucker (KKT) conditions over problem (1.2.1). Details about the mathematical formulation of the conditions can be reviewed in the next chapter. Many of those methods are based in the calculation of a *reduced gradient* [39]. For this reason, the reduced gradient can be considered as a kernel calculation in PDECO algorithms, so a well understanding of its implementation and performance into the in-house parallel PDE simulator is of central importance for future developments. In the next chapter we will include the theoretical description of the reduced gradient, and its utilization in different PDECO algorithms.

1.3 Objectives

Our main objective is to implement a framework to solve the discrete version of problem (1.2.1) in which the final user will be able to set the cost functional $\mathcal{J}(u, d)$ and the governing PDE constraint $\mathcal{R}(u, d) = 0$. The proposed framework must be based in the calculation of reduced gradients on top of a legacy highly-parallel PDE simulator called Alya [30, 29], developed in the CASE department of BSC, which solves incompressible and compressible fluid dynamics, solid mechanics, thermal and electromagnetic flows and a large variety of problems. The framework has to work in concordance with the parallelization strategy used in Alya, which is a distributed-memory domain decomposition, keeping the scalability achieved by the linear system solvers and also maintaining the software design already adopted in the legacy code.

In order to fulfill our main objective, three tasks are proposed:

1. Resolution of a test problem:

   The first task is related to the implementation of all the steps involved in the calculation of the reduced gradient described in further chapters using a basic stationary PDE model. To test our implementation we will design a 2D synthetic optimization example with unique solution based in a convection-diffusion-reaction linear operator, commonly known as *transport equation*.

2. Resolution of an applied problem:

   The second task is based in the application of the previously developed routines into a larger example. It is based in a 3D controlled source electromagnetic inversion in which the underlying PDE model is a complex-valued stationary *Helmholtz equation* and has useful applications in hydrocarbon exploration.

3. Integration of developed routines:

   With all the knowledge generated in the previous tasks, our final task is to integrate all the routines and programs developed before into the legacy PDE simulator Alya. The proposed integration must be delivered as a *service* that can be used by other researchers. It has to be well programmed following the guidelines of the main software architects of Alya, adding minimal modifications to the existing design of the main application.

1.4 About Barcelona Supercomputing Center

Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC) hosts the supercomputer MareNostrum. It also has well-known supercomputing research groups that develop tools for academia and industry. BSC focuses its research areas in Computer Sciences, Life and Earth Sciences and Computer Applications in Science and Engineering. In the context of this multi-disciplinary approach, BSC has more than 350 researchers and experts in HPC and 100 of those are from outside Spain. BSC was constituted as a public consortium formed by the current Spanish Ministry of Economy and Competitivity (Ministerio de Economía y Competitividad), the Department of Economy and Knowledge of the Catalan Government and the Technical University of Catalonia - Barcelona Tech (UPC), and is headed by Professor Mateo Valero.

In 2011, the BSC-CNS was recognized as a *Severo Ochoa Centre of Excellence* for its contributions and research agenda in the area of computing and applications. In the first edition of the Severo Ochoa programme, the Ministry of Science and Innovation selected 8 research centres and units in Spain to be among the best in the world in their respective fields. More information: `htpp://www.bsc.es`.

Chapter 2

Theoretical background

In this chapter we will describe the theoretical background relative to PDECO algorithms, the discretization scheme used in this work, called finite element method, and the basic algorithm to calculate the reduced gradient of a cost function in a PDECO problem. In the last section we include a brief summary of the proposed algorithm. It will be the basis of our proposed implementation described in the next chapter.

2.1 PDE-constrained optimization

The discrete version of problem (1.2.1) can be formulated as:

$$\begin{aligned} \underset{(\mathbf{u},\mathbf{d})\in\mathbb{R}^{n_u}\times\mathbb{R}^{n_d}}{\text{minimize}} \quad & J(\mathbf{u},\mathbf{d}) \\ \text{subject to} \quad & \mathbf{R}(\mathbf{u},\mathbf{d}) = \mathbf{0} \end{aligned} \tag{2.1.1}$$

with $J : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \to \mathbb{R}$ a discrete version of the cost functional and $\mathbf{R} : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \to \mathbb{R}^{n_u}$ defined as the discretized PDE with its boundary conditions. Considering that $\mathbf{u}$ depends implicitly on $\mathbf{d}$ as $\mathbf{u} := \mathbf{u}(\mathbf{d})$, we can define a new cost function $j : \mathbb{R}^{n_d} \to \mathbb{R}$, called *reduced cost function*, that satisfies $j(\mathbf{d}) = J(\mathbf{u}(\mathbf{d}), \mathbf{d})$ and the constrained optimization problem (2.1.1) becomes the following unconstrained optimization problem:

$$\underset{\mathbf{d}\in\mathbb{R}^{n_d}}{\text{minimize}} \quad j(\mathbf{d}) \tag{2.1.2}$$

A detailed taxonomy of algorithms to solve the discrete PDECO problem can be reviewed in [51], but mainly two general approaches are available. The first approach is also referred to as *nested analysis and design* (**NAND**) because a complete PDE resolution, represented by $\mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{0}$, is nested inside of each evaluation of the cost function $j(\mathbf{d})$. The second approach is known as *simultaneous analysis and design* (**SAND**), because starting with an initial guess $(\mathbf{u}^0, \mathbf{d}^0)$ where $\mathbf{R}(\mathbf{u}^0, \mathbf{d}^0) \neq \mathbf{0}$, simultaneous feasibility and optimality conditions are forced in each step of the optimization. The first approach

is more common and easy to tackle using legacy codes, and the second approach gives more flexibility and speed of convergence in the overall solution, requiring more software development and delivering less general solutions (application dependency).

If we can compute the gradient of j with respect to $\mathbf{d}$, $\nabla_{\mathbf{d}} j(\mathbf{d})$, one can apply *gradient-based* methods [41], as described in algorithm 1 for **NAND** approach and algorithm 2 for **SAND** approach, which iteratively update the design vector $\mathbf{d}$ using the formula $\mathbf{d}^{k+1} = \mathbf{d}^k + \alpha^k \mathbf{p}^k$ with $\alpha^k > 0$ a parameter obtained using a line-search strategy and $\mathbf{p}^k \in \mathbb{R}^{n_d}$ a descent direction based in the calculated gradient. In the **SAND** approach, we can see an additional update of the state vector $\mathbf{u}$ using the formula $\mathbf{u}^{k+1} = \mathbf{u}^k + \alpha^k(\mathbf{u}_N^k + \nabla_{\mathbf{d}}\mathbf{u}(\mathbf{d}^k)\mathbf{p}^k)$ described in detail in the appendix A.1.

Input: initial guess $\mathbf{d}^0 \in \mathbb{R}^{n_d}$, tolerance $\epsilon > 0$, $k \leftarrow 0$

Compute reduced gradient $\nabla_{\mathbf{d}} j(\mathbf{d}^k)$;
while $\|\nabla_{\mathbf{d}} j(\mathbf{d}^k)\| > \epsilon$ **do**
  Compute descent direction $\mathbf{p}^k \in \mathbb{R}^{n_d}$ such that $\nabla_{\mathbf{d}} j(\mathbf{d}^k)^T \mathbf{p}^k < 0$;
  Compute step length $\alpha^k > 0$ such that $j(\mathbf{d^k} + \alpha^k \mathbf{p^k}) < j(\mathbf{d}^k)$;
  Update design vector $\mathbf{d}^{k+1} \leftarrow \mathbf{d}^k + \alpha^k \mathbf{p}^k$;
  $k \leftarrow k + 1$;
  Compute reduced gradient $\nabla_{\mathbf{d}} j(\mathbf{d}^k)$;
end

Output: Solution $(\mathbf{u}(\mathbf{d}^k), \mathbf{d}^k)$

Algorithm 1: Gradient-based optimization using **NAND** approach

Input: initial guess $(\mathbf{u}^0, \mathbf{d}^0) \in \mathbb{R}^{n_d}$, tolerances $\epsilon_u > 0$, $\epsilon_j > 0$, $k \leftarrow 0$

Compute reduced gradient $\nabla_{\mathbf{d}} j(\mathbf{d}^k)$ and $\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k)$;
while $\|\nabla_{\mathbf{d}} j(\mathbf{d}^k)\| > \epsilon_j$ *and* $\|\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k)\| > \epsilon_u$ **do**
  Compute descent direction $\mathbf{p}^k \in \mathbb{R}^{n_d}$ such that $\nabla_{\mathbf{d}} j(\mathbf{d}^k)^T \mathbf{p}^k < 0$;
  Compute step length $\alpha^k > 0$ such that $j(\mathbf{d^k} + \alpha^k \mathbf{p^k}) < j(\mathbf{d}^k)$;
  Update design vector $\mathbf{d}^{k+1} \leftarrow \mathbf{d}^k + \alpha^k \mathbf{p}^k$;
  Update state vector $\mathbf{u}^{k+1} \leftarrow \mathbf{u}^k + \alpha^k(\mathbf{u}_N^k + \nabla_{\mathbf{d}}\mathbf{u}(\mathbf{d}^k) \cdot \mathbf{p}^k)$;
  $k \leftarrow k + 1$;
  Compute reduced gradient $\nabla_{\mathbf{d}} j(\mathbf{d}^k)$ and $\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k)$;
end

Output: Solution $(\mathbf{u}^k, \mathbf{d}^k)$

Algorithm 2: Gradient-based optimization using **SAND** approach

Some examples of gradient-based well known methods to build $\mathbf{p}^k$ are *steepest descent*, *conjugate gradient* or *quasi-Newton methods*. If second-order derivatives can be obtained, $\nabla^2_{\mathbf{d}} j(\mathbf{d}^k)$ (Hessian of j), we can apply *Newton* methods to build the descent direction, which guarantees to converge to optimal values in less steps than using only the

gradient. However, in order to obtain the gradient $\nabla_{\mathbf{d}} j(\mathbf{d})$, we require the knowledge of the derivative of $\mathbf{u}(\mathbf{d})$ with respect to $\mathbf{d}$ and this information is not often available from many traditional PDE simulators. Furthermore, it is often extremely difficult if not impossible as a practical matter, to modify the PDE simulator to compute this information. The ability to do this opens up a wide variety of more efficient optimization techniques and provides the tools to address much larger problems.

2.1.1 Levels of optimization

The taxonomy proposed in [51], in order to understand the landscape of PDECO algorithms, is provided here:

Level-0 **NAND** approach that do not compute gradients. Only computations of $j(\mathbf{d})$ are available, with a nested resolution of $\mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{0}$ for each different input $\mathbf{d}$.

Level-1 **NAND** approach that use finite differences. A finite-difference approximation of the reduced gradient $\nabla_{\mathbf{d}} j(\mathbf{d})$ is available. With this approximation, gradient-based methods can be implemented (line-search + descent direction). The major drawback of this level relies on the finite-difference reduced gradient computation, because it requires n_d resolutions of $\mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{0}$ per optimization iteration and the accuracy of the computed optimal solution is degraded because of the truncation error involved with finite differences.

Level-2 **NAND** approach that use direct sensitivity to obtain the reduced gradient. If we apply the chain rule in the definition of the reduced cost function, the reduced gradient can be formulated as:

$$\nabla_{\mathbf{d}} j(\mathbf{d}) = \nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d}) \cdot \nabla_{\mathbf{d}} \mathbf{u}(\mathbf{d}) + \nabla_{\mathbf{d}} J(\mathbf{u}, \mathbf{d}) \tag{2.1.3}$$

The term $\nabla_{\mathbf{d}} \mathbf{u}$ is called *direct sensitivity matrix* and can be obtained using the implicit function theorem described in the appendix A.2 as:

$$\nabla_{\mathbf{d}} \mathbf{u}(\mathbf{d}) = -[\nabla_{\mathbf{u}} \mathbf{R}(\mathbf{u}, \mathbf{d})]^{-1} \nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) \tag{2.1.4}$$

Level-3 **NAND** approach that use adjoint sensitivity to obtain the reduced gradient. A different approach to calculate the first term of equation (2.1.3), $\nabla_{\mathbf{u}} J(\mathbf{u}(\mathbf{d}), \mathbf{d}) \cdot \nabla_{\mathbf{d}} \mathbf{u}(\mathbf{d})$, is based in the first-order necessary conditions, also known as the Karush-Kuhn-Tucker (KKT) conditions, defined in the appendix A.3 with Lagrange multiplier or *adjoint vector* $\boldsymbol{\lambda} \in \mathbb{R}^{n_u}$, applied to problem (2.1.1):

$$\nabla_{(\mathbf{u},\mathbf{d})} J(\mathbf{u}, \mathbf{d}) - \boldsymbol{\lambda}^T \nabla_{(\mathbf{u},\mathbf{d})} \mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{0}_{1 \times (n_u + n_d)} \tag{2.1.5}$$

Using only the columns related to the state vector $\mathbf{u}$, we have:

$$\begin{aligned} \nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d}) - \boldsymbol{\lambda}^T \nabla_{\mathbf{u}} \mathbf{R}(\mathbf{u}, \mathbf{d}) &= \mathbf{0}_{1 \times n_u} \\ \nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d}) &= \boldsymbol{\lambda}^T \nabla_{\mathbf{u}} \mathbf{R}(\mathbf{u}, \mathbf{d}) \end{aligned} \tag{2.1.6}$$

Equation (2.1.6) is commonly known as *adjoint problem*. Replacing (2.1.6) into (2.1.3) together with the direct sensitivity matrix (2.1.4), we have an adjoint-based reduced gradient expression:

$$\nabla_{\mathbf{d}} j(\mathbf{d}) = -\boldsymbol{\lambda}^T \nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d}) + \nabla_{\mathbf{d}} J(\mathbf{u}, \mathbf{d}) \tag{2.1.7}$$

Level-4 **SAND** approach that use direct sensitivity to obtain the reduced gradient. The direct sensitivity is the same calculated in level-2.

Level-5 **SAND** approach that use adjoint sensitivity to obtain the reduced gradient. The adjoint sensitivity is the same calculated in level-3.

Level-6 **SAND** approach that use second-order derivatives, or approximations to them. The second-order derivatives are used to build the descent direction $\mathbf{p}^k$ solving Newton equations using the Hessian and gradient of j.

In this report, we will focus our development in level-3 optimization methods, based in adjoint sensitivities, because they represent the best trade-off between development effort and computational efficiency. This affirmation is based in the estimated cost that each level of optimization delivers in each iteration. In the next subsection we present a detailed justification of the delivered costs, in order to support our decision.

2.1.2 Computational cost of each optimization level

We will assume that the PDE simulator uses iterative methods for sparse linear systems, which are typically based in several matrix-vector operations of cost $m << 2n_u^2$ floating-point operations (flops). We will assume that N matrix vector operations of cost m flops must be performed each time a PDE simulation is executed.

Level-0 methods doesn't scale appropriately when the size of the design space is large, but may be the only option for PDE codes where the complexity of the physics precludes the calculation of analytic derivatives and where standard approximations are poor. Because of their lack of scalability, we will not focus our research in this kind of methods.

Level-1 methods are also not suited for large design spaces because each derivative calculation needs at least $2n_d$ resolutions of the PDE constraint as $\mathbf{R}(\mathbf{u}, \mathbf{d}^k \pm h\mathbf{e}_i) = \mathbf{0}$ with $h > 0$ small scalar and $\mathbf{e}_i$ the canonical vector in the i-th coordinate. With the assumption of cost for a PDE simulation of N matrix-vector operations of cost m flops, the total estimated cost per optimization iteration is at least $2 \times n_d \times N \times m$ flops.

Level-2 methods need to solve n_d linear systems with different right-hand sides corresponding to columns of $\nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d})$. To obtain each column only one matrix-vector operation is needed. After that, for each linear system solution, a vector-vector operation with cost n_u flops is needed. The total estimated cost per iteration is $n_d \times (m + N \times m + n_u)$ flops.

On the other hand, level-3 methods need to solve only one linear system in order to obtain the adjoint vector $\boldsymbol{\lambda}$, and then n_d vector-vector operations with cost n_u flops. As in level-2 methods, each vector-vector operation is performed using a column of $\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d})$, which is obtained using one matrix-vector operation. The total estimated cost per iteration is $N \times m + n_d \times (m + n_u)$ flops.

Comparing the estimated costs of the previous levels, we can conclude that level-3 is the best alternative if n_d is considerably large. This will be our case of interest for large applications with hundreds of thousands of design variables.

Levels 4, 5 and 6 present similar complexities to levels 2 and 3 with better convergence properties and faster execution. However, as we mentioned before, the amount of development that needs to be done, and the loss of generality of the implemented algorithms makes them unfeasible for our purposes.

2.1.3 Descent direction calculation

Once we have the value of the derivative $\nabla_{\mathbf{d}} j(\mathbf{d})$, we can perform a gradient-based optimization using a descent direction as described in algorithms 1 and 2. Important gradient-based methods extracted from [41] are:

- Steepest descent:

$$\mathbf{p}^k = -\nabla_{\mathbf{d}} j(\mathbf{d}^k)$$

- Quasi-Newton:
  In these methods, the update step is performed as

$$\mathbf{d}^{k+1} = \mathbf{d}^k - \alpha^k \mathbf{B}_k^{-1} \nabla_{\mathbf{d}} j(\mathbf{d}^k)$$

  where the matrix $\mathbf{B}_k$ is an approximation of the Hessian matrix of $j(\mathbf{d})$. The most popular Quasi-Newton methods are (using the general notation $y_k = \nabla_{\mathbf{d}} j(\mathbf{d}^{k+1}) - \nabla_{\mathbf{d}} j(\mathbf{d}^k)$ and $\Delta x_k = \mathbf{d}^{k+1} - \mathbf{d}^k$):

  - Davidon-Fletcher-Powell (**DFP**):

$$\mathbf{B}_{k+1} = \left(I - \frac{y_k \Delta x_k^T}{y_k^T \Delta x_k}\right) \mathbf{B}_k \left(I - \frac{\Delta x_k y_k^T}{y_k^T \Delta x_k}\right) + \frac{y_k y_k^T}{y_k^T \Delta x_k}$$

$$\mathbf{H}_{k+1} = \mathbf{B}_{k+1}^{-1} = \mathbf{H}_k + \frac{\Delta x_k \Delta x_k^T}{y_k^T \Delta x_k} - \frac{\mathbf{H}_k y_k y_k^T \mathbf{H}_k^T}{y_k^T \mathbf{H}_k y_k}$$

  - Broyden-Fletcher-Goldfarb-Shanno (**BFGS**):

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{y_k y_k^T}{y_k^T \Delta x_k} - \frac{\mathbf{B}_k \Delta x_k (\mathbf{B}_k \Delta x_k)^T}{\Delta x_k^T \mathbf{B}_k \Delta x_k} \tag{2.1.8}$$

$$\mathbf{H}_{k+1} = \mathbf{B}_{k+1}^{-1} = \left(I - \frac{y_k \Delta x_k^T}{y_k^T \Delta x_k}\right)^T \mathbf{H}_k \left(I - \frac{y_k \Delta x_k^T}{y_k^T \Delta x_k}\right) + \frac{\Delta x_k \Delta x_k^T}{y_k^T \Delta x_k} \quad (2.1.9)$$

  - Broyden:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{y_k - \mathbf{B}_k \Delta x_k}{\Delta x_k^T \Delta x_k} \Delta x_k^T$$

$$\mathbf{H}_{k+1} = \mathbf{B}_{k+1}^{-1} = \mathbf{H}_k + \frac{(\Delta x_k - \mathbf{H}_k y_k) y_k^T \mathbf{H}_k}{y_k^T \mathbf{H}_k \Delta x_k}$$

  - Symmetric Rank 1:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{(y_k - \mathbf{B}_k \Delta x_k)(y_k - \mathbf{B}_k \Delta x_k)^T}{(y_k - \mathbf{B}_k \Delta x_k)^T \Delta x_k}$$

$$\mathbf{H}_{k+1} = \mathbf{B}_{k+1}^{-1} = \mathbf{H}_k + \frac{(\Delta x_k - \mathbf{H}_k y_k)(\Delta x_k - \mathbf{H}_k y_k)^T}{(\Delta x_k - \mathbf{H}_k y_k)^T y_k}$$

- Conjugate gradient methods:
  These algorithms are based in a classical method to solve linear and nonlinear systems in the form $F(x) = 0$, called conjugate gradient method. Here we present two extensions of this method aimed to solve a nonlinear unconstrained optimization problem.

  - Fletcher-Reeves:

$$\mathbf{p}^{k+1} = -\nabla_{\mathbf{d}} j(\mathbf{d}^{k+1}) + \beta_{FR}^{k+1} \mathbf{p}^k$$

    with β_{FR}^{k+1} defined as

$$\beta_{FR}^{k+1} = \frac{\nabla_{\mathbf{d}} j(\mathbf{d}^{k+1})^T \nabla_{\mathbf{d}} j(\mathbf{d}^{k+1})}{\nabla_{\mathbf{d}} j(\mathbf{d}^k)^T \nabla_{\mathbf{d}} j(\mathbf{d}^k)}$$

    and $\mathbf{p}^0 = -\nabla_{\mathbf{d}} j(\mathbf{d}^0)$ initial steepest descent direction.

  - Polak-Ribière:
    The formula to obtain the descent direction is the same as in Fletcher-Reeves, the only difference is in the β value:

$$\beta_{PR}^{k+1} = \frac{\nabla_{\mathbf{d}} j(\mathbf{d}^{k+1})^T (\nabla_{\mathbf{d}} j(\mathbf{d}^{k+1}) - \nabla_{\mathbf{d}} j(\mathbf{d}^{k+1}))}{\nabla_{\mathbf{d}} j(\mathbf{d}^k)^T \nabla_{\mathbf{d}} j(\mathbf{d}^k)}$$

- Newton:
  Using second-order derivatives, with the Hessian $H(\mathbf{d}^k) = \nabla_{\mathbf{d}}^2 j(\mathbf{d}^k)$, the Newton method is defined:

$$\mathbf{d}^{k+1} = \mathbf{d}^k - \alpha^k H(\mathbf{d}^k)^{-1} \nabla_{\mathbf{d}} j(\mathbf{d}^k)$$

In this report, we will focus on steepest descent and conjugate gradient methods, because their simplicity makes them usable for test problems, focus our work in the reduced gradient calculation, which is a kernel calculation in all of the descent direction calculation algorithms.

2.1.4 Line-search strategy

In the last step of the gradient-based methods we need to perform a line-search through the obtained descent direction $\mathbf{p}^k$. This problem can be stated as:

$$\min_{\alpha>0} j(\mathbf{d}^k + \alpha \mathbf{p}^k) \tag{2.1.10}$$

Several strategies can be adopted to obtain a solution α_* to this problem. In [41] we can find the *Wolfe conditions* that must satisfy the step α_* in order to give *sufficient decrease* to the problem (2.1.10):

$$j(\mathbf{d}^k + \alpha_* \mathbf{p}^k) \leq j(\mathbf{d}^k) + c_1 \alpha_* \mathbf{p}^k \cdot \nabla_{\mathbf{d}} j(\mathbf{d}^k) \tag{2.1.11a}$$

$$\mathbf{p}^k \cdot \nabla_{\mathbf{d}} j(\mathbf{d}^k + \alpha_* \mathbf{p}^k) \geq c_2 \mathbf{p}^k \cdot \nabla_{\mathbf{d}} j(\mathbf{d}^k) \tag{2.1.11b}$$

for some constants $c_1 \in (0, 1)$ and $c_2 \in (c_1, 1)$. The first inequality is called *Armijo condition* and the second is called *curvature condition*.

Given a descent direction $\mathbf{p}^k$, all line-search procedures require an initial step α_1^k, and generate a sequence

$$\{\alpha_i^k\}_{i>1} = \{\alpha_2^k, \alpha_3^k, \dots\}$$

that either terminates with a step length α_{i*}^k satisfying the conditions specified by the user (for example, the Wolfe conditions) or determines that such step length does not exist. In this case, a restart of the line-search can be performed, using a new initial step α_1^k that can be setted as $\alpha_1^k \leftarrow C * \alpha_1^k$, with $C > 1$.

Typical procedures consist of two phases: a *bracketing* phase that finds an interval $[a, b]$ containing acceptable step lengths, and a *selection* phase that zooms in to locate the final step length. The selection phase usually reduces the bracketing interval during its search for the desired step length and interpolates some of the function and derivative information gathered on earlier steps to guess the location of the minimizer. Details of proposed methods can be reviewed in [41]. Additionally, application dependent information can be used to accelerate the search, for example, using geometrical information or material properties of the physical objects in the problem, in order to pre-conditioning the value of the descent direction $\mathbf{p}^k$.

In this report, we will focus on simpler methods, such as backtracking or multipling-by-two

$$\alpha_1^k = 1\,, \qquad \{\alpha_i^k\}_{i>1} = \left\{\frac{\alpha_1^k}{2}, \frac{\alpha_1^k}{4}, \frac{\alpha_1^k}{8}, \dots\right\}$$

$$\alpha_1^k = 1\,, \qquad \{\alpha_i^k\}_{i>1} = \left\{2\alpha_1^k, 4\alpha_1^k, 8\alpha_1^k, \dots\right\}$$

using preconditioned and non-preconditioned descent directions, leaving sophisticated line-search strategies for future development. Details of the implemented methods will be explained in further chapters.

2.2 Finite element method

In the previous section, we have explained how to solve the PDECO problem using several algorithms. However, the reliability and efficiency of those algorithms are based in the resolution of the PDE constraint $\mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{0}$ by the preferred PDE simulator. This procedure is commonly known as solving the *forward* problem. Let us recall that the PDE solution $\mathbf{u}$ is called *state* vector and the input of the equation $\mathbf{d}$ is called *design* vector. Our legacy PDE simulator implements the *finite element* method, which is capable to solve several types of PDEs: transient or stationary, linear or nonlinear, it can contains single or multiple physics, and can be real or complex valued. For simplicity, in this work we will focus our research on stationary linear single physics equations, real or complex valued. Finite element method applied to this kind of equations leads to a linear system of equations as:

$$\mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{A}(\mathbf{d})\mathbf{u} - \mathbf{b}(\mathbf{d}) \tag{2.2.1}$$

with $\mathbf{A}(\mathbf{d}) \in \mathbb{K}^{n_u \times n_u}$ non-singular, $\mathbf{u}, \mathbf{b}(\mathbf{d}) \in \mathbb{K}^{n_d}$ and $\mathbb{K} = \mathbb{R}$ or $\mathbb{C}$. The matrix $\mathbf{A}(\mathbf{d})$ and vector $\mathbf{b}(\mathbf{d})$ are historically known as *stiffness matrix* and *load vector*.

All technical details involved in this method can be studied in many sources [54, 18, 32]), and because of its strong mathematical orientation, we will not include further definitions related to it. However, there are two aspects of the method of central importance for our work: the assembling of stiffness matrix and load vector, and the resolution of the resulting linear system.

An example of the assembling process is described briefly in algorithm 3. The input of the process is a set of nodes and elements, together with empty stiffness matrix and load vector. The nodes and elements are defined in a previous process called *mesh generation* in which a domain Ω is transformed into a *mesh*, a partition into n_{elem} subsets called *elements*. Each element is defined by the coordinates of their nodes. Some popular software utilities to perform this task are Gmsh [22], ANSYS ICEM [1] and GiD [2]. A large list of public domain and commercial software packages for mesh generation can be viewed in [3].

As mentioned in line 3 of algorithm 3, in order to calculate the local contributions $\mathbf{A}_e$ and $\mathbf{b}_e$ we need to use quadrature points to approximate integral calculations. Theoretical details of this procedure can be found in the literature previously mentioned. In practice, it corresponds to perform a 3-level nested loop that first traverses a set of predefined points inside the element, the so called quadrature points, and then traverses each pair of nodal points of the element, accumulating the resulting values of floating-point operations into arrays corresponding to $\mathbf{A}_e$ and $\mathbf{b}_e$. Line 4 can be viewed as a 1-level and 2-level loops, which traverses the nodal points (pairs of values or single values for matrix or vector respectively) of the element calculating their global index using a map $\{1, 2, 3\} \times \{1, \ldots, n_{elem}\} \rightarrow \{1, \ldots, n_{node}\}$ and storing the local values into the global matrix and vector using the mapped index. Line 6 can be thought as a 1-level loop that traverses all values of $\mathbf{A}$ and $\mathbf{b}$ asking if the corresponding nodal point belongs to a boundary. If

Input: n_{node} nodes, n_{elem} elements, empty matrix $\mathbf{A} \in \mathbb{R}^{n_{node} \times n_{node}}$ and empty vector $\mathbf{b} \in \mathbb{R}^{n_{node}}$

1 **for** $e = 1 : n_{elem}$ **do**
2 Obtain nodal coordinates for each nodal point in e-th element;
3 Calculate local stiffness matrix $\mathbf{A}_e \in \mathbb{R}^{3\times 3}$ and local load vector $\mathbf{b}_e \in \mathbb{R}^3$ using quadrature points to approximate integrals;
4 Add $\mathbf{A}_e$ and $\mathbf{b}_e$ to $\mathbf{A}$ and $\mathbf{b}$ respectively, using the global index $\{1, \ldots, n_{node}\}$ of the nodal points of e-th element;
5 **end**
6 Include PDE boundary conditions into $\mathbf{A}$ and $\mathbf{b}$;

Output: global stiffness matrix $\mathbf{A}$ and load vector $\mathbf{b}$

Algorithm 3: Assembling process in the finite element method for a 2D domain using triangular elements (3 nodes per element)

it belongs, its value is changed for a predefined value previously loaded from an input file. In further chapters we will show explicitly how this process is implemented, using a distributed-memory approach.

The assembling procedure is important in our work because we can obtain columns of $\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}, \mathbf{d})$, in the form

$$[\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}, \mathbf{d})]_{\cdot,j} = \left(\frac{\partial}{\partial d_j}\mathbf{A}(\mathbf{d})\right)\mathbf{u} - \frac{\partial}{\partial d_j}\mathbf{b}(\mathbf{d}),$$

assembling $\frac{\partial}{\partial d_j}\mathbf{A}(\mathbf{d})$ and $\frac{\partial}{\partial d_j}\mathbf{b}(\mathbf{d})$ using a modified version of the original assembling routine. We will explain this topic in further chapters.

The second important aspect is related to the resolution of the linear system $\mathbf{A}(\mathbf{d})\mathbf{u} = \mathbf{b}(\mathbf{d})$, obtained from the stiffness matrix and load vector. Both components are sparse and can be stored in a proper way, in order to save memory resources and accelerate computations. Using a block compressed row storage format [10], our legacy PDE simulator implements a toolbox of iterative methods for sparse linear systems [46], which are typically based in the resolution of several matrix-vector operations. In further chapters we will give details on its parallel implementation using a hybrid distributed-shared memory approach.

Relative to our work, we will need to use iterative solvers, matrix-vector and vector-vector multiplication operations from the toolbox at disposal, in order to calculate the reduced gradient explained in the previous section. For this reason, we need to study the current implementation of the mentioned matrix operations, and eventually we will need to modify an existing solver to make it usable for our purposes (for example, to implement a transposed linear system solver).

2.3 Reduced gradient calculation

With the main theoretical framework already explained, in this section we will show details about the reduced gradient calculation, which is a kernel calculation for the majority of algorithms detailed in section 2.1.1 (from level-2 to level-6). As we explained in section 2.1.2, our work will be focused in the adjoint sensitivity approach to obtain the reduced gradient. Some earlier works in this approach can be found in [24], [25] and [23], and posteriorly in [28].

2.3.1 Example

In order to understand all the steps involved in the calculation, a step by step example will be presented. Let us consider the following partial differential equation to be solved in a domain $\Omega \in \mathbb{R}^2$, with u solution of the equation and f a source (we assume that u and f belong to suitable functional spaces):

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{2.3.1}$$

After discretization using the finite element method the following (possibly after reordering columns and rows) linear system arises: $\mathbf{Au} = \mathbf{b}$ with $\mathbf{A} \in \mathbb{R}^{n_u \times n_u}$ and $\mathbf{u}, \mathbf{b} \in \mathbb{R}^{n_u}$, being $\mathbf{u}$ the unknown vector.

Now, let us consider in equation (2.3.1) a design vector $\mathbf{d} \in \mathbb{R}^{n_d}$:

$$\begin{aligned} -\Delta u &= f(\mathbf{d}) \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{2.3.2}$$

For example, if the source function has the form $f(x, y) = x^2 + y^2$, the design vector can be interpreted as a weight in each term, $f(x, y, \mathbf{d}) = d_1x^2 + d_2y^2$, with $\mathbf{d} = (d_1, d_2) \in \mathbb{R}^{n_d}$, $n_d = 2$. If we modify the design vector $\mathbf{d}$, we can obtain different values of the solution u. The inverse problem in this case, for a L^2-norm cost functional with data function u^*, defined as

$$\mathcal{J}(u, \mathbf{d}) = \frac{1}{2}\int_\Omega (u - u^*)^2 dV \tag{2.3.3}$$

is as follows:

$$\begin{aligned} \underset{(u,\mathbf{d})\in\mathcal{U}\times\mathbb{R}^{n_d}}{\text{minimize}} \quad & \frac{1}{2}\int_\Omega (u - u^*)^2 dV \\ \text{subject to} \quad & \begin{aligned} -\Delta u &= f(\mathbf{d}) \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \end{aligned} \tag{2.3.4}$$

The solution of this problem can be viewed as the answer to the following question:

Which is the optimal value of $\mathbf{d}$,
that makes the PDE solution u *as similar as possible to a data function* u^**?*

Applying the discretization, problem (2.3.4) becomes:

$$\begin{aligned} \underset{(\mathbf{u},\mathbf{d})\in\mathbf{R}^{n_u}\times\mathbb{R}^{n_d}}{\text{minimize}} \quad & J(\mathbf{u},\mathbf{d}) \\ \text{subject to} \quad & \mathbf{R}(\mathbf{u},\mathbf{d}) = \mathbf{0} \end{aligned} \tag{2.3.5}$$

with $J : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \to \mathbb{R}$ a discrete version of the cost functional

$$J(\mathbf{u},\mathbf{d}) = \frac{1}{2}(\mathbf{u}-\mathbf{u}^*)^T\mathbf{Q}(\mathbf{u}-\mathbf{u}^*) \tag{2.3.6}$$

$\mathbf{Q} \in \mathbb{R}^{n_u\times n_u}$ symmetric and $\mathbf{R} : \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \to \mathbb{R}^{n_u}$ defined as $\mathbf{R}(\mathbf{u},\mathbf{d}) = \mathbf{A}\mathbf{u}-\mathbf{b}(\mathbf{d})$. We can observe that the matrix $\mathbf{A}$ doesn't have a dependency on the design vector $\mathbf{d}$, because in the PDE (2.3.2) the design variables only appear in the right-hand side. Considering that $\mathbf{u}$ depends on $\mathbf{d}$ through $\mathbf{u}(\mathbf{d}) = \mathbf{A}^{-1}\mathbf{b}(\mathbf{d})$ ($\mathbf{A}^{-1}$ is never calculated numerically), we can define the reduced cost function $j : \mathbb{R}^{n_d} \to \mathbb{R}$, $j(\mathbf{d}) := J(\mathbf{u}(\mathbf{d}),\mathbf{d})$ and the constrained optimization problem (2.3.5) will have the same solution as the following unconstrained optimization problem:

$$\underset{\mathbf{d}\in\mathbb{R}^{n_d}}{\text{minimize}} \quad j(\mathbf{d}) \tag{2.3.7}$$

According to the adjoint sensitivity calculation of section 2.1.1, we need to obtain the adjoint vector $\boldsymbol{\lambda}$ solving the system described in equation (2.1.6):

$$\nabla_{\mathbf{u}} J(\mathbf{u},\mathbf{d}) = \boldsymbol{\lambda}^T \nabla_{\mathbf{u}} \mathbf{R}(\mathbf{u},\mathbf{d})$$

In our example, this system is equal to:

$$(\mathbf{u}-\mathbf{u}^*)^T\mathbf{Q} = \boldsymbol{\lambda}^T\mathbf{A} \tag{2.3.8}$$

or equivalently

$$\mathbf{A}^T\boldsymbol{\lambda} = \mathbf{Q}(\mathbf{u}-\mathbf{u}^*) \tag{2.3.9}$$

After that, we need to obtain $\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d})$, which in our example is equal to:

$$\begin{aligned} \nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d}) &= \nabla_{\mathbf{d}}\left(\mathbf{A}\mathbf{u}-\mathbf{b}(\mathbf{d})\right) \\ &= \mathbf{0} - \nabla_{\mathbf{d}}\mathbf{b}(\mathbf{d}) \end{aligned} \tag{2.3.10}$$

With these two components, we can calculate the reduced gradient defined in equation (2.1.7):

$$\nabla_{\mathbf{d}} j(\mathbf{d}) = -\boldsymbol{\lambda}^T\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d}) + \nabla_{\mathbf{d}} J(\mathbf{u},\mathbf{d})$$

In our example, the reduced gradient will have the following form

$$\nabla_{\mathbf{d}} j(\mathbf{d}) = -\boldsymbol{\lambda}^T\nabla_{\mathbf{d}}\mathbf{b}(\mathbf{d}) + \mathbf{0} \tag{2.3.11}$$

The second term is zero because there are no explicit values of the design vector in the cost function.

2.3.2 Numerical requirements

Three elements are needed to obtain the reduced gradient (2.1.7): $\boldsymbol{\lambda}$, $\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d})$ and $\nabla_{\mathbf{d}}J(\mathbf{u},\mathbf{d})$. Additionally, two extra elements are needed to build the adjoint system (2.1.6): $\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u},\mathbf{d})$ and $\nabla_{\mathbf{u}}J(\mathbf{u},\mathbf{d})$. To summarize, we need to calculate four explicit derivatives and solve one additional transposed linear system with the same dimensions as the forward problem.

As we mentioned in the previous section, this linear system can be solved using *almost* the same routines used to solve the forward problem $\mathbf{R}(\mathbf{u},\mathbf{d})=\mathbf{0}$. The only modification that needs to be done in an existing iterative linear system solver is relative to the matrix-vector operations used inside of it. In order to have a transposed solver re-using the implemented iterative methods, we need to use transposed matrix-vector operations.

The construction of the explicit derivatives can be done in several ways. If we don't have access to the code that calculates $J(\mathbf{u},\mathbf{d})$ and $\mathbf{R}(\mathbf{u},\mathbf{d})$, running several times each routine and calculating finite differences with those results can be done to obtain an estimation of the derivatives. On the other hand, if we have access to the code the best alternative is to use *automatic differentiation* (AD) [26] to generate the code of routines that calculates those derivatives. If the code is not well structured, doesn't follow programming patterns needed by AD tools, or is intrinsically parallel, we can still calculate the derivatives *by hand*, carefully studying the code and identifying the parts of it where the corresponding variables are used, and then changing the values by the explicit derivatives. This topics will be discussed in further chapters, with concrete examples.

2.3.3 Algorithm

The steps involved in the calculation of $\nabla_{\mathbf{d}}j(\mathbf{d})$ can be viewed in algorithm 4.

Input: $\mathbf{d}$ design vector

1 $\mathbf{u} \leftarrow$ solve $\mathbf{R}(\mathbf{u},\mathbf{d})=\mathbf{0}$; (*forward problem*)
2 Calculate explicit derivatives $\nabla_{\mathbf{u}}J(\mathbf{u},\mathbf{d})$ and $\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u},\mathbf{d})$;
3 $\boldsymbol{\lambda} \leftarrow$ solve $\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u},\mathbf{d})^T\boldsymbol{\lambda} = \nabla_{\mathbf{u}}J(\mathbf{u},\mathbf{d})^T$; (*adjoint problem*)
4 Calculate explicit derivatives $\nabla_{\mathbf{d}}J(\mathbf{u},\mathbf{d})$ and $\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d})$;
5 $\nabla_{\mathbf{d}}j(\mathbf{d}) = -\boldsymbol{\lambda}^T\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d}) + \nabla_{\mathbf{d}}J(\mathbf{u},\mathbf{d})$;

Output: $\nabla_{\mathbf{d}}j(\mathbf{d})$

Algorithm 4: Reduced gradient calculation using adjoint sensitivity

The implementation of this algorithm on top of our legacy PDE simulator, which uses a distributed-memory approach to solve the forward problem, will be our main task to solve in this report. After this implementation works fine for our test problems, we can move on to implement descent direction algorithms described in section 2.1.3, and line-search strategies as described in section 2.1.4.

Chapter 3

Implementation

In this chapter we include a description of Alya [30, 29], the PDE simulator in which we are going to integrate our developments. We also include the main implementation aspects of the parallel resolution of a linear system $\mathbf{Au} = \mathbf{b}$ using sparse iterative methods, which are the core operations performed by Alya. For simplicity, our focus will be in Alya's aspects related to linear stationary PDE resolution with Dirichlet boundary conditions.

After that, we include a description of the main implementation aspects of our work: the reduced gradient, descent direction and line-search calculations.

Due to its simplicity and flexibility, we will make use of UML diagrams [45] to explain architecture or communication details of the proposed implementations.

3.1 Alya

Alya, the PDE simulator designed in the CASE department of BSC, solves incompressible and compressible fluid dynamics, solid mechanics, thermal flows and a large variety of problems. It is written in Fortran 90 following an imperative programming paradigm [43], was conceived as a parallel application from scratch (using MPI and OpenMP as main parallel programming models) and currently has more than 750000 lines of code. As we can see in figure 3.1, it is composed by three architectural layers: *Physics*, *Services* and *Kernel*.

Physics includes each of the individual physical problems, coded as internal *modules*. Multi-physical problems can be solved by coupling the solutions of several modules.

