UNIVERSIDADE FEDERAL FLUMINENSE

MATHEUS DE CASTRO RIBEIRO

Toward an ultrasonic inspecting method to detect and classify adhesive bonding defects in real time

> NITERÓI 2019

UNIVERSIDADE FEDERAL FLUMINENSE

MATHEUS DE CASTRO RIBEIRO

Toward an ultrasonic inspecting method to detect and classify adhesive bonding defects in real time

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Computação Científica e Sistemas de Potência

Orientador: Prof. Dr. Ricardo Leiderman

Co-orientador: Prof. Dr. Esteban Walter Gonzalez Clua

> NITERÓI 2019

MATHEUS DE CASTRO RIBEIRO

"Toward an ultrasonic inspecting method to detect and classify adhesive bonding defects in real time"

> de Mestrado apresentada Dissertação Pós-Graduação de em Programa ao Universidade Federal Computação daFluminense como requisito parcial para a obtenção do Grau de Mestre em Área de concentração: Computação. Computação Científica e Sistemas de Potência

Aprovada em Agosto de 2019.

BANCA EXAMINADORA

Prof. Dr. Ricardo Leiderman - Orientador, UFF

Prof. Dr. Esteban Walter Gonzalez Clua - Co-orientador, UFF

d'Mar B2 Fr

Prof. Dr. André Maues Brabo Pereira, UFF

Prof. Dr. Daniel Alves Castello, UFRJ

Niterói 2019

Ficha catalográfica automática - SDC/BEE Gerada com informações fornecidas pelo autor

R484t Ribeiro, Matheus de Castro Toward an ultrasonic inspecting method to detect and classify adhesive bonding defects in real time / Matheus de Castro Ribeiro ; Ricardo Leiderman, orientador ; Esteban Walter Gonzalez Clua, coorientador. Niterói, 2019. 76 f. : il. Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2019. DOI: http://dx.doi.org/10.22409/PGC.2019.m.13728150746 1. Modelagem por ultrassom. 2. Juntas coladas. 3. Espalhamento acúsitco. 4. Computação em GPU. 5. Produção intelectual. I. Leiderman, Ricardo, orientador. II. Gonzalez Clua, Esteban Walter, coorientador. III. Universidade Federal Fluminense. Instituto de Computação. IV. Título. CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

Resumo

Juntas coladas são um método eficiente de unir diferentes componentes em projetos estruturais. No entanto, a ausência de um método de inspeção não-destrutivo confiável capaz de atestar a integridade de juntas coladas é ainda um problema em aberto. Nas últimas décadas, muitos pesquisadores se esforçaram no sentido de atender essa demanda e, nesse contexto, métodos que utilizam ultrassom para realizar a inspeção emergiram como os mais promissores. É um consenso que a capacidade de modelar matematicamente e computacionalmente a interação entre ondas ultrassônicas e as juntas de adesão tem um papel crucial no desenvolvimento de qualquer método de inspeção por ultrassom. Tendo isso em mente, num trabalho anterior foi desenvolvido e implementado um algoritmo que simula numericamente a interação entre ondas ultrassônicas e juntas coladas (defeituosas). A implementação foi desenvolvida em MATLAB e leva por volta de um minuto para executar uma simulação típica, tornando-o impraticável para resolver o problema inverso em tempo real. No presente trabalho, nós revisitamos o algoritmo e desenvolvemos uma nova implementação baseada na estratégia de paralelismo SIMT (Instrução Única, Múltiplos Threads), adequada para um implementação paralela em GPU. Desta forma, nós executamos diversas simulações afim de comparar o tempo de execução dessa nova implementação com a anteriormente proposta. A grosso modo, nossa nova implementação foi capaz de reduzir o tempo de execução num fator de 25, abrindo a possibilade do problema ser resolvido em tempo real. Tanto quanto sabemos, essa é a primeira vez na literatura em que GPUs são aplicadas para resolver este problema de propagação de onda em particular.

Palavras-chave: Espalhamento acústico, ultrassom, juntas coladas, computação em GPU, otimização em performance, sistemas industriais em tempo-real.

Abstract

Adhesive bonding is an efficient method to join different components in structural design. However, a reliable non-destructive inspecting method to attest the integrity of adhesive bonds is still an open task. In the last few decades, many researchers have put effort into addressing this demand, and the methods based on ultrasound have emerged as the most promising ones. It is consensual that the capability of modeling both mathematically and computationally the interaction between ultrasonic waves and adhesive bonds will play a crucial role in the development of any ultrasonic inspecting method. In that sense, in a previous work, an algorithm to numerically simulate the interaction between ultrasonic waves and localized interfacial adhesion defects was developed and implemented. The implementation was developed using MATLAB and takes around a minute to run, making it non feasible to solve the correlated inverse problem in real time. In the present work, we develop a novel GPU parallel implementation, aiming to reduce considerably the execution time. We then perform several simulations to compare the execution time of our new enhanced code with the previous implementation. Roughly speaking, our new implementation has reduced the execution time by a factor of 25, opening the possibility for solving the inverse problem in real time. To the best of our knowledge, this is the first time in the literature that GPU is employed to solve this particular wave propagation problem.

Keywords: Scattering problem, ultrasound, adhesive bonds, GPU computing, performance optimization, real-time industrial systems.