Services is a toolbox that provides a variety of independent procedures to be called by the physical modules and the kernel. Inside this level we can find the service *Parall*, which has a fundamental importance because it is in charge of the parallelization management. The parallelization scheme can be loop-external, loop-internal or hybrid. In the loop-external scheme, a work distribution is achieved by sub-division of the original problem

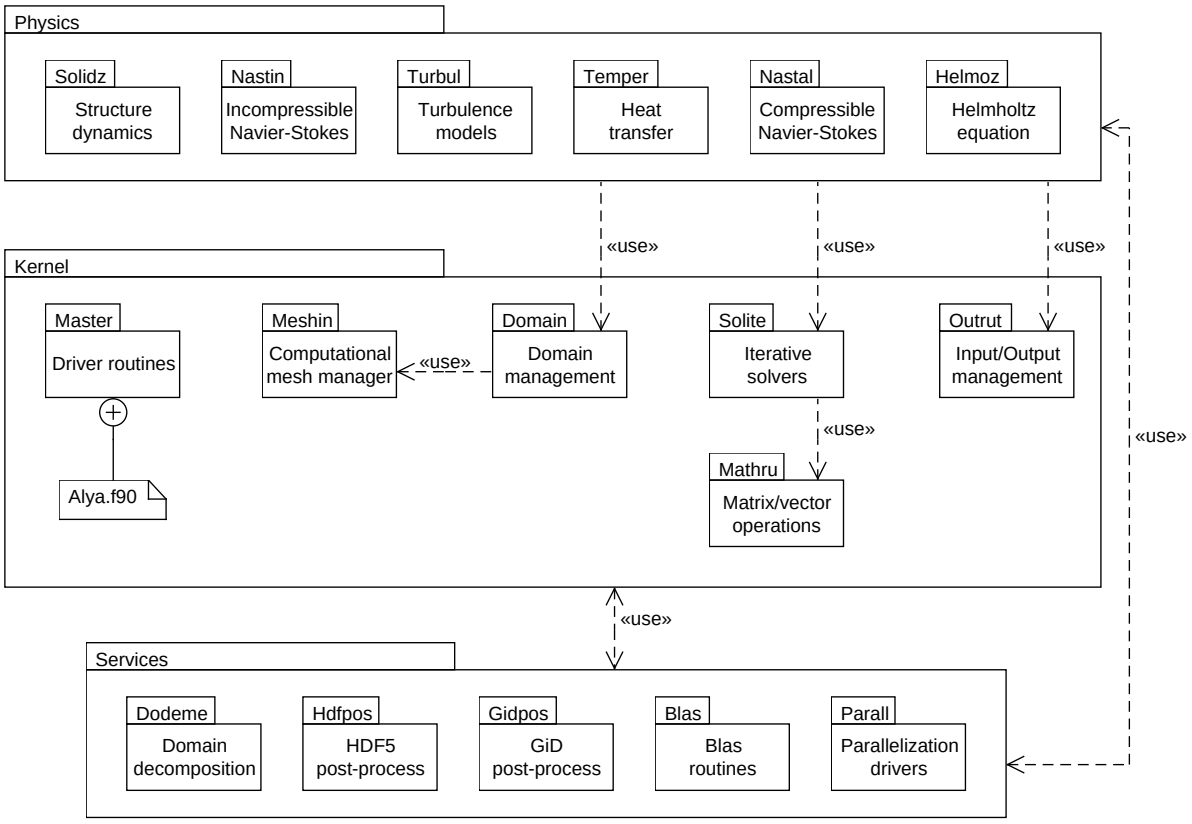

Figure 3.1: Simplified model of the three architectural layers of Alya

in smaller problems running concurrently with all the inter-process communication done using the MPI programming model. Details about this aspect are included in the next subsections. The loop-internal scheme is based on the parallelization of code loops using OpenMP. In the hybrid scheme both parallelization strategies are used at the same time.

Kernel contains the core of Alya, where all the common procedures lies. It has all the basic tools to be linked with at least one physical module in order to solve numerically the associated PDE described in it. As mentioned in section 2.2, the finite element method is implemented as discretization scheme. In order to apply this method, the first step is to read the computational mesh from input files and perform its partitioning. This task is performed by the packages *Meshin* and *Domain*. *Meshin* handles the external library METIS [34] that is specially designed for mesh partitioning tasks. *Domain* handles all the data structures obtained as result of the mesh partitioning, for example the nodal and element index arrays, or map functions between local and global numbering.

Today there are more than 30 researchers and post-graduates working in Alya, either programming or running large instances of a specific physical module, or developing core routines in the Kernel or Services layers. A control version system together with

a nightly test suite verification are used in order to manage the code modifications and error detections.

3.1.1 Main driver

The main driver of Alya is located in the kernel file `Alya.f90`. It performs calls to other kernel driver routines, delegating the main workload to physics modules loaded in input files.

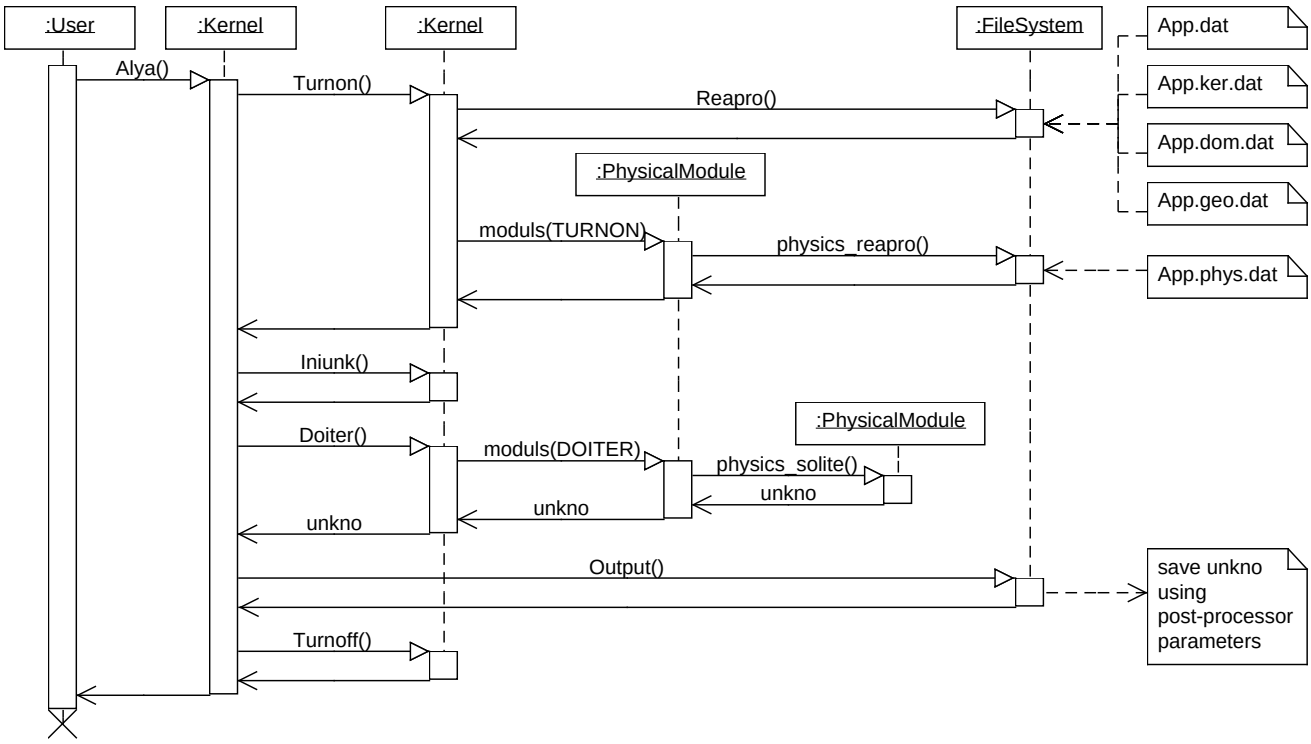

Figure 3.2: Sequence diagram of the main driver routine `Alya()`

In figure 3.2 we can view a simplified sequence diagram of the main driver routine `Alya()`. The description of each step is as follows:

- The kernel launches the routine `Turnon()` which reads input files (through routine `Reapro()`), allocates memory for the main variables and handles the mesh partitioning process, which will be described in the next subsection.
- The routine `Reapro()` handles the interaction with the file system to read general input files, such as kernel and domain parameters, and the computational mesh. It reads which physics modules will be loaded in the execution of Alya (specified in a list inside of the file `App.dat`).

- After that, the kernel launches the interface routine `moduls(TASK)` which receives a specific task identifier as input. Through this interface plus the identifier `TURNON`, each loaded physics module launches the file system interaction routine `physics_reapro()` to read its characteristic parameters.
- Once all necessary variables have been allocated, the routine `Iniunk()` initializes them.
- After the initialization, the routine `Doiter()` is launched by the kernel, in order to run the main routines involved in the discretization scheme and linear system solving. This step will be explained in the next subsection. The output of this routine is the array `unkno` which stores the values of the state vector $\mathbf{u}$ solution of the system $\mathbf{Au} = \mathbf{b}$.
- Finally the routine `Output()` handles the interaction with the file system to write the solution stored in the array `unkno`, and the routine `Turnoff()` closes the execution, deallocating variables and writing execution reports.

3.1.2 Assembling and linear system solving

As we mentioned in section 2.2, concerning Alya's internal structure, our work is focused in the study of two procedures: assembling and linear system solving. To fully understand these procedures, we need to review the basics of the parallelization strategy implemented in Alya. According to [53], the parallelization follows a data partitioning strategy, in which the computational domain or *mesh* is divided into several sub-domains with balanced number of elements on each one. The initial mesh reading and partitioning is performed by a master process using the external library METIS [34]. After that, the master sends to slave processes information about their corresponding sub-domain parameters (number and indexes of elements, internal and external boundaries, and other useful information). These tasks are performed by several internal routines from packages *Meshin* and *Domain*, which are handled inside `Turnon()` (for simplicity we haven't included these internal routines into diagram of figure 3.2). Once each slave has its sub-domain information and also knows its neighbouring sub-domains (with whom it will share boundary information), the assembling of a distributed linear system can start. The resulting system $\mathbf{Au} = \mathbf{b}$ is solved using iterative methods.

A simplified sequence diagram of the assembling and linear system solving can be viewed in figure 3.3. The description of each step is as follows:

- Each slave executes the same physics module calling the routine `physics_solite()` (hosted in the same module) which returns a distributed array denoted `unkno`. The size of this array may be different in each slave.

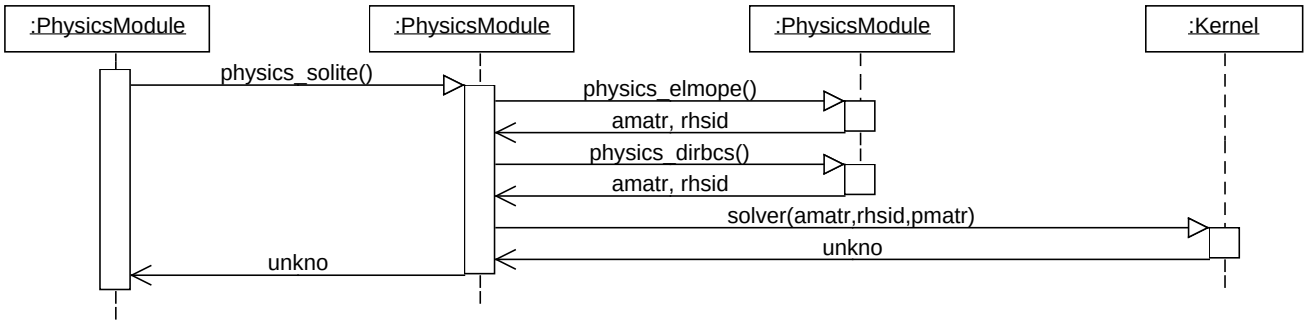

Figure 3.3: Sequence diagram of the assembling and linear system solving procedures in Alya performed in the routine `physics_solite()`

- Routine `physics_elmope()` performs the tasks detailed in lines 1-5 from algorithm 3 of section 2.2. Each slave p stores its local contributions $\mathbf{A}_p$ and $\mathbf{b}_p$ in arrays `amatr` and `rhsid` respectively.
- Routine `physics_dirbcs()` performs the task detailed in line 6 from algorithm 3 of section 2.2, in which boundary conditions are imposed over arrays `amatr` and `rhsid`.
- Finally, the kernel routine `solver(...)` is launched by the physics module, using and extra parameter `pmatr` corresponding to a precondition matrix. Using local contributions $\mathbf{A}_p$ and $\mathbf{b}_p$ on each slave p, a wide range of iterative strategies can be followed in order to solve the linear system $\mathbf{Au} = \mathbf{b}$. Each slave keeps a part $\mathbf{u}_p$ of the global solution $\mathbf{u}$, storing it in the array `unkno`.

`physics_elmope()`

In figure 3.4 we can observe the sequence diagram of the routine `physics_elmope()`. The description of each step is as follows:

- First of all, using the element index `e`, all local information is extracted from global data structures previously allocated and initialized by internal kernel packages (*Meshin* and *Domain*). Also, physics-dependent information relative to this element is also extracted, for example, material properties or velocity fields.
- Using the local information, the physics module launches the kernel routine denoted `elmcar(local_data)` which calculates the volume, cartesian derivatives and Hessian matrix associated to this element. This values are dependent on geometric data and the type of element (trial function basis) chosen in the kernel input files.

- With all this information, the routine `physics_elmloc(...)` calculates the local stiffness matrix $\mathbf{A}_e$ and local load vector $\mathbf{b}_e$, storing them in the arrays `elmat` and `elrhs`.
- Finally the physics module launches the routine `physics_assemb(...)` which adds the local arrays to the global arrays `amatr` and `rhsid` using a map denoted `ipoin = lnods(inode)` which maps the node `inode` of the current element to a global index node `ipoin`.

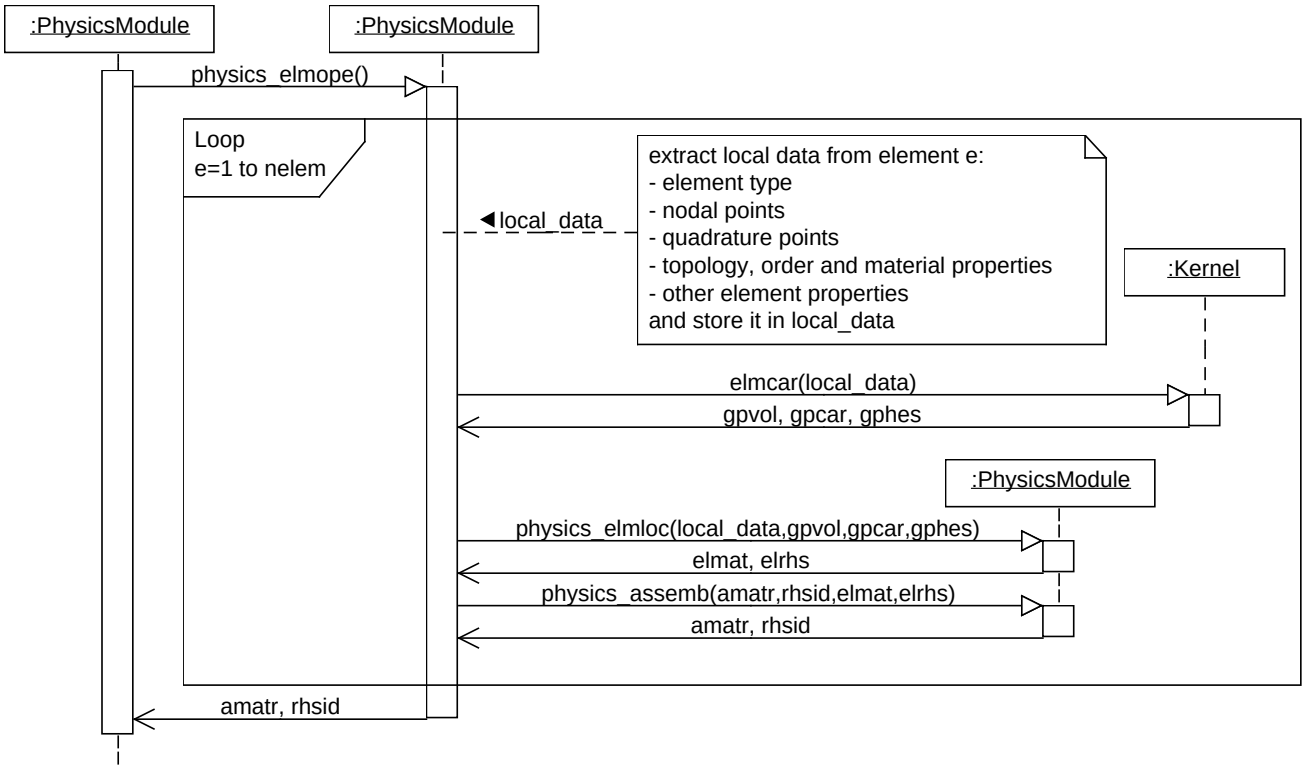

Figure 3.4: Sequence diagram of the routine `physics_elmope()`

physics_dirbcs()

The routine `physics_dirbcs()` modifies the right-hand side $\mathbf{b}$ stored in `rhsid` fixing the values which belong to an external boundary, and also modifies the rows and columns of the matrix $\mathbf{A}$ stored in `amatr`. For example, if the nodal point indexed by i belongs to a boundary Γ with Dirichlet condition $u = u_0$, the matrix and right-hand side must be modified as:

$$\left[\begin{array}{c|c|c} & a_{1,i} & \\ & \vdots & \\ & a_{i-1,i} & \\ \hline a_{i,1} \ \dots \ a_{i,i-1} & a_{i,i} & a_{i,i+1} \ \dots \ a_{i,n_u} \\ \hline & a_{i+1,i} & \\ & \vdots & \\ & a_{n_u,i} & \end{array}\right] \longrightarrow \left[\begin{array}{c|c|c} & 0 & \\ & \vdots & \\ & 0 & \\ \hline 0 \ \dots \ 0 & 1 & 0 \ \dots \ 0 \\ \hline & 0 & \\ & \vdots & \\ & 0 & \end{array}\right] \tag{3.1.1}$$

$$\left(\begin{array}{c} \vdots \\ \hline b_i \\ \hline \vdots \end{array}\right) \longrightarrow \left(\begin{array}{c} \vdots \\ \hline u_0 \\ \hline \vdots \end{array}\right) \tag{3.1.2}$$

solver()

The routine `solver()` works as an interface for several iterative solvers. In Alya's kernel layer, we can find several algorithms already implemented such as GMRES, Bi-CGSTAB, CG [46] or Deflated CG [47], [38]. Preconditioning methods are used in order to get better conditioned systems. A linear system is *preconditioned* if $\mathbf{A}$ is left or right multiplied by a non-singular matrix $\mathbf{P}$, resulting in systems $\mathbf{P}^{-1}\mathbf{A}\mathbf{u} = \mathbf{P}^{-1}\mathbf{b}$ or $\mathbf{A}\mathbf{P}^{-1}\mathbf{y} = \mathbf{b} \wedge \mathbf{P}\mathbf{u} = \mathbf{y}$ respectively.

The implementations are adapted to the block compressed row storage (BCRS) format [10], used to store each slave's local contribution $\mathbf{A}_p$ into the array `amatr`. This storage format is based in the compressed row storage (CRS) format, which puts the subsequent nonzeros of the matrix rows in contiguous memory locations. Three arrays are needed to store a sparse matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$: one to store the nonzero floating-point numbers of the matrix (`amatr`), and the other two to store index integers (`c_dom`, `r_dom`). The array `amatr` stores the nonzero values as they are traversed in a row-wise fashion (left-right and top-down). Array `c_dom` stores the column indexes of the elements in `amatr`. That is, if `amatr(`k`)` $= a_{i,j}$ then `c_dom(`k`)` $= j$. Array `r_dom` stores the locations in `amatr` that start a row, that is, if `amatr(`k`)` $= a_{i,j}$ then `r_dom(`i`)` $\leq k <$ `r_dom(`$i+1$`)`. By convention, we add and extra element to the tail of `r_dom`, $n_{nz} + 1$, where n_{nz} is the

number of nonzeros in the matrix. The storage savings for this approach is significant. Instead of storing n^2 elements, we need only $2n_{nz} + n + 1$ storage locations. In figures 3.5 and 3.6 we can see examples of the CRS and BCRS formats.

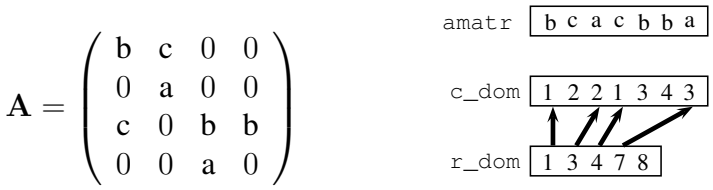

Figure 3.5: Example of compressed row storage format

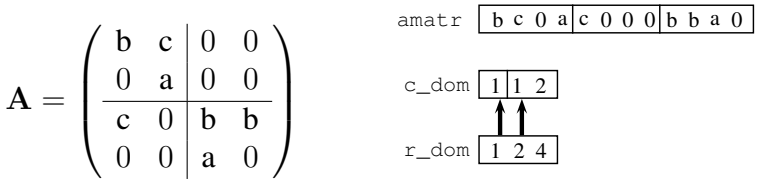

Figure 3.6: Example of block compressed row storage format, using 2×2 blocks

A central routine intensively used by almost all iterative solvers is the sparse matrix-vector multiplication (SpMV). In algorithms 5 and 6 we can see the matrix-vector operation procedures using CRS and BCRS formats. The BCRS version works like the CRS version with the addition of an inner block iteration depicted in lines 6-11 of algorithm 6.

In parallel implementations of iterative solvers, each slave p performs matrix-vector operations using local data $\mathbf{A}_p$ and $\mathbf{b}_p$, obtaining $\mathbf{u}_p$. In each iteration of those solvers, an update operation must be done over $\mathbf{u}_p$, adding contributions from the internal boundary nodes, shared with other slaves. These communications are performed using MPI calls to `MPI_Allreduce` and `MPI_SendRecv`. Details about the parallel implementation are explained in the next subsection.

Input: matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ stored in (amatr, c_dom, r_dom), vectors $\mathbf{x}$ and $\mathbf{y}$

```
1  for i = 1 to n do
2  |   y[i] ← 0;
3  |   for j = r_dom[i] to r_dom[i + 1] − 1 do
4  |   |   y[i] ← y[i] + amatr[j] ∗ x[c_dom[j]];
5  |   end
6  end
```

Output: $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$

Algorithm 5: Procedure for computing $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is stored using CRS format

Input: matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ stored in (amatr, c_dom, r_dom), vectors $\mathbf{x}$ and $\mathbf{y}$

```
1  z ← 0;
2  n_b ← n/r_b;
3  for b = 1 to n_b do
4  |   y[r_b ∗ (b − 1) + 1 : r_b ∗ b] ← 0;
5  |   for j = r_dom[b] to r_dom[b + 1] − 1 do
6  |   |   for k = 1 to r_b do
7  |   |   |   for t = 1 to c_b do
8  |   |   |   |   y[r_b ∗ (b − 1) + k] ←
   |   |   |   |   y[r_b ∗ (b − 1) + k] + amatr[z] ∗ x[c_b ∗ (c_dom[j] − 1) + t];
9  |   |   |   |   z + +;
10 |   |   |   end
11 |   |   end
12 |   end
13 end
```

Output: $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$

Algorithm 6: Procedure for computing $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is stored using BCRS format with block size $r_b \times c_b$

3.1.3 Communication types and scheduling

In the assembling procedure, each slave hasn't any communication with the other slaves and the scalability only depends on the load balancing.

Inside of the iterative solvers, the communications are performed as follows:

- At the very beginning, before starting the main loop of iterations, each slave exchanges entries of the local $\mathbf{b}_p$ vector that contain values on the shared nodes (the nodes belonging to more than one sub-domain) with all the neighbouring sub-domains. This exchange is performed using `MPI_Sendrecv`. Having all the contributions from the neighbouring sub-domains, a slave sums them up and gets the global $\mathbf{b}$ values on the shared nodes.
- Each slave performs matrix-vector multiplications locally and then exchanges and adds contributions on the shared nodes using `MPI_Sendrecv`, so that each slave has the global product values on the shared nodes.
- Each slave performs scalar vector products locally and then the master assembles and sums contributions from all the slaves using `MPI_Allreduce`, so that each slave has the global value of the calculated dot product of two vectors.
- Each slave calculates linear combination of vectors, which is done locally without any communication. This operation is perfectly parallel.

The necessity of the `MPI_Sendrecv` communication can be explained through the principles of the domain decomposition technique on a simple example. Let us consider a domain Ω which is divided into two disjoint sub-domains Ω_1 and Ω_2 with interface Γ_{12}. This is shown in figure 3.7, where $\mathbf{u}_1$, $\mathbf{u}_2$, and $\mathbf{u}_3$ are the unknowns in Ω_1, Ω_2, and on Γ_{12}, respectively. A renumbering process of nodal points must be performed in order to keep local vectors updated of the sub-domain actual size. Let us take any linear PDE represented by the system $\mathbf{Au} = \mathbf{b}$ in the discretized domain. Furthermore, let us use a

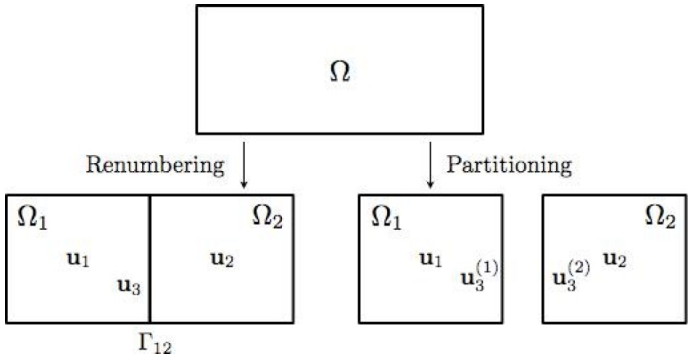

Figure 3.7: Node renumbering and domain partitioning

simple iteration (Richardson) without preconditioning for solving the system:

$$\mathbf{u}^{k+1} = \mathbf{u}^k + (\mathbf{b} - \mathbf{A}\mathbf{u}^k) \tag{3.1.3}$$

Using the sub-domain renumbering of figure 3.7, equation (3.1.3) can be written as:

$$\begin{aligned}
\mathbf{u}^{k+1} &= \mathbf{u}^k + \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_3 \\ \mathbf{b}_2 \end{bmatrix} - \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{13} & \mathbf{0} \\ \mathbf{A}_{31} & \mathbf{A}_{33} & \mathbf{A}_{32} \\ \mathbf{0} & \mathbf{A}_{23} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1^k \\ \mathbf{u}_3^k \\ \mathbf{u}_2^k \end{bmatrix} \\
&= \mathbf{u}^k + \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_3^{(1)} + \mathbf{b}_3^{(2)} \\ \mathbf{b}_2 \end{bmatrix} - \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{13} & \mathbf{0} \\ \mathbf{A}_{31} & \mathbf{A}_{33}^{(1)} + \mathbf{A}_{33}^{(2)} & \mathbf{A}_{32} \\ \mathbf{0} & \mathbf{A}_{23} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1^k \\ \mathbf{u}_3^k \\ \mathbf{u}_2^k \end{bmatrix}
\end{aligned} \tag{3.1.4}$$

where the contribution for the interface unknown, $\mathbf{A}_{33}$, has been split in contributions from Ω_1 and Ω_2, which are $\mathbf{A}_{33}^{(1)}$ and $\mathbf{A}_{33}^{(2)}$, respectively. The same has been carried out for the right-hand side. In the finite-element context, these contributions reflect the values of the test functions of both elements that share the interface. Now, we can perform a partition which is based on elements of the mesh. Consequently, the nodes on the boundary are repeated and noted as $\mathbf{u}_3^{(1)}$ Ω_1, and $\mathbf{u}_3^{(2)}$, in Ω_2. Let us apply the matrix-vector operation as in the non-partitioned case (we put a zero coefficient in the matrix when the contribution is not available in the sub-domain):

$$\mathbf{u}^{(1)^{k+1}} = \mathbf{u}^{(1)^k} + \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_3^{(1)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{13} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{31} & \mathbf{A}_{33}^{(1)} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1^k \\ \mathbf{u}_3^{(1)^k} \\ \mathbf{u}_3^{(2)^k} \\ \mathbf{u}^k \end{bmatrix} \tag{3.1.5}$$

$$= \mathbf{u}^{(1)^k} + \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_3^{(1)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{3.1.6}$$

$$\mathbf{u}^{(2)^{k+1}} = \mathbf{u}^{(2)^k} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{b}_3^{(2)} \\ \mathbf{b}_2 \end{bmatrix} - \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{33}^{(2)} & \mathbf{A}_{32} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{23} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1^k \\ \mathbf{u}_3^{(1)^k} \\ \mathbf{u}_3^{(2)^k} \\ \mathbf{u}^k \end{bmatrix} \tag{3.1.7}$$

$$= \mathbf{u}^{(2)^k} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{r}_3^{(2)} \\ \mathbf{r}_2 \end{bmatrix} \tag{3.1.8}$$

The idea of the domain decomposition is to recover the result of equation (3.1.3) each time we perform the matrix-vector operation. In order to have an equivalent result in Ω_1 to the original one in equation (3.1.3), the contribution of Ω_2 has to be added to the

interface node $\mathbf{u}_3^{(1)}$. In the distributed memory context, this operation can be carried out by sending the residual $\mathbf{r}_3^{(2)}$, from Ω_2 to Ω_1. The same thing stands for Ω_2. Therefore, the residual exchange can be carried out using `MPI_Sendrecv` between Ω_1 and Ω_2. After the exchange, the following actualization is performed:

$$\mathbf{u}^{(1)^{k+1}} = \mathbf{u}^{(1)^k} + \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_3^{(1)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{r}_3^{(2)} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{3.1.9}$$

$$\mathbf{u}^{(2)^{k+1}} = \mathbf{u}^{(2)^k} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{r}_3^{(2)} \\ \mathbf{r}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{r}_3^{(1)} \\ \mathbf{0} \end{bmatrix} \tag{3.1.10}$$

A key issue in the communication between slaves is the message scheduling algorithm [16]. Figure 3.8-top shows the kind of problem that can arise when data transfer is not properly scheduled. The red crosses indicate a spot where a two-way communication could be performed, but instead only one-way messages were delivered. A bad scheduling can strongly penalize the scalability of the execution, overloading the interconnection network or filling the buffers of the network interface cards (NIC) of the compute nodes. In Alya, an optimal communication scheduling is implemented, as showed in figure 3.8-bottom, which minimizes the number of simultaneous sending or receiving by one slave, allowing faster transmission between nodes and reducing the overall overhead generated from communications.

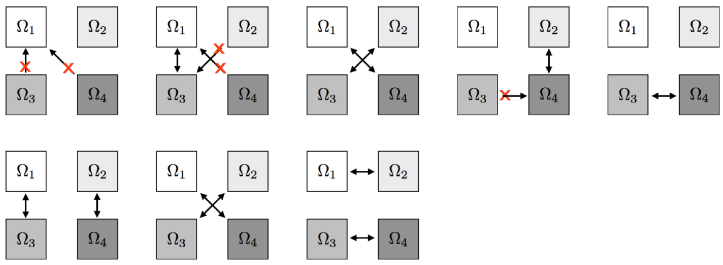

Figure 3.8: Schematic example of Alya's parallel communications with 4 subdomains

An execution trace of Alya using 9 slaves, generated with a performance analysis tool called Paraver [37], can be viewed in figure 3.9. In the X-axis we have the execution time in microseconds and in the Y-axis we have the states of the master (row 1) and the slaves (rows 2 to 9). We can see in the first steps the file reading and mesh partitioning performed by the master process. After the master sends the corresponding sub-domain information to each slave process, each one assembles its corresponding left and right-hand side, solves the associated linear system exchanging boundary information with their neighbours using the MPI routine `MPI_SendRecv`, and writes its solution.

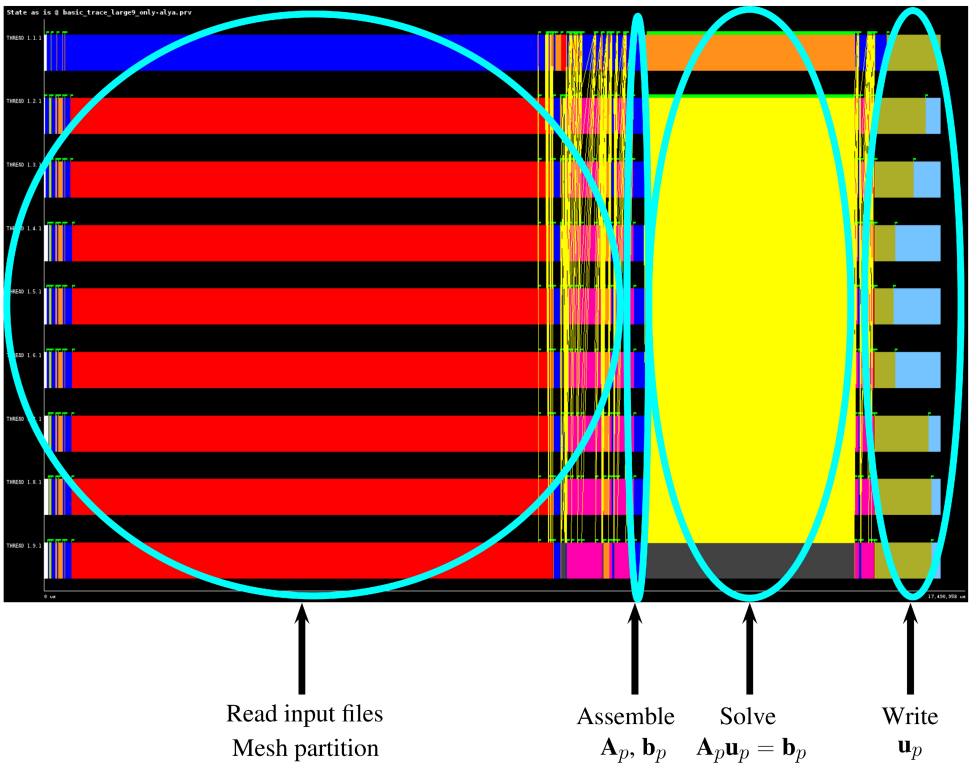

Figure 3.9: Alya's execution trace using 9 processes (1 master + 8 slaves). Paraver view *State as is* with yellow lines as MPI messages, and red and blue colors as idle and running states of the processes

The parallelization strategy allows Alya to have an almost linear scalability for several problems (see figure 3.10). This means that the amount of execution time is reduced almost in the same proportion as the number of processes used to solve the problem (if p processes are running, the execution time is p times faster).

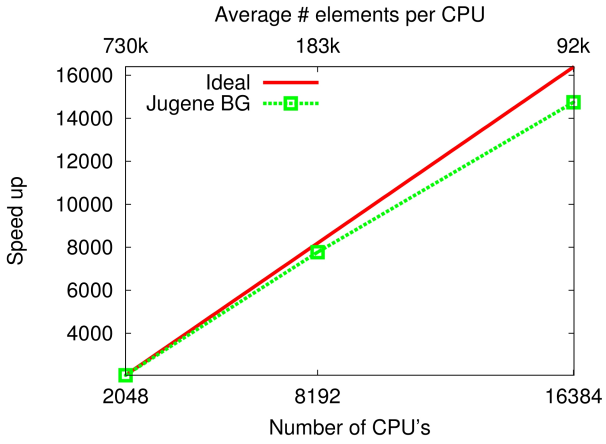

Figure 3.10: Benchmark results in a mesh of 1.6 billion of tetrahedra, running Alya to solve an incompressible flow problem on an complex aneurism geometry using the computational resources of supercomputer Jugene BlueGene/P from Jülich Supercomputing Centre [29]

Finally, concerning our purposes, we are only interested in the aspects relative to the sparse matrix-vector operation routines, because we will have to modify them in order to adapt the iterative solvers to their transposed versions, solving $\mathbf{A}^T\mathbf{u} = \mathbf{c}$.

3.2 Reduced gradient calculation

In this section we will give details about the implementation of the reduced gradient described in algorithm 4, using Alya's linear stationary PDE resolution with Dirichlet boundary conditions as basic scenario. The UML diagrams depicted in the previous section will serve us as base for our development and to easily explain our contributions.

Algorithm 4 modified for the linear stationarity behaviour is detailed in algorithm 7.

Input: $\mathbf{d}$ design vector

1 $\mathbf{u} \leftarrow$ solve $\mathbf{A}(\mathbf{d})\mathbf{u} = \mathbf{b}(\mathbf{d})$; (*forward problem*)
2 Calculate explicit derivatives $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d}) = \mathbf{c}(\mathbf{u}, \mathbf{d})^T$ and $\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{A}(\mathbf{d})$;
3 $\boldsymbol{\lambda} \leftarrow$ solve $\mathbf{A}(\mathbf{d})^T\boldsymbol{\lambda} = \mathbf{c}(\mathbf{u}, \mathbf{d})$; (*adjoint problem*)
4 Calculate explicit derivatives $\nabla_{\mathbf{d}} J(\mathbf{u}, \mathbf{d})$ and $\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}, \mathbf{d})$;
5 $\nabla_{\mathbf{d}} j(\mathbf{d}) = -\boldsymbol{\lambda}^T \nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}, \mathbf{d}) + \nabla_{\mathbf{d}} J(\mathbf{u}, \mathbf{d})$;

Output: $\nabla_{\mathbf{d}} j(\mathbf{d})$

Algorithm 7: Reduced gradient calculation using adjoint sensitivity for the linear stationary PDE case

As we reviewed in the previous section, we will use ideas from the assembling procedure and linear system resolution. The routines that are in charge of these procedures are three: `physiscs_elmope()`, `physics_dirbcs()` and `solve()`, with the first two implemented in the specific physical modules, and the third implemented in the kernel layer, working as an interface for several iterative solvers. All of these routines are managed by the physical module routine `physics_solite()` which delivers the linear system solution $\mathbf{u}$ stored into the array `unkno`. This can be viewed as the implementation of the forward problem, described in step 1 from algorithm 7.

Our proposed implementation uses `physics_solite()` as base, depicted in figure 3.3, adding the steps 2-5 from algorithm 7. The sequence diagram of the modified version of this routine can be viewed in figure 3.11.

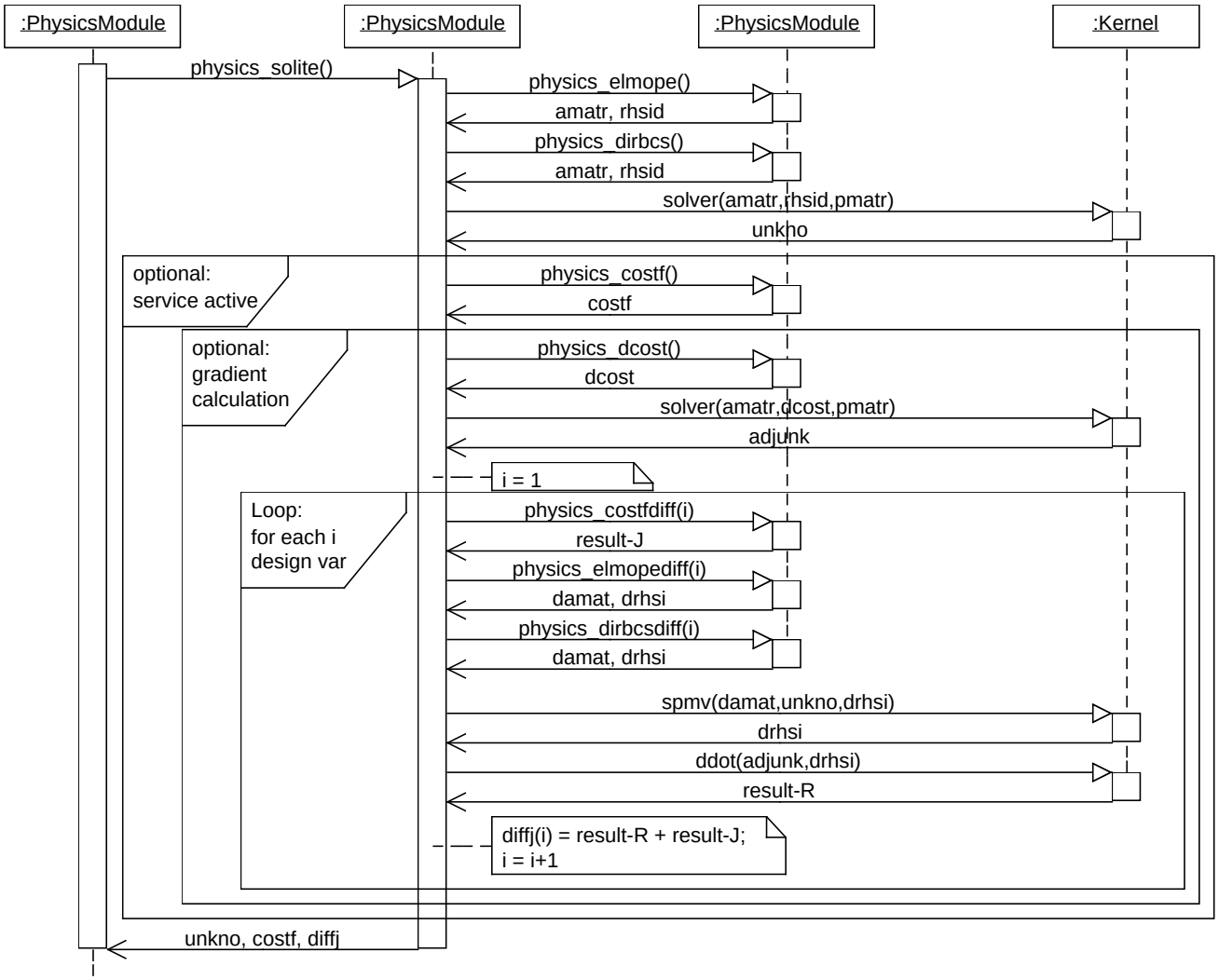

Figure 3.11: Sequence diagram of the modified routine `physics_solite()`, with reduced gradient calculation added

The description of each step of this diagram is as follows:

- The first three steps are almost the same as in the original routine `physics_solite()`, with the only difference that a special treatment of the design variables must be done in `physics_elmope()`. This will be explained in further subsections.
- An optional block with the legend *optional: service active* is depicted next. A binary flag determines if the optimization service is active in Alya's configuration files. If it is active, we can proceed to the next steps of the routine.
- If the service is active, a new routine denoted `physics_costf()` is launched. This routine calculates the cost function value $J(\mathbf{u}, \mathbf{d})$ and stores it in the scalar variable `costf`. Technical details of its implementation will be given in further subsections.
- Next we can see another optional block with the legend *optional: gradient calculation*. In line-search strategies, typically we don't need to calculate the value of the reduced gradient in each evaluation of the cost function. For this reason, another binary flag is used to determine if the current execution corresponds to a pure line-search evaluation or if it corresponds to a gradient calculation execution. Inside of this block, we can see a new routine denoted `physics_dcost()` which assembles $\nabla_{\mathbf{u}} J(\mathbf{u}, \mathbf{d})$, denoted by $\mathbf{c}(\mathbf{u}, \mathbf{d})$, and stores it in the array `dcost`, with the same dimensions of the array `rhsid`.
- After that, a call to the kernel routine `solver(...)` is performed using as parameters `amatr`, `dcost` and `pmatr`. In this step we need to solve the adjoint problem of line 3 from algorithm 7. Now the right-hand side is the array $\mathbf{c}(\mathbf{u}, \mathbf{d})$ stored in `dcost` and the matrix corresponds to the transpose of $\mathbf{A}(\mathbf{d})$. The result $\boldsymbol{\lambda}$, known as adjoint vector, is stored in the array `adjunk`, with the same dimensions of the array `unkno`. Implementation details of this step will be given in further subsections.
- The next step is another block that corresponds to a loop through the design vector entries, or design variables. For each design variable d_i, we need to calculate $\frac{\partial j(\mathbf{d})}{\partial d_i}$ using the formula from line 5 of algorithm 7. We can see that 4 sub-steps are needed to obtain this value:
  - First, we need to calculate $\frac{\partial J(\mathbf{u},\mathbf{d})}{\partial d_i}$. This task is performed by the new routine `physics_costfdiff(i)` which is a modified version of the routine `physics_costf()`. The result is stored in a scalar variable denoted `result-J`. Details about this routine will be discussed in further subsections.
  - Second, we need to calculate $\frac{\partial \mathbf{R}(\mathbf{u},\mathbf{d})}{\partial d_i}$ which corresponds to

$$\frac{\partial \mathbf{R}(\mathbf{u}, \mathbf{d})}{\partial d_i} = \left(\frac{\partial}{\partial d_i} \mathbf{A}(\mathbf{d}) \right) \mathbf{u} - \frac{\partial}{\partial d_i} \mathbf{b}(\mathbf{d}) \tag{3.2.1}$$

In order to obtain this value, we need to assemble $\frac{\partial}{\partial d_i}\mathbf{A}(\mathbf{d})$ and $\frac{\partial}{\partial d_i}\mathbf{b}(\mathbf{d})$, and store them in the arrays `damat` and `drhsi` respectively. These tasks are performed by the new physics module routines `physics_elmopediff(i)` and `physics_dirbcsdiff(i)`, which correspond to a modified version of the assembling routines `physics_elmope()` and `physics_dirbcs()`. The modification involves the application of the derivative $\frac{\partial}{\partial d_i}$ directly in those codes, making the necessary transformations manually. Details about this topic will be discussed in further subsections.

- Using $\frac{\partial}{\partial d_i}\mathbf{A}(\mathbf{d})$ and $\frac{\partial}{\partial d_i}\mathbf{b}(\mathbf{d})$, a matrix-vector product and vectorial sum are performed to obtain $\frac{\partial \mathbf{R}(\mathbf{u},\mathbf{d})}{\partial d_i}$ from equation (3.2.1). This task is executed by the kernel routine `spmv(...)`. The array `drhsi` will be re-used to store the result of this operation.
- Finally, a dot-product operation is performed between $\boldsymbol{\lambda}$ and $\frac{\partial \mathbf{R}(\mathbf{u},\mathbf{d})}{\partial d_i}$ through the kernel routine `ddot(...)`. The result, stored in the scalar `result-R`, is added to `result-J`, and stored in the i-th entry of the array `diffj`.

3.2.1 Design variable update

The design vector $\mathbf{d} \in \mathbf{R}^{n_d}$ is stored into the global array `design`. The design space size n_d is specified in the configuration files. This array has the same value in each process and it will be used in the optimization update steps described in section 3.3.

In the first optimization step, when the first reduced gradient is being calculated, a link between the user-defined physical/geometrical design variables and the array `design` must be performed.

For example, let us assume that the design variable corresponds to the electric conductivity σ of each nodal point in the domain Ω. In the configuration files the user must specify the size of the design space n_d (in this example the design space size is the same as the state space size, i.e. the total nodal points in Ω) and the initial electric conductivity values σ_0. After the configuration file reading, the electric conductivity values σ_0 are stored into a physical module variable, in this case, we will denote this variable as `sigma_phys`. The size of this variable is different in each process because different number of nodal points can reside in different sub-domains.

The key step to link the variables `sigma_phys` and `design` is included at the beginning of the routine `physics_elmope()`, as depicted in the code of the next paragraph. In this code we can observe the link in the lines 5-13. If the first iteration is running, we store the initial values of σ_0 into `design` using a local-to-global map for nodal point indexes denoted `lninv_loc`. This map is necessary because the size of `sigma_phys`, denoted `npoin` (number of nodal points in each sub-domain) is different in each process and the size of `design` is the same for each process (total number of nodal points). If any other iteration is running, we need to recover the updated design variables from the

last iteration, stored in `design`, and assign them to the physical variable `sigma_phys`, in order to run the subsequent procedures.

```
subroutine physics_elmope()
   ! params and variables
   ! ...

   if(first_iteration) then
      do i=1,npoin
         design(lninv_loc(i)) = sigma_phys(i)
      end do
   else ! other iteration
      do i=1,npoin
         sigma_phys(i) = design(lninv_loc(i))
      end do
   end if

   ! code
   ! ...
end subroutine physics_elmope
```

It is important to remark that the size of the array `design` is passed as input in the configuration files, and it doesn't depend on any physical parameter. The user must take into account the size of the design space depending on the specific problem that must be solved, and the intricacies of its own routine implementation. If design variables are included into some boundary condition, `physics_dirbcs()` must be modified with the proposed update.

3.2.2 Cost function evaluation

The routine `physics_costf()` calculates the cost function value $J(\mathbf{u}, \mathbf{d})$ specified by the user. Going back to the continuous definition of a PDECO problem, from equation (1.2.1), we can assume that the continuous cost functional can be expressed as an integral of some *proper* function f over the domain as

$$\mathcal{J}(u, d) = \int_{\Omega} f(u, d) dV \tag{3.2.2}$$

The function f must be differentiable w.r.t. u and d, and also it must be locally bounded. This roughly means that its integral over each element defined in a domain discretization is finite and can be approximated using quadrature points associated to those elements. If we use a partitioned discrete domain $\Omega_h = \bigcup_{p=1}^{n_{procs}} \Omega_p$, with n_{procs} the number of slaves involved in the parallel execution, the corresponding discrete cost function can be approximated as

$$J(\mathbf{u}, \mathbf{d}) = \sum_{p=1}^{n_{procs}} \sum_{e \in \Omega_p} \mathbf{f}_e(\mathbf{u}, \mathbf{d}) \tag{3.2.3}$$

For example, in the section 2.3.1, the cost functional is:

$$\mathcal{J}(u,d) = \frac{1}{2}\int_{\Omega}(u-u^*)^2 dV \tag{3.2.4}$$

In this case we have $f(u,d) = \frac{1}{2}(u-u^*)^2$ with its finite element discrete version

$$\mathbf{f}_e(\mathbf{u},\mathbf{d}) = \frac{1}{2}\sum_{i=1}^{n_{nodes(e)}}\sum_{j=1}^{n_{quad(i)}} w_{i,j}(u_i - u_i^*)^2 \tag{3.2.5}$$

where $n_{node(e)}$ and $n_{quad(i)}$ are the number of element nodes and the number of quadrature points associated to each node, $w_{i,j}$ are weights associated to test functions values in the quadrature points and element volume terms, and u_i and u_i^* are nodal values of $\mathbf{u}$ and the observed data $\mathbf{u}^*$.

According to this, each slave p can calculate the discrete version of $\sum_{e\in\Omega_p}\mathbf{f}_e(\mathbf{u},\mathbf{d})$ and then an `MPI_Allreduce` operation must be performed. In order to calculate the discretized integral over each sub-domain, the assembling routine `physics_elmope()` can be used as basis, because it provides an easy way to accumulate a value through all elements. A general framework to perform the integral calculation is depicted in figure 3.12. The steps in this diagram are as follows:

- A scalar variable `costf` accumulates the values $\mathbf{f}_e(\mathbf{u},\mathbf{d})$ through the elements in the sub-domain Ω_p, managed by each slave p.
- Local data is extracted from the element e, in the same way that is performed in the routine `physics_elmope()`.
- After that, local element values of $\mathbf{u}$ are extracted from the array `unkno` using the local-global map `ipoin = lnods(inode)`.
- In the same way as `physics_elmope()`, using local information, the physics module launches the kernel routine denoted `elmcar(local_data)` which calculates the volume, cartesian derivatives and Hessian matrix associated to this element.
- With all the previous local information, the local contribution to the cost function, denoted $\mathbf{f}_e(\mathbf{u},\mathbf{d})$, is calculated and stored in the scalar variable `f_e`. Immediately after its calculation, the accumulation in `costf` proceeds.
- The last step in this routine is the execution of the MPI routine `MPI_Allreduce` in order to add the contributions in each slave to the variable `costf`. This routine is managed by the service *Parall*.

Some considerations to take into account in the implementation of this routine:

- The continuous cost functional must be sufficiently smooth, at least $\mathcal{C}^1$ (continuous and differentiable) in both variables, state and design vectors. Also, it is recommended that it will be a convex or strictly-convex function, in order to guarantee the existence of a global optimal value. If this last condition is not possible, it is still possible to find local optimal values, but a posteriori interpretation of the result will be needed.

- It is not recommended to access external information sources during the cost function evaluation, for example, extract database values for the design vector. This could downgrade the gradient-based optimization to a local search (from level-1 or higher, to level-0, using the taxonomy of section 2.1.1).

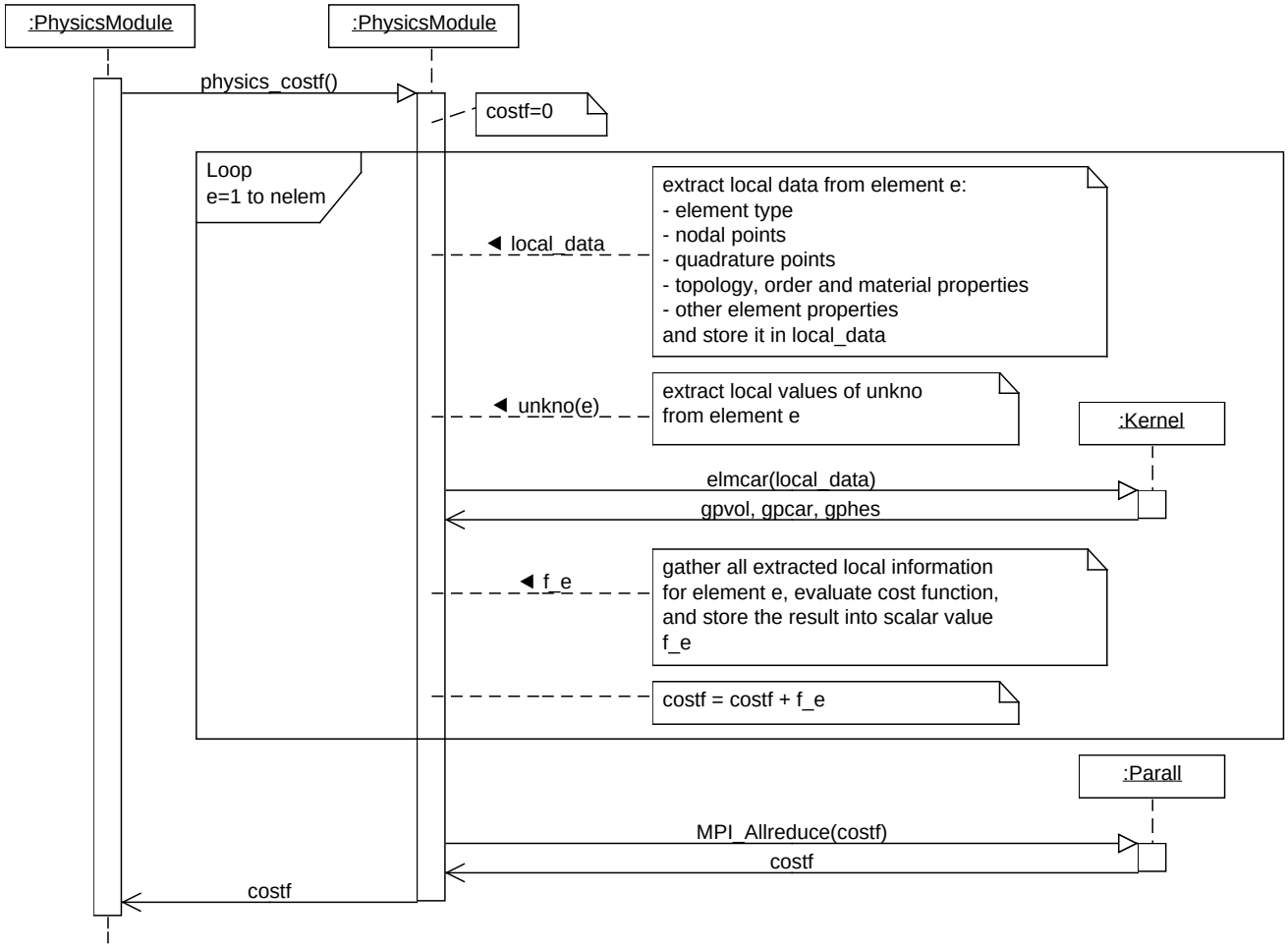

Figure 3.12: Sequence diagram of the new routine `physics_costf()`

3.2.3 Cost function differentiation w.r.t. state vector

Once the evaluation of $J(\mathbf{u}, \mathbf{d})$ is implemented in the routine `physics_costf()`, we need to calculate its gradient w.r.t. state vector $\mathbf{u}$. Using the cost function as defined in equation (3.2.3), we can take the derivative w.r.t. a nodal value u_i:

$$\begin{aligned} \frac{\partial}{\partial u_i} J(\mathbf{u}, \mathbf{d}) &= \frac{\partial}{\partial u_i} \sum_{p=1}^{n_{procs}} \sum_{e \in \Omega_p} \mathbf{f}_e(\mathbf{u}, \mathbf{d}) \\ &= \sum_{p=1}^{n_{procs}} \sum_{e \in \Omega_p} \frac{\partial \mathbf{f}_e(\mathbf{u}, \mathbf{d})}{\partial u_i} \end{aligned} \tag{3.2.6}$$

The above expression indicates that each slave p is in charge of the calculation of $\frac{\partial \mathbf{f}_e(\mathbf{u},\mathbf{d})}{\partial u_i}$ for all nodes i which belong to the sub-domain Ω_p. For example, using the expression of equation (3.2.5), the derivatives calculated by slave p are:

$$\begin{aligned} \frac{\partial}{\partial u_i} J(\mathbf{u}, \mathbf{d}) &= \sum_{e \in \Omega_p} \frac{\partial}{\partial u_i} \left(\frac{1}{2} \sum_{k=1}^{n_{nodes(e)}} \sum_{j=1}^{n_{quad(k)}} w_{k,j} (u_k - u_k^*)^2 \right) \\ &= \sum_{\{e \in \Omega_p :\, node(i) \in e\}} \frac{\partial}{\partial u_i} \left(\frac{1}{2} \sum_{k=1}^{n_{nodes(e)}} \sum_{j=1}^{n_{quad(k)}} w_{k,j} (u_k - u_k^*)^2 \right) \\ &= \sum_{\{e \in \Omega_p :\, node(i) \in e\}} \sum_{j=1}^{n_{quad(i)}} w_{i,j} (u_i - u_i^*) \end{aligned} \tag{3.2.7}$$

The set $\{e \in \Omega_p : \, node(i) \in e\}$ denotes the elements e of Ω_p that have some nodal point corresponding to $node(i)$.

Using the code of routines `physics_costf()` and `physics_elmope()` (specially right-hand side assembling) as base, we can assemble the array `dcost` which stores the values $\frac{\partial \mathbf{f}_e(\mathbf{u},\mathbf{d})}{\partial u_i}$ for each nodal point indexed by i. Details of this procedure, applied to all sub-domain nodes, are depicted in the sequence diagram of figure 3.13.

In this diagram, we can see the following steps:

- A vector variable `dcost` accumulates the values $\frac{\partial \mathbf{f}_e(\mathbf{u},\mathbf{d})}{\partial u_i}$ through the elements in the sub-domain Ω_p, managed by each slave p.
- Local data is extracted from the element e, in the same way that is performed in the routine `physics_elmope()`.
- After that, local element values of $\mathbf{u}$ are extracted from the array `unkno` using the local-global map `ipoin = lnods(inode)`.

- In the same way as `physics_elmope()`, using local information, the physics module launches the kernel routine denoted `elmcar(local_data)` which calculates the volume, cartesian derivatives and Hessian matrix associated to this element.
- An internal loop is performed through the nodal points i of element e.
- With all the previous local information, for each nodal point i the local contribution to the cost function derivative, denoted $\frac{\partial \mathbf{f}_e(\mathbf{u},\mathbf{d})}{\partial u_i}$, is calculated and stored in the scalar variable `df_e`. Immediately after its calculation, the accumulation in `dcost` proceeds, using the global index of the node i.

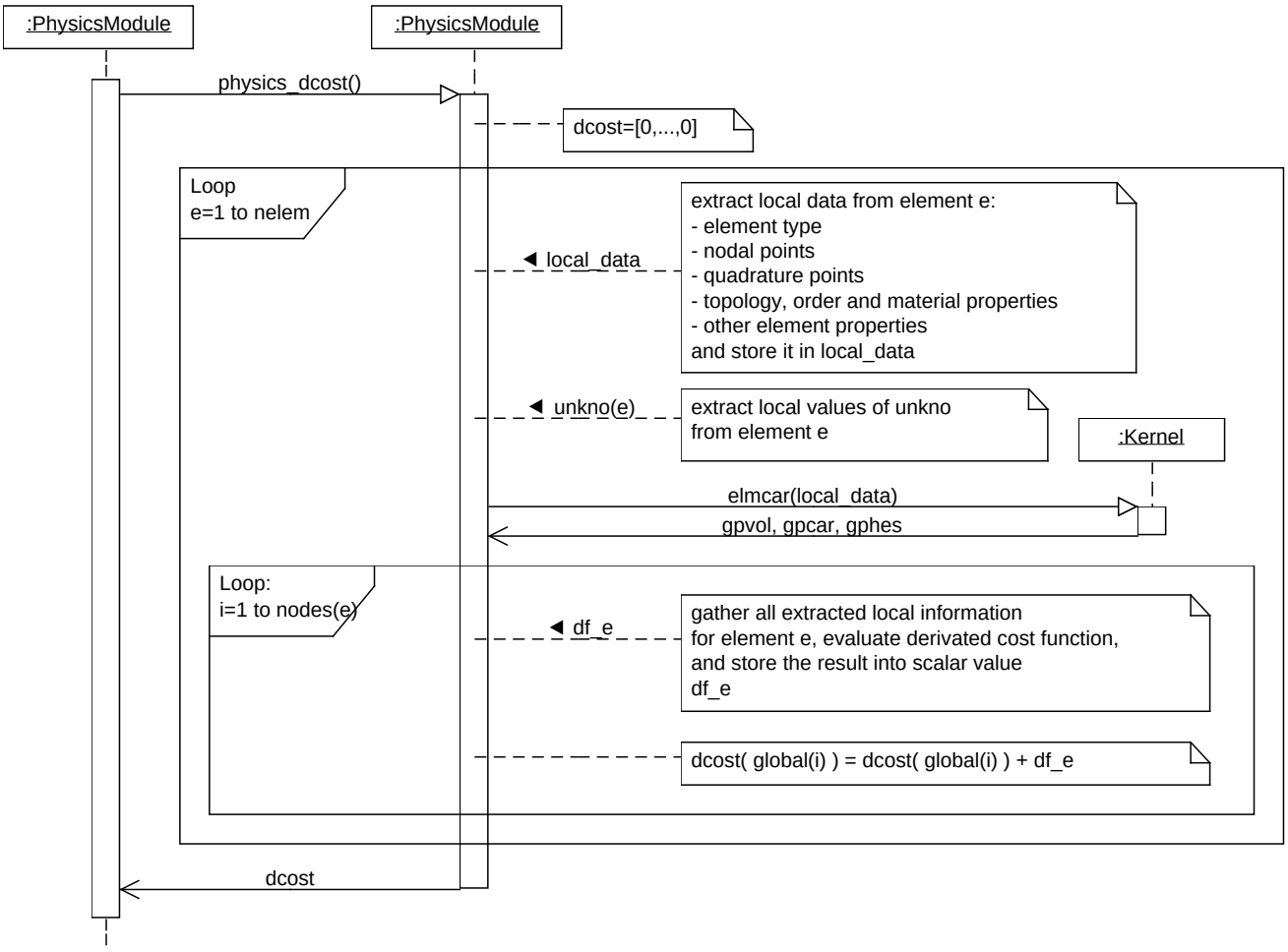

Figure 3.13: Sequence diagram of the new routine `physics_dcost()`

3.2.4 Adjoint problem resolution

In the adjoint problem, the right-hand side corresponds to the array `dcost`, described in the previous subsection, and the matrix corresponds to $\mathbf{A}(\mathbf{d})^T$, where $\mathbf{A}(\mathbf{d})$ is stored in BCRS format in the arrays `amatr`, `c_dom` and `r_dom`. The result $\boldsymbol{\lambda}$, known as adjoint vector, is stored in the array `adjunk`, with the same dimensions of the array `unkno`.

As we explained in section 3.1.2, `solver(...)` routine acts as an interface for several iterative solvers implemented in Alya, which are based in matrix-vector operations using the BCRS format as described in algorithm 6. In order to re-use those solvers in the adjoint problem, we need to replace the matrix-vector operations $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ into $\mathbf{y} \leftarrow \mathbf{A}^T\mathbf{x}$ in the iterative solvers that will be used.

According to [10], the modified versions of the matrix-vector products described in algorithms 5 and 6, can be viewed in algorithms 8 and 9. We can see that the main modifications involve the exchange of indexes between vectors $\mathbf{x}$ and $\mathbf{y}$. Our implementation follows algorithm 9, modifying the existing kernel routine `spmv(...)` which implements algorithm 6. Each iterative solver can use different versions of the matrix-vector products. There may be some physical modules that use 1×1 blocks, recovering the CRS format and using the standard algorithms 5 and 8, or others that use larger block sizes. For this reason, several implementations of the matrix-vector products will be necessary in order to cover all possible iterative solver choices.

Regarding the performance, both direct and transpose matrix-vector CRS products of algorithms 5 and 8 have largely the same structure, and both use indirect addressing, through an extra array access to `c_dom` in order to obtain $\mathbf{x}[\texttt{c\_dom}[j]]$ or $\mathbf{y}[\texttt{c\_dom}[i]]$. Hence, their vectorizability is limited, and even not possible, due to their non-contiguous memory accesses produced by the indirect addressing. However, the direct product $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ has a more favorable memory access pattern (per iteration of the outer loop), it reads two vectors of data (a row of matrix $\mathbf{A}$ and the input vector $\mathbf{x}$) and writes one scalar. The transposed product $\mathbf{y} \leftarrow \mathbf{A}^T\mathbf{x}$ on the other hand reads one element of the input vector $\mathbf{x}$, one row of matrix $\mathbf{A}$, and both reads and writes the result vector $\mathbf{y}$. This loss of performance is also present in the matrix-vector BCRS products of algorithms 6 and 9, multiplying the reads and writes described before by the block size $r_b \times c_b$.

In order to see where we need to modify the iterative solvers, we show the conjugate gradient method as sample. The steps of this methods can be viewed in algorithm 10. In lines 1, 5 and 7, we can see that a matrix-vector product is needed. In algorithm 11 we can see the modified method using transposed matrix-vector operations in lines 1, 5 and 7. All the other steps of the transposed algorithm are the same as in the original version.

Input: matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ stored in (`amatr`, `c_dom`, `r_dom`), vectors $\mathbf{x}$ and $\mathbf{y}$

```
1  for i = 1 to n do
2  |  y[i] ← 0;
3  end
4  for j = 1 to n do
5  |  for i = r_dom[j] to r_dom[j + 1] − 1 do
6  |  |  y[c_dom[i]] ← y[c_dom[i]] + amatr[i] ∗ x[j];
7  |  end
8  end
```

Output: $\mathbf{y} \leftarrow \mathbf{A}^T\mathbf{x}$

Algorithm 8: Procedure for computing $\mathbf{y} = \mathbf{A}^T\mathbf{x}$, where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is stored using CRS format

Input: matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ stored in (`amatr`, `c_dom`, `r_dom`), vectors $\mathbf{x}$ and $\mathbf{y}$

```
1  z ← 0;
2  n_b ← n/r_b;
3  for i = 1 to n do
4  |  y[i] ← 0;
5  end
6  for b = 1 to n_b do
7  |  for i = r_dom[b] to r_dom[b + 1] − 1 do
8  |  |  for k = 1 to r_b do
9  |  |  |  for t = 1 to c_b do
10 |  |  |  |  y[c_b ∗ (c_dom[i] − 1) + t] ←
   |  |  |  |  y[c_b ∗ (c_dom[i] − 1) + t] + amatr[z] ∗ x[r_b ∗ (b − 1) + k];
11 |  |  |  |  z + +;
12 |  |  |  end
13 |  |  end
14 |  end
15 end
```

Output: $\mathbf{y} \leftarrow \mathbf{A}^T\mathbf{x}$

Algorithm 9: Procedure for computing $\mathbf{y} = \mathbf{A}^T\mathbf{x}$, where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is stored using BCRS format with block size $r_b \times c_b$

Input: matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, initial vector $\mathbf{x}^0$ and right-hand side $\mathbf{b}$

$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}_0$;
$\mathbf{p}_0 \leftarrow \mathbf{r}_0$;
$k \leftarrow 0$;
repeat
  $\alpha_k \leftarrow \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$;
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$;
  $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$;
  if r_{k+1} *is sufficiently small* **then**
    Tolerance reached and solution found. Exit;
  end
  $\beta_k \leftarrow \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$;
  $\mathbf{p}_{k+1} \leftarrow \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$;
  $k \leftarrow k + 1$;
until *tolerance not reached*;

Output: $\mathbf{x}^k$ solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$

Algorithm 10: Conjugate gradient iterative method to solve $\mathbf{A}\mathbf{x} = \mathbf{b}$

Input: matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, initial vector $\mathbf{x}^0$ and right-hand side $\mathbf{b}$

$\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{A}^T\mathbf{x}_0$;
$\mathbf{p}_0 \leftarrow \mathbf{r}_0$;
$k \leftarrow 0$;
repeat
  $\alpha_k \leftarrow \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}^T\mathbf{p}_k}$;
  $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{p}_k$;
  $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha_k \mathbf{A}^T\mathbf{p}_k$;
  if r_{k+1} *is sufficiently small* **then**
    Tolerance reached and solution found. Exit;
  end
  $\beta_k \leftarrow \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$;
  $\mathbf{p}_{k+1} \leftarrow \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$;
  $k \leftarrow k + 1$;
until *tolerance not reached*;

Output: $\mathbf{x}^k$ solution of $\mathbf{A}^T\mathbf{x} = \mathbf{b}$

Algorithm 11: Conjugate gradient iterative method to solve $\mathbf{A}^T\mathbf{x} = \mathbf{b}$

3.2.5 Cost and constraint function differentiation w.r.t. design vector

The differentiation of $J(\mathbf{u},\mathbf{d})$ and $\mathbf{R}(\mathbf{u},\mathbf{d})$ with respect to a design variable d_i involves the development of codes that compute the derivatives $\frac{\partial J(\mathbf{u},\mathbf{d})}{\partial d_i}$ and $\frac{\partial \mathbf{R}(\mathbf{u},\mathbf{d})}{\partial d_i}$, using `physics_costf()`, `physics_elmope()` and `physics_dirbcs()` as base of the derivative codes. According to [31, 26], four possible options are available in order to obtain the desired codes:

- *Finite difference approximation*: this approach suffers from the fact that the values being computed are approximations, not true derivatives. If the step-size used for the divided differences is too large or too small, the approximations can be grossly inaccurate. Furthermore it is difficult to assess the accuracy of the approximation a priori.
- *Developing derivative code by hand*: this approach provides efficient and accurate derivatives but can be tedious, error-prone, and extremely time-consuming if the programmer doesn't have prior knowledge of the original code.
- *Symbolic differentiation*: symbolic manipulators, such as Mathematica [4] or Matlab [5] can compute derivatives directly or generate a program to compute them. However, traditional symbolic manipulation is memory intensive and is often not feasible for large functions or functions best expressed as procedures with many loops and branches.
- *Automatic differentiation (AD)*: this approach produces derivative code that computes accurate (within the limits of finite precision arithmetic) derivatives, and can be applied to arbitrarily complex programs with minimal effort on the part of the programmer. Using advanced techniques, the performance of AD can rival or exceed that of code developed by hand.

Among these options, AD is the one with the largest practical and theoretical background and it also holds a robust community of tool developers and researchers. Sequential codes can be easily adapted to be scanned by an AD tool, in order to generate derivative codes. However, when dealing with parallel codes, the adaptation process is not straightforward. In Alya we can find several global scalar and vector variables, with different sizes and values depending on the slave that uses them. In some cases it could be necessary to study the nature of the design variables in the physics module, because they can be defined only in a few sub-domains, or they can also be defined in some boundaries belonging to one or more sub-domains. Further study on this kind of issues is needed (some experiences have been carried in [31]), but this is out of the scope of our work.

In order to avoid this problems, we choose to obtain the derivative codes *by hand*, using our knowledge gained studying Alya's structure and behaviour.

Methodology

To fix ideas, in the next paragraphs we will describe our methodology to produce derivative codes by hand in Alya. This methodology is strongly influenced by ideas extracted from AD, code optimization and parsing topics from compiler design [6]. For each design variable, we need to apply three steps in order to modify the current code:

- identify the scope of the output variables for each slave,
- find the path that will be affected by the derivative w.r.t. d_i,
- modify the scope applying the derivative $\frac{\partial}{\partial d_i}$ in the calculated path.

In order to understand these steps, we present an example with the parallel routine `foo` (with MPI as programming model), with 1000 design variables stored in the array `design`,

$$\mathbf{d} = (\texttt{design(1)}, \ldots, \texttt{design(1000)}) \in \mathbb{R}^{1000}$$

In the following, we will use indistinctly d_i or `design(i)`.

```
1  subroutine foo( in_x, in_y, in_z, design, out_w )
2     real, intent(in)  :: in_x, in_y, in_z, design(1000)
3     real, intent(out) :: out_w(npoin)
4     real              :: tmp
5     tmp = x_in
6     if(idslave==2) then
7        do i=1,1000
8           tmp = tmp + sqrt(design(i))
9        end do
10    end if
11    do i=1,npoin
12       out_w(i) = tmp*in_x + in_y*in_y + in_z*in_z*design(10)
13    end do
14 end subroutine foo
```

In this code we can see two global variables: `idslave` and `npoin` which have different values in each slave execution. The variable `idslave` represents the slave identifier and `npoin` represents the size of the array `out_w` in each slave. We will assume that all slaves have the same values in the array `design` and scalars `in_x`, `in_y` and `in_z`, and also that these scalars doesn't have previous dependency on any design variable.

Mathematically, the routine `foo` represents a function $\texttt{foo} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{1000} \to \mathbb{R}^{\texttt{npoin}}$ where `npoin` depends on the slave which is executing it:

$$\texttt{foo}\begin{pmatrix} \texttt{in\_x} \\ \texttt{in\_y} \\ \texttt{in\_z} \\ \texttt{design(1)} \\ \vdots \\ \texttt{design(1000)} \end{pmatrix} = \begin{pmatrix} \texttt{out\_w(1)} \\ \vdots \\ \texttt{out\_w(npoin)} \end{pmatrix}$$

Applying the derivative $\frac{\partial}{\partial d_i}$ the resulting vector is as follows:

$$\frac{\partial}{\partial d_i}\texttt{foo} = \begin{pmatrix} \frac{\partial}{\partial d_i}\texttt{out\_w(1)} \\ \vdots \\ \frac{\partial}{\partial d_i}\texttt{out\_w(npoin)} \end{pmatrix}$$

And the complete derivative w.r.t. design vector $\mathbf{d}$:

$$\nabla_{\mathbf{d}}\texttt{foo} = \left[\begin{array}{c|c|c} \frac{\partial}{\partial d_1}\texttt{out\_w(1)} & \dots & \frac{\partial}{\partial d_{1000}}\texttt{out\_w(1)} \\ \vdots & \dots & \vdots \\ \frac{\partial}{\partial d_1}\texttt{out\_w(npoin)} & \dots & \frac{\partial}{\partial d_{1000}}\texttt{out\_w(npoin)} \end{array}\right]$$

We can see that the derivative must be applied to each output variable, interpreted as a function of the design variables.

The three steps to obtain the derivative code that calculates $\frac{\partial}{\partial d_i}\texttt{foo}$ are as follows:

- The *first step* is to identify the scope of the output variables. A clean strategy to visualize this scope is to plot the computation graph of the output variables for each slave, in a bottom-up fashion. In figure 3.14 we can see the computation graph of variable `out_w(k)` for all slaves, using a fixed index `k`.
- The *second step* is the identification of the path that will be affected by the derivative w.r.t. d_i. In the computation graphs of figure 3.15, we can see the affected path for the design variable $d_i =$ `design(10)` in dashed lines. In this path, the terms that are multiplying some expression containing d_i are included, i.e. terms as $expr(d_i) * term$, for some expression $expr$. The additive terms that doesn't include any expression using d_i are not included. For example, the expression `in_z * in_z * design(10)` is included in the path, but the term `in_y* in_y` is not included. This process is repeated until we have reached the input variables (parameters of the routine or global variables).
- The *third step* is the modification of the scope using the differentiation w.r.t d_i. In figure 3.16 we can see the path affected by the derivative w.r.t. `design(10)`. For slave `idslave` $= 2$ we can see that the squared root operator $\sqrt{}$ is transformed into $\frac{1}{2\sqrt{}}$ and the term `in_z * in_z * design(10)` is transformed into `in_z * in_z * 1.0`.

With this three steps applied, the derivative code of routine `foo` with respect to the design variable `design(10)` is as follows:

```
1  subroutine foodiff( in_x, in_y, in_z, design, out_w)
2     real, intent(in)  :: in_x, in_y, in_z, design(1000)
3     real, intent(out) :: out_w(npoin)
4     real              :: tmp
5     tmp = x_in
6     if(idslave==2) then
7        tmp = tmp + 0.5*(1.0/sqrt(design(10)))
8     else
9        tmp = 0.0
10    end if
11    do i=1,npoin
12       out_w(i) = tmp*in_x + 0.0 + in_z*in_z*1.0
13    end do
14 end subroutine foodiff
```

Using an extra input `d_i`, which indicates the index of the design variable d_i that will be used as derivative parameter, the general derivative code is as follows:

```
1  subroutine foodiff( in_x, in_y, in_z, design, out_w, d_i)
2     real,    intent(in)  :: in_x, in_y, in_z, design(1000)
3     integer, intent(in)  :: d_i
4     real,    intent(out) :: out_w(npoin)
5     real                 :: tmp, tmp2
6     tmp = x_in
7     if(idslave==2) then
8        tmp = tmp + 0.5*(1.0/sqrt(design(indvar)))
9     else
10       tmp = 0.0
11    end if
12    if(d_i==10) then
13       tmp2 = in_z*in_z*1.0
14    else
15       tmp2 = 0.0
16    end
17    do i=1,npoin
18       out_w(i) = tmp*in_x + 0.0 + tmp2
19    end do
20 end subroutine foodiff
```

Mathematically the derivative calculated by this code is as follows:

$$\frac{\partial}{\partial d_i}\texttt{foo} \;=\; \begin{pmatrix} \texttt{tmp*in\_x} + \texttt{tmp2} \\ \vdots \\ \texttt{tmp*in\_x} + \texttt{tmp2} \end{pmatrix}$$

where `tmp` and `tmp2` take different values depending on the slave identifier or the design variable index.

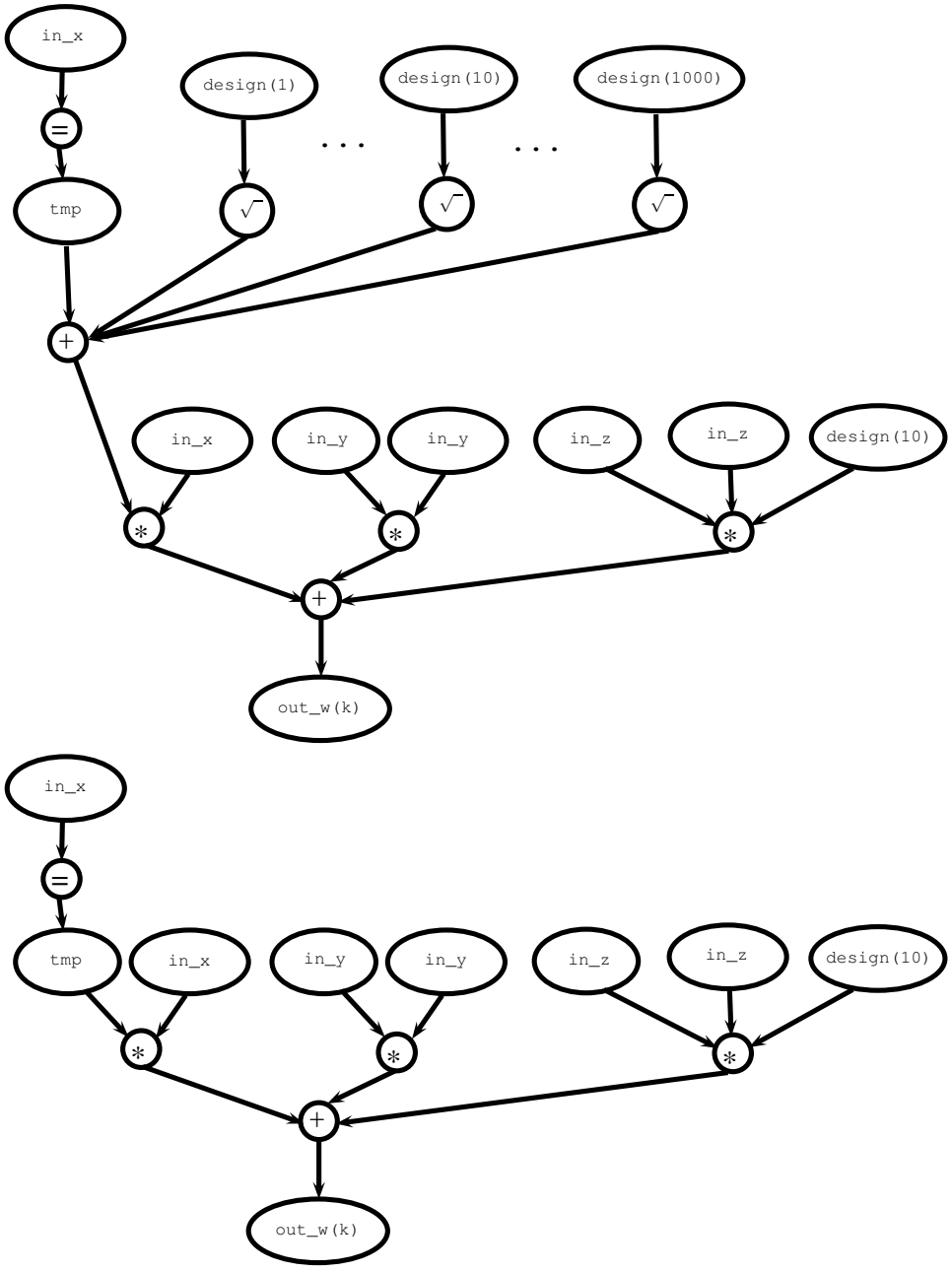

Figure 3.14: Computation graphs of variable `out_w(k)` using the example routine `foo`, for slaves `idslave` $= 2$ (top) and `idslave` $\neq 2$ (bottom)

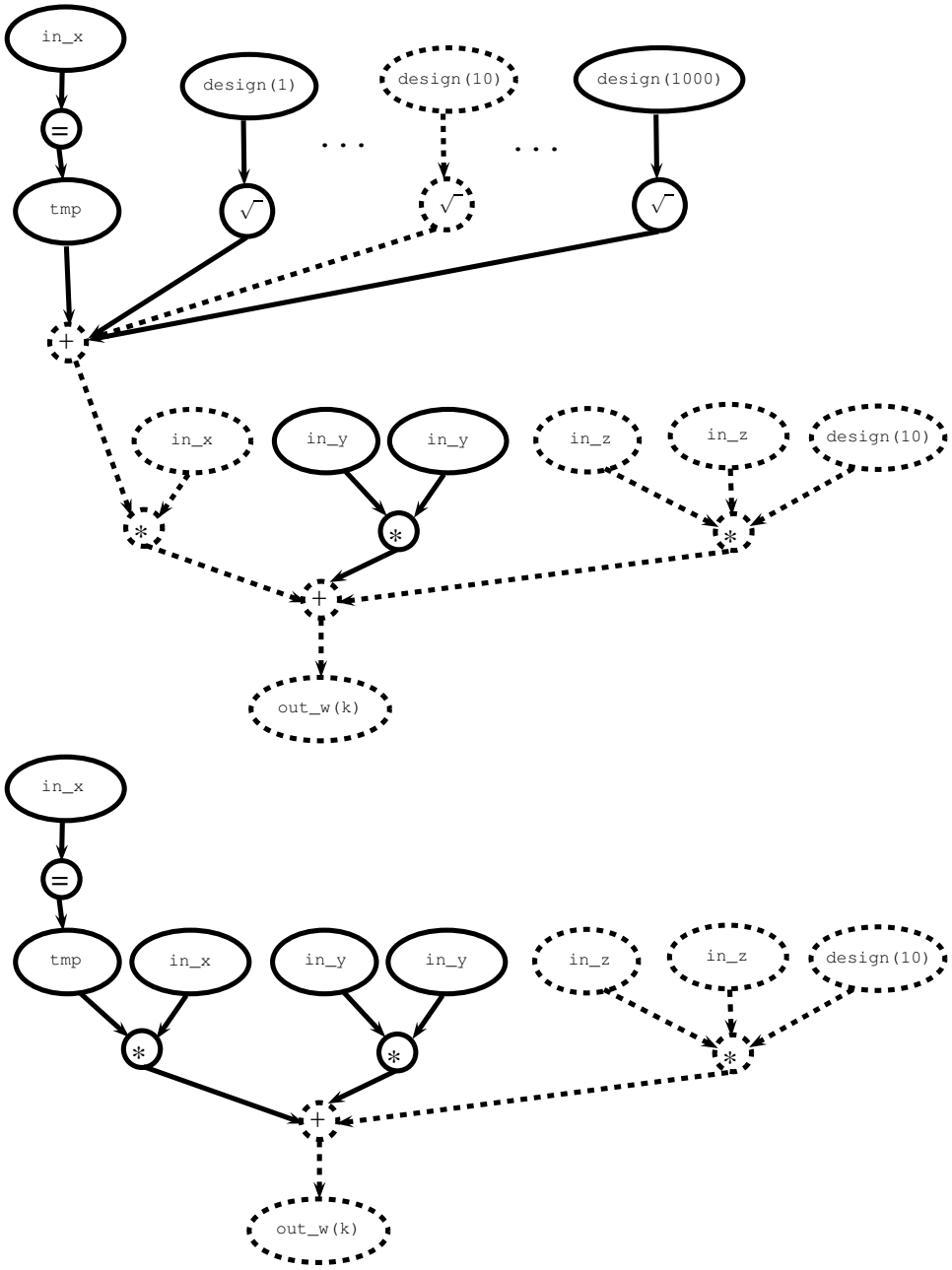

Figure 3.15: Affected path of variable `design(10)` (dashed lines), in the computation graphs of `out_w(k)` using the example routine `foo`, for slaves `idslave` $= 2$ (top) and `idslave` $\neq 2$ (bottom)

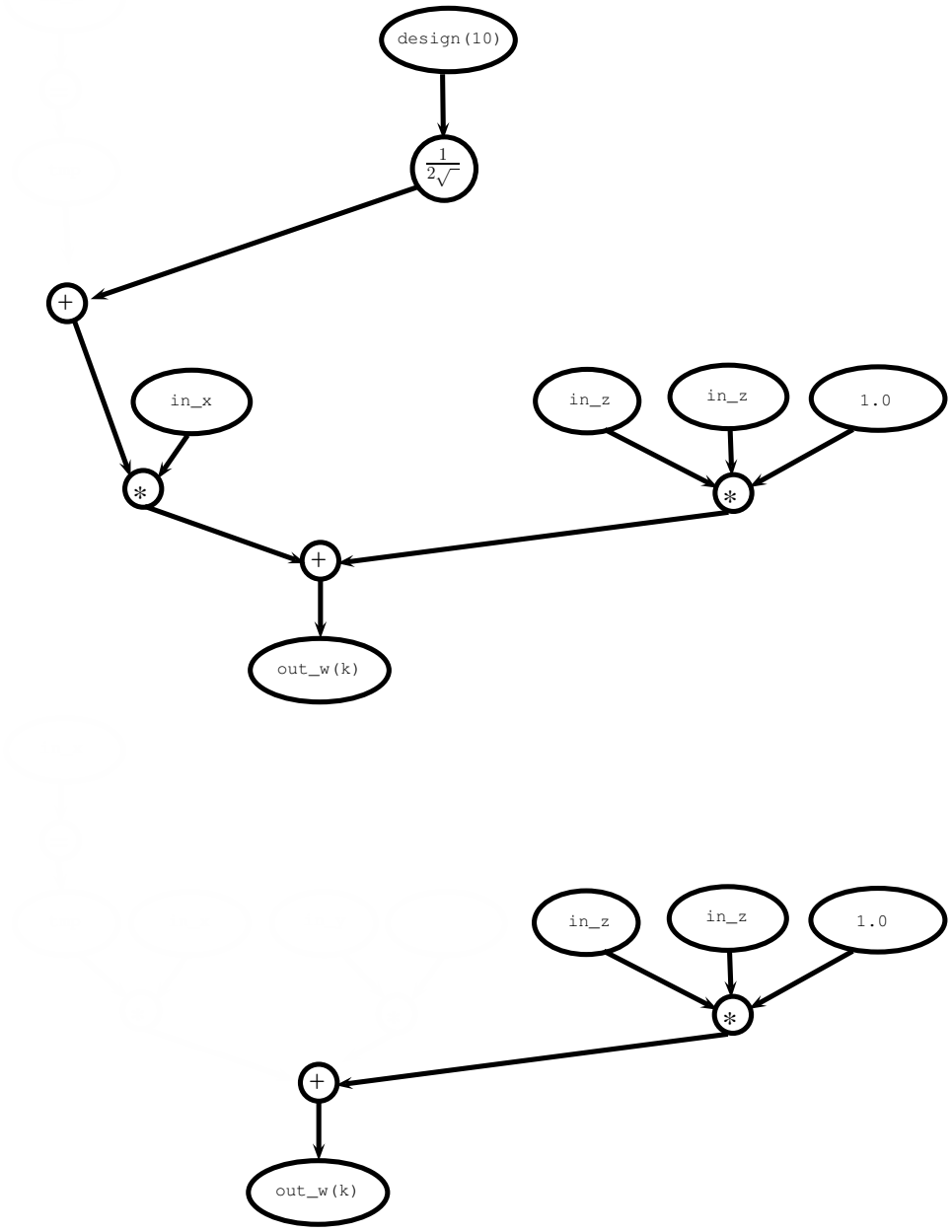

Figure 3.16: Differentiation w.r.t. d_i applied on computation graphs of variable `out_w(k)` using the example routine `foo`, for slaves `idslave` $=$ 2 (top) and `idslave` $\neq 2$ (bottom)

Cost function differentiation

Concerning the cost function $J(\mathbf{u}, \mathbf{d})$, the routine `physics_costf()` is differentiated by hand following the proposed methodology, resulting in a new routine denoted `physics_costfdiff(i)`, which calculates $\frac{\partial}{\partial d_i} J(\mathbf{u}, \mathbf{d})$, with `i` the design variable index. Each physics module can have several cost functions implemented, for this reason a detailed study using the proposed methodology is necessary in each routine. In some applications, we will never find design variables embedded into the cost function in an explicit way. For these applications the value returned by `physics_costfdiff(i)` will be always zero. However, there are other applications in which we can find regularization terms in the cost function. For example, in the following cost functional

$$\mathcal{J}(u, d) = \frac{1}{2} \int_{\Omega} (u - u^*)^2 dV + \frac{\beta}{2} \|d - d^{ref}\|^2 \tag{3.2.8}$$

the second term is usually called the *regularization term*, and is used to incorporate external information d^{ref} about the objective value for the design variables. This external information, or *reference model*, may accelerate the convergence of the optimization process, through the calculation of *better* gradients and descent directions. In our work, we only consider cost functions without design variables embedded, so the differentiated routine will return zero in any case.

Constraint function differentiation

As we mentioned in section 3.2, the differentiated constraint function $\frac{\partial}{\partial d_i}\mathbf{R}(\mathbf{u}, \mathbf{d})$ requires several steps, being the assembling of $\frac{\partial}{\partial d_i}\mathbf{A}(\mathbf{d})$ and $\frac{\partial}{\partial d_i}\mathbf{b}(\mathbf{d})$ the basic steps in order to calculate equation (3.2.1):

$$\frac{\partial \mathbf{R}(\mathbf{u}, \mathbf{d})}{\partial d_i} = \left(\frac{\partial}{\partial d_i} \mathbf{A}(\mathbf{d}) \right) \mathbf{u} - \frac{\partial}{\partial d_i} \mathbf{b}(\mathbf{d})$$

The routine that compute $\mathbf{A}(\mathbf{d})$ and $\mathbf{b}(\mathbf{d})$ is denoted `physics_elmope()`, which assembles stiffness matrix and load vector, and stores them into the arrays `amatr` and `rhsid`. The routine `physics_dirbcs()` imposes boundary conditions on the assembled elements. A detailed description of these routines can be reviewed in section 3.1.2.

Both routines don't have input parameters. Instead, they use several global variables, scalar and vectors, including the arrays `amatr` and `rhsid`. Mathematically, we can consider the routine `physics_elmope()` as a function

$$\texttt{elmope} : \mathbb{R}^m \rightarrow \mathbb{R}^{\texttt{nnz}} \times \mathbb{R}^{\texttt{npoin}}$$

with m the total number of global inputs used in the assembling, `nnz` and `npoin` the size of both global arrays `amatr` and `rhsid` respectively. The routine returns both arrays as

follows:

$$\texttt{elmope}\begin{pmatrix}\texttt{input\_1}\\ \vdots \\ \texttt{input\_m}\end{pmatrix} = \begin{pmatrix}\texttt{amatr(1)}\\ \vdots \\ \texttt{amatr(nnz)}\\ \texttt{rhsid(1)}\\ \vdots \\ \texttt{rhsid(npoin)}\end{pmatrix}$$

Applying the derivative $\frac{\partial}{\partial d_i}$ for some design variable d_i, the differentiated routine is as follows:

$$\frac{\partial}{\partial d_i}\texttt{elmope} = \begin{pmatrix}\frac{\partial}{\partial d_i}\texttt{amatr(1)}\\ \vdots \\ \frac{\partial}{\partial d_i}\texttt{amatr(nnz)}\\ \frac{\partial}{\partial d_i}\texttt{rhsid(1)}\\ \vdots \\ \frac{\partial}{\partial d_i}\texttt{rhsid(npoin)}\end{pmatrix}$$

At this point, we can apply the proposed methodology to find the scope of each variable `amatr(k)` and `rhsid(k)` for all indexes `k`, calculate the paths affected by the derivative $\frac{\partial}{\partial d_i}$ and apply the modifications in the code of `physics_elmope()`. The resulting new code is denoted `physics_elmopediff(i)` with `i` the design variable index of d_i. Two auxiliary global vectors are needed to store the derivate values: `damat` and `drhsi`, used as follows:

$$\frac{\partial}{\partial d_i}\texttt{elmope} = \begin{pmatrix}\frac{\partial}{\partial d_i}\texttt{amatr(1)}\\ \vdots \\ \frac{\partial}{\partial d_i}\texttt{amatr(nnz)}\\ \frac{\partial}{\partial d_i}\texttt{rhsid(1)}\\ \vdots \\ \frac{\partial}{\partial d_i}\texttt{rhsid(npoin)}\end{pmatrix} = \begin{pmatrix}\texttt{damat(1)}\\ \vdots \\ \texttt{damat(nnz)}\\ \texttt{drhsi(1)}\\ \vdots \\ \texttt{drhsi(npoin)}\end{pmatrix}$$

With the differentiated arrays `damat` and `drhsi` already calculated, we need to apply the differentiated boundary conditions, through a modified version of `physics_dirbcs()`, denoted `physics_dirbcsdiff(i)`. As we viewed in equations (3.1.1) and (3.1.2), the stiffness matrix and load vector are modified in order to impose the boundary condition $u = u_0$ in Γ. In the derivative code the modification is performed as depicted in equations (3.2.9) and (3.2.10), with $\tilde{a}_{k,j} = \frac{\partial}{\partial d_i} a_{k,j}$. If u_0 doesn't depend on any design variable, the right side of equation (3.2.10) is $\mathbf{0}_n$.

$$\left[\begin{array}{c|c|c} & \tilde{a}_{1,i} & \\ & \vdots & \\ & \tilde{a}_{i-1,i} & \\ \hline \tilde{a}_{i,1} \ \dots \ \tilde{a}_{i,i-1} & \tilde{a}_{i,i} & \tilde{a}_{i,i+1} \ \dots \ \tilde{a}_{i,n_u} \\ \hline & \tilde{a}_{i+1,i} & \\ & \vdots & \\ & \tilde{a}_{n_u,i} & \end{array}\right] \longrightarrow \left[\begin{array}{c|c|c} & 0 & \\ & \vdots & \\ & 0 & \\ \hline 0 \ \dots \ 0 & \frac{\partial}{\partial d_i} 1 & 0 \ \dots \ 0 \\ \hline & 0 & \\ & \vdots & \\ & 0 & \end{array}\right] = \mathbf{0}_{n\times n} \tag{3.2.9}$$

$$\begin{pmatrix} \vdots \\ \hline b_i \\ \hline \vdots \end{pmatrix} \longrightarrow \begin{pmatrix} \vdots \\ \hline \frac{\partial}{\partial d_i} u_0 \\ \hline \vdots \end{pmatrix} \tag{3.2.10}$$

3.2.6 Performance analysis

In this subsection we will show a performance analysis of the reduced gradient implementation, based in algorithm 4. Our interest is to have a mathematical expression of the computation/communication ratio, using as parameters the size of state and design vectors, n_u and n_d respectively, the number of slaves P, and the number of iterations in the linear solvers N_{fwd} and N_{adj}. The value of n_u can be different in each slave, but we will assume that all values are the same, approximately $n_u \approx n/P$ with n the number of nodal points in the whole mesh.

Computation time

As mentioned in section 2.1.2, our forward problem simulator uses iterative methods for sparse linear systems, which are typically based in several matrix-vector operations of cost $m << 2n_u^2$ floating-point operations (flops). We will assume that N_{fwd} matrix-vector operations of cost m flops must be performed each time the forward solver is executed. The corresponding execution time of the matrix-vector operations is $t_{mat-vec}$. Analogously, we will assume that N_{adj} matrix vector operations of cost $(1+\delta)m$ flops must be performed each time the adjoint solver is executed. The scalar $\delta > 0$ represents a small increment in the execution time using the transposed matrix-vector operation of algorithm 9.

- The first computation is related with the assembling of $\mathbf{A}_p(\mathbf{d})$ and $\mathbf{b}_p(\mathbf{d})$ for each slave $p \in \{1, \dots, P\}$:

$$t_{comp-fwd-assemble} = O(n_u) \tag{3.2.11}$$

This value represents the loop through all elements of routine `physics_elmope()` from figure 3.4. This loop is algorithmically similar to a loop based in nodal points.

- The second computation is related with the resolution of the forward problem, previously discussed:

$$t_{comp-fwd-solver} = N_{fwd} \times t_{mat-vec} \tag{3.2.12}$$

- The third computation is related with the computation and assembling of the cost function and its derivative w.r.t. state vector, performed by routines `physics_costf()` and `physics_dcost()`. Those routines are algorithmically similar to the basic assembling routine `physics_elmope()` with the corresponding execution time:

$$\begin{aligned} t_{comp-cost-assemble} &= O(n_u) + O(n_u) \\ &= O(n_u) \end{aligned} \tag{3.2.13}$$

- The fourth computation is related with the resolution of the adjoint problem, previously discussed:

$$t_{comp-adj-solver} = N_{adj} \times (1+\delta) t_{mat-vec} \tag{3.2.14}$$

- In order to calculate the reduced gradient, for each design variable d_i, we need to assemble $\frac{\partial}{\partial d_i} J(\mathbf{u}, \mathbf{d})$, $\frac{\partial}{\partial d_i} \mathbf{A}(\mathbf{d})$ and $\frac{\partial}{\partial d_i} \mathbf{b}(\mathbf{d})$, solve a matrix-vector and dot products:

$$\begin{aligned} t_{comp-grad} &= n_d \times (3 \times O(n_u) + t_{mat-vec} + O(n_u)) \\ &= n_d \times (O(n_u) + t_{mat-vec}) \end{aligned} \tag{3.2.15}$$

According to the previous expressions, the overall computation time is:

$$\begin{aligned} t_{computation} &= O(n_u) + N_{fwd} \times t_{mat-vec} + N_{adj} \times (1+\delta) t_{mat-vec} \\ &\quad + n_d \times (O(n_u) + t_{mat-vec}) \\ &= n_d \times O(n_u) + (N_{fwd} + N_{adj} \times (1+\delta) + n_d) \times t_{mat-vec} \end{aligned} \tag{3.2.16}$$

Communication time

We will assume that each operation `MPI_Sendrecv` exchanges an array of size d and takes the following execution time:

$$t_{send/recv} = t_{startup} + d \times t_{data} \tag{3.2.17}$$

with $t_{startup}$ a constant time portion of the transmission and t_{data} the time to transmit one data word. In the operation `MPI_Allreduce`, in its most basic implementation, the

masters gathers $P-1$ arrays of size d from each slave, calculates the selected operation, for example a vectorial sum, and sends back the results with a broadcast operation:

$$\begin{aligned} t_{allreduce} &= (P-1)\times(t_{startup}+d\times t_{data})+P\times d\times t_{flop}+t_{startup}+d\times t_{data} \\ &= P\times(t_{startup}+d\times t_{data}+d\times t_{flop}) \end{aligned} \tag{3.2.18}$$

with t_{flop} a constant time representing a floating-point operation. Using the previous assumptions, the communication time is composed by:

- The assembling of $\mathbf{A}_p(\mathbf{d})$ and $\mathbf{b}_p(\mathbf{d})$ doesn't have any communication between processes.
- The forward problem resolution uses N_{fwd} matrix-vector operations, and each one uses an exchange operation `MPI_Sendrecv` exchanging at most n_u values (internal boundary nodes are sent/received by neighbour sub-domains). Additionally, a reduction is used to update the residual value $\|\mathbf{b}-\mathbf{A}\mathbf{u}\|^2$, through an operation `MPI_Allreduce`, where each slave calculates the scalar $\|\mathbf{b}_p-\mathbf{A}_p\mathbf{u}_p\|^2$ and then the master adds all slave values (i.e. $d=1$):

$$\begin{aligned} t_{comm-fwd-solver} &= N_{fwd}\times(t_{send/recv}+t_{allreduce}) \\ &= N_{fwd}\times(t_{startup}+n_u\times t_{data} \\ &\quad +P\times(t_{startup}+t_{data}+t_{flop})) \\ &= N_{fwd}\times(t_{startup}\times(P+1)+t_{data}\times(P+n_u)+t_{flop}\times P) \end{aligned} \tag{3.2.19}$$

- The computation of the cost function uses one scalar `MPI_Allreduce` adding the values of $J(\mathbf{u},\mathbf{d})$ in each sub-domain. The assembling of the cost function derivative doesn't have any communications between processes:

$$t_{comm-costf} = P\times(t_{startup}+t_{data}+t_{flop}) \tag{3.2.20}$$

- The adjoint problem resolution uses N_{adj} transposed matrix-vector operations, and following equation (3.2.19) the communication time is:

$$t_{comm-adj-solver} = N_{adj}\times(t_{startup}\times(P+1)+t_{data}\times(P+n_u)+t_{flop}\times P) \tag{3.2.21}$$

- In the reduced gradient calculation, for each design variable d_i, the only operations with communications between processes are the matrix-vector product and the vector-vector product, each performing a `MPI_Sendrecv` and `MPI_Allreduce` respectively:

$$\begin{aligned} t_{comm-grad} &= n_d\times(t_{send/recv}+t_{allreduce}) \\ &= n_d\times(t_{startup}+n_u\times t_{data}+P\times(t_{startup}+t_{data}+t_{flop})) \\ &= n_d\times(t_{startup}\times(P+1)+t_{data}\times(P+n_u)+t_{flop}\times P) \end{aligned} \tag{3.2.22}$$

According to the previous expressions, but omitting the contribution of $t_{comm-costf}$ to simplify the final results (this contribution is neglecting in comparison with the other communication contributions), the overall communication time is:

$$\begin{aligned} t_{communication} &= (N_{fwd} + N_{adj} + n_d) \\ &\quad \times (t_{startup} \times (P+1) + t_{data} \times (P + n_u) + t_{flop} \times P) \end{aligned} \tag{3.2.23}$$

Computation/Communication ratio

Using the expressions in equations (3.2.16) and (3.2.23), the computation/communication ratio is as follows:

$$\begin{aligned} ratio &= \frac{t_{computation}}{t_{communication}} \\ &= \frac{n_d \times O(n_u) + (N_{fwd} + N_{adj} \times (1+\delta) + n_d) \times t_{mat-vec}}{(N_{fwd} + N_{adj} + n_d) \times (t_{startup} \times (P+1) + t_{data} \times (P + n_u) + t_{flop} \times P)} \end{aligned}$$

In this expression, it is reasonable to assume that $t_{startup} \approx t_{data}$, $t_{flop} << t_{data}$, $(N_{fwd} + N_{adj} \times (1+\delta) + n_d)/(N_{fwd} + N_{adj} + n_d) \approx 1$, $O(n_u) \approx C_1 n_u$ and $t_{mat-vec} \approx C_2 n_u$ with $C_2 < C_1 << n_u$ using sufficiently sparse matrices (C_1 and C_2 are constants which depend linearly on the element-node connectivity of the computational domain), resulting in a more compact expression:

$$\begin{aligned} ratio &\approx \frac{n_d \times C_1 n_u}{(N_{fwd} + N_{adj} + n_d) \times (t_{data} \times (2P + n_u))} + \frac{C_2 n_u}{t_{data} \times (2P + n_u)} \\ &\approx \left(\frac{C_1 n_d}{N_{fwd} + N_{adj} + n_d} + C_2 \right) \times \frac{n/P}{t_{data} \times (2P + n/P)} \\ &\approx \left(\frac{C_1 \frac{n_d}{n}}{N_{fwd} + N_{adj} + n_d} + \frac{C_2}{n} \right) \times \frac{\frac{n}{P}}{t_{data} \times (2\frac{P}{n} + \frac{1}{P})} \\ &\approx \left(\frac{C_1 \frac{n_d}{n}}{N_{fwd} + N_{adj} + n_d} \right) \times \frac{\frac{n}{P}}{t_{data} \times (2\frac{P}{n} + \frac{1}{P})} \\ &\approx \frac{\frac{C_1}{2} \times n_d \times \frac{n}{P}}{P \times (N_{fwd} + N_{adj} + n_d) \times t_{data}} \end{aligned} \tag{3.2.24}$$

This ratio tells us that the computation depends on the number of state and design variables, $n_u = n/P$ and n_d, and the communication depends on the number of processes, solver iterations and number of design variables, together with the interconnection parameter t_{data}. Each application can have different values for this parameters, but a careful study of this ratio can help us to see if we have to put effort on optimize computation calculations or enhance the communication properties of the implementation, compressing vector values in order to reduce the number of messages that each slave sends/receives or using optimized versions of the MPI routines `MPI_Sendrecv` or `MPI_Allreduce`.

Extrae/Paraver analysis

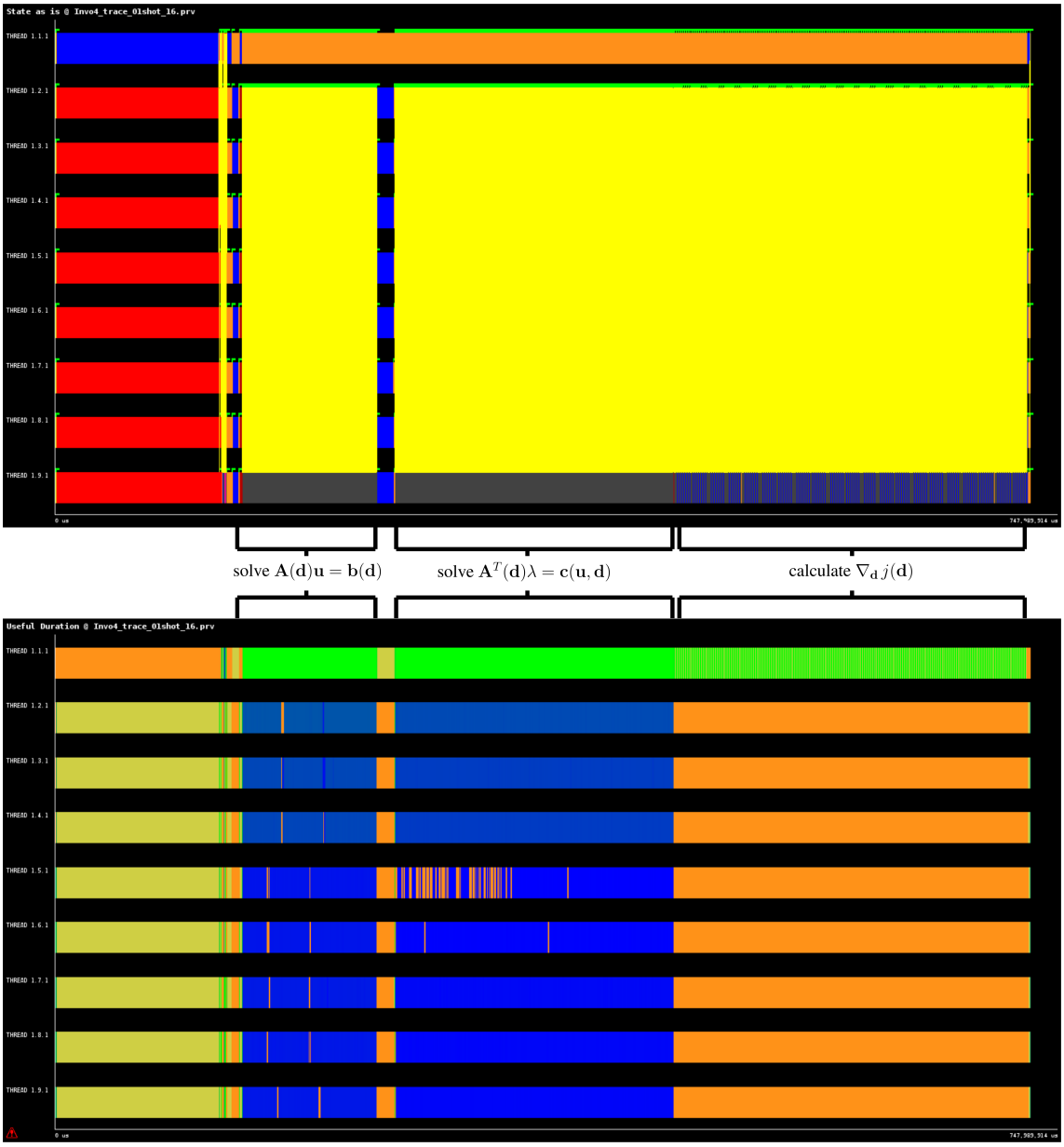

Figure 3.17: Alya's execution trace with reduced gradient calculation using 9 processes (1 master + 8 slaves). Paraver view *State as is* (top) shows yellow lines as MPI messages, and red and blue colors as idle and running states of the processes. Paraver view *Useful duration* (bottom) shows the compute burst intensity on each processor and each sample time step. Colors yellow, green, blue and orange represent low, medium-low, high and very-high compute burst intensity respectively

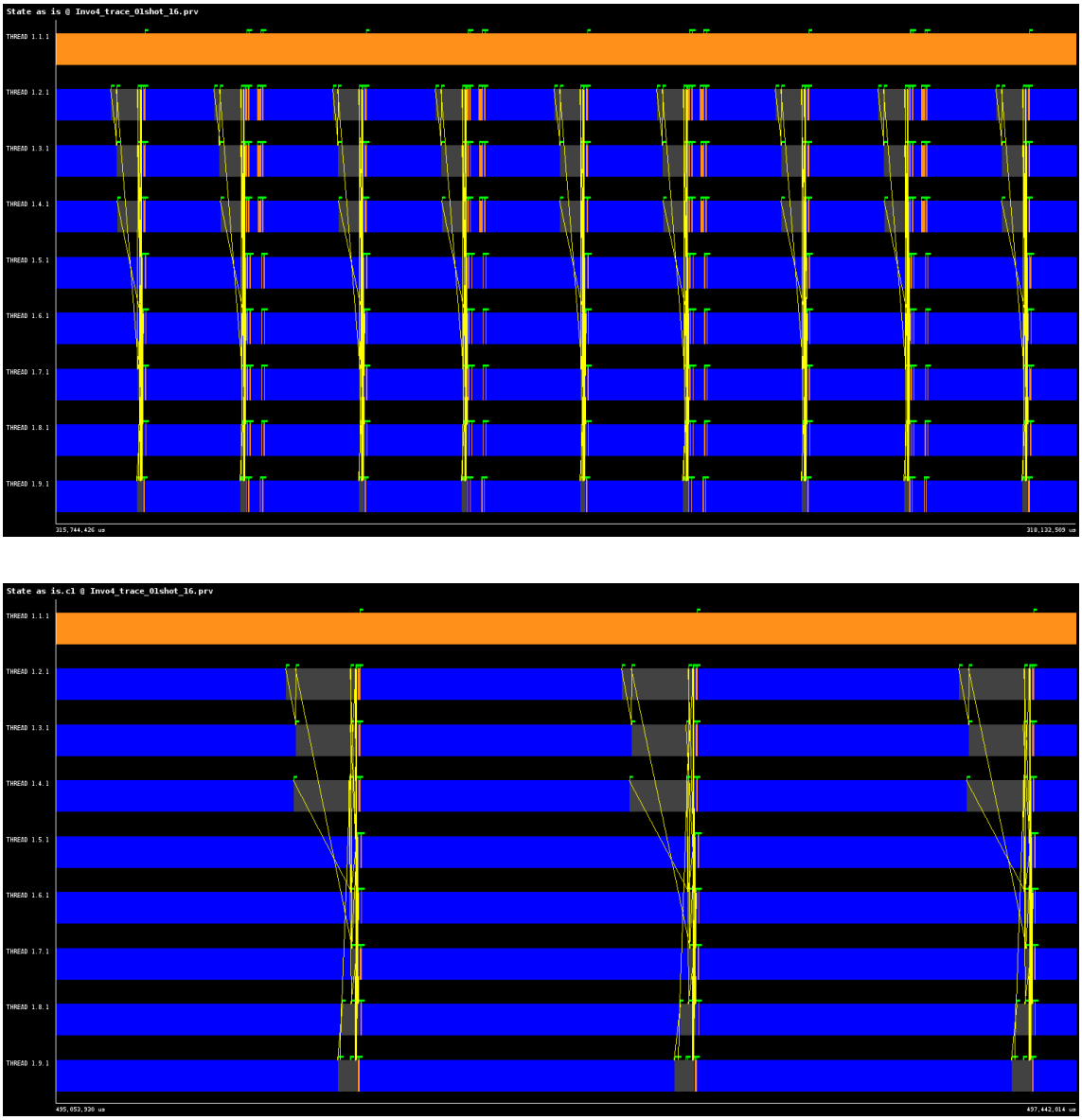

Figure 3.18: Zoom into *State as is* view from figure 3.17. On top we can see one of the solver regions (forward and adjoint are similar, only varies in the number of iterations) and on bottom we can see a few iterations of the loop through design variables to calculate $\nabla_{\mathbf{d}} j(\mathbf{d})$. Both zoom images use the same time duration

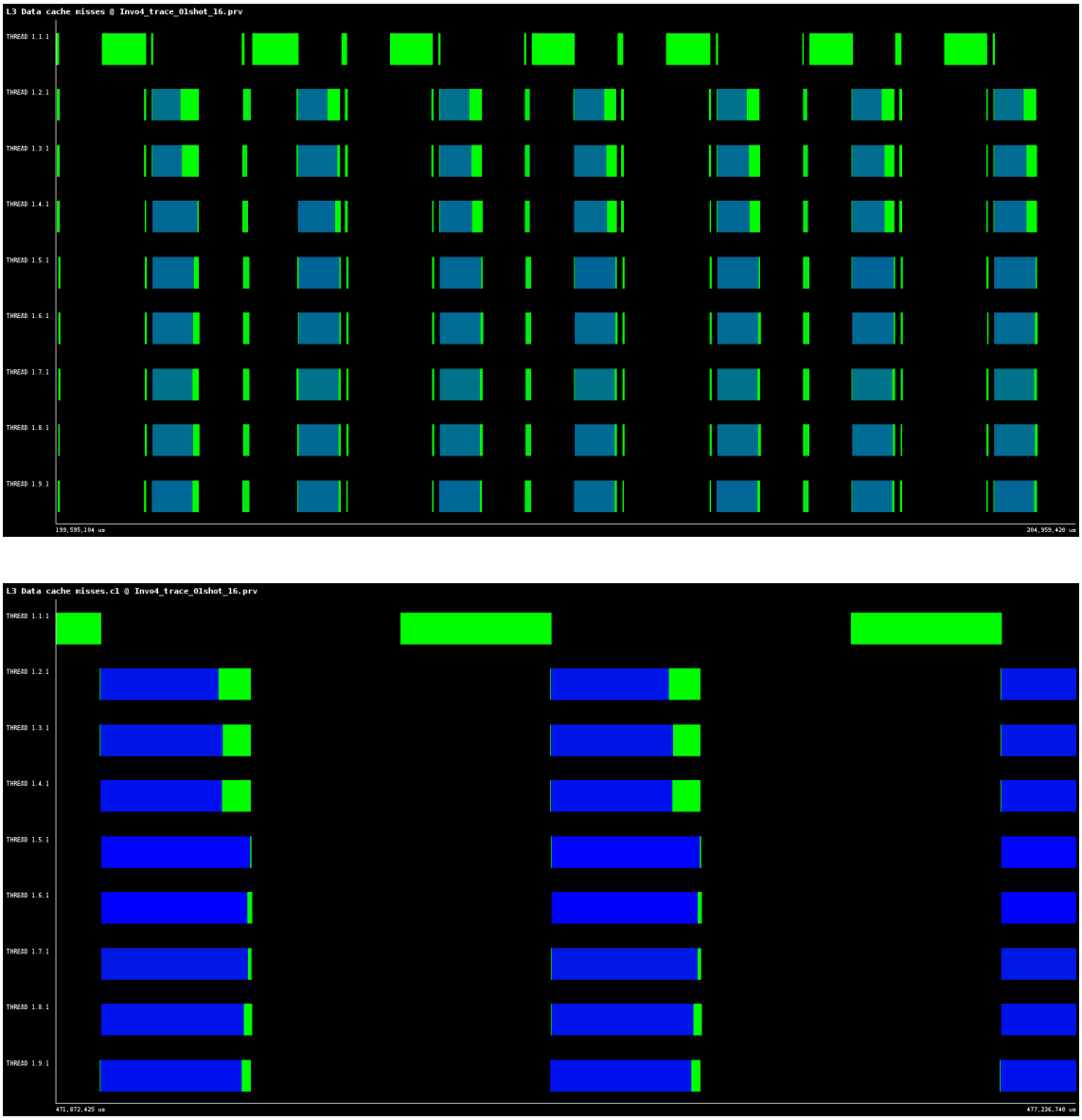

Figure 3.19: Zoom into *L3_Data_cache_misses* view. On top we can see one of the solver regions and on bottom we can see a few iterations of the loop through design variables to calculate $\nabla_{\mathbf{d}} j(\mathbf{d})$. Both zoom images use the same time duration. Colors light-green, pale-blue and navy-blue represent low, medium and high number of L3 cache misses in each corresponding CPU. The number of cache misses is higher in the design vector loop because in each iteration three assembling operations must be performed, removing from the CPU cache any previous values

3.2.7 Comments

As we saw in the previous subsections, relative to the reduced gradient calculation, several implementation steps are necessary to have $\nabla_{\mathbf{d}} j(\mathbf{d})$ at our disposal. This calculation is the most important in the whole optimization process, because it is a kernel element in several high-level PDE-constrained optimization algorithms. In some applications, using this gradient we can perform a sensitivity analysis to get useful information on how to modify the design vector. This kind of analysis may be the only alternative due to the complexity to obtain good gradients, making difficult or even impossible, to perform the optimization process.

The most difficult task by far, is related with the differentiation by hand of the routine `physics_elmope()`, because the programmer needs a high-level knowledge of each user-defined implementation. Different physical modules can be implemented with different styles and techniques, using different design variables, making this task very time-consuming and error-prone. In this work, we developed two applications using different physical modules, each one using different programming styles and subroutines. After this experience, we can recommend that, if a new physical module will be used as constraint in a PDE-constrained optimization problem, a lot of effort must be spent by the programmer in the study of the PDE discretization (finite element method in our case), and also to get insight of the actual implementation `physics_elmope()`, before start to work in other tasks.

The development of the cost function routine `physics_costf()` and its derivative `physics_dcost()` have a medium level of difficulty. This is due to the experience gained by the programmer in prior studies of the assembling routines. Additionally, these routines doesn't have to interact with other subroutines that may be strange or unknown to the programmer, and as consequence, he/she can base these implementations in the routine `physics_elmope()` only focusing in a few important parts of the code.

The implementation of the transposed iterative methods has also a medium level of difficulty, because it requires to study the implementation of the desired iterative solver, identify the spots where the matrix-vector operations are used, and change them by their transposed counterparts.

Regarding the performance, in figures 3.17, 3.18 and 3.19, we can see a sample behaviour of the proposed implementation of $\nabla_{\mathbf{d}} j(\mathbf{d})$. The forward and adjoint solvers inherit the scalability from the solvers already implemented in Alya. The design vector loop shows a more complex behaviour, with higher computation bursts per iteration and higher number of L3 cache misses. This misses are consequence of the assembling procedures performed in each iteration. All in all, the showed sample's efficiency obtained in the execution time using 9 processes (1 master + 8 slaves) was 72% in *running* state and 28% in *idle* state, which is similar to the efficiency obtained by Alya running a similar instance, solving only the forward problem. As a final remark, if better efficiency or less execution time is needed, the proposed implementation can be optimized studying the internal code of Alya, according to each specific application that will be solved.

3.3 Optimization service

In this section we will explain the details concerning the implementation of different descent directions $\mathbf{p}^k$ and line-search strategies with different steps α^k, according to the mathematical theory presented in sections 2.1.3 and 2.1.4. We start with a description of the service implemented in Alya which holds all the routines involved in these calculations. After that we will give details about the necessary modifications that must be introduced in Alya's main driver routine `Alya.f90`. Finally, we will explain the internal implementation of the most important routines inside of the proposed service.

3.3.1 *Optsol* service structure

Once a physical module have delivered the gradient $\nabla_{\mathbf{d}} j(\mathbf{d})$, we need to perform calculations relative to optimization methods, namely the descent direction calculation and line-search strategy. Our aim is to keep apart the reduced gradient calculation and optimization calculations, creating a new service in Alya's *services* layer. This new service is denoted *Optsol*, and its main routines are:

- `Optsol(TASK)`: this routine acts as an interface to manage the tasks requested by other routines. `TASK` is an identifier of the tasks that the service could perform. The most important ones are `BEGOPT` (begin optimization initializing some flags and variables), `DOOPTI` (perform descent direction calculation and/or calculate temporal updated design variables) and `ENDOPT` (test convergence conditions, calculate line-search step and accept/reject temporal updated variables).
- `opt_begopt()`: initializes the control flags and some internal variables. There are 1 control flag and 3 iteration counters:
  - `kfl_goopt`: Control flag. Indicates the termination of the optimization due to convergence conditions.
  - `kfl_curstp_opt`: Counter. Keeps track of the gradient calculation iterations. Each time a gradient calculation is performed, this counter is increased.
  - `kfl_curlin_opt`: Counter. Keeps track of the number of line-search evaluations. Each time a gradient calculation is performed, this counter is reseted to 1 (the first line-search evaluation).
  - `kfl_curres_opt`: Counter. Keeps track of the number of line-search restarts performed. When a line-search through a descent direction is not able to decrease the cost function, a restart of the initial step value is performed. Each time a gradient calculation is performed, this counter is reseted to 1 (the first line-search evaluation).

Each counter has a maximum number of iterations, defined in the input files.

Mathematically, using the notation defined in section 2.1.4, we can map the indexes k and i to the counters `kfl_curstp_opt` and `kfl_curlin_opt` respectively, according to the following update equation:

$$\mathbf{d}_i^k = \mathbf{d}^k + \alpha_i^k \mathbf{p}^k \tag{3.3.1}$$

In the following table we can view a sample of the counter's evolution, according to the fulfillment of termination criteria, for example, the Wolfe conditions from section 2.1.4:

kfl_curstp_opt	kfl_curlin_opt	kfl_curres_opt	$\mathbf{d}_i^k$	line-search termination
1	1	0	$\mathbf{d}_1^1 = \mathbf{d}^1 + \alpha_1^1 \mathbf{p}^1$	No
1	2	0	$\mathbf{d}_2^1 = \mathbf{d}^1 + \alpha_2^1 \mathbf{p}^1$	No
1	3	0	$\mathbf{d}_3^1 = \mathbf{d}^1 + \alpha_3^1 \mathbf{p}^1$	Yes, $\mathbf{d}^2 = \mathbf{d}_3^1$
2	1	0	$\mathbf{d}_1^2 = \mathbf{d}^2 + \alpha_1^2 \mathbf{p}^2$	No
2	2	0	$\mathbf{d}_2^2 = \mathbf{d}^2 + \alpha_2^2 \mathbf{p}^2$	No
2	3	0	$\mathbf{d}_3^2 = \mathbf{d}^2 + \alpha_3^2 \mathbf{p}^2$	No
2	4	0	$\mathbf{d}_4^2 = \mathbf{d}^2 + \alpha_4^2 \mathbf{p}^2$	Yes, $\mathbf{d}^3 = \mathbf{d}_4^2$
3	1	0	$\mathbf{d}_1^3 = \mathbf{d}^3 + \alpha_1^3 \mathbf{p}^3$	No
3	2	0	$\mathbf{d}_2^3 = \mathbf{d}^3 + \alpha_2^3 \mathbf{p}^3$	No
⋮	⋮	⋮	⋮	⋮
3	N	0	$\mathbf{d}_N^3 = \mathbf{d}^3 + \alpha_N^3 \mathbf{p}^3$	No, restart $\alpha_1^3 \leftarrow 10\alpha_1^3$
3	1	1	$\mathbf{d}_1^3 = \mathbf{d}^3 + \alpha_1^3 \mathbf{p}^3$	No
3	2	1	$\mathbf{d}_2^3 = \mathbf{d}^3 + \alpha_2^3 \mathbf{p}^3$	No
⋮	⋮	⋮	⋮	⋮

- `opt_doopti()`: this routine performs two different tasks, depending on the value of the counter `kfl_curlin_opt`.

  If `kfl_curlin_opt` $= 1$, we need to calculate the descent direction $\mathbf{p}^k$ using the reduced gradient as input. It will be stored in a global variable (vector) with the same size as `diffj`, denoted `descdir`. Additionally, with this descent direction, the first update candidate must be calculated, denoted $\mathbf{d}_1^k = \mathbf{d}^k + \alpha_1^k \mathbf{p}^k$, with an initial step value of α_1^k. The update candidate $\mathbf{d}_1^k$ is stored in a global variable (vector) denoted `designtmp`.

  If `kfl_curlin_opt` > 1, we only need to calculate the update candidate $\mathbf{d}^k + \alpha_i^k \mathbf{p}^k$ using an updated step α_i^k, with $i > 1$, and the current descent direction $\mathbf{p}^k$ previously calculated in the first line-search step. The update candidate will be stored in the global variable (vector) `designtmp`, re-writing the memory locations.

  Three internal routines are used to perform the previous tasks: `opt_descdir()`, `opt_algorithm(ALGO)` and `opt_updde s()`. A detailed explanation of each one will be given in a further subsection.

- `opt_endopt()`: manages the control flow of the optimization, using the control flags and counters previously described. Additionally, this routine is in charge of accept or reject the proposed updated design vector candidates $\mathbf{d}_i^k$, stored in `designtmp`. This task is done by testing user defined line-search termination criteria. It also updates the step length for the next iteration. Further details will be given in the corresponding subsection.

3.3.2 Main driver modifications

In section 3.1.1 we reviewed the current implementation of Alya's main driver routine, `Alya()`, with its sequence diagram depicted in figure 3.2. Based in this routine, we have developed a modified driver routine including some aspects necessary to perform calls to the service *Optsol*, with a corresponding optimization loop. The modified sequence diagram is depicted in figure 3.20, and the description of these steps is as follows:

- The routine `Turnon()` is performed in the same way as in the original routine. The only modifications are related with input files reading, which now includes parameters related to the new service *Optsol*, such as convergence tolerance, maximum number of iterations, size of design vector array, adjoint iterative solver method parameters, and others.
- After that, the kernel routine `Begopt()` is launched. It delegates its work in the routine `Optsol(BEGOPT)` managed by the same service, which acts as interface to the service routine `opt_begopt()`.
- A loop block starts, iterating indefinitely if the condition `flag==true` holds. The flag corresponds to the control flag `kfl_goopt`.
- Inside of this block, the routines `Iniunk()` and `Doiter()` are performed in the same way as in the original routine, with the addition of the reduced gradient and cost function calculation by the physical module, stored in the variables `costf` (scalar) and `diffj` (vector). All details about the reduced gradient and cost function calculation were given in section 3.2.
- The routine `Doopti()` is launched by the kernel, delegating its work in the routine `Optsol(DOOPTI)`, which acts as interface to the service routine `opt_doopti()`. It returns the variables `designtmp` (vector) and `descdir` (vector). The first stores the temporal value of the updated design vector $\mathbf{d}_i^k$, that need to be validated. The second stores the value of the current descent direction $\mathbf{p}^k$ being used by the line-search.
- The routine `Endopt()` is launched by the kernel, delegating its work in the routine `Optsol(ENDOPT)`, which acts as interface to the service routine `opt_endopt()`. It returns the variables `flag` (binary flag), `design` (vector) and `step` (scalar).

The binary flag corresponds to the updated value of the control flag `kfl_goopt`. If `kfl_goopt==false` the optimization stops. The `design` vector corresponds to the updated value for the design vector, that will be used in the next iteration of the optimization, in order to calculate a cost function evaluation, and a reduced gradient calculation (only if a new line-search is launched). Finally, the `step` scalar stores the step value α_{i+1}^{k} for the next iteration of the line search, or α_{1}^{k+1} if termination criteria has been fulfilled by the current step value.

- The last routines `Output()` and `Turnoff()` work in the same way as in the original routine. The only modification is related with the output of the design vector, that is managed by the post-processing services of Alya.

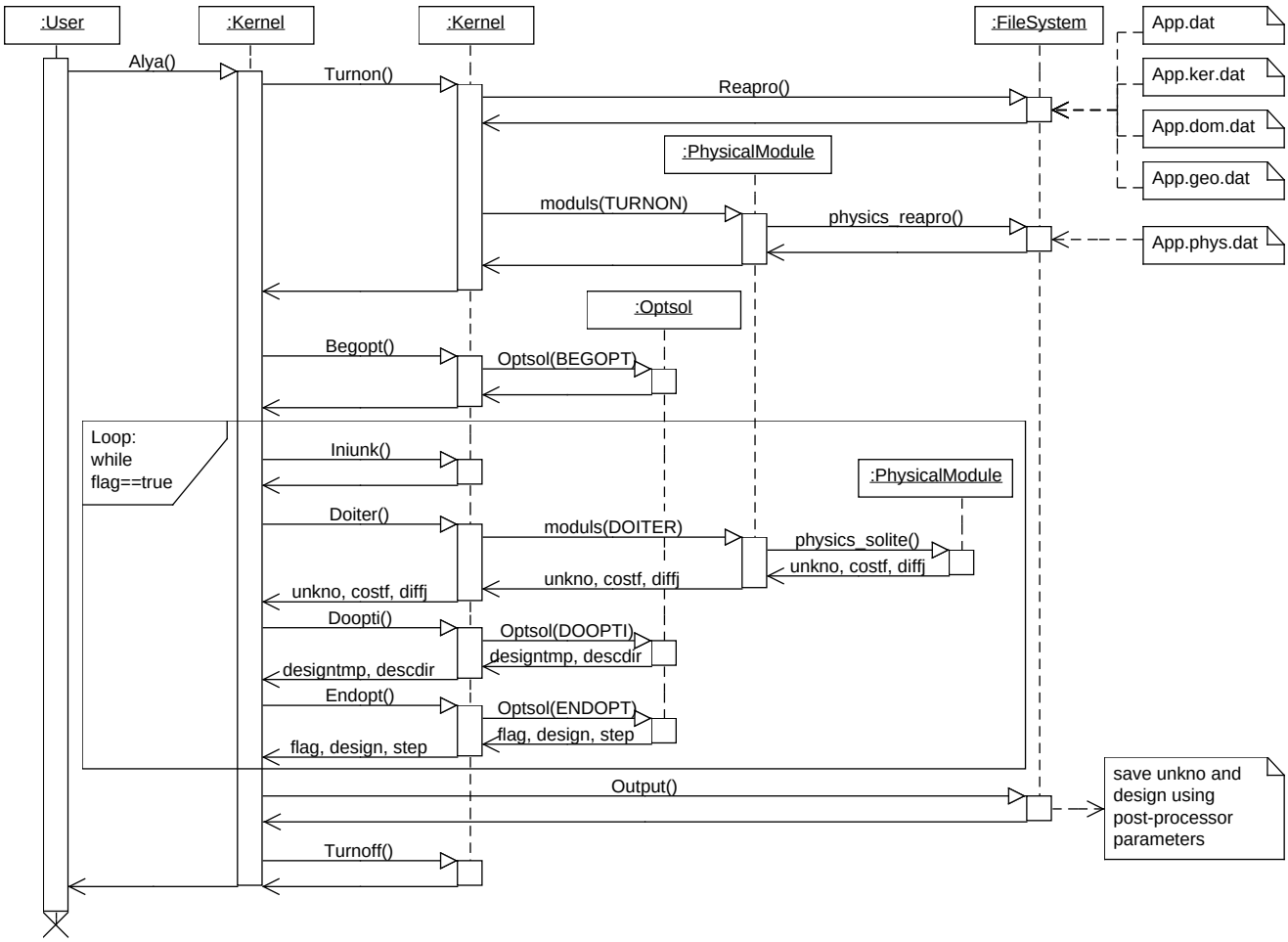

Figure 3.20: Sequence diagram of the modified main driver routine `Alya()` including calls to the optimization service *Optsol*

3.3.3 Descent direction and design vector candidate calculation

In figure 3.21 we can view the sequence diagram of the routine `opt_doopti()`. We can see clearly two subroutines launched by the service, denoted `opt_descdir()` and `opt_upddes()`. And also, inside of `opt_descdir()`, we can see another subroutine denoted `opt_algorithm(ALGO)`. The steps in this diagram are as follows:

- The routine `opt_descdir()` is launched by the service. An optional block can be executed when `kfl_curlin_opt=1`, and is associated with the calculation of the descent direction $\mathbf{p}^k$.
- Inside of this block, the routine `opt_algorithm(ALGO)` is launched by the service, using an index parameter `ALGO`, which represents the identifier number of the algorithm used to calculate $\mathbf{p}^k$. Possible algorithms to be implemented in this routine can be viewed in section 2.1.3. Currently, there are two algorithms implemented: steepest descent and conjugate gradient using the Polak-Ribière update formula. It uses the current reduced gradient $\nabla_{\mathbf{d}} j(\mathbf{d}^k)$ as input, stored in the global variable `diffj` (vector). The output is stored in the global variable `descdir` (vector), with the same size of `diffj`.

  In order to store the current descent direction through all the associated line-search iterations, we use an auxiliary global variable denoted `descdirprev` (vector), with the same size as `descdir`. This auxiliary variable is useful in some calculations when we need to use a previous value of the descent direction in order to calculate its current value, for example in the conjugate gradient update

  $$\mathbf{p}^k = -\nabla_{\mathbf{d}} j(\mathbf{d}^k) + \beta^k \mathbf{p}^{k-1}.$$

  After that, when `kfl_curlin_opt>=1`, we need to assign the previously stored value `descdirprev` into `descdir` (when `kfl_curlin_opt=1` we are performing an extra assignment, but this is left in this form in order to clarify the code).

- After that, the routine `opt_upddes()` is launched by the service. If the descent direction calculation is active, i.e. `kfl_curlin_opt=1`, we need to store the current value of the global variable `design` into an auxiliary global variable `designprev`. This variable is useful when we need to use previous values of the design variable in some internal calculation.

  After that, when `kfl_curlin_opt>=1`, the update candidate is calculated through the formula

  $$\texttt{designtmp} = \texttt{designprev} + \texttt{step} * \texttt{descdir}$$

  which is equivalent to

  $$\mathbf{d}_i^k = \mathbf{d}^k + \alpha_i^k \mathbf{p}^k$$

  The output corresponds to the global variable `designtmp` (vector), with the same size of `design`.

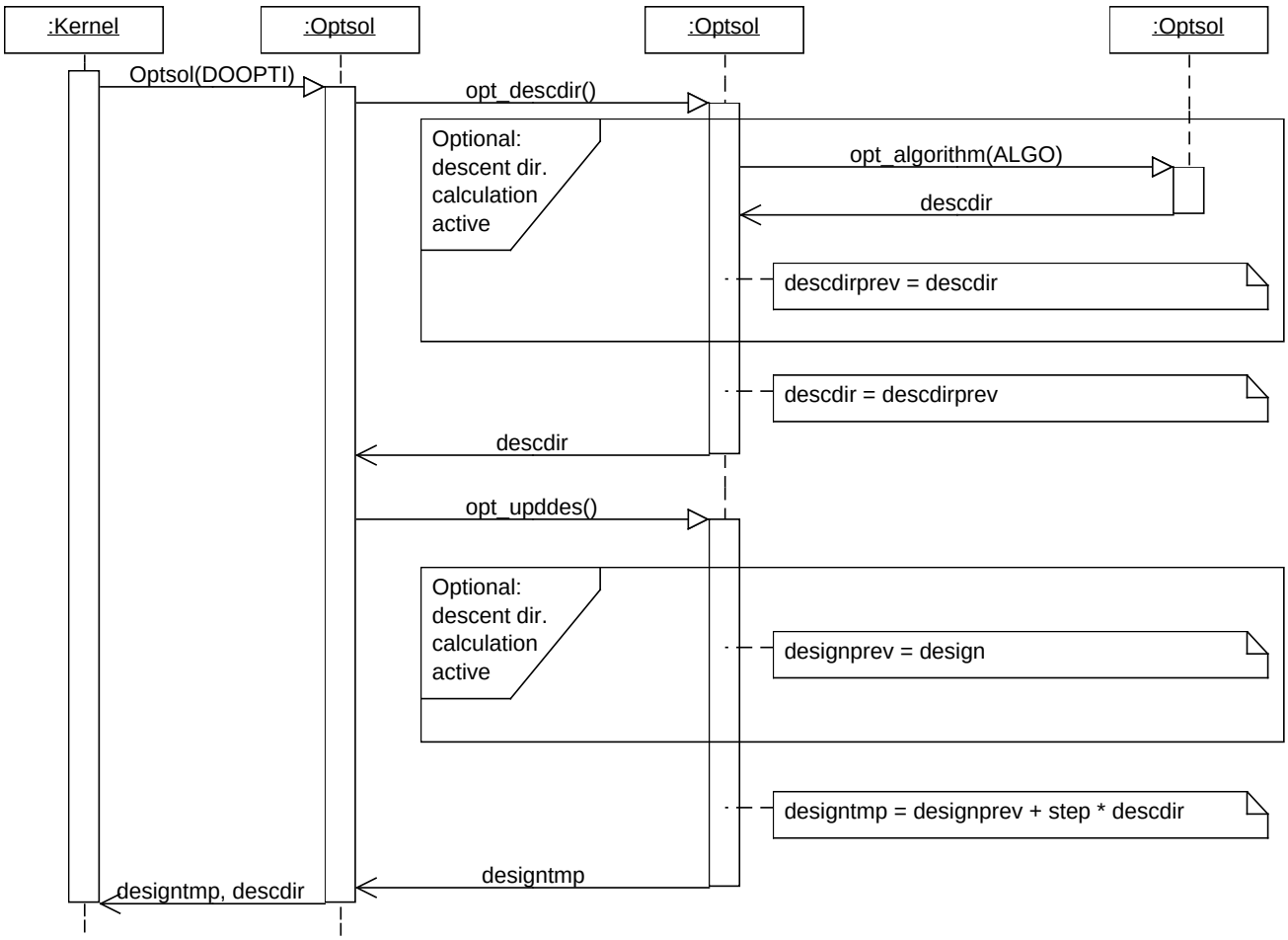

Figure 3.21: Sequence diagram of the routine `opt_doopti()`

3.3.4 Flow control and updates

The last part of the optimization process is related with the test of overall convergence/termination and line-search termination criteria using the update candidate $\mathbf{d}_i^k$, descent direction $\mathbf{p}^k$ and reduced gradient $\nabla_{\mathbf{d}} j(\mathbf{d})$ as input, stored in the global variables `designtmp`, `descdir` and `diffj` respectively. These tasks are included into the routine `opt_endopt()`.

Several nested tests are performed, starting with the overall convergence/termination. Each test can evaluate control flags, counters, tolerances or maximum number of iterations. A detailed description is presented in algorithm 12:

- In line 1 we perform a convergence test, calculating the reduced gradient norm $\|\nabla_{\mathbf{d}} j(\mathbf{d}^k)\|$ and checking if a tolerance ϵ_{grad} is achieved. Additionally, we test the current number of iterations stored in the counter `kfl_curstp_opt`, checking if a maximum number has been reached. If some test is not passed, the optimization ends by setting the binary flag `kfl_goopt` to 0.

- In line 2 we test if this is the very first iteration of the algorithm (first descent direction calculation and first line-search evaluation). An update of the design vector and step is performed in lines 3. These operations are defined as:

```
update(design):={
      design = designtmp
}

update(step):={
      // one of this updates is performed
      //step = 0.5 * step
      //step = 2.0 * step
}
```

  The step update corresponds to a backtracking search, according to the theory presented in section 2.1.4. Finally, an increment in the line-search counter is performed, `kfl_curlin_opt++`.

- In line 7 we test the line-search termination criteria using the step α_i^k, stored in the global variable `step` (scalar). In this case, the Armijo condition is evaluated, as shown in equation (2.1.11a):

$$Armijo: \; j(\mathbf{d}^k + \alpha_i^k \mathbf{p}^k) \leq j(\mathbf{d}^k) + c_1 \alpha_i^k \mathbf{p}^k \cdot \nabla_{\mathbf{d}} j(\mathbf{d}^k)$$

  The curvature condition (2.1.11b) is stronger, so we decided not to evaluate it in our tests. In order to evaluate the Armijo condition, we need to store previously the cost function value $j(\mathbf{d}^k)$, which will be saved in a global variable denoted `costfprev`

(scalar). The global variable `costf` (scalar) contains the value $j(\mathbf{d}^k + \alpha_i^k \mathbf{p}^k)$, `descdir` contains the value of $\mathbf{p}^k$ and `diffj` contains the value of $\nabla_{\mathbf{d}} j(\mathbf{d}^k)$.

If this condition holds, the line-search ends. An update of the design vector is performed using the `update(design)` operation, and also a reset of the step length using the operation `reset(step)` defined as:

```
reset(step):={
      step = step_initial
}
```

With `step_initial` the initial value α_1^k for each line-search execution (it can be different in each line-search, but we have considered only a fixed initial value $\alpha_1^k = \alpha_1$). The line-search counter is reseted, `kfl_curlin_opt=1`, and the gradient calculation counter is incremented, `kfl_curstp_opt++`.

- If the Armijo condition is not fulfilled, in line 13 we test if the current number of line-search evaluations has reached a maximum number. If this limit has not been reached yet, we perform an update of the design vector and step length, as in lines 3-4 described before.

- If the maximum number of line-search evaluations has been reached, in line 18 we test if the current number of line-search restarts has reached a maximum number. If this limit has not been reached yet, we reset the design vector using the operation `reset(design)` and restart the step length using `restart(step)`, both operations defined as:

```
reset(design):={
      design = designprev
}

restart(step):={
      // one of this restarts is performed
      //step = 10.0 * step_initial
      //step = 0.1 * step_initial
}
```

The line-search counter is reseted, `kfl_curlin_opt=1` and the line-search restart counter is incremented, `kfl_curres_opt++`. If the maximum number of line-search restarts has been reached, the optimization ends by setting the binary flag `kfl_goopt` to 0.

```
  // Test convergence and max number of iterations
1 if ‖∇_d j(d^k)‖ > ε_grad .and. kfl_curstp_opt < max-curstp then
    // Test very first iteration
2   if kfl_curstp_opt = 1 and kfl_curlin_opt = 1 then
3     update(design);update(step);
4     kfl_curlin_opt++;
5     Exit;
6   else
      // Test line-search termination (Armijo condition)
7     if Armijo condition holds then
8       update(design);reset(step);
9       kfl_curlin_opt=1;
10      kfl_curstp_opt++;
11      Exit;
12    else
        // Test line-search max number of iterations
13      if kfl_curlin_opt < max-curlin then
14        update(design);update(step);
15        kfl_curlin_opt++;
16        Exit;
17      else
          // Test restart max number of iterations
18        if kfl_curres_opt < max-curres then
19          reset(design);restart(step);
20          kfl_curlin_opt=1;
21          kfl_curres_opt++;
22          Exit;
23        else
            // End of optimization
24          kfl_goopt = 0;
25          Exit;
26        end
27      end
28    end
29  end
30 else
    // End of optimization
31  kfl_goopt = 0;
32  Exit;
33 end
```

Algorithm 12: Flow control and updates of the optimization process implemented in the routine `opt_endopt()`

Chapter 4

Applications

4.1 Source parameter estimation in 2D transport equation

4.1.1 Problem formulation

In order to test if the proposed implementation allows us to obtain reasonable convergence results, and also preserving the scalability of Alya, a synthetic problem with unique solution is proposed. The test case problem with domain $\Omega \subset \mathbb{R}^2$ is:

$$\begin{array}{lll} \underset{\mathbf{d}\in\mathbb{R}^{n_d}}{\text{minimize}} & \frac{1}{2}\int_\Omega (u(x,y)-u^{obs}(x,y))^2 dxdy & \\ \text{subject to} & \mathcal{L}(u) = f(\mathbf{d}) & \text{in } \Omega \\ & u = u^{obs} & \text{in } \partial\Omega \end{array} \tag{4.1.1}$$

with the differential operator and source function as:

$$\mathcal{L}(u) = \rho c_p \vec{v}\cdot\nabla u - \nabla\cdot(\kappa\nabla u) + s\,u \tag{4.1.2}$$

$$f(\mathbf{d}) = \sum_{i=1}^{n_d} (d_i - d_i^{target})^2\, p_i(x,y) \tag{4.1.3}$$

The operator $\mathcal{L}$ is a stationary convection-diffusion-reaction linear operator, commonly know as *transport equation*, with $\kappa > 0$, $\rho, c_p, s \in \mathbb{R}$ and $\vec{v} : \mathbb{R}^2 \to \mathbb{R}^2$, and the source f is defined using p_i as quadratic polynomials in the coordinates x and y, independent of $\mathbf{d}$. The design variable vector is $\mathbf{d} = (d_1, \ldots, d_{n_d})$.

According to the weak maximum principle applied to elliptic operators [21], this problem has a unique theoretical solution:

$$(u^*, \mathbf{d}^*) = (u^{obs}, \mathbf{d}^{target}) \tag{4.1.4}$$

4.1.2 Parameters and system settings

Physical and computational mesh parameters

We choose a constant value for $u_{obs} = 100$ that acts as an *observed* data. Three different sizes of the computational mesh described in figure 4.1 were tested: 330.000 elements (small), 1.300.000 elements (medium) and 3.700.000 elements (large). Using each mesh we test 4 different source functions with 1, 5, 10 and 50 design variables.

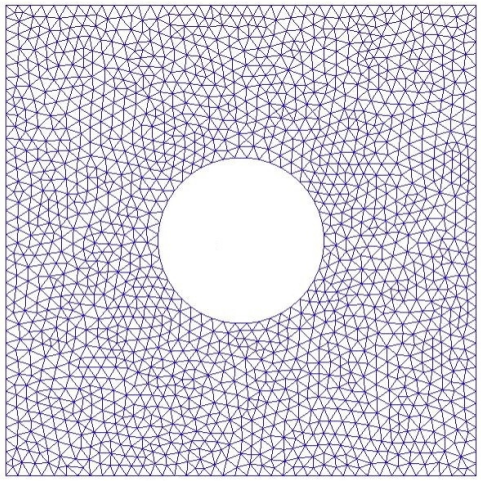

Figure 4.1: 2D mesh of the source parameter estimation problem

Hardware settings

The number of processes (CPUs) available for each run were 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512. A distributed-memory cluster of compute nodes called Marenostrum-II was used to execute the tests. It is based on 2560 compute nodes of IBM PowerPC 970MP 4-core CPUs at 2.3GHz connected through a Myrinet interconnection network, with 32GB of memory per node.

Iterative solver parameters

The iterative method chosen to solve forward and adjoint linear systems was GMRES using Jacobi ($\mathbf{P} = diag(\mathbf{A}_p)$) as preconditioner, with initial values of $\mathbf{u}^0 = \mathbf{0}$ and $\boldsymbol{\lambda}^0 = \mathbf{0}$ in both problems respectively. The parameters *maximum number of iterations* and *residual convergence tolerance* ϵ of GMRES (measured as $r_k = \|\mathbf{P}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{u}^k)\|$) are setted as 1000 iterations and $\epsilon = 10^{-20}$ respectively.

Optimization parameters

The optimization method implemented is a steepest descent ($\mathbf{p} = -\nabla_{\mathbf{p}} j$) with tolerance $\epsilon_{grad} = 10^{-10}$, and a maximum of 30 optimization iterations and a backtracking line-search with a maximum of 10 iterations.

4.1.3 Results

Each test, identified by the size of the mesh, the number of design variables and the number of processes, was executed 30 times taking the average execution time as final value.

An example of initial and final states of the system can be viewed in figure 4.2.

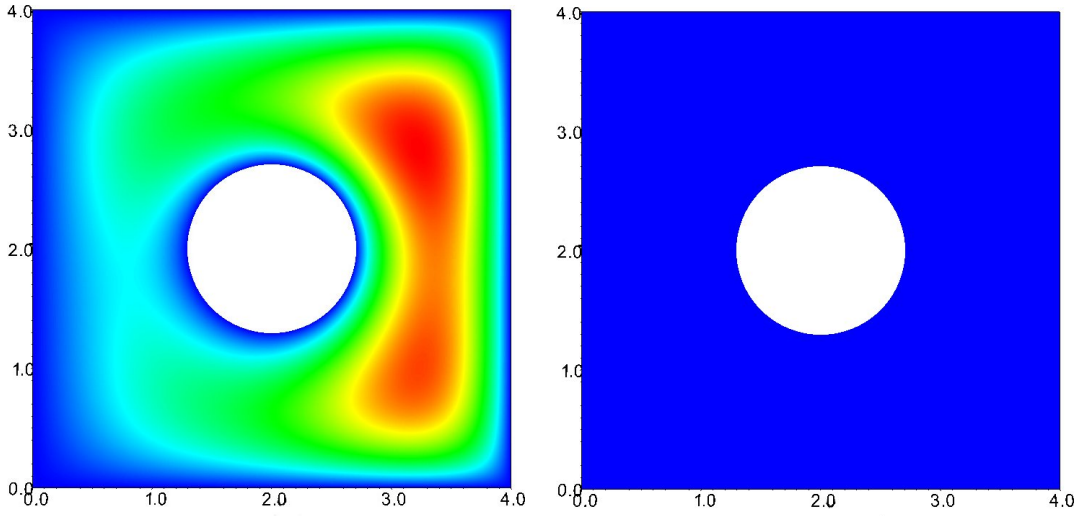

Figure 4.2: Possible initial state (left) and final state (right) using $u_{obs} = 100$. The values of the state variable are: $u_{\max} = 2471$, $u_{\min} = 100$ for the initial state, $u_{\max} = 100$, $u_{\min} = 100$ for the final state (blue color represents value $u = 100$)

Samples of convergence plots, showing the evolution of the cost function j and the squared norm of its reduced gradient, are included in figures 4.3 and 4.4. An example of the evolution of a design variable vector $\mathbf{d}$ as the optimization process advances is presented in figure 4.5. The execution times and speed-up results using the large mesh (3.7M elements) are presented in figures 4.6 and 4.7.

According to our tests, the proposed implementation achieves acceptable levels concerning the convergence of the optimization process and the preservation of the scalability. All tests converged to the desired target values, however the achieved convergence is strongly dependent on the quality of the residual delivered by the parallel iterative solver,

the size of the computational mesh and the number of design variables. We choose GMRES as main solver for the forward and adjoint problems because it allows to solve non-symmetrical linear systems, but one of the disadvantages of this solver is its poor convergence if the mesh step size decreases, as in the large mesh scenario. When the step-size decreases, more iterations are needed in order to reach smaller values of the residual r_k. Using this large mesh combined with a fixed number of solver iterations (1000 iterations), GMRES only reaches a residual of order 10^{-1}. A consequence of this bad behaviour is an increment in the number of optimization iterations (line-searches and gradient calculations) and in some cases no convergence at all was achieved. On the other hand, if more solver iterations are used, good residual values are obtained from the forward and adjoint problems, enhancing the convergence rate of the optimization method, with the payoff of performing more iterations in the linear system resolution.

One of the most important factors that determines the quality of the obtained residual is the preconditioner P used in the iterative solver. In our tests we choose the Jacobi (diagonal) preconditioner, but strong evidence indicates that several tests must be done using different preconditioners (such as ILU, block-ILU [47] or multigrid methods [27, 52], among others) in order to get lower values for the residual, and accelerate the convergence of the optimization process.

Concerning the scalability, we can see that for the large mesh it is preserved up to 128. As we explained before, this behaviour is a consequence of the useful work performed by each process versus the amount of overhead time introduced by inter-process communication in the linear system resolution. The amount of work that each process does is proportional to the amount of elements in their corresponding sub-domains. The average number of elements per process is shown on the top x-axis of the speedup plot in figure 4.7. We can see that the scalability decreases when the elements per process goes approximately below 25K. As we mentioned in the description of the method, two linear systems must be solved in each optimization step, while only one matrix assembling is done. The parallel solver used to solve the adjoint problem doubles the amount of inter-process communications by means of MPI messages, overloading the interconnection network in some cases, increasing even more the overhead time as a consequence. The ratio (useful work):(overhead introduced by inter-process communications) is critical in the preservation of the scalability, and for this particular method, a further study on this aspect is needed in order to get insight on the limits of the parallelism that can be used in it.

Concerning the physical memory of the computational resources (in this case, RAM of each compute node), a nice feature of the method is its use of the same matrix already stored in memory for the resolution of the adjoint problem. This feature reduces in part the utilization of extra memory and allows us to use the same data structures that are being used in the forward problem. However, if $\mathbf{d}$ is large, a possible limitation has to do with the storage of $\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u},\mathbf{d})$ and $\nabla_{\mathbf{d}}j(\mathbf{d})$. Until now, the storage of each derivative $\frac{\partial}{\partial d_i}\mathbf{R}(\mathbf{u},\mathbf{d})$ is done using a vector of size n_u denoted `drhsi`. Different values of d_i use the same vector to store its values, re-writing the memory locations. To store $\nabla_{\mathbf{d}}j(\mathbf{d})$,

we use a global vector of size n_d denoted `diffj` not distributed among the slaves, which limits its size to the available memory per node. Efficient and distributed ways of storing this derivatives have to be implemented, in order to avoid this limitation, so a further study on this aspect is also needed.

A typical pitfall in parallel computing has to do with the numerical accuracy obtained with several processes. In our case, floating-point non-commutativity can affect the number of optimization steps. The value of $\nabla_{\mathbf{d}} j(\mathbf{d})$ can vary if `MPI_Allreduce` accumulates the sum in different orders on different processors, since floating-point addition is not associative. The result of this behaviour is that slightly differences, typically in very small orders of magnitude, are accumulated through the optimization iterations and convergence can be achieved in different steps using different numbers of processes.

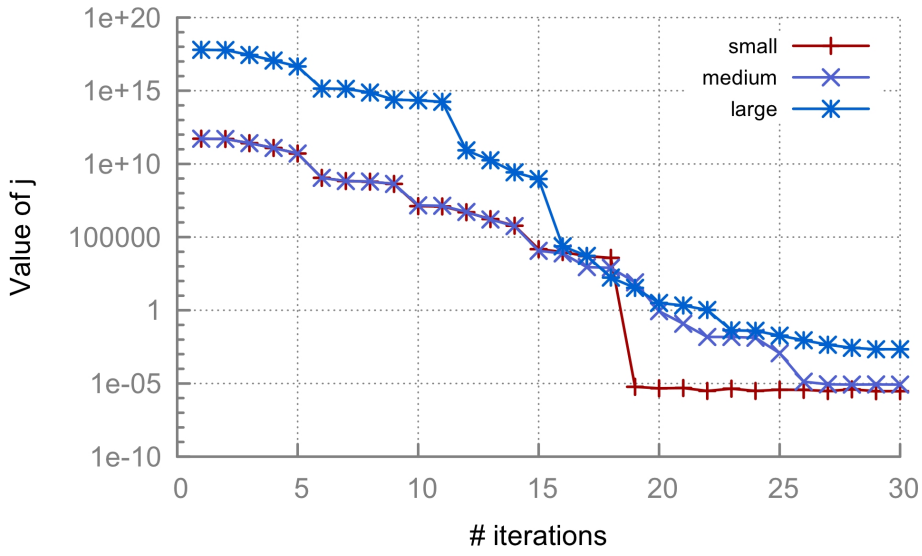

Figure 4.3: Convergence of cost function j with 5 design variables. The design variables start at initial values $\mathbf{d}^0 = (100, 100, 100, 100, 100)$ and converge to the target values $\mathbf{d}^{target} = (50, 80, 20, 0, -80)$.

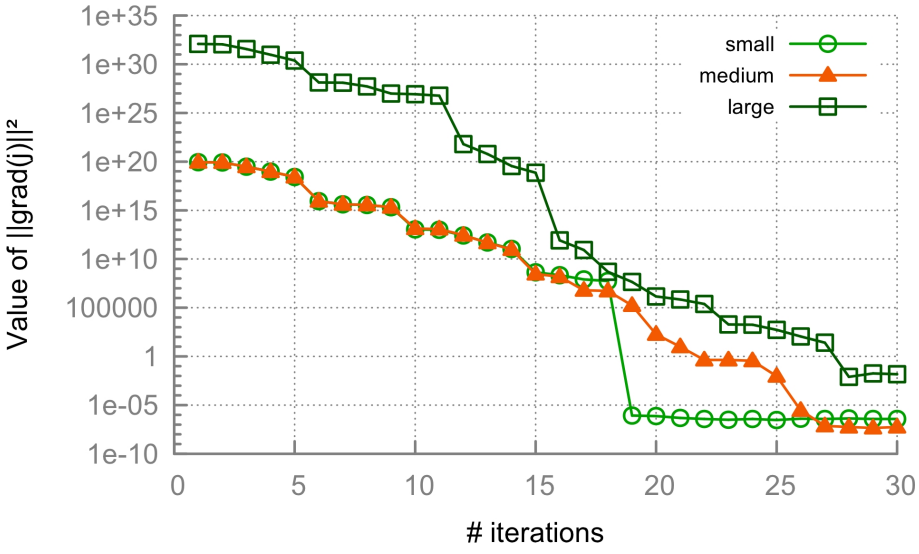

Figure 4.4: Convergence of $\|\nabla_{\mathbf{d}} j\|$ with 5 design variables. The design variables start at initial values $\mathbf{d}^0 = (100, 100, 100, 100, 100)$ and converge to the target values $\mathbf{d}^{target} = (50, 80, 20, 0, -80)$.

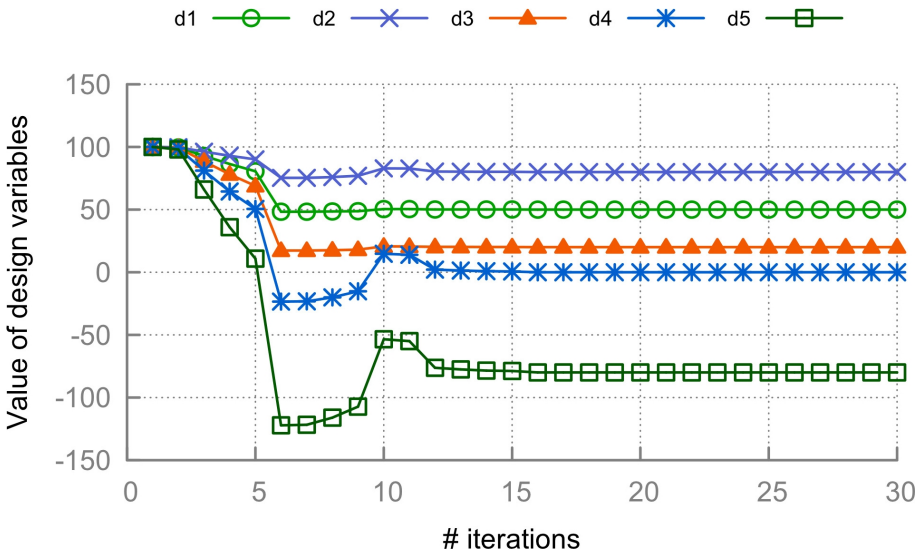

Figure 4.5: Evolution of 5 design variables as the optimization process advances using a large mesh (3.7M elements).The design variables start at initial values $\mathbf{d}^0 = (100, 100, 100, 100, 100)$ and converge to the target values $\mathbf{d}^{target} = (50, 80, 20, 0, -80)$.

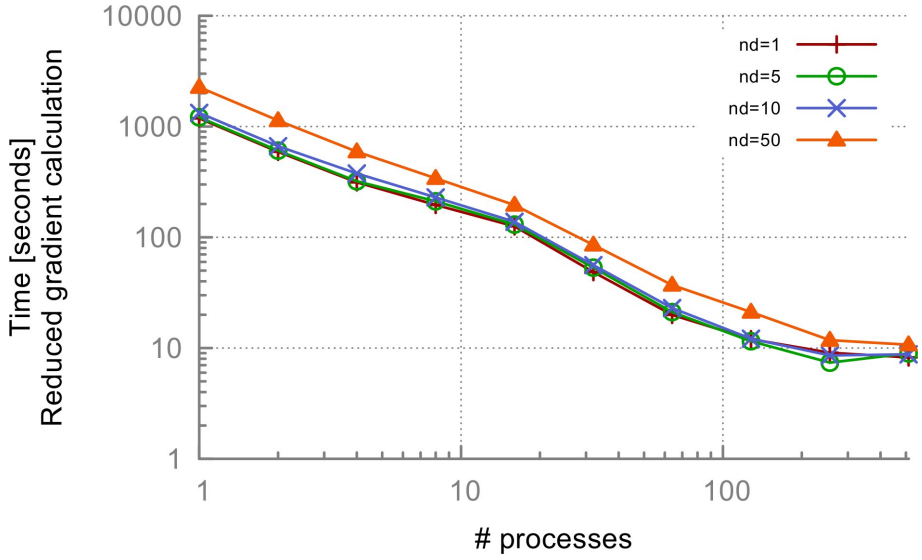

Figure 4.6: Execution time using a 2D mesh with 3.7M tetrahedral elements

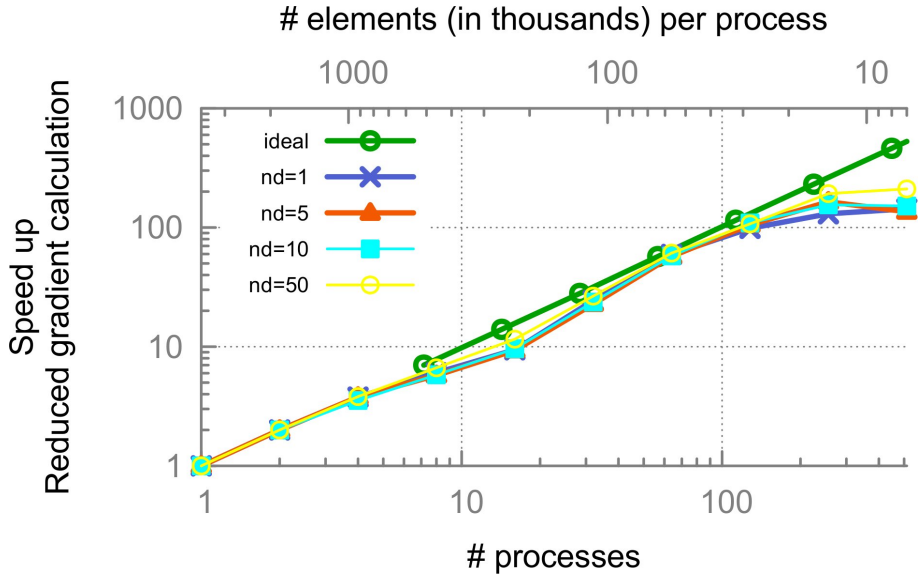

Figure 4.7: Speed-up results using a 2D mesh with 3.7M tetrahedral elements

4.2 Hydrocarbon exploration by 3D EM inversion

4.2.1 Problem formulation

Electromagnetic Modelling in Geophysics (extracted from [36])

Exploration geophysics is the applied branch of geophysics which uses various physical measurements to obtain information about the subsurface of the Earth that is not available from surface geological observation. By using surface methods, geophysicists measure physical data at the surface of the Earth in order to detect or infer the presence and position of ore minerals, hydrocarbons, geothermal reservoirs, groundwater reservoirs and other buried geological structures. The ultimate goal of a geophysical analysis is to build a constrained model of geology, lithology and fluid properties based upon which commercial decisions about reservoir exploration, development and management can be made. To achieve this, the Earth can be interrogated with a number of tools, such as seismic methods, controlled-source electromagnetic (CSEM) methods, magnetotellurics (MT), well-logging, magnetic methods, gravity methods, etc. Each of these techniques obtains a corresponding data type. Each data type must be interpreted or inverted within an integrated framework so that the resultant Earth model is consistent with all the data used in its construction.

Among all the above-mentioned methods, seismic has a special place. It is widely accepted that seismic methods are extremely powerful and generally applicable. They have become the hydrocarbon industries standard method for obtaining high-resolution images of structure and stratigraphy which can guide exploration, appraisal and development projects. However, there are some situations when seismic data fail to answer the geophysical questions of interest. In these situations, complementary sources of data must be used to obtain the required information. For example, seismic methods are very effective at mapping geological reservoir formations. However, because of extremely poor sensitivity of seismic properties to changes in a type of fluids, such as brine, water, oil and gas, in many situations it is difficult or even impossible to extract information on fluids trapped in the subsurface from seismic data. The fact that these established seismic methods are not good at recognizing different types of reservoir fluids contained in rock pores has encouraged the development of new geophysical techniques that can be combined with them in order to image fluids directly. A range of techniques, which have appeared recently and have shown considerable potential, use electromagnetic (EM) waves to map variations in subsurface electrical conductivity, σ (S/m), of oil and gas offshore prospects. For hydrocarbon imaging, a quantity that is more meaningful than electrical conductivity is its reciprocal, $1/\sigma$ (Ω m), called electrical resistivity, ρ. Namely, a resistivity measurement shows a high degree of sensitivity to fluid content. For example, in an oil-saturated region of a reservoir, the resistivity can be 1-2 orders of magnitude higher than in surrounding water-saturated sediments, where the variation in resistivity is small. Therefore, an increase of resistivity, in comparison with resistivity values of the surrounding geolog-

ical strata, may directly indicate potential hydrocarbon reservoirs. EM methods allow remote measurements of resistivity which is, as explained, extremely sensitive to properties and a distribution of fluids in a structure. In addition, the important contrast between resistivities of a hydrocarbon and a fluid like brine or water, makes EM-field measurements desirable for detecting hydrocarbon locations. Moreover, the resistivity information from EM surveys is complementary to seismic data and can improve the constraints on the fluid properties when used in an integrated geophysical interpretation. This is just an example of why EM methods have come to exploration geophysics to stay, and furthermore, of why they have been gaining increasing significance over the past decades.

In general, the use of EM exploration methods has been motivated by their ability to map electrical conductivity, dielectric permittivity and magnetic permeability. The knowledge of these EM properties is of great importance since they can be used in hydrological modelling, chemical and nuclear waste site evaluations, reservoir characterisation, as well as mineral, oil and gas exploration. Nowadays, there is a great diversity of EM methods, each of which has some primary field of application. However, many of them can be used in considerably wide range of different fields. For example, the EM mapping, on land, produces a resistivity map which can detect boundaries between different types of rocks and directly identify local three-dimensional (3-D) targets, such as base-metal mineral deposits, which are much more conductive than the host rocks in which they are found. This method is also used as a tool in a detection of sub-sea permafrost, as well as as a supplementary technique to seismic in offshore oil exploration. Furthermore, physical properties such as porosity, bulk density, water content and compressional wave velocity may be estimated from a profile of the electrical conductivity with depth.

The marine controlled-source electromagnetic (CSEM) method is nowadays a well-known geophysical exploration tool in the offshore environment and a commonplace in the industry (e.g. [19]). In CSEM, also referred to as seabed logging (SBL), the subseabed structure is explored by emitting low-frequency signals from a high-powered electric dipole (figure 4.8) source close to the seabed. By studying the received signal, thin resistive layers beneath the seabed could be detected at scales of a few tens of meters to depths of several kilometres. Operating frequencies of a transmitter in CSEM may range between 0.1 and 10 Hz, and the choice depends on dimensions of a model. In most studies, typical frequencies vary from 0.25 to 1 Hz, which means that for source-receiver offsets up to 10-12 km a penetration depth of the method can extend to several kilometres below the seabed. Figure 4.9 shows marine controlled-source EM method in combination with marine magnetotelluric method. The CSEM method has long been used to study the electrical conductivity of the oceanic crust and upper mantle. However, more recently, an intense commercial interest has arisen to apply the method to detect offshore hydrocarbon reservoirs. Also, the method has proven effective for characterization of gas hydrate-bearing shallow sediments. Moreover, during the last decade, CSEM has been considered as an important tool for reducing ambiguities in data interpretation and reducing exploratory risk.

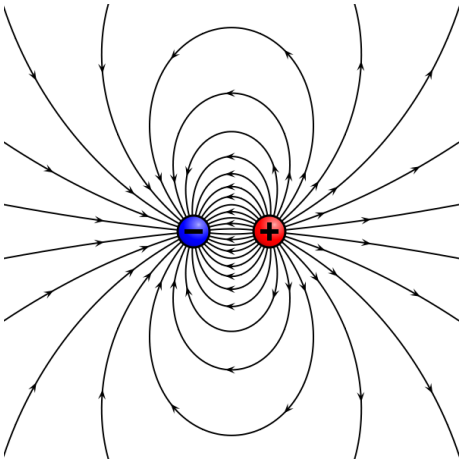

Figure 4.8: Electric field lines of two opposing charges separated by a finite distance (dipole)

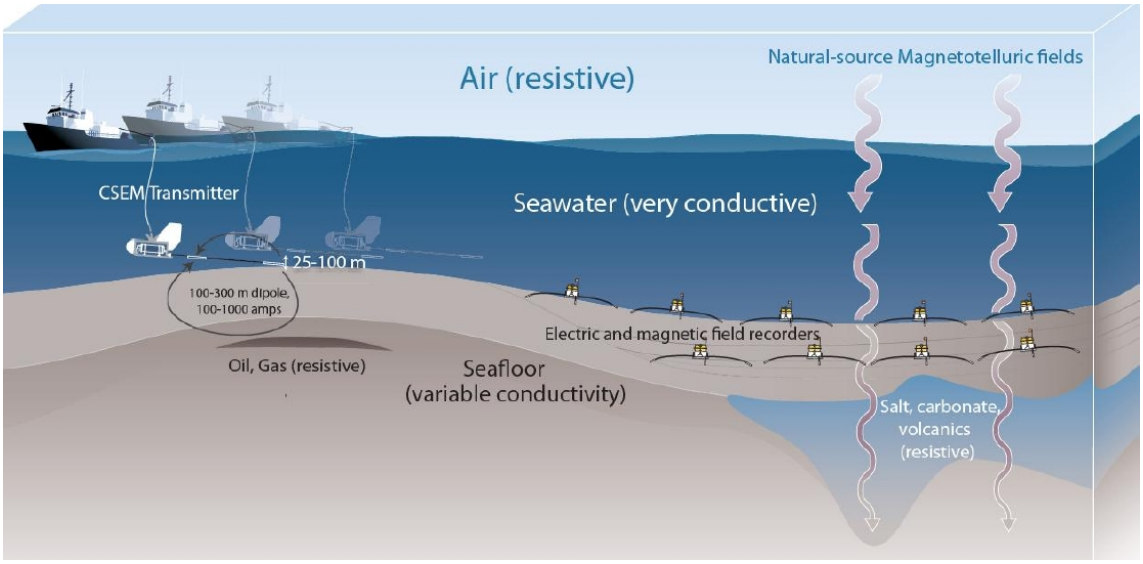

Figure 4.9: Marine Controlled-source Electromagnetic survey. The field recorders, or receivers, are typically placed in a grid formation. The transmitter dipole is towed by a vessel *shooting* EM pulses at a fixed frequency in an ordered configuration

EM Forward modeling

The forward model studied, described in [44] following ideas from [8], is formulated in terms of secondary Coulomb-gauged EM potentials $(\mathbf{A}_s, \nabla\phi_s)$ with Dirichlet boundary conditions in a 3D domain Ω as follows:

$$\nabla^2\mathbf{A}_s + i\omega\mu_0\sigma(\mathbf{A}_s + \nabla\phi_s) = -i\omega\mu_0\delta\sigma(\mathbf{A}_p + \nabla\phi_p) \tag{4.2.1}$$

$$\nabla\cdot(i\omega\mu_0\sigma(\mathbf{A}_s + \nabla\phi_s)) = -\nabla\cdot(i\omega\mu_0\delta\sigma(\mathbf{A}_p + \nabla\phi_p)) \tag{4.2.2}$$

The design variables in this problem are modeled as the variations $\delta\sigma$ in the electrical conductivity $\sigma(x,y,z) := \sigma_{base}(x,y,z) + \delta\sigma(x,y,z) \in \mathbb{R}^{3\times3}$ (diagonal).

Using a finite element discretization with N nodes, efficiently implemented in Alya (details of the implementation can be viewed in [36]), we can obtain a $4N \times 4N$ linear system as follows:

$$\mathbf{K}(\mathbf{d})\mathbf{z} = \mathbf{f}(\mathbf{d}) \tag{4.2.3}$$

with $\mathbf{d}$ as the discrete values of $\delta\sigma$ in the domain, $\mathbf{z} = (\mathbf{A}_s^1, \mathbf{A}_s^2, \mathbf{A}_s^3, \phi_s)^T \in \mathbb{C}^{4N}$, $\mathbf{K}(\mathbf{d}) \in \mathbb{C}^{4N\times4N}$ and $\mathbf{f}(\mathbf{d}) \in \mathbb{C}^{4N}$. We will refer to vector $\mathbf{z}$ as *state vector* and vector $\mathbf{d}$ as *design vector*, and the number of state variables as $n_u = 4N$ and the number of design variables as n_d. For example, assuming isotropy in the electrical conductivity and $\delta\sigma$ being uniform in all the domain (only one material), the number of design variables is $n_d = 1$. On the contrary, assuming anisotropy and $\delta\sigma := \delta\sigma(x,y,z)$ dependent on the spatial coordinates, the number of design variables can be $n_d \geq N$.

EM Inverse modeling

Using the previous notation, we can define a *misfit* or cost function $J : \mathbb{C}^{n_u}\times\mathbb{R}^{n_d} \to \mathbb{R}$ which will be used in a minimization process.

$$\begin{aligned} J(\mathbf{z},\mathbf{d}) &= \sum_{k\in OBS} \delta\overline{(z_k - z_k^{obs})}(z_k - z_k^{obs}) \\ &= \overline{(\mathbf{z}-\mathbf{z}^{obs})}^T\mathbf{M}^{obs}(\mathbf{z}-\mathbf{z}^{obs}) \end{aligned} \tag{4.2.4}$$

where OBS is the set of nodal indexes with observations, M^{obs} is a diagonal real matrix of size $4N \times 4N$ with $M_k^{obs} = \delta > 0$ if $k \in OBS$ and $M_k^{obs} = 0$ if $k \notin OBS$. With the equations (4.2.3) and (4.2.4) we can define the following minimization problem:

$$\begin{array}{ll} \underset{(\mathbf{z},\mathbf{d})\in\mathbb{C}^{n_u}\times\mathbb{R}^{n_d}}{\text{minimize}} & J(\mathbf{z},\mathbf{d}) \\ \text{subject to} & \mathbf{K}(\mathbf{d})\mathbf{z} = \mathbf{f}(\mathbf{d}) \end{array} \tag{4.2.5}$$

As explained in previous chapters, we can consider the linear constraints as a linear function $\mathbf{R}(\mathbf{z},\mathbf{d}) = \mathbf{K}(\mathbf{d})\mathbf{z} - \mathbf{f}(\mathbf{d})$ and the problem takes the form of (2.1.1). For technical

reasons, detailed in [14], we need to define the cost function and the constraint function in terms of $\mathbf{z}$ and its conjugate $\overline{\mathbf{z}}$, so the constrained problem becomes:

$$\begin{array}{cll} \underset{(\mathbf{z},\overline{\mathbf{z}},\mathbf{d})\in\mathbb{C}^{n_u}\times\mathbb{C}^{n_u}\times\mathbb{R}^{n_d}}{\text{minimize}} & J(\mathbf{z},\overline{\mathbf{z}},\mathbf{d}) & \\ \text{subject to} & \mathbf{R}(\mathbf{z},\overline{\mathbf{z}},\mathbf{d}) & = \mathbf{0}_{s\times 1} \end{array} \tag{4.2.6}$$

And finally if we consider the dependency of $\mathbf{z}$ on $\mathbf{d}$ (because the linear system has unique solution), $\mathbf{z} := \mathbf{z}(\mathbf{d})$, the constrained optimization problem becomes an unconstrained optimization problem with a new cost function $j : \mathbb{R}^{n_d} \to \mathbb{R}$ as defined in (2.1.2). The next step is the calculation of $\nabla_{\mathbf{d}}\, j(\mathbf{d})$ and this is done as follows:

$$\begin{aligned} \nabla_{\mathbf{d}}\, j(\mathbf{d}) &= \nabla_{\mathbf{d}}\, J(\mathbf{z}(\mathbf{d}),\overline{\mathbf{z}}(\mathbf{d}),\mathbf{d}) \\ &= \nabla_{\mathbf{z}} J\cdot\nabla_{\mathbf{d}}\mathbf{z} + \nabla_{\overline{\mathbf{z}}} J\cdot\nabla_{\mathbf{d}}\overline{\mathbf{z}} + \nabla_{\mathbf{d}} J \\ &= \overline{(\mathbf{z}-\mathbf{z}^{obs})}^{T}\mathbf{M}^{obs}\nabla_{\mathbf{d}}\mathbf{z} + \\ &\quad (\mathbf{z}-\mathbf{z}^{obs})^{T}\mathbf{M}^{obs}\nabla_{\mathbf{d}}\overline{\mathbf{z}} + 0 \end{aligned} \tag{4.2.7}$$

As we saw in the example of the introductory section, we will avoid the terms $\nabla_{\mathbf{d}}\mathbf{z}$ and $\nabla_{\mathbf{d}}\overline{\mathbf{z}}$ using the implicit function theorem described in appendix A.2 applied to $\mathbf{R}(\mathbf{z},\overline{\mathbf{z}},\mathbf{d})$ (the steps to obtain them are essentially the same as in level-3 optimization of section 2.1.1):

$$\nabla_{\mathbf{d}}\mathbf{z} = -\mathbf{K}(\mathbf{d})^{-1}\underbrace{\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{z}(\mathbf{d}),\overline{\mathbf{z}}(\mathbf{d}),\mathbf{d})}_{\nabla_{\mathbf{d}}\mathbf{R}} \tag{4.2.8}$$

An equivalently for its conjugate $\overline{\mathbf{R}(\mathbf{z}(\mathbf{d}),\overline{\mathbf{z}}(\mathbf{d}),\mathbf{d})} = \overline{\mathbf{K}(\mathbf{d})}\overline{\mathbf{z}} - \overline{\mathbf{f}(\mathbf{d})}$:

$$\nabla_{\mathbf{d}}\overline{\mathbf{z}} = -\left[\overline{\mathbf{K}(\mathbf{d})}\right]^{-1}\underbrace{\nabla_{\mathbf{d}}\overline{\mathbf{R}(\mathbf{z}(\mathbf{d}),\overline{\mathbf{z}}(\mathbf{d}),\mathbf{d})}}_{\nabla_{\mathbf{d}}\overline{\mathbf{R}}} \tag{4.2.9}$$

Using the expressions in (4.2.8) and (4.2.9), the derivative of the cost function j can be expressed as:

$$\begin{aligned} \nabla_{\mathbf{d}}\, j &= -\overline{(\mathbf{z}-\mathbf{z}^{obs})}^{T}\mathbf{M}^{obs}\mathbf{K}(\mathbf{d})^{-1}\nabla_{\mathbf{d}}\mathbf{R} + \\ &\quad -(\mathbf{z}-\mathbf{z}^{obs})^{T}\mathbf{M}^{obs}\left[\overline{\mathbf{K}(\mathbf{d})}\right]^{-1}\nabla_{\mathbf{d}}\overline{\mathbf{R}} \end{aligned} \tag{4.2.10}$$

In order to obtain the adjoint system and the final expression of the previous derivative, we need to calculate the Karush-Kuhn-Tucker conditions (appendix A.3) for the constrained optimization problem (4.2.6) using the following real-valued Lagrangian function (in appendix A.4 we can find a detailed justification of this expression for the Lagrangian):

$$L(\mathbf{z},\overline{\mathbf{z}},\mathbf{d},\boldsymbol{\lambda}) = J(\mathbf{z},\overline{\mathbf{z}},\mathbf{d}) - 2\,\mathrm{Re}\left\{\boldsymbol{\lambda}^{T}\,\overline{\mathbf{R}(\mathbf{z},\overline{\mathbf{z}},\mathbf{d})}\right\} \tag{4.2.11}$$

the adjoint system can be expressed as $\overline{\mathbf{K}(\mathbf{d})}^T \boldsymbol{\lambda} = \mathbf{M}^{obs}(\mathbf{z} - \mathbf{z}^{obs})$ or equivalently

$$\mathbf{K}(\mathbf{d})^T \overline{\boldsymbol{\lambda}} = \mathbf{M}^{obs}\overline{(\mathbf{z} - \mathbf{z}^{obs})} \tag{4.2.12}$$

and the derivative of the cost function j as:

$$\nabla_{\mathbf{d}}\, j(\mathbf{d}) = -2\,\mathrm{Re}\left\{\overline{\boldsymbol{\lambda}}^T \nabla_{\mathbf{d}}\mathbf{R}(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d})\right\} \tag{4.2.13}$$

To handle the positiveness of the conductivity $\sigma = \sigma_{base} + \delta\sigma > 0$ we choose a natural logarithmic transformation $\gamma_i = \ln(\sigma_{base} + d_i)$, with $d_i \in (-\sigma_{base}, +\infty]$ and $\gamma_i \in \mathbb{R}$ for all i. With this transformation, we need to incorporate the derivative of the inverse change of variable in the final version of $\nabla_\gamma\, j$ to get a well-posed unconstrained minimization problem:

$$\nabla_\gamma\, j(\gamma) = -2\,\mathrm{Re}\left\{\overline{\boldsymbol{\lambda}}^T \nabla_{\mathbf{d}}\mathbf{R}(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d})\right\} \cdot \nabla_\gamma \mathbf{d}(\gamma) \tag{4.2.14}$$

The reduced gradient of equation (4.2.14) represents the sensitivity using only one transmitter and several receivers. As we mentioned previously, the electrical dipole is towed around a predefined configuration, which can be interpreted as if several transmitters were sending pulsations in different parts of the seabed. Mathematically, we need to calculate a multiple shot cost function as:

$$\begin{aligned} J(\mathbf{z}, \mathbf{d}) &= \sum_{i=1}^{SHOTS} J_i(\mathbf{z}, \mathbf{d}) \\ &= \sum_{i=1}^{SHOTS} \sum_{k \in OBS_i} \overline{\delta(z_k - z_k^{obs_i})}(z_k - z_k^{obs_i}) \\ &= \sum_{i=1}^{SHOTS} \overline{(\mathbf{z} - \mathbf{z}^{obs_i})}^T \mathbf{M}^{obs_i}(\mathbf{z} - \mathbf{z}^{obs_i}) \end{aligned} \tag{4.2.15}$$

with its corresponding adjoint system using several transmitters:

$$\mathbf{K}(\mathbf{d})^T \overline{\boldsymbol{\lambda}} = \sum_{i=1}^{SHOTS} \mathbf{M}^{obs_i}\overline{(\mathbf{z} - \mathbf{z}^{obs_i})} \tag{4.2.16}$$

Unfortunately, our current implementation, based in Alya, only supports one transmitter at a time, so an equivalent formulation, is as follows:

$$\mathbf{K}(\mathbf{d})^T \overline{\boldsymbol{\lambda}_i} = \mathbf{M}^{obs_i}\overline{(\mathbf{z} - \mathbf{z}^{obs_i})},\ \forall i = 1, \ldots, SHOTS \tag{4.2.17a}$$

$$\overline{\boldsymbol{\lambda}} = \sum_{i=1}^{SHOTS} \overline{\boldsymbol{\lambda}}_i \tag{4.2.17b}$$

This formulation implies (4.2.16) because matrix-vector product is a linear operation on vectors, i.e.

$$\begin{aligned}\mathbf{K}(\mathbf{d})^T\,\overline{\boldsymbol{\lambda}} &= \mathbf{K}(\mathbf{d})^T \sum_{i=1}^{SHOTS} \overline{\boldsymbol{\lambda}}_i \\ &= \sum_{i=1}^{SHOTS} \mathbf{K}(\mathbf{d})^T\,\overline{\boldsymbol{\lambda}}_i \\ &= \sum_{i=1}^{SHOTS} \mathbf{M}^{obs_i}\overline{(\mathbf{z}-\mathbf{z}^{obs_i})} \end{aligned} \tag{4.2.18}$$

Using the alternative formulation, the reduced gradient of equation (4.2.14) is as follows:

$$\begin{aligned}\nabla_\gamma\, j(\gamma) &= -2\,\mathrm{Re}\left\{\overline{\boldsymbol{\lambda}}^T\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{z},\overline{\mathbf{z}},\mathbf{d})\right\}\cdot\nabla_\gamma\mathbf{d}(\gamma) \\ &= -2\,\mathrm{Re}\left\{\left(\sum_{i=1}^{SHOTS}\overline{\boldsymbol{\lambda}}_i\right)^T\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{z},\overline{\mathbf{z}},\mathbf{d})\right\}\cdot\nabla_\gamma\mathbf{d}(\gamma) \\ &= \sum_{i=1}^{SHOTS} -2\,\mathrm{Re}\left\{\overline{\boldsymbol{\lambda}}_i^T\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{z},\overline{\mathbf{z}},\mathbf{d})\right\}\cdot\nabla_\gamma\mathbf{d}(\gamma) \\ &= \sum_{i=1}^{SHOTS}\nabla_\gamma\, j_i(\gamma) \end{aligned} \tag{4.2.19}$$

The multiple shot formulation of equation (4.2.19) tells us that we need to calculate $SHOTS$ reduced gradients, in order to obtain the overall reduced gradient $\nabla_\gamma\, j(\gamma)$. This new feature has several software implications, in the sense that some sequence diagrams of chapter 3 can be modified, and new subroutines must be created in order to support the new requirements of this particular application.

4.2.2 Parameters and system settings

Physical and computational mesh parameters

In order to test our implementation, we design a synthetic scenario of isotropic electrical conductivity with a 3D mesh depicted in figures 4.10, 4.11 and 4.12 (1,059,738 nodal points and 4,765,287 tetrahedral elements) with two main regions: the marine subsurface $\{(x,y,z)\in\Omega : z\leq 0\}$ with $\sigma_{subsurface} = 1.2$ and water on top of the subsurface with $\sigma_{water} = 3.3$. The transmitters correspond to horizontal electric dipole of length 1.0 meter and current 1 A, placed 50 meters above the interface separating the subsurface and the water regions. In figure 4.11 we can see the local mesh refinements in the line of transmitters. This local refinements are needed in order to have good numerical behaviour

in the forward problem resolution. A line of receivers is displayed at the seabed, in the same line of the trasmitters, using a separation step of 500 meters (21 receivers). We can observe that the mesh is strongly regular at the subsurface. This is due to the fast propagation of the electric fields, that are affected by three mechanisms depending on the offset transmitter-receiver: geometric spreading from a dipole, galvanic change in electric field as current crosses a conductivity boundary, and inductive attenuation (see [19] for more details). However, optimal meshes have a local refinement depending on the depth. The closer to the seabed, the finer the mesh elements. Unfortunately the complexity of this mesh design is out of the scope of our work, and further tests are needed in order to use optimal meshes.

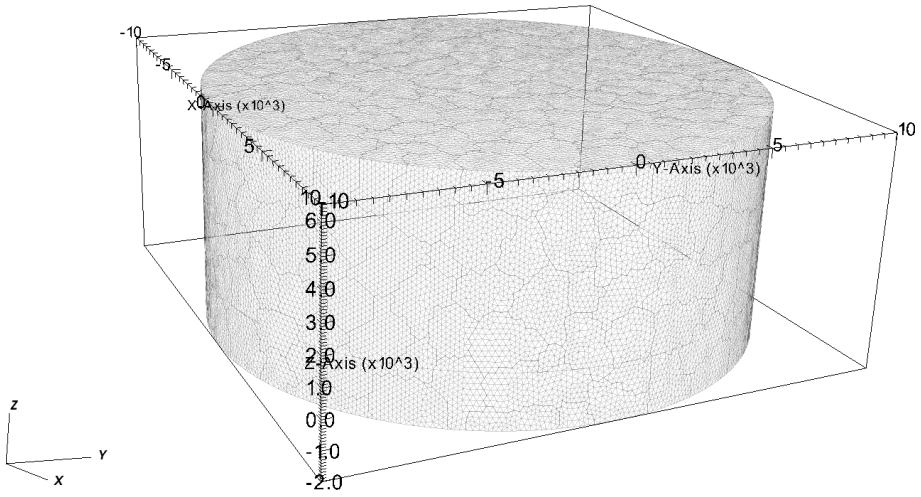

Figure 4.10: 3D mesh of the synthetic example of CSEM inversion

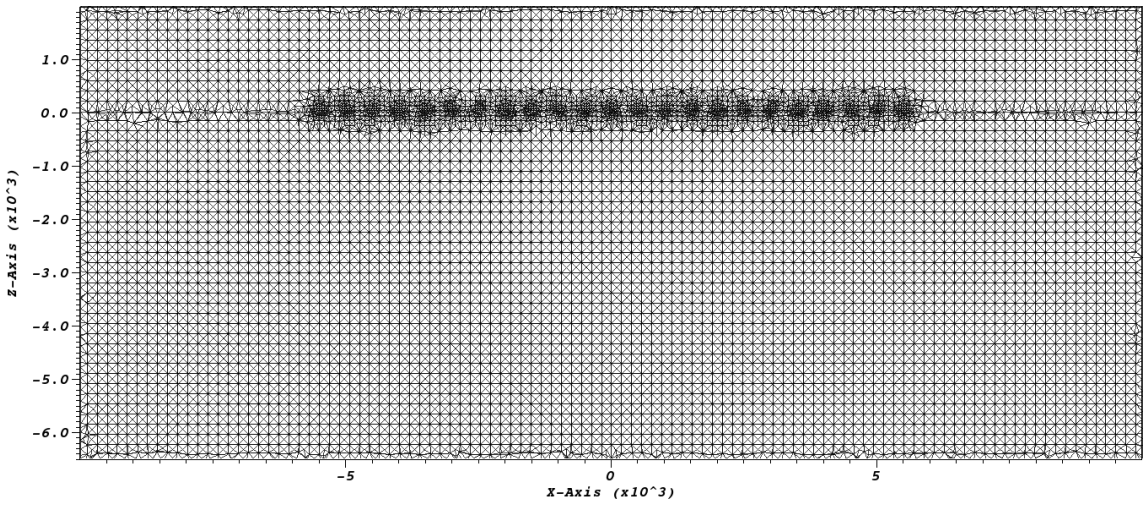

Figure 4.11: 3D mesh of the synthetic example of CSEM inversion, slice in Y-plane at the origin

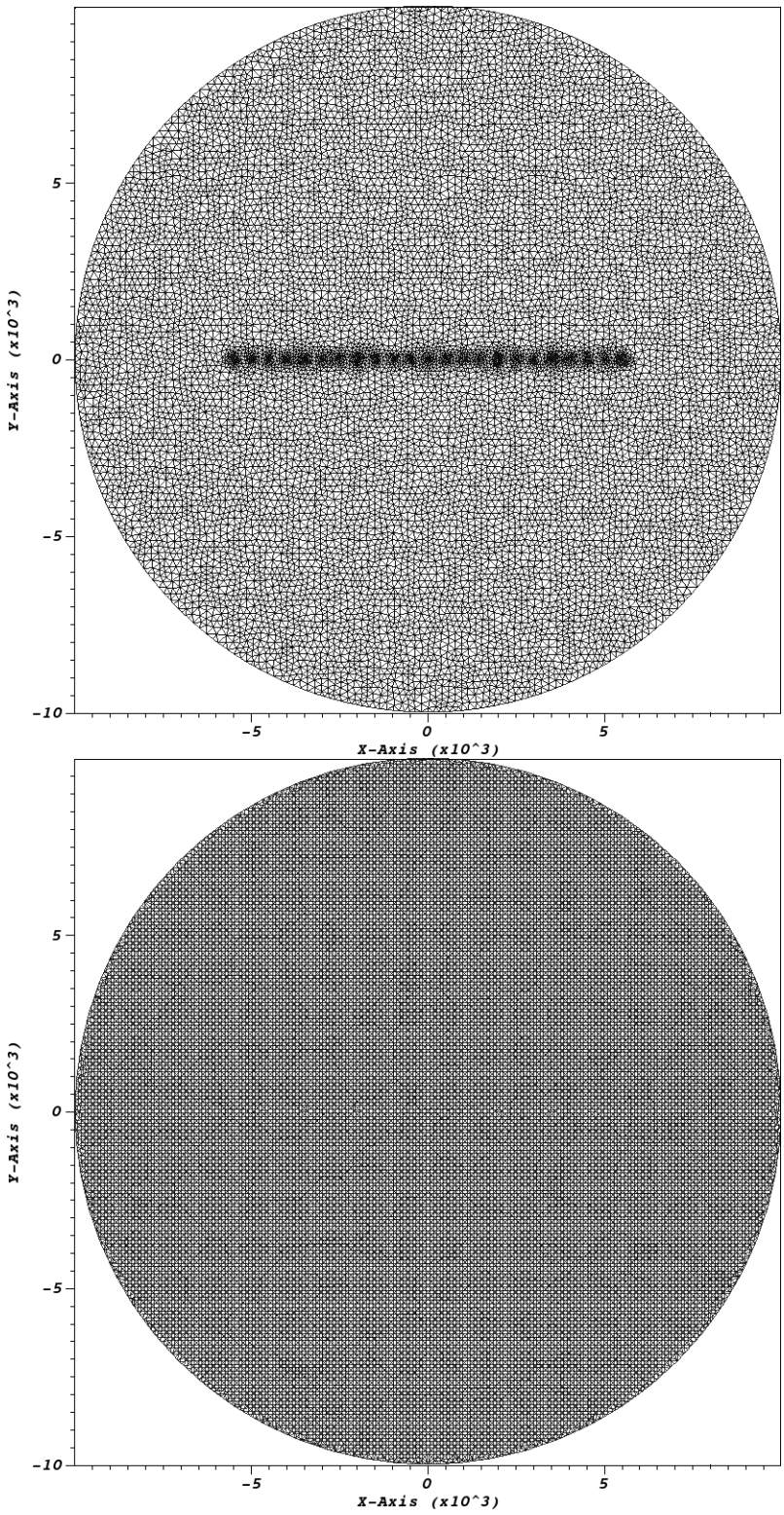

Figure 4.12: 3D mesh of the synthetic example of CSEM inversion, slice in Z-plane at $z = 0$ (top) and $z = -500$ (bottom)

The design variables correspond to the value of the electric conductivity in each node of the mesh. However, it is not necessary to calculate the reduced gradient in each nodal point, because there are regions of the domain which are not interesting from the point of view of the inversion calculation. For this reason, we define a *region of interest*, which is a box located below the seabed, at some defined depth. The size of the region of interest is variable, and can range from 1 nodal point to all mesh nodes, i.e. $n_d \in \{1, \ldots, 1059738\}$. The only limitation of the size of $\mathbf{d}$ is the physical memory of each compute node. In this test, the maximum size of n_d delivers approximately 8Mb of space, so there are no memory space problems. This issue must be considered if larger design variable spaces must be modeled.

Hardware settings

The number of processes (CPUs) available for each run were 16, 32, 64, 128 and 256. A distributed-memory cluster of compute nodes called Marenostrum-III was used to execute the tests. It is based on 3028 compute nodes of Intel SandyBridge-EP E5-2670 2x 8-cores CPUs at 2.6 GHz connected through Infiniband and Gigabit Ethernet interconnection networks, with 32GB of memory per node. We set the tests to allow 2 MPI processes per compute node, in order to get fair results introducing less possible contentions or accelerations due to intra-node communications. This is intended to help in some cases where intra-node communications generates several contention incidents which slow down the obtained measurements. It is also assumed that the interconnection network generates the same amount of contention incidents in every inter-node communication.

Iterative solver parameters

The iterative method chosen to solve forward and adjoint complex-valued linear systems was BiCGSTAB using Jacobi ($\mathbf{P} = diag(\mathbf{A}_p)$) as preconditioner, with initial values of $\mathbf{z}^0 = \mathbf{0}$ and $\boldsymbol{\lambda}^0 = \mathbf{0}$ in both problems respectively. The parameters *maximum number of iterations* and *residual convergence tolerance* ϵ of BiCGSTAB (measured as $r_k = \|\mathbf{P}^{-1}(\mathbf{f} - \mathbf{K}\mathbf{z}^k)\|$) are setted as 5000 iterations and $\epsilon = 10^{-10}$ respectively.

Optimization parameters

The optimization method implemented is a steepest descent ($\mathbf{p} = -\nabla_{\mathbf{p}} j$) with tolerance $\epsilon_{grad} = 10^{-35}$, and a maximum of 5 optimization iterations and a line-search with a maximum of 20 iterations. These parameters are intended to be used in a basic scenario, but in the next subsection (*Inversion results*) we will describe the particular features of each optimization execution.

4.2.3 Results

Forward simulations

First of all, it will be useful to visualize samples of the forward simulation, in order to get a priori information about the state variable values and how they can affect the current cost function.

In figures 4.13, 4.14 and 4.15 we can see samples of simulated secondary vector and scalar potentials $(\mathbf{A}_s, \phi_s)$ using a transmitter source with frequency $f = 0.2$Hz, current 1A and located at position $(1500, 0, -50)$. Each potential has real and imaginary components, which have different behaviours, as we can see in the previous figures. These potentials are defined as our state variables, stored in the state vector $\mathbf{z}$. An interesting property regarding the values of each potential component can be observed in the scale values of figures 4.13, 4.14 and 4.15. We can observe that the real and imaginary parts of $\mathbf{A}_s$ and the real and imaginary parts of ϕ_s have values of order 10^{-10}, 10^{-11}, 10^{-9} and 10^{-7} respectively. This orders of magnitude are replicated, with more or less accuracy, in each test that we have run. The fact that the secondary scalar potential ϕ_s is 2-3 orders of magnitude larger than the secondary vector potential $\mathbf{A}_s$ tells us that we have to modify the cost function in order to measure values of the same order of magnitude. For this reason, we have chosen to discard ϕ_s from the observed data of our cost function.

In figure 4.16 we can see a sample of our design variables, represented as the logarithm of electric conductivity $\ln(\sigma)$, stored in the design vector $\boldsymbol{\gamma}$. In this figure we can see the different values of σ for each different material in the simulation. Also, we can see how a component of the state variables, the imaginary part of $\mathbf{A}_s$ denoted $\mathrm{Im}\{\mathbf{A}_s\}$, traverses through the anomaly, represented by a black region with $\sigma = 0.5$ $(\ln(\sigma) \approx -0.6931)$ embedded into the subsurface with $\sigma = 1.2$ $(\ln(\sigma) \approx 0.1823)$ and the sea water on top with $\sigma = 3.3$ $(\ln(\sigma) \approx 1.1939)$.

Based on the previous observation, we have decided to use only the imaginary part of $\mathbf{A}_s^1$, denoted $\mathrm{Im}\{\mathbf{A}_s^1\}$, as our unique observable measurement. With this decision, if the state vector is defined as $\mathbf{z} = \mathbf{A}_s^1$, the associated cost function is:

$$\begin{aligned} J(\mathbf{z}, \mathbf{d}) &= \sum_{k \in OBS} \delta(\mathrm{Im}\, z_k - \mathrm{Im}\, z_k^{obs})^2 \\ &= \mathrm{Im}(\mathbf{z} - \mathbf{z}^{obs})^T \mathbf{M}^{obs}\, \mathrm{Im}(\mathbf{z} - \mathbf{z}^{obs}) \\ &= \frac{-1}{4}\Big[(\mathbf{z} - \mathbf{z}^{obs})^T \mathbf{M}^{obs}(\mathbf{z} - \mathbf{z}^{obs}) + \overline{(\mathbf{z} - \mathbf{z}^{obs})}^T \mathbf{M}^{obs}\overline{(\mathbf{z} - \mathbf{z}^{obs})} \\ &\quad - \overline{(\mathbf{z} - \mathbf{z}^{obs})}^T \mathbf{M}^{obs}(\mathbf{z} - \mathbf{z}^{obs}) - (\mathbf{z} - \mathbf{z}^{obs})^T \mathbf{M}^{obs}\overline{(\mathbf{z} - \mathbf{z}^{obs})}\Big] \end{aligned} \quad (4.2.20)$$

This definition is based in the identity $\mathrm{Im}\, z = \frac{z - \overline{z}}{2i}$. The corresponding derivative w.r.t. state vector is:

$$\begin{aligned} \nabla_{\mathbf{z}} J(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}) &= \frac{-1}{4}\left\{2\mathbf{M}^{obs}(\mathbf{z} - \mathbf{z}^{obs}) - 2\mathbf{M}^{obs}\overline{(\mathbf{z} - \mathbf{z}^{obs})}\right\} \\ &= -i\mathbf{M}^{obs}\, \mathrm{Im}\,(\mathbf{z} - \mathbf{z}^{obs}) \end{aligned} \quad (4.2.21)$$

With this derivative, the adjoint problem associated to the cost function of equation (4.2.20) is as:

$$\mathbf{K}(\mathbf{d})^T \overline{\boldsymbol{\lambda}} \;=\; i\mathbf{M}^{obs}\,\mathrm{Im}\,(\mathbf{z}-\mathbf{z}^{obs}) \tag{4.2.22}$$

The algebra needed to obtain $\nabla_{\boldsymbol{\gamma}} j(\boldsymbol{\gamma})$ is the same as we explained in equation (4.2.14) (with 1 shot) and (4.2.19) (with multiple shots).

As we mentioned in the introductory paragraphs, CSEM inversion is used as a complementary technique to characterize fluids in the subsurface, mapping the electric conductivity of each material. A typical scenario consists in the generation of subsurface images using alternative techniques, such as seismic imaging. These images can give accurate information on the geometrical and geological description of the anomalies in the subsurface, however they doesn't get depth insight about the actual material properties of the anomalies. Using as initial model the seismic images with ideal conductivity values, CSEM inversion can determine if the anomalies correspond to hydrocarbon resources, sub-salt reservoirs or other materials.

In terms of software design, the proposed changes of equations (4.2.20), (4.2.21) and (4.2.22) are translated in the development of two new routines `physics_costf_v2()` and `physics_dcost_v2()` which calculate the new cost function value and its derivate w.r.t. state vector, respectively. The rest of the software design is not affected by this modification, in fact, the software design supports several cost functions with their corresponding derivatives, in a way that is clean and transparent to the final user.

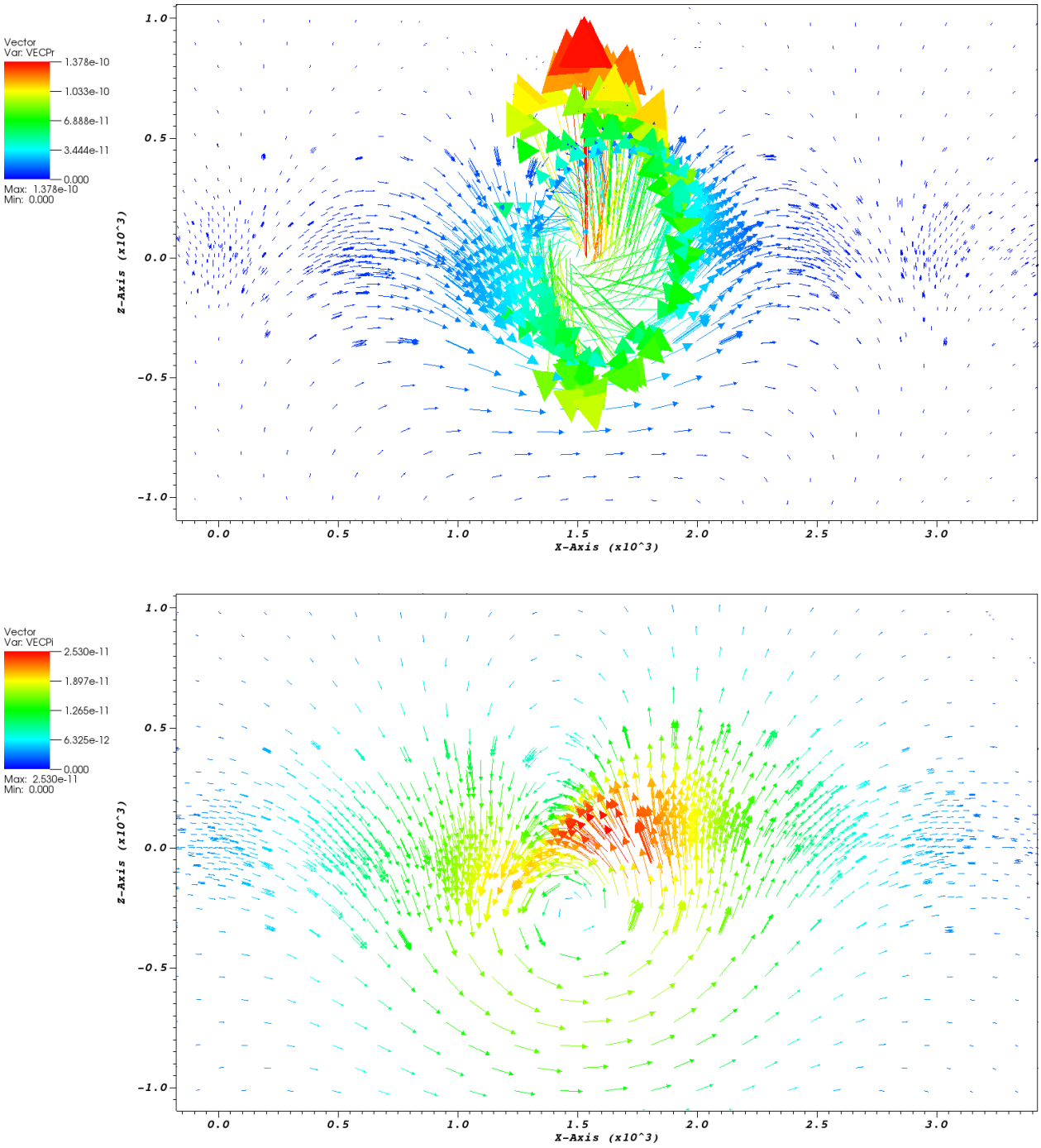

Figure 4.13: Sample of real and imaginary parts of secondary vector potential $\mathbf{A}_s$ (top and bottom respectively) generated by a transmitter source with frequency $f = 0.2$Hz, current 1A and located at $(1500, 0, -50)$. Slices in plane Y at $y = 0$

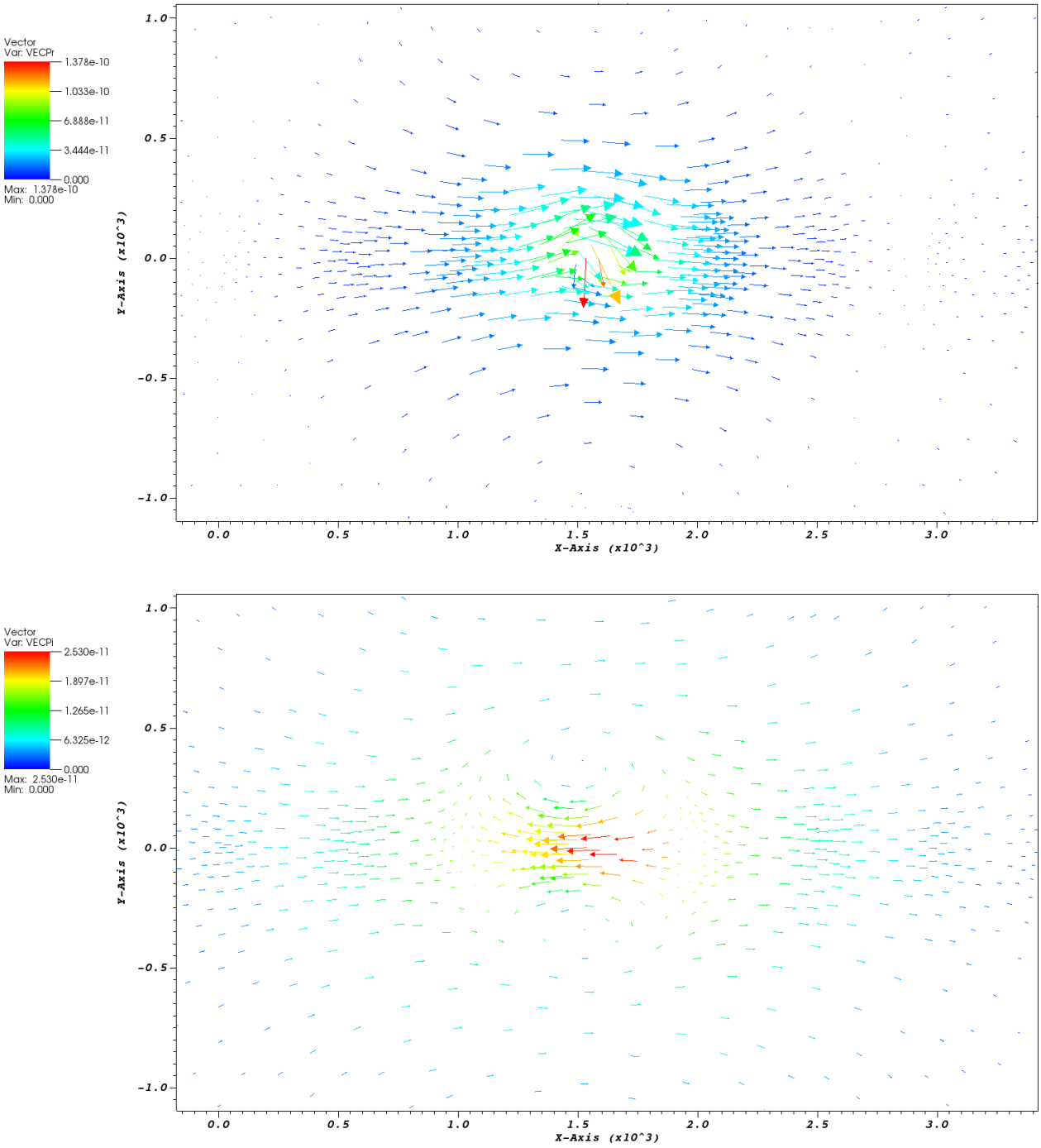

Figure 4.14: Sample of real and imaginary parts of secondary vector potential $\mathbf{A}_s$ (top and bottom respectively) generated by a transmitter source with frequency $f = 0.2$Hz, current 1A and located at $(1500, 0, -50)$. Slices in plane Z at $z = 0$

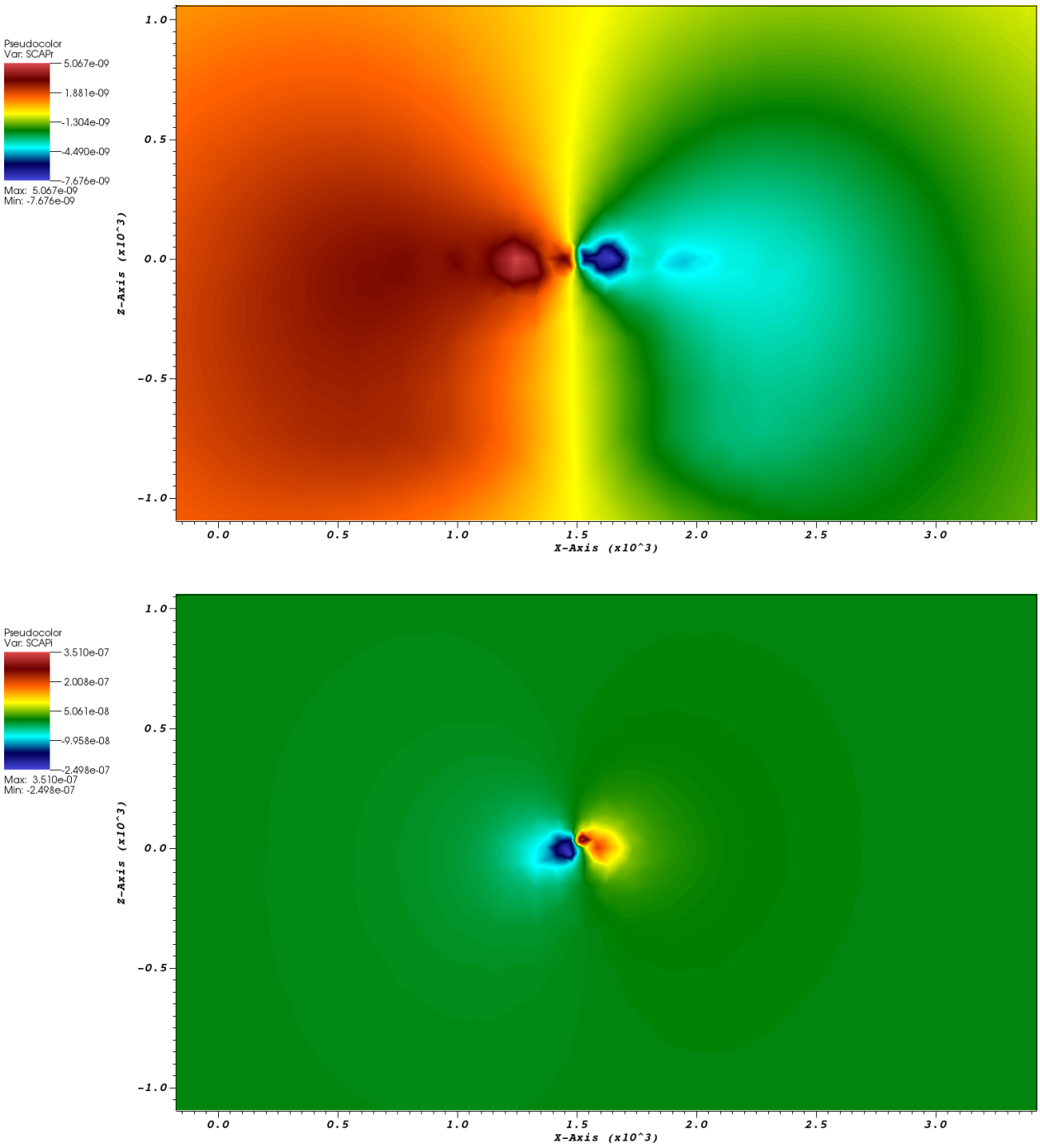

Figure 4.15: Sample of real and imaginary parts of secondary scalar potential ϕ_s (top and bottom respectively) generated by a transmitter source with frequency $f = 0.2$Hz, current 1A and located at $(1500, 0, -50)$. Slices in plane Y at $y = 0$

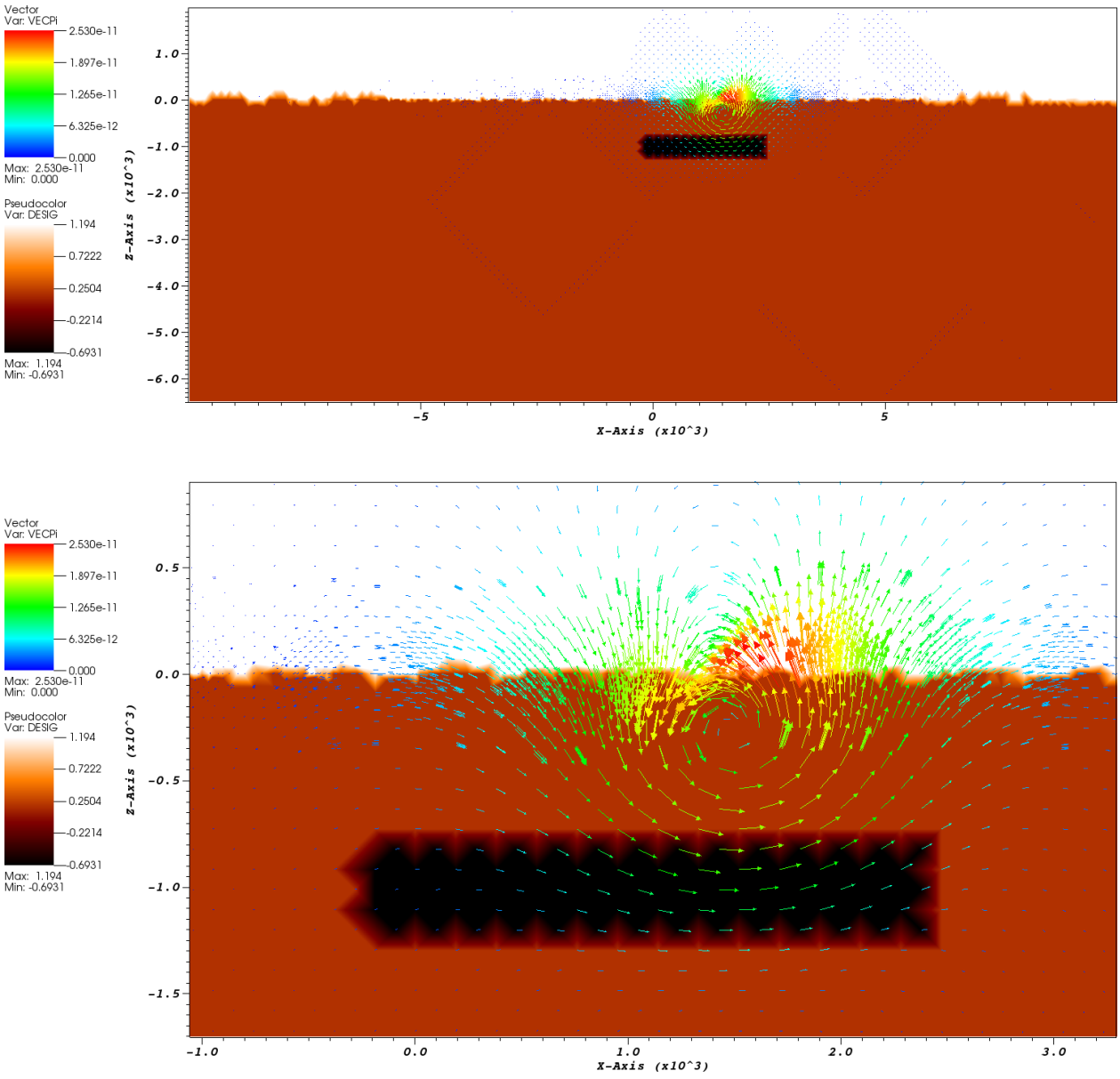

Figure 4.16: Sample of a design variable (top: the whole domain; bottom: zoom in the anomaly region), represented by an electric conductivity model, under the effect of an electric field, represented by the imaginary part of the secondary vector potential $\mathbf{A}_s$ generated by a transmitter source with frequency $f = 0.2$Hz, current 1A and located at $(1500, 0, -50)$. Slice in plane Y at $y = 0$

Sensitivity analysis

Before start with a full optimization process, a sensitivity analysis is required. This analysis involves the study of the reduced gradient before applying the descent direction algorithms. According to equation (4.2.19), for each i-th transmitter, also denoted as *shot*, we need to calculate the reduced gradient $\nabla_{\boldsymbol{\gamma}} j_i(\boldsymbol{\gamma})$ using the algorithm 4 from section 3.2, and add it to the accumulated reduced gradient. If several design variables are included in the region of interest and several transmitters are involved in the simulation, this process can take a considerable amount of execution time. For this reason, we need to check if the first reduced gradient gives us useful information. The negative reduced gradient, $-\nabla_{\boldsymbol{\gamma}} j(\boldsymbol{\gamma})$, acts as a basic source of information, because it represents the steepest descent direction. If the sign of the steepest descent direction indicates a decrement or increment in some nodal values of the electric conductivity, we can infer that the cost function will be reduced and a successful line-search can be performed.

In this part, a *target model* refers to a defined geological scenario embedded in the subsurface. A typical target model involves an *anomaly* placed somewhere below the seabed, with fixed values of electric conductivity in all its internal nodal points. Our aim is to discover an unknown target model, starting from an initial model and using data observed in the receivers generated by each shot individually. A *starting* or *initial model* refers to an initial geological configuration, that works as our initial guess of the contents in the sub-seabed. Several starting models need to be tested, in order to obtain a *good* starting design vector $\boldsymbol{d}^0$ in the optimization process. The selection of the starting model is out of the scope of our research, so we will use simple and synthetic target and starting models, studying the sign of the reduced gradient inside of the anomaly.

In figures 4.17 and 4.18 we can see two target models, the first with a homogeneous electric conductivity and the second with two regions of different values. Both anomalies have the same dimensions $[-2400, 2400] \times [-400, 400] \times [-600, -1200]$, located at depth 600 meters below the seabed.

In figure 4.19 we can see the reduced gradient obtained using 4 different transmitters with frequency 0.2Hz and current 1A, with target model depicted in figure 4.17 (homogeneous electric conductivity) and initial model with the same geometry but different electric conductivity, with value $\sigma_{start} = 1.1$ or $\ln(\sigma_{start}) \approx 0.0953$. From the point of view of the final user (a geophysicist or a geologist), he/she sets an initial guess of electric conductivity with value $\sigma_{start} = 1.1$, without knowing the true value σ_{target}, hoping that $\sigma_{start} \approx \sigma_{target}$ holds. The user runs the reduced gradient algorithm, in which the cost function measures the difference between the observed values (with $\sigma_{target} = 0.01$) and simulated values (with $\sigma_{start} = 1.1$), and also a 3D image of $\nabla_{\boldsymbol{\gamma}} j(\boldsymbol{\gamma})$ is generated (2D slices depicted in figure 4.19). In the last picture of this figure, we can see the accumulated value of the reduced gradient: $\nabla_{\boldsymbol{\gamma}} j(\boldsymbol{\gamma}) = \sum_{i=1}^{4} \nabla_{\boldsymbol{\gamma}} j_i(\boldsymbol{\gamma})$. The values of this gradient through the line labeled with the letter *E* are depicted in figure 4.22. We can see that all of these values are positive, which indicate that the steepest descent direction $-\nabla_{\boldsymbol{\gamma}} j(\boldsymbol{\gamma})$ is negative and possibly a line-search using this direction will decrease the values of σ_{start}

inside of the anomaly. This result is in concordance with our expectations, because the target model has a lower value of σ in the anomaly, so we need to decrease the initial guess σ_{start}.

In figure 4.20 we can see the reduced gradient obtained using 4 different transmitters with frequency 0.2Hz and current 1A, with target model depicted in figure 4.17 (homogeneous electric conductivity) and initial model with the same geometry but different electric conductivity, with value $\sigma_{start} = 0.001$ or $\ln(\sigma_{start}) \approx -6.9077$. This scenario is the opposite of the one described in the previous paragraph, because now the initial guess is lower than the target value $\sigma_{target} = 0.01$. As mentioned before, the final user sets an initial guess of electric conductivity with value $\sigma_{start} = 0.001$, without knowing the true value σ_{target}, hoping that $\sigma_{start} \approx \sigma_{target}$ holds. In the last picture of figure 4.20 we can see the accumulated value of the reduced gradient. The values of this gradient through the line labeled with the letter *E* are depicted in figure 4.23. We can see that all of these values are negative, which indicate that the steepest descent direction $-\nabla_{\boldsymbol{\gamma}} j(\boldsymbol{\gamma})$ is positive and possibly a line-search using this direction will increase the values of σ_{start} inside of the anomaly. This result is in concordance with our expectations, because the target model has a higher value of σ in the anomaly, so we need to increase the initial guess σ_{start}.

In figure 4.21 we can see the reduced gradient obtained using 4 different transmitters with frequency 0.2Hz and current 1A, with target model depicted in figure 4.18 (two homogeneous regions of electric conductivity) and initial model with the same geometry but different electric conductivity, with value $\sigma_{start} = 0.05$ or $\ln(\sigma_{start}) \approx -2.9957$. This scenario combines both scenarios described in the previous paragraphs, because now in the left-region of the anomaly the initial guess is lower than the target value $\sigma_{target_1} = 0.1$, and in the right-region of the anomaly the initial guess is higher than the target value $\sigma_{target_2} = 0.01$. As mentioned before, the final user sets an initial guess of electric conductivity with value $\sigma_{start} = 0.05$, without knowing the true value σ_{target}, hoping that $\sigma_{start} \approx \sigma_{target}$ holds. In the last picture of figure 4.21 we can see the accumulated value of the reduced gradient. The values of this gradient through the line labeled with the letter *E* are depicted in figure 4.24. We can see that the values in the left-region are negative and in the right-region are positive, which indicate that the steepest descent direction $-\nabla_{\boldsymbol{\gamma}} j(\boldsymbol{\gamma})$ is positive in the left-region and negative in the right-region, and possibly a line-search using this direction will increase the values of σ_{start} inside the left-region of the anomaly, and will decrease the values of σ_{start} inside the right-region of the anomaly. This result is in concordance with our expectations, because the target model has a higher value of σ in the left-region of the anomaly, and lower value of σ in the right-region of the anomaly, so we need to increase and decrease the initial guess σ_{start} in each corresponding region.

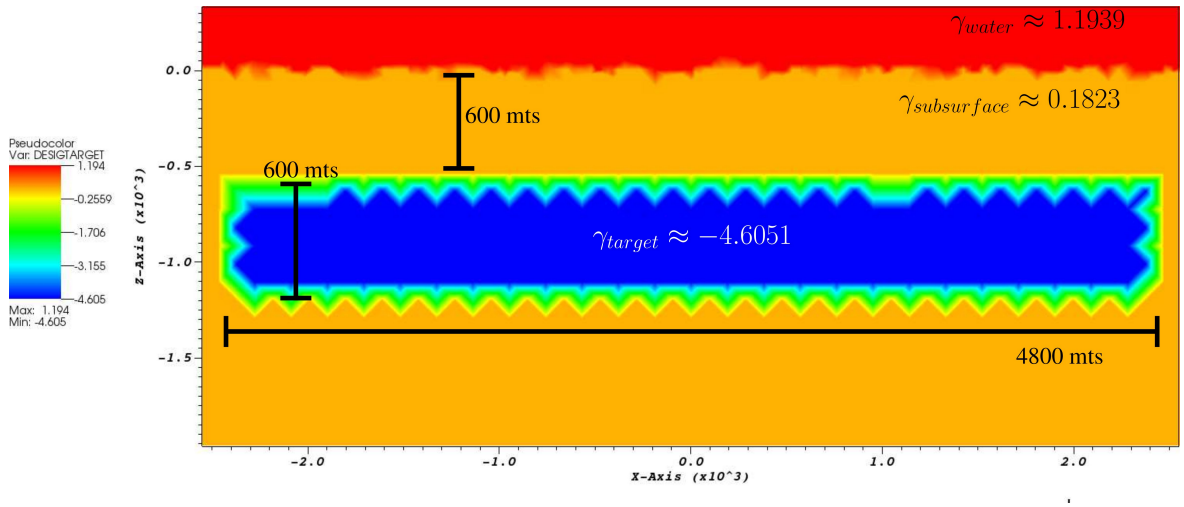

Figure 4.17: Target model of scenario 1, slice in plane Y at $y = 0$. The anomaly has homogeneous electric conductivity with value $\sigma_{target} = 0.01$ ($\ln(\sigma_{target}) \approx -4.6051$)

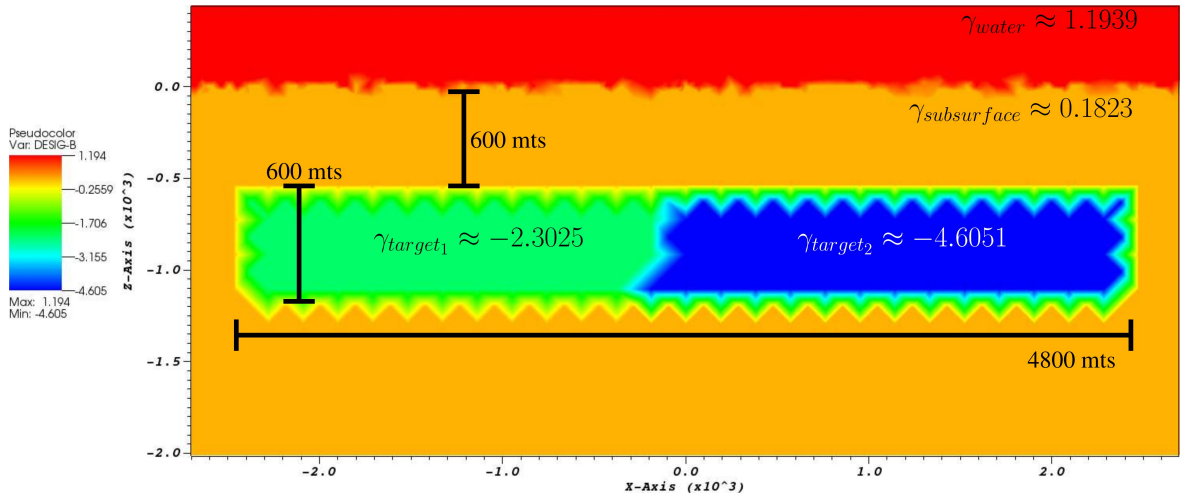

Figure 4.18: Target model of scenario 2, slice in plane Y at $y = 0$. The anomaly has two homogeneous regions of electric conductivity with values $\sigma_{target_1} = 0.1$ ($\ln(\sigma_{target_1}) \approx -2.3025$) and $\sigma_{target_2} = 0.01$ ($\ln(\sigma_{target_2}) \approx -4.6051$)

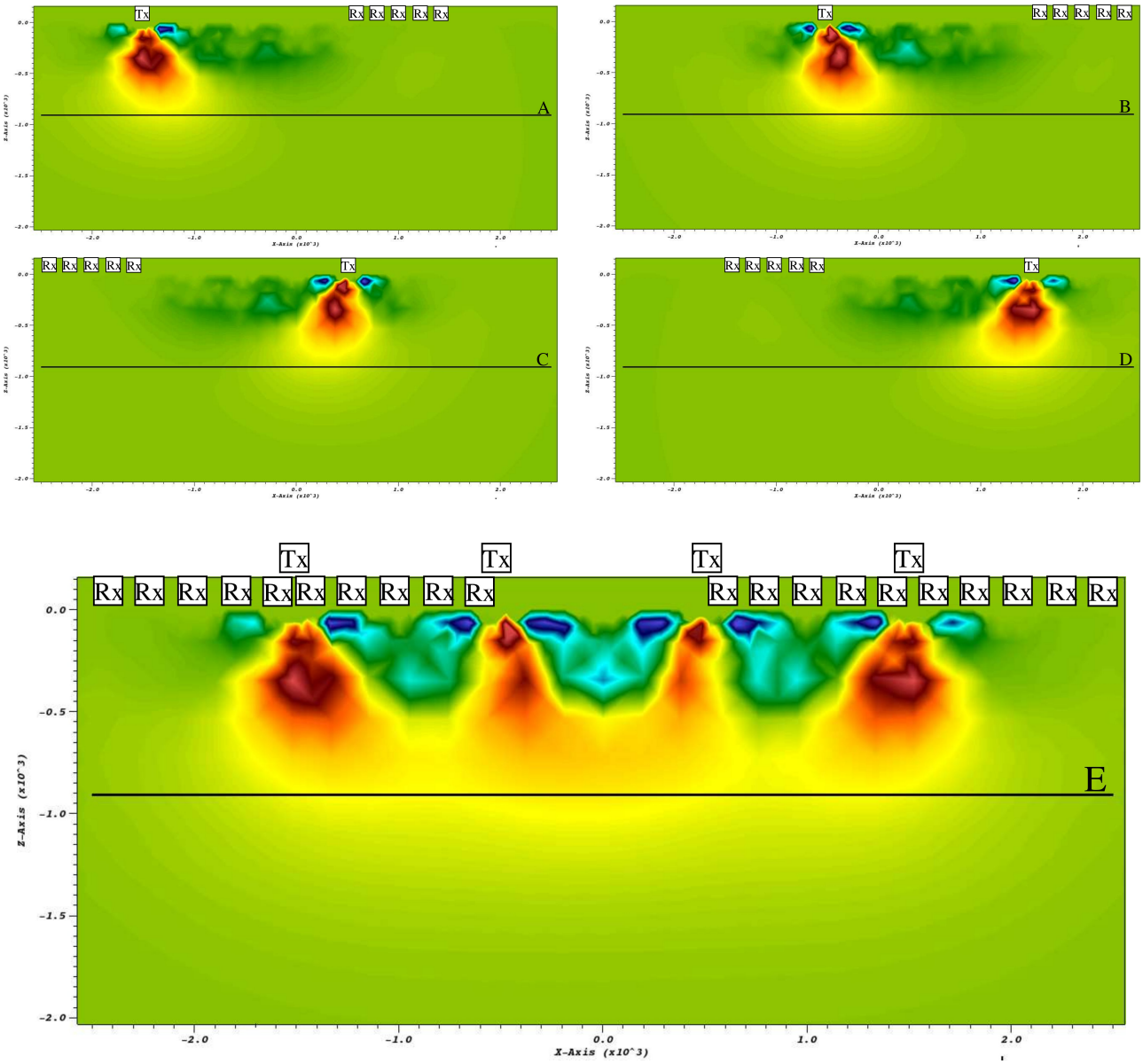

Figure 4.19: Reduced gradient of scenario 1 using a starting model with the same geometry of the target model and homogeneous electric conductivity with value $\sigma_{start} = 1.1$ ($\ln(\sigma_{start}) = 0.0953$). Four transmitters (1000 meters of separation) are used to calculate the gradient, each individually and all of them accumulated

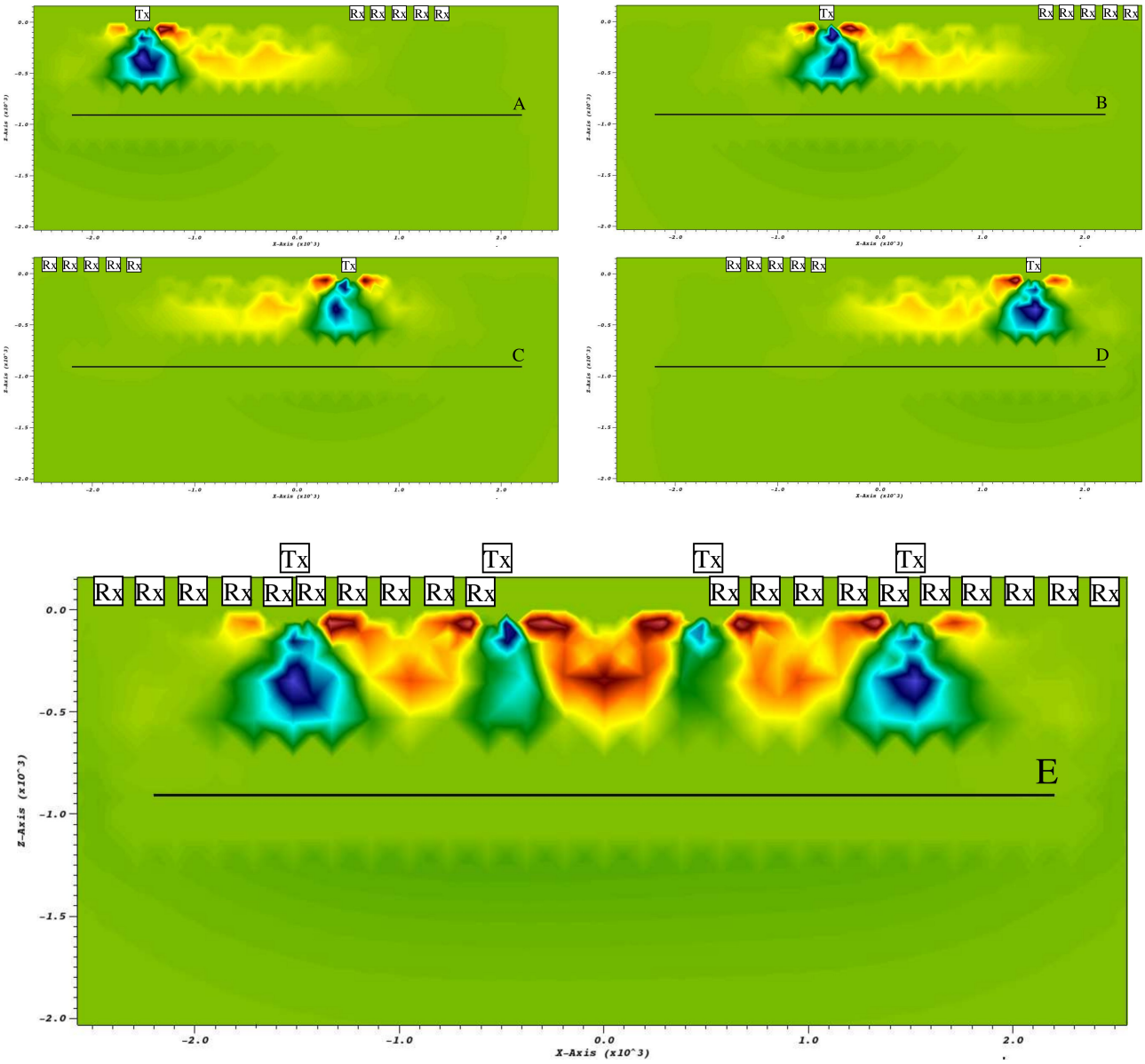

Figure 4.20: Reduced gradient of scenario 1 using a starting model with the same geometry of the target model and homogeneous electric conductivity with value $\sigma_{start} = 0.001$ ($\ln(\sigma_{start}) = -6.9077$). Four transmitters (1000 meters of separation) are used to calculate the gradient, each individually and all of them accumulated

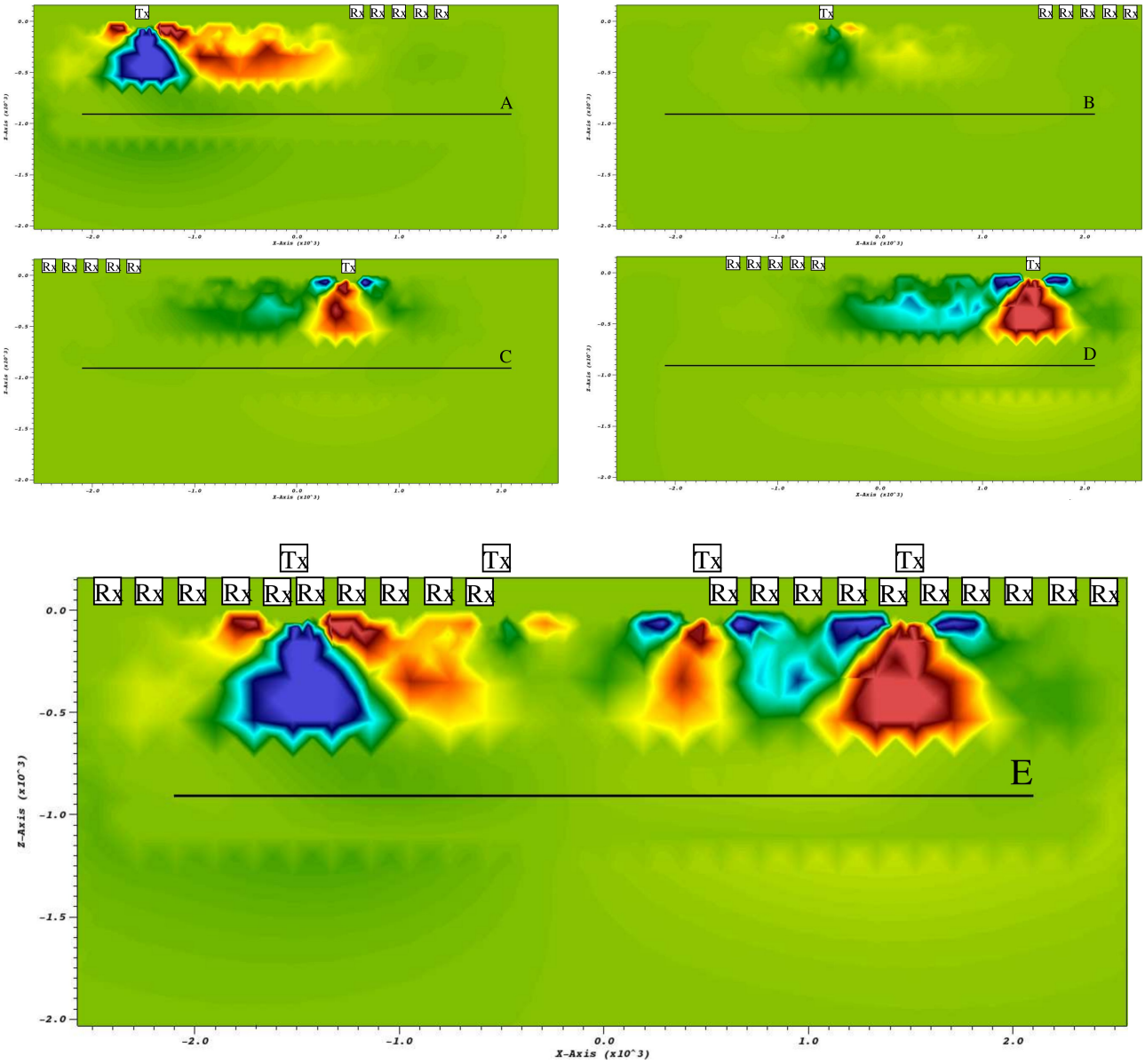

Figure 4.21: Reduced gradient of scenario 2 using a starting model with the same geometry of the target model and homogeneous electric conductivity with value $\sigma_{start} = 0.05$ ($\ln(\sigma_{start}) = -2.9957$). Four transmitters (1000 meters of separation) are used to calculate the gradient, each individually and all of them accumulated

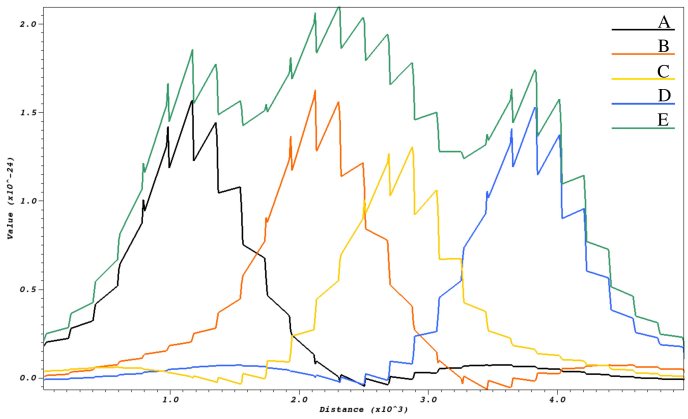

Figure 4.22: Values of reduced gradient depicted in figure 4.19 through lines A, B, C, D, E, from left to right

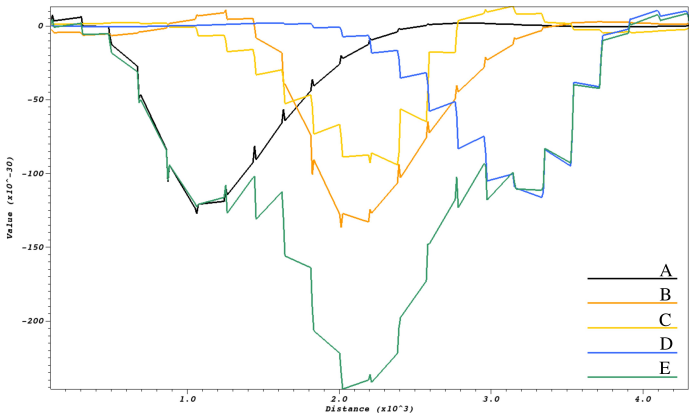

Figure 4.23: Values of reduced gradient depicted in figure 4.20 through lines A, B, C, D, E, from left to right

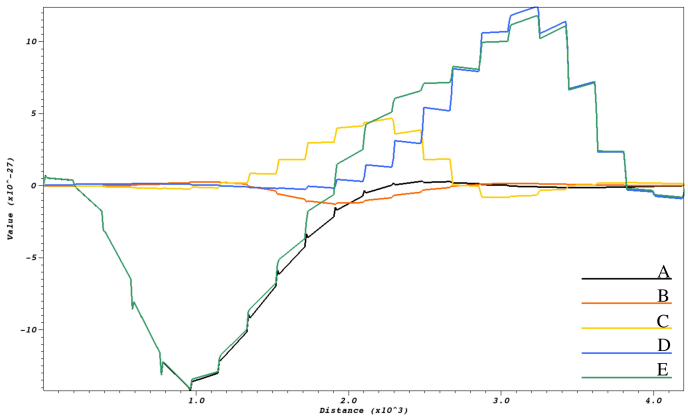

Figure 4.24: Values of reduced gradient depicted in figure 4.21 through lines A, B, C, D, E, from left to right

Inversion results

The sensitivity analysis tells us that the reduced gradient gives us useful information in order to perform the optimization process, also known as *inversion* of the log-conductivity model. The inversion includes reduced gradient and descent direction calculations, line-search executions and design variable updates. We choose three test samples in order to validate our implementation. In figures 4.25, 4.26 and 4.27 we can see the results. On top, middle and bottom, we have the target, initial and inverted (final) values for the design variables, represented by the log-conductivity $\ln(\sigma)$ in each nodal point of the region of interest, which matches with the anomaly in study. Each inversion uses a different number of transmitters with frequency 0.2Hz and current 1A, and each one has associated 5 receivers with an offset of 2000 meters (distance between the closest receiver and the transmitter). The receiver-receiver separation is 200 meters. As we described before, in each inversion we have used 5 optimization iterations with maximum of 20 iterations per line-search and a gradient tolerance of $\epsilon_{grad} = 10^{-35}$.

The first inversion (figure 4.25) has 4 transmitters located at positions $(-1500, 0, -50)$ $(-500, 0, -50)$, $(500, 0, -50)$ and $(1500, 0, -50)$. The target model of log-conductivity corresponds to the scenario 1 depicted in figure 4.17 and the initial log-conductivity is $\ln(\sigma_{initial}) \approx -2.9957$, with $\sigma_{initial} = 0.05$.

The second inversion (figure 4.26) has 8 transmitters located at positions $(-2000, 0, -50)$, $(-1500, 0, -50)$, $(-1000, 0, -50)$, $(-500, 0, -50)$, $(500, 0, -50)$, $(1000, 0, -50)$, $(1500, 0, -50)$ and $(2000, 0, -50)$. The target model of log-conductivity corresponds to the scenario 1 depicted in figure 4.17 and the initial log-conductivity is $\ln(\sigma_{initial}) \approx -5.2983$, with $\sigma_{initial} = 0.005$.

The third inversion (figure 4.27) has 4 transmitters located at positions $(-1500, 0, -50)$ $(-500, 0, -50)$, $(500, 0, -50)$ and $(1500, 0, -50)$. The target model of log-conductivity corresponds to the scenario 2 depicted in figure 4.18 (two region non-homogeneous conductivity) and the initial log-conductivity is $\ln(\sigma_{initial}) \approx -2.9957$, with $\sigma_{initial} = 0.05$.

Sample nodal values of each inversion can be viewed in plots of figures 4.28, 4.29 and 4.30, associated to the lines depicted in the corresponding figures 4.25, 4.26 and 4.27. As we expected, the inverted values are close to the target values. However, more optimization iterations and more transmitters/receivers are necessary to obtain better results, hopefully reaching convergence in the gradient tolerance, and minimizing the value of the cost function.

The evolution of the cost function and reduced gradient for the three samples can be viewed in figures 4.31 and 4.32. We can see that the reduced gradient is calculated only in the first step of each line-search. This is due to its high computational cost in terms of execution time, so instead of evaluate the norm of the reduced gradient in each evaluation point, we just evaluate it in the initial points of the line-searches. We can observe that both the cost function and reduced gradient decrease their values in a sustained form. Several more tests are necessary to completely validate our implementation, but these

initial samples show us that using a few optimization steps the values of the initial log-conductivity models are modified in a *reasonable* manner.

The execution times and speed-up results using the mesh of 4.7M tetrahedral elements are presented in figures 4.33 and 4.34. Three design space sizes were tested: 1000, 100000 and 1000000. We can observe that the scalability starts decaying using 128 processes and further. This behaviour is probably due to the decay in the elements handled in each sub-domain. With 128 processes, each sub-domain handles approximately 32000 elements and 6000 nodal points, and this implies that less computation time and more communication time is being spent by each process's matrix-vector operations, performed inside of the design variable loop (depicted in the inner block of sequence diagram from figure 3.11). This tells us that larger meshes must be loaded in order to preserve the linear scalability of the reduced gradient calculation.

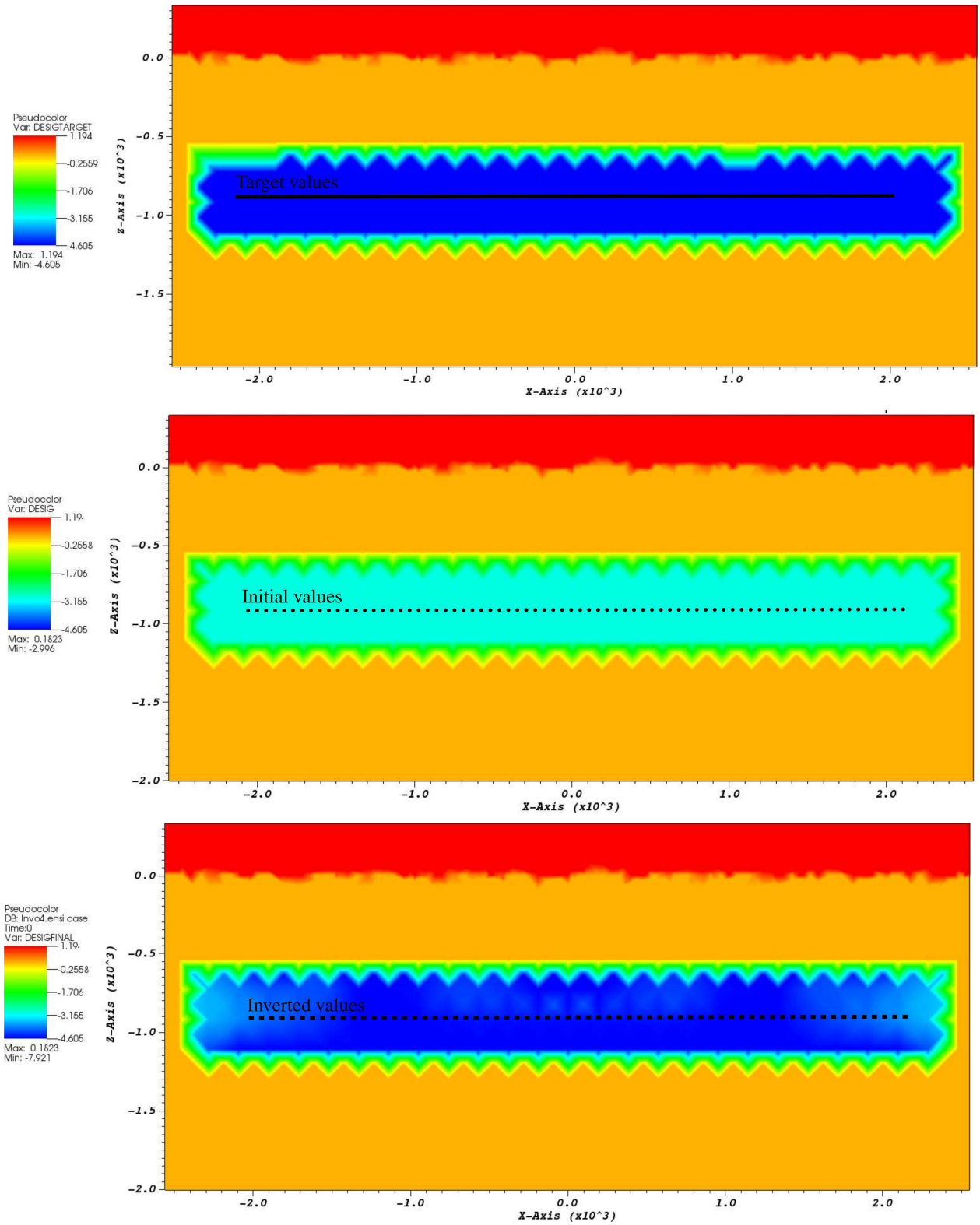

Figure 4.25: First inversion results: log-conductivity models. Slice in plane $Y = 0$.

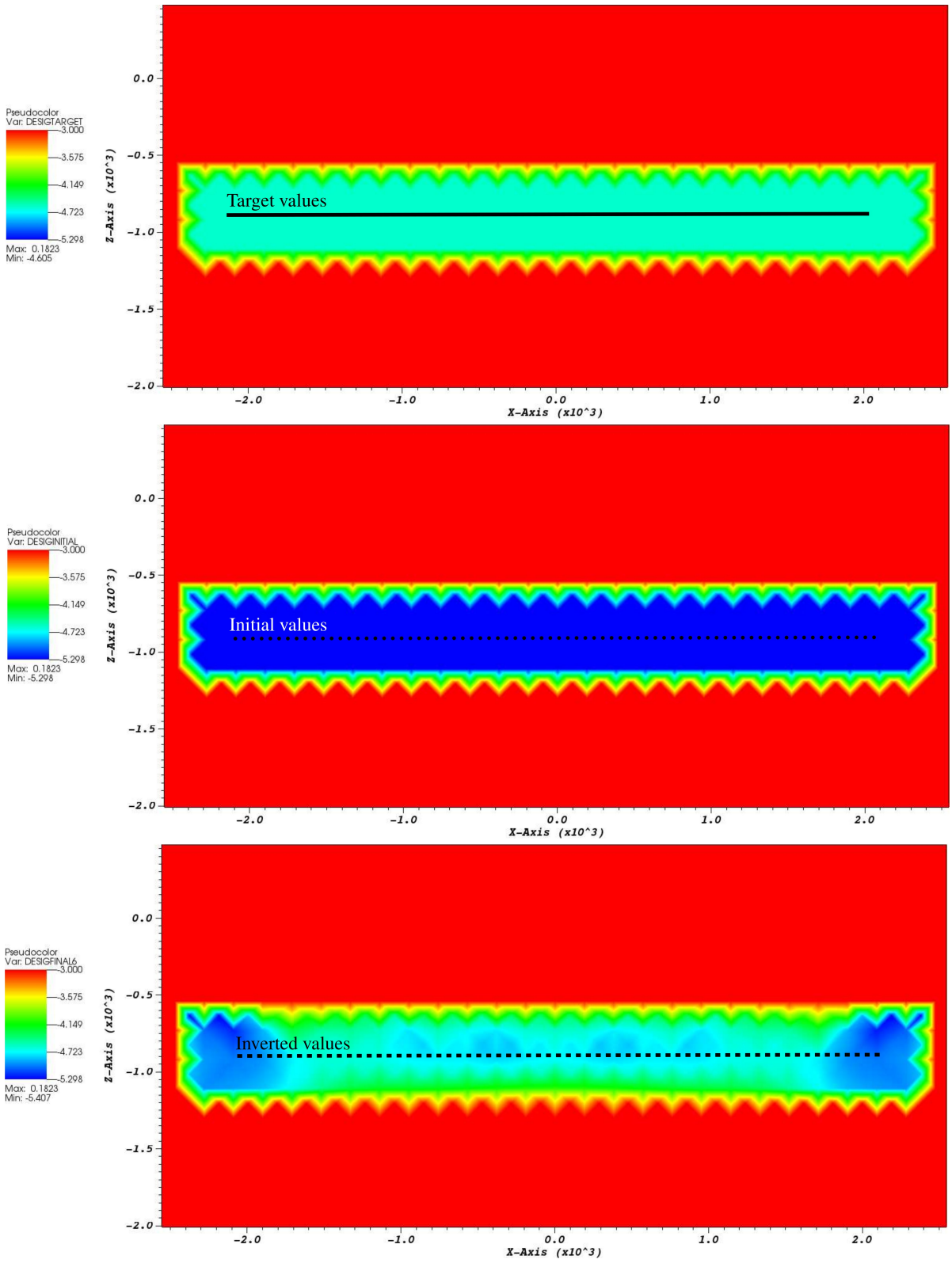

Figure 4.26: Second inversion results: log-conductivity models. Slice in plane $Y = 0$.

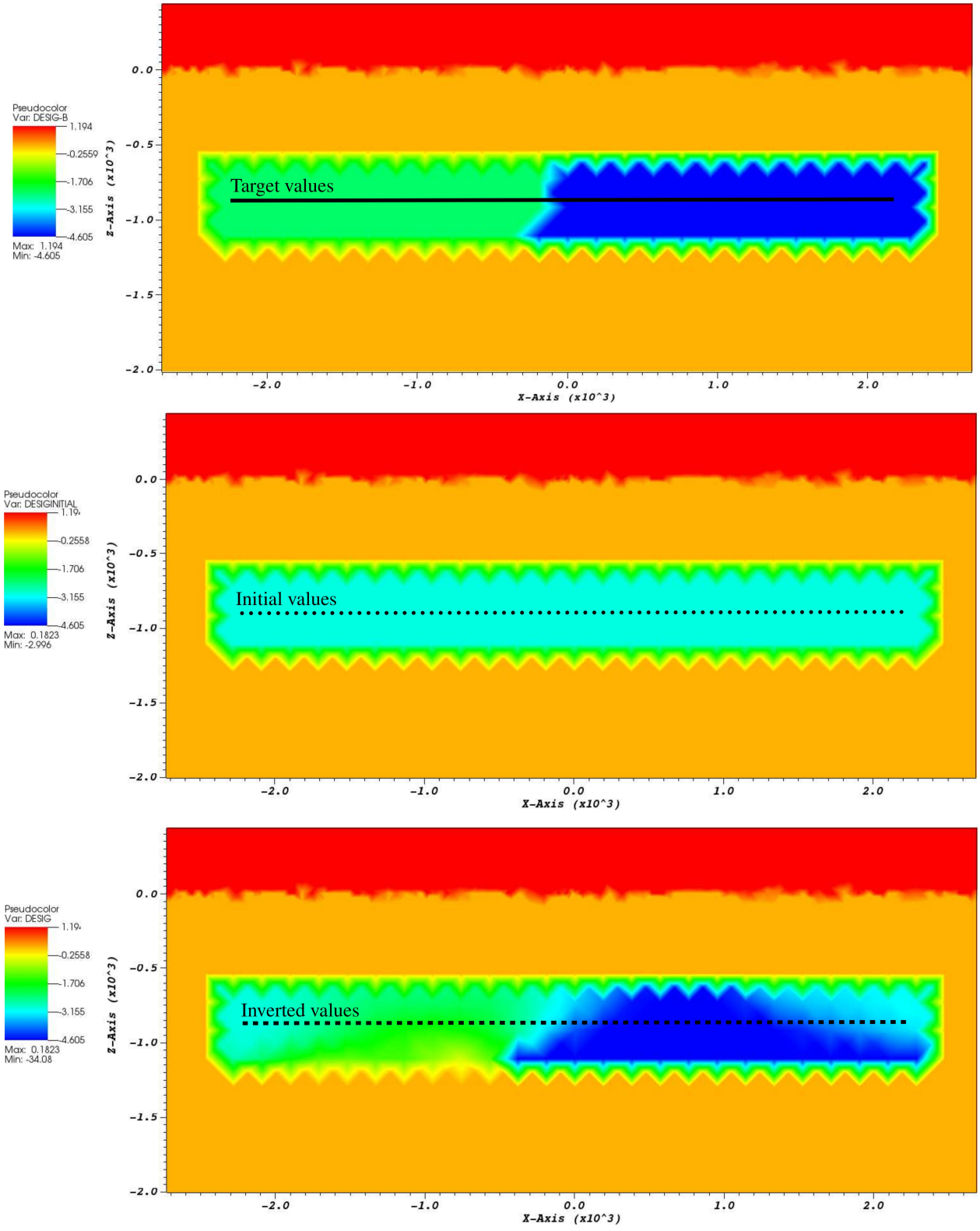

Figure 4.27: Third inversion results: log-conductivity models. Slice in plane $Y = 0$.

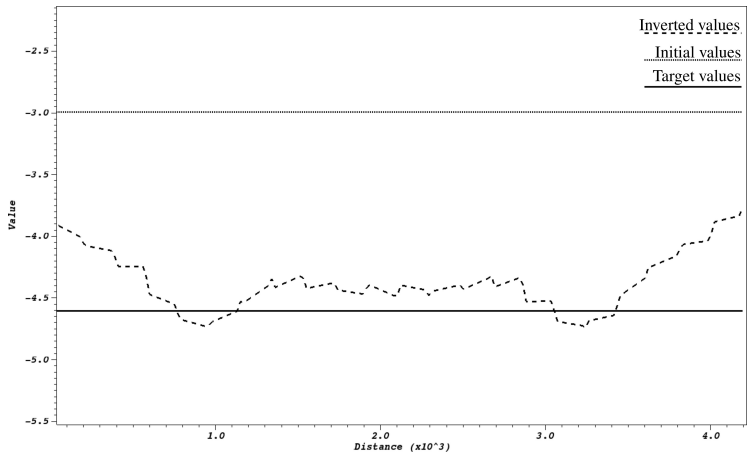

Figure 4.28: First inversion results: values trough a line from figure 4.25.

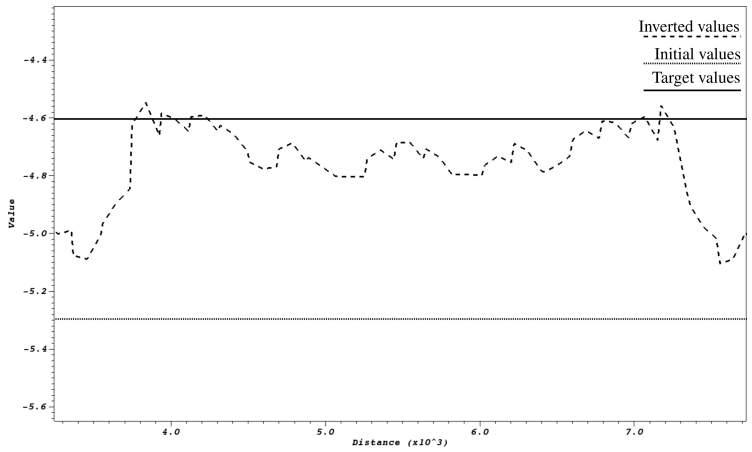

Figure 4.29: Second inversion results: values trough a line from figure 4.26.

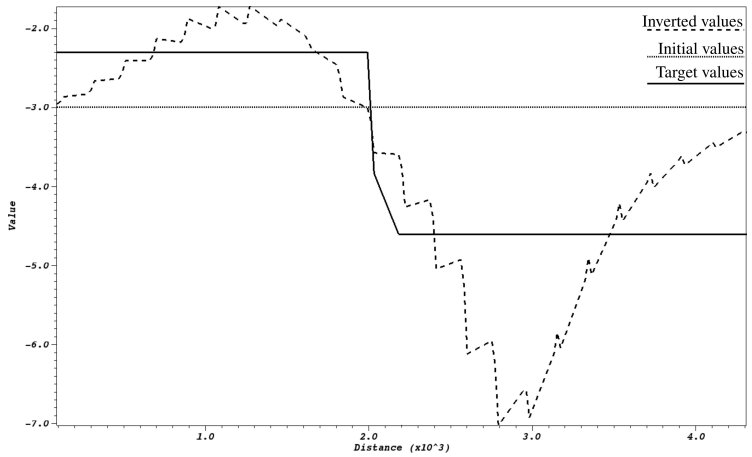

Figure 4.30: Third inversion results: values trough a line from figure 4.27.

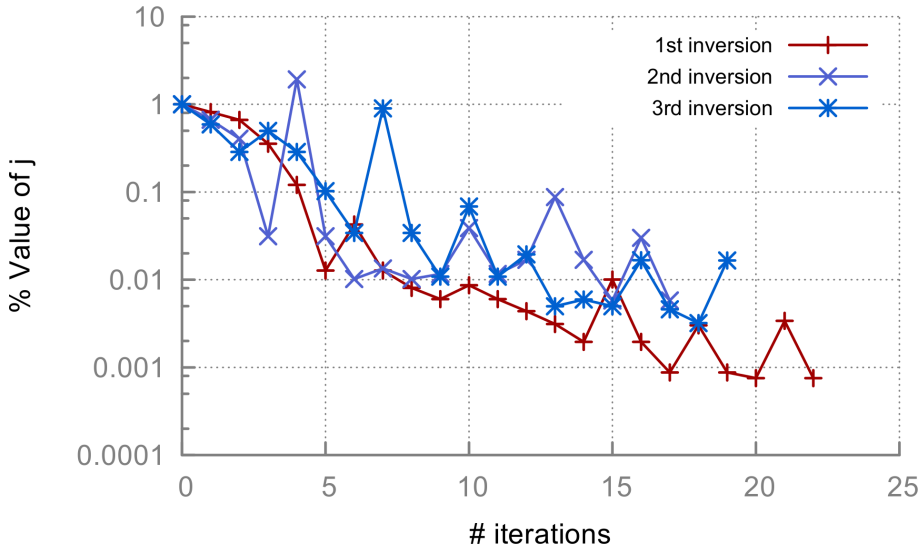

Figure 4.31: Convergence of cost function j in the three sample inversions.

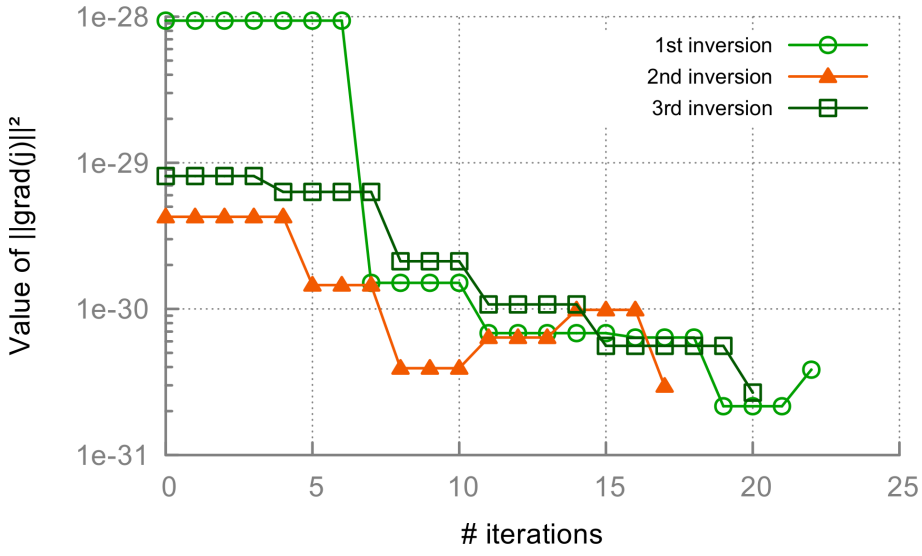

Figure 4.32: Convergence of $\|\nabla_{\mathrm{d}} j\|$ in the three sample inversions.

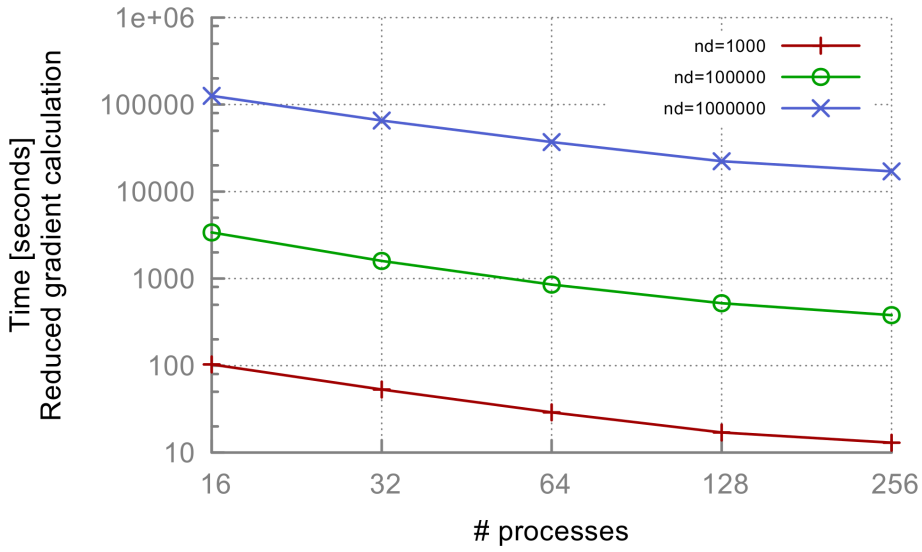

Figure 4.33: Execution time using a 3D mesh with 4.7M tetrahedral elements

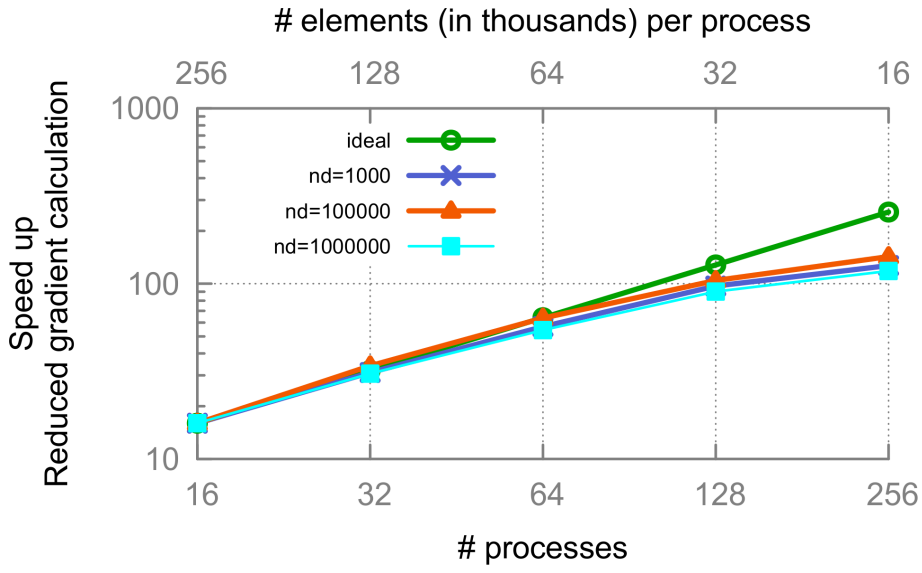

Figure 4.34: Speed-up results using a 3D mesh with 4.7M tetrahedral elements

Chapter 5

Conclusions

5.1 Overview

The proposed implementation solves PDECO problems in the following form:

$$\begin{array}{lll} \underset{(\mathbf{u},\mathbf{d})\in\mathbb{R}^{n_u}\times\mathbb{R}^{n_d}}{\text{minimize}} & J(\mathbf{u},\mathbf{d}) & \\ \text{subject to} & \mathbf{A}(\mathbf{d})\mathbf{u} \;=\; \mathbf{b}(\mathbf{d}) & \end{array} \tag{5.1.1}$$

Its resolution is based in the reduced gradient described in section 2.3. A development framework was proposed in order to allow the final user to communicate its physical models with the reduced gradient calculation routines. With this gradient, the user is able to obtain descent directions and perform line-searches through this direction. This process is repeated until convergence or a maximum number of iterations is achieved. Alya, the underlying PDE simulator, assembles the constraint equation of the problem (5.1.1) and solves it using parallel computing techniques. The parallelization allows the user to run large scale simulations using high performance computing facilities reaching almost linear scalability for several instances and models.

According to the results obtained in sections 4.1.3 and 4.2.3, the proposed implementation achieves acceptable levels concerning the convergence of the optimization process and the preservation of the scalability. The convergence of the optimization process is strongly dependent on the quality of the residual delivered by the parallel iterative solver, the size of the computational mesh and the number of design variables.

5.2 Implementation

Undoubtedly, the most important contribution of the present work is related to the software design proposed for the implementation of the problem (5.1.1). In the hosting institution, Computer Applications in Science and Engineering (CASE) department of

Barcelona Supercomputing Center, the state of the field in this kind of problems was limited to a few individual applications, which were not designed to be re-used in an integrated framework for simulation and design into Alya. For this reason, the design and implementation of reduced gradient calculation and *Optsol* service, described in chapter 3, will be the basis for future improvements and new problems that will be solved in the department.

Regarding implementation aspects, as mentioned before in section 3.2.7, the most difficult part to be tackled by the programmer is related with the differentiation by hand of the routines `physics_elmope()` and `physics_costf()`. If no automatic tools are available, each differentiation must be performed carefully, because different physical modules could have been implemented with different styles and techniques, making this task very time-consuming and error-prone. Additionally, the programmer needs to have a basic knowledge of the discretization method implemented in the underlying simulator, in our case, the finite element method.

Regarding performance aspects, an important topic is the study of the computation/-communication ratio of the reduced gradient calculation. As we mentioned in section 3.2.6, in the general case it is estimated as

$$ratio_{comp/comm} \approx \frac{C \times n_d \times n}{P^2 \times (N_{fwd} + N_{adj} + n_d) \times t_{data}}$$

with P the number of processes, n the number of nodes in the mesh, n_d the number of design variables, N_{fwd} and N_{adj} the number of iterations in the forward and adjoint problems, t_{data} the estimated time of travel of a single word through the interconnection network, and C a constant that depends on the elements per node of the mesh. We can see that if the number of processes P increases, the communication factor increases by P^2. This behaviour can affect the execution of the reduced gradient calculation if the values n and n_d are sufficiently small. For this reason the final user must perform a careful and detailed study of the performance of the application, varying the sizes of the state and design variable spaces, together with the number of processes.

5.3 Applications

The proposed implementation solves PDECO problems where the physical model is represented by a stationary linear real/complex valued PDE with Dirichlet boundary conditions. The choice of this underlying PDE model was based in the need of relatively simple routines which assemble and solve the resulting linear system $\mathbf{A(d)u = b(d)}$. This simplicity is intended to help in the development process by diminishing the necessity to understand complex physical models. In this way the programmer can put all the effort in the main objective of this report, which is the development of a framework that is able to calculate reduced gradients, descent directions, and also to perform line-searches.

Two applications whose physical model fits in this scheme are: source parameter estimation in 2D transport equation and hydrocarbon exploration by 3D controlled-source electromagnetic (CSEM) inversion.

On the first application, detailed in section 4.1, we have tested the earlier versions of the proposed implementation, analyzing convergence aspects and the relationship between the quality of the underlying iterative solver and the quality of the optimization solution. This study gives us useful insight on how to set the numerical parameters of Alya in order to use larger computational domains and larger design variable spaces.

On the second application, detailed in section 4.2, we have tunned the implementation in order to solve synthetic scenarios of large-scale CSEM inversion problems. It is important to mention that the resolution of geophysical inverse problems using parallel codes is among the hardest problems in terms of number of computations and communications between compute nodes.

In order to solve PDECO problems with other underlying physical PDE models, such as transient or non-linear equations, several steps must be re-designed and re-implemented. Eventually, new routines and services will be added to the current implementation, but using the proposed software design as base for future developments.

5.4 Future work

Several avenues for future work are open. The most promising one in the long-term involves the resolution of multidisciplinary design optimization and large-scale inverse problems. The first kind involves several coupled physical systems, which can be non-linear or transient equations. The second kind involves transient physical systems with several coupled right-hand sides. Both problems have large state and design spaces, ranging from thousands to millions of variables. In order to expand this work to more complex physical models, several tasks must be performed. Among them, in ascending order, we can mention:

- Adaptation of software design to allow stationary non-linear physical models. Particularly, the Navier-Stokes PDE model must be studied in detail, calculating its adjoint problem and setting the routines needed to obtain the reduced gradient in terms of the pressure only, velocity only, or both coupled. The future design must be integrated with the current one, in order to re-use parts of the framework presented in this report.

- Adaptation of software design to allow transient linear/non-linear physical models. This task must be studied using both linear and non-linear models concurrently, in order to have an integrated design for both problems.

- Adaptation of software design to allow stationary/transient linear/non-linear multiphysics models. With the first and second task completed, this task includes the

communication between different physical modules inside of one simulation, in a sequential way. For example, the Navier-Stokes equation may require the temperature in some region as input in order to be solved, and the temperature is obtained previously solving the Heat equation. This kind of problem represents the ultimate challenge in terms of software design integration, because it involves the main aspects that can occur as a constraint in a PDECO problem.

- Integration of enhanced optimization methods. This task can be performed in parallel, without dependency on the previous ones. It involves the study, design and implementation of more sophisticated optimization algorithms, as described in section 2.1.1. Enhanced line-search strategies adapted to specific problems and the inclusion of external constraints for design variables, can also be beneficial for the design of the complete framework.

Appendix A

Mathematical tools

A.1 Update state vector in SAND approach

The first-order Taylor expansion of the constraint function $\mathbf{R}(\mathbf{u}, \mathbf{d})$ centered in $(\mathbf{u}^k, \mathbf{d}^k)$ is defined as:

$$\begin{aligned} \mathbf{R}(\mathbf{u}, \mathbf{d}) &= \mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \\ &\quad +\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \cdot (\mathbf{u} - \mathbf{u}^k) \\ &\quad +\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \cdot (\mathbf{d} - \mathbf{d}^k) \\ &\quad +O(\|\mathbf{u} - \mathbf{u}^k\|^2) + O(\|\mathbf{d} - \mathbf{d}^k\|^2) \end{aligned} \tag{A.1.1}$$

Evaluating in a sufficiently near feasible point $(\mathbf{u}^{k+1}, \mathbf{d}^{k+1})$ such that $\mathbf{R}(\mathbf{u}^{k+1}, \mathbf{d}^{k+1}) = 0$, $\mathbf{d}^{k+1} = \mathbf{d}^k + \alpha^k \mathbf{p}^k$ and the second-order terms can be dropped out, we have:

$$\begin{aligned} \mathbf{0} &= \mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \\ &\quad +\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \cdot (\mathbf{u}^{k+1} - \mathbf{u}^k) \\ &\quad +\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \cdot \alpha^k \mathbf{p}^k \end{aligned} \tag{A.1.2}$$

Or equivalently:

$$\begin{aligned} \mathbf{u}^{k+1} &= \mathbf{u}^k - [\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k)]^{-1}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \\ &\quad -[\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k)]^{-1}\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k) \cdot \alpha^k \mathbf{p}^k \end{aligned} \tag{A.1.3}$$

Using the definition of the direct sensitivity matrix

$$\nabla_{\mathbf{d}}\mathbf{u}(\mathbf{d}) = -[\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u}, \mathbf{d})]^{-1}\nabla_{\mathbf{d}}\mathbf{R}(\mathbf{u}, \mathbf{d})$$

and denoting

$$\mathbf{u}_N^k = -\frac{1}{\alpha^k}[\nabla_{\mathbf{u}}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k)]^{-1}\mathbf{R}(\mathbf{u}^k, \mathbf{d}^k)$$

we can formulate the update step for the state vector in the **SAND** approach as:

$$\mathbf{u}^{k+1} \leftarrow \mathbf{u}^k + \alpha^k(\mathbf{u}_N^k + \nabla_{\mathbf{d}}\mathbf{u}(\mathbf{d}^k) \cdot \mathbf{p}^k) \tag{A.1.4}$$

A.2 Implicit function theorem

Extracted from [41], 2nd edition, pages 630-631.

Theorem A.2.1. *Let $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ be a function such that*

- *$h(z^*, 0) = 0$ for some $z^* \in \mathbb{R}^n$,*
- *the function $h(\cdot, \cdot)$ is continuously differentiable in some neighbourhood of $(z^*, 0)$, and*
- *$\nabla_z h(z, t)$ is nonsingular at the point $(z, t) = (z^*, 0)$.*

Then there exist open sets $\mathcal{N}_z \subset \mathbb{R}^n$ and $\mathcal{N}_t \subset \mathbb{R}^m$ containing z^* and 0, respectively, and a continuous function $z : \mathcal{N}_t \to \mathcal{N}_z$ such that $z^* = z(0)$ and $h(z(t), t) = 0$ for all $t \in \mathcal{N}_t$. Further, $z(t)$ is uniquely defined. Finally, if h is p times continuously differentiable with respect to both its arguments for some $p > 0$, then $z(t)$ is also p times continuously differentiable with respect to t, and we have

$$\nabla z(t) = -\nabla_t h(z(t), t)[\nabla_z h(z(t), t)]^{-1}$$

for all $t \in \mathcal{N}_t$.

In our case this theorem helps us to express the state vector $\mathbf{u}$, solution of the discretized PDE constraint $\mathbf{R}(\mathbf{u}, \mathbf{d}) = \mathbf{0}$, as a continuously differentiable function of $\mathbf{d}$, denoted by $\mathbf{u}(\mathbf{d})$. Its derivative with respect to $\mathbf{d}$ can be expressed as

$$\nabla_{\mathbf{d}} \mathbf{u}(\mathbf{d}) = -[\nabla_{\mathbf{u}} \mathbf{R}(\mathbf{u}, \mathbf{d})]^{-1} \nabla_{\mathbf{d}} \mathbf{R}(\mathbf{u}, \mathbf{d})$$

A.3 Karush-Kuhn-Tucker conditions

Extracted from [41], 2nd edition, page 321.

First, let us define the general formulation for constrained optimization problems:

$$\begin{array}{lll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & f(x) & \\ \text{subject to} & c_i(x) = 0, & i \in \mathcal{E} \\ & c_i(x) \geq 0, & i \in \mathcal{I} \end{array} \tag{A.3.1}$$

where f and the functions c_i are all smooth, real-valued functions on a subset of $\mathbb{R}^n$, and $\mathcal{I}$ and $\mathcal{E}$ are two finite sets of indices.

Additionaly, we need to define the *Lagrangian function* for the general problem (A.3.1):

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x) \tag{A.3.2}$$

Also, we say that the *linear independence constraint qualification* (LICQ) holds in a feasible point x if the set of active constraints gradients

$$\{\nabla c_i(x) : i \in \mathcal{A}(x)\}$$

is linearly independent, with $\mathcal{A}(x) = \mathcal{E} \cup \{i \in \mathcal{I} : c_i(x) = 0\}$ the *active set* of constraints.

Theorem A.3.1. *Suppose that x^* is a local solution of* (A.3.1)*, that the functions f and c_i in* (A.3.1) *are continuously differentiable, and that the LICQ holds at x^*. Then there is a Lagrange multiplier vector λ^*, with components λ_i^*, $i \in \mathcal{E} \cup \mathcal{I}$, such that the following conditions are satisfied at (x^*, λ^*):*

$$\nabla_x \mathcal{L}(x^*, \lambda^*) = 0, \tag{A.3.3a}$$
$$c_i(x^*) = 0, \quad \textit{for all } i \in \mathcal{E} \tag{A.3.3b}$$
$$c_i(x^*) \geq 0, \quad \textit{for all } i \in \mathcal{I} \tag{A.3.3c}$$
$$\lambda_i^* \geq 0, \quad \textit{for all } i \in \mathcal{I} \tag{A.3.3d}$$
$$\lambda_i^* c_i(x^*) = 0, \quad \textit{for all } i \in \mathcal{E} \cup \mathcal{I} \tag{A.3.3e}$$

A.4 Lagrangian using complex-valued state variables

The Lagrangian definition of equation (4.2.11) comes from the following fact. If we consider the cost function $J(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d})$ as a function of $\mathbf{x}, \mathbf{y} \in \mathbb{R}^s$ with $\mathbf{z} = \mathbf{x} + i\mathbf{y}$, we can rename it as

$$J(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}) = \tilde{J}(\mathbf{x}, \mathbf{y}, \mathbf{d})$$

and the constraints function $\mathbf{R}$ can be renamed as

$$\mathbf{R}(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}) = \tilde{\mathbf{R}}^{real}(\mathbf{x}, \mathbf{y}, \mathbf{d}) + i\tilde{\mathbf{R}}^{imag}(\mathbf{x}, \mathbf{y}, \mathbf{d})$$

and the optimization problem (4.2.5) can be stated as:

$$\begin{array}{ll} \underset{\mathbf{d} \in \mathbb{R}^{n_d}}{\text{minimize}} & \tilde{J}(\mathbf{x}, \mathbf{y}, \mathbf{d}) \\ \text{subject to} & \tilde{\mathbf{R}}^{real}(\mathbf{x}, \mathbf{y}, \mathbf{d}) = \mathbf{0}_{s\times 1} \\ & \tilde{\mathbf{R}}^{imag}(\mathbf{x}, \mathbf{y}, \mathbf{d}) = \mathbf{0}_{s\times 1} \end{array} \tag{A.4.1}$$

with $\tilde{J} : \mathbb{R}^{n_u} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \to \mathbb{R}$, $\tilde{\mathbf{R}}^{real} : \mathbb{R}^{n_u} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \to \mathbb{R}^{n_u}$ and $\tilde{\mathbf{R}}^{imag} : \mathbb{R}^{n_u} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_d} \to \mathbb{R}^{n_u}$. The Lagrangian function asociated to this problem, with $\phi, \psi \in \mathbb{R}^{n_u}$ the corresponding Lagrange multipliers, is as follows:

$$\begin{aligned} L(\mathbf{x}, \mathbf{y}, \mathbf{d}, \phi, \psi) &= \tilde{J}(\mathbf{x}, \mathbf{y}, \mathbf{d}) + \phi^T \tilde{\mathbf{R}}^{real}(\mathbf{x}, \mathbf{y}, \mathbf{d}) + \psi^T \tilde{\mathbf{R}}^{imag}(\mathbf{x}, \mathbf{y}, \mathbf{d}) \\ &= J(\mathbf{x}, \mathbf{y}, \mathbf{d}) + \mathrm{Re}\left\{(\phi + i\psi)^T (\mathbf{R}^{real}(\mathbf{x}, \mathbf{y}, \mathbf{d}) - i\mathbf{R}^{imag}(\mathbf{x}, \mathbf{y}, \mathbf{d}))\right\} \end{aligned} \tag{A.4.2}$$

Denoting $\boldsymbol{\lambda} = \phi + i\psi$ and recovering the actual names for $\tilde{J}$, $\tilde{\mathbf{R}}^{real}$ and $\tilde{\mathbf{R}}^{imag}$, we can obtain

$$L(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}, \boldsymbol{\lambda}) \;=\; J(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}) + \mathrm{Re}\left\{\boldsymbol{\lambda}^T \overline{\mathbf{R}(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d})}\right\} \tag{A.4.3}$$

We can multiply the Lagrangian multiplier by any scalar constant because by definition it is an unconstrained variable, so multiplying $\boldsymbol{\lambda}$ by the scalar -2 doesn't affect the previous argument. This scaling is due to aesthetic reasons, in order to keep clean the following algebraic steps.

After the previous explanation, we can unroll the previous expression for the Lagrangian with the aim of taking derivatives with respect to $\mathbf{z}$ or $\overline{\mathbf{z}}$:

$$\begin{aligned} L(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}, \boldsymbol{\lambda}) &= J(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}) - 2\,\mathrm{Re}\left\{\boldsymbol{\lambda}^T\, \overline{\mathbf{R}(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d})}\right\} && \text{(A.4.4)} \\ &= J(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}) - \boldsymbol{\lambda}^T\, \overline{\mathbf{R}(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d})} - \overline{\boldsymbol{\lambda}}^T\, \mathbf{R}(\mathbf{z}, \overline{\mathbf{z}}, \mathbf{d}) && \text{(A.4.5)} \\ &= \overline{(\mathbf{z} - \mathbf{z}^{obs})}\mathbf{M}^{obs}(\mathbf{z} - \mathbf{z}^{obs}) \\ &\quad -\boldsymbol{\lambda}^T\, (\overline{\mathbf{K}(\mathbf{b})}\overline{\mathbf{z}} - \overline{\mathbf{f}(\mathbf{d})}) \\ &\quad -\overline{\boldsymbol{\lambda}}^T\, (\mathbf{K}(\mathbf{b})\mathbf{z} - \mathbf{f}(\mathbf{d})) && \text{(A.4.6)} \end{aligned}$$

The derivative with respect to $\mathbf{z}$ can be set equal to zero by KKT conditions (let us assume all the hypothesis are satisfied):

$$\begin{aligned} \nabla_{\mathbf{z}} L &= \mathbf{0}_{s\times 1} \\ \mathbf{M}^{obs}\overline{(\mathbf{z} - \mathbf{z}^{obs})} - \mathbf{K}(\mathbf{d})^T\, \overline{\boldsymbol{\lambda}} &= \mathbf{0}_{s\times 1} \\ \mathbf{K}(\mathbf{d})^T\, \overline{\boldsymbol{\lambda}} &= \mathbf{M}^{obs}\overline{(\mathbf{z} - \mathbf{z}^{obs})} \end{aligned} \tag{A.4.7}$$

Bibliography

[1] ANSYS ICEM CFD meshing software. http://www.ansys.com/Products/Other+Products/ANSYS+ICEM+CFD, 2013. 21

[2] GiD. The Personal Pre and Post Processor. http://gid.cimne.upc.es/, 2013. 21

[3] List of public domain and commercial mesh generators. Mantained by Dr. Robert Schneiders. http://www.robertschneiders.de/meshgeneration//software.html, 2013. 21

[4] Mathematica 9, Wolfram Reseach. http://www.wolfram.com/mathematica/, 2013. 51

[5] MATLAB R2013a, MathWorks. http://www.mathworks.com/products/matlab/, 2013. 51

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. 52

[7] V. Akçelik, G. Biros, O. Ghattas, J. Hill, D. Keyes, and B. van Bloemen Waanders. Parallel Algorithms for PDE-Constrained Optimization. In M. A. Heroux, P. Raghavan, and H. D. Simon, editors, *Parallel Processing for Scientific Computing*, chapter 16, pages 291–322. Society for Industrial and Applied Mathematics, January 2006. 12

[8] E. A. Badea, M. E. Everett, G. A. Newman, and O. Biro. Finite-element analysis of controlled-source electromagnetic induction using Coulomb-gauged potentials. *Geophysics*, 66(3):786–799, 2001. 87

[9] P. R. Bannister. Determination of the electrical conductivity of the sea bed in shallow waters. *Geophysics*, 33:995–1002, 1968. 11

[10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994. 22, 32, 48

[11] G. Biros and O. Ghattas. Parallel preconditioners for KKT systems arising in optimal control of viscous incompressible flows. In *In Proceedings of Parallel CFD '99*. North Holland, 1999. 11

[12] G. Biros and O. Ghattas. Parallel Lagrange-Newton-Krylov-Schur methods for PDE constrained optimization. Part I: The Krylov-Schur solver. Technical report, Laboratory for Mechanics, Algorithms, and Computing, Carnegie Mellon University, 2000. 11

[13] G. Biros and O. Ghattas. Parallel Lagrange-Newton-Krylov-Schur methods for PDE constrained optimization. Part II: The Lagrange-Newton solver, and its application to optimal control of steady viscous flows. Technical report, Laboratory for Mechanics, Algorithms, and Computing, Carnegie Mellon University, 2000. 11

[14] D.H. Brandwood. A complex gradient operator and its application in adaptive array theory. *IEE Proceedings F (Communications, Radar and Signal Processing)*, 130(1):11–16, 1983. 88

[15] L. Brock-Nannestad. Determination of the electrical conductivity of the seabed in shallow waters with varying conductivity profile. *Electronics Letters*, 1(10):274–276, 1965. 11

[16] P. Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 2001. 37

[17] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Pub. Inc., San Francisco, CA, USA, 2001. 11

[18] P. G. Ciarlet. Basic Error Estimates for Elliptic Problems. In P. G. Ciarlet and J. L. Lions, editors, *Handbook of Numerical Analysis, Vol. II. Finite Element Methods (Part 1)*, pages 17–351. North-Holland, Amsterdam, 1991. 21

[19] S. Constable. Ten years of marine CSEM for hydrocarbon exploration. *Geophysics*, 75(5):75A67–75A81, 2010. 85, 91

[20] R. Daley. *Atmospheric data analysis*. Cambridge University Press, Cambridge, 1991. 11

[21] L. C. Evans. *Partial Differential Equations (Graduate Studies in Mathematics, V. 19) GSM/19*. American Mathematical Society, June 1998. 77

[22] C. Geuzaine and J. F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 2009. 21

[23] M. B. Giles, M. C. Duta, J. Müller, and N. Pierce. Algorithm developments for discrete adjoint methods. *AIAA Journal*, 41(2):198–205, February 2003. 23

[24] M. B. Giles and N. A. Pierce. Adjoint equations in CFD: duality, boundary conditions, and solution behaviour. In *13th AIAA Computational Fluid Dynamics Conference*, number AIAA–97–1850, Snowmass Village, CO, June 1997. 23

[25] M. B. Giles and N. A. Pierce. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion*, 65:393–415, 2000. 23

[26] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008. 25, 51

[27] W. Hackbusch. *Multi-grid methods and applications*, volume 4 of *Springer series in computational mathematics*. Springer, Berlin [u.a.], 1985. 80

[28] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich. *Optimization with PDE Constraints*. Springer, 2008. 23

[29] G. Houzeaux, R. de la Cruz, H. Owen, and M. Vázquez. Parallel uniform mesh multiplication applied to a navier-stokes solver. Available at http://www.sciencedirect.com/science/article/pii/S0045793012001569, preprint (2012). 5, 12, 26, 38

[30] G. Houzeaux, M. Vázquez, R. Aubry, and J. M. Cela. A massively parallel fractional step solver for incompressible flows. *Journal of Computational Physics*, 228(17):6316 – 6332, 2009. 12, 26

[31] P. D. Hovland. *Automatic differentiation of parallel programs*. PhD thesis, University of Illinois, Urbana-Champaign, 1997. PhD : computer science. 51

[32] T. J. R. Hughes. *The finite element method : linear static and dynamic finite element analysis*. Englewood Cliffs, N.J. Prentice-Hall International, 1987. 21

[33] A. Jameson. Aerodynamic design via control theory. *Journal of Scientific Computing*, 3(3):233–260, 1988. 11

[34] G. Karypis and V. Kumar. MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`, 1995. 27, 29

[35] D. Kirk and W. Hwu. *Programming Massively Parallel Processors - A Hands-on Approach.* Morgan Kaufmann, 2012. 11

[36] J. Koldan. *Numerical Solution of 3-D Electromagnetic Problems in Exploration Geophysics and its Implementation on Massively Parallel Computers*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona, 2013. PhD : computer science. 84, 87

[37] J. Labarta, S. Girona, V. Pillet, T. Cortes, and J. M. Cela. A parallel program development environment. In *In proceedings of the 2th. Int. Euro-Par Conference*, pages 665–674. Springer, 1995. 37

[38] R. Löhner, F. Mut, J. Cebral, R. Aubry, and G. Houzeaux. Deflated preconditioned conjugate gradient solvers for the pressure-poisson equation: Extensions and improvements. *Int. J. Numer. Meth. Engn.*, 87:2–14, 2011. 32

[39] R. M. Lewis. Practical aspects of variable reduction formulations and reduced basis algorithms in multidisciplinary design optimization. Technical report, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1995. 12

[40] G. A. Newman and D. L. Alumbaugh. Three-dimensional massively parallel electromagnetic inversion—I. Theory. *Geophysical Journal International*, 128(2):345–354, 1997. 11

[41] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 1999. 15, 18, 20, 120

[42] O. Pironneau. On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64(1):97–110, 1974. 11

[43] T. W. Pratt and M. V. Zelkowitz. *Programming Languages: Design and Implementation*. Pearson Education, Inc., 4 edition, 2001. 26

[44] V. Puzyrev, J. Koldan, J. de la Puente, G. Houzeaux, M. Vázquez, and J. M. Cela. A parallel finite-element method for 3D controlled-source electromagnetic forward modeling. *Geophysical Journal International*, 193(2):678–693, 2013. 87

[45] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Boston, MA, 2 edition, 2005. 26

[46] Y. Saad. *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, 2 edition, April 2003. 22, 32

[47] Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc'h. A deflated version of the conjugate gradient algorithm. *SIAM J. Sci. Comput.*, 21(5):1909–1926, December 1999. 32, 80

[48] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998. 11

[49] J. R. Stewart and H. C. Edwards. *The SIERRA framework for developing advanced parallel mechanics applications*, pages 301–315. Berlin: Springer, 2003. 12

[50] A. Tarantola. Inversion of seismic reflection data in the acoustic approximation. *Geophysics*, 49(8):1259–1266, 1984. 11

[51] B. van Bloemen Waanders, Bartlett R., Long K., Boggs P., and Salinger A. Large Scale Non-Linear Programming for PDE Constrained Optimization. Technical report, Sandia National Laboratories, 2002. 12, 14, 16

[52] P. Wesseling. *An introduction to multigrid methods*. R.T. Edwards, 2004. 80

[53] B. Wilkinson and A. Michael. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ: Prentice Hall, second edition, 2005. 29

[54] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*, volume 1. McGraw Hill Book Company, London, 4th edition, 1989. 21