List of Figures

1.1	Multilayered composite.	1
1.2	Inspecting scheme.	4
2.1	scenario considered in the present work. \mathbf{u}_2^+ is the incident displacement field, \mathbf{u}_1^+ is the reflected displacement field and \mathbf{u}_1^- is the transmitted dis- placement field.	5
2.2	Decomposition of wavenumber vector k	7
2.3	The Quasistatic approximation (QSA). We replace the interface by a set of tangential and normal springs.	8
2.4	The scattering effect observed	10
3.1	Flow chart of the Scattering Algorithm	15
4.1	Parallel LU factorization	36
4.2	Illustrations of computation of submatrices U_{01} and L_{10}	37
4.3	Scheme that illustrates how matrix A is factorized in the parallel LU al- gorithm. Darker areas of matrix A means factorization has been already	
	performed	38
4.4	Illustration of the "Null Guess" optimization.	42
5.1	Material structure considered in the numerical simulations	45
5.2	Interfacial stiffness considered in example 1	47
5.3	The spectrum of the scattered displacement field in the z direction for simmulations of case 1	48
5.4	Interfacial stiffness considered in example 2	50
5.5	The spectrum of the scattered displacement field in the z direction in the upper half-space for simulations of example 2	52

5.6	Interfacial stiffness considered in example 3	54
5.7	The spectrum of the scattered displacement field in the z direction in the upper half-space for simmulations of example 3	55
5.8	Interfacial stiffness considered in example 4	57
5.9	The spectrum of the scattered displacement field in the z direction in the upper half-space for simmulations of example 4	58
5.10	Interfacial stiffness considered in example 5	60
5.11	The spectrum of the scattered displacement field in the z direction in the upper half-space for simmulations of example 5	61

List of Tables

3.1	Summary of theoretical complexity analysis for each stage of the Scattering Algorithm	25
3.2	Time/cost analysis for each stage	25
5.1	Characteristics of system environments used in simulations	44
5.2	Mechanical properties of layers represented in the model	46
5.3	Numerical information of simulations for example 1	47
5.4	Execution time of simulations for example 1 in different implementations	49
5.5	Numerical information of simulations for example 2	51
5.6	Execution time of simulations for example 2 in different implementations	53
5.7	Execution time of simulations for example 3 in different implementations	56
5.8	Execution time of simulations for example 4 in different implementations	59
5.9	Execution time of simulations for example 5 in different implementations	62
5.10	. Duration of actions performed in GPU during a 5000-points simulation $% \mathcal{A} = \mathcal{A}$.	63
5.11	. Kernels	64
5.12	Execution time of implementation using LAPACK library to solve the linear system in Stage 4	66
5.13	Execution time of implementation using cuSOLVER library to solve the linear system in Stage 4	66
5.14	Execution time of implementation using cuSOLVER library to solve the linear system in Stage 4.	67

Contents

1	1 Introduction					
2	The	Theoretical Formulation				
	2.1	Elastic half-spaces	5			
	2.2	Adhesion Interfaces	7			
	2.3	Scattering from imperfectly bonded half-spaces	10			
3	Seri	Serial Implementation and Bottleneck Analysis				
	3.1	Computing \mathbf{Z}_1 and \mathbf{Z}_2	16			
	3.2	Computing the residual column vector $\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots$	19			
	3.3	Computation and assembly of the Jacobian matrix	20			
	3.4	Computation and assembly of the scattered displacement field $\ldots \ldots \ldots$	21			
	3.5	Finding the bottleneck of the Scattering Algorithm	22			
4	Usir	ng GPU solution in the critical path	26			
	4.1	Solving a Dense Linear System	26			
	4.2	LU factorization	29			
		4.2.1 Permutation matrices	32			
	4.3	Parallel LU factorization	33			
	4.4	"Null Guess" optimization	41			
5	Res	ults and discussion	44			
	5.1	First case	46			

AĮ	Appendix A - Impedance Tensors in Isotropic Media 7						
Re	References						
6	6 Conclusion and future works						
	5.8	Final consideration	68				
	5.7	Other Simulations	65				
	5.6	Profiling the GPU	62				
	5.5	Fifth case	59				
	5.4	Forth case	56				
	5.3	Third case	53				
	5.2	Second case	50				

Chapter 1

Introduction

Adhesive bonds are an efficient method to join different components in structural design. Its emerging utilization is intimately related with the advantage that this solution presents in opposition to traditional bonding methods such as riveted or welded joints: the efficient load transfer between its constituents elements. This is, for instance, the situation presented in multilayered composites, when many constituent layers of different materials are bonded together in order to create a different material, as schematically represented in Figure 1.1.



Figure 1.1: Multilayered composite.

In terms of mechanical properties, five regions can be distinguished in any adhesive bond: the adherents, the adhesive layer itself, and two thin adhesion interfaces between the adhesive layer and the adherents. As an illustrative example, a typical adhesive layer thickness is on the order of hundred of microns, whereas a typical adhesion interface thickness is on the order of a few microns. However, as small as it seems, it is observed that degradation of this thin interface, rather than the entire adhesive layer, can lead to disastrous failure [23]. In that context, the absence of a reliable non-destructive inspection method to attest the integrity of such a thin and difficult-to-access region is the main factor that prevents further application of this type of bonding. The implementation of such a inspecting method is still an open issue.

Among possible solutions for this particular task, the use of ultrasound techniques to inspect the interfaces have emerged and been proposed by many researchers in the last few decades (see, e.g., [16, 17, 9, 15, 7, 1, 22, 10]). In this context, it is consensual that the capability of modeling both mathematically and computationally the interaction between ultrasonic waves and adhesion interfaces will play a crucial role in the development of any ultrasonic inspecting method.

So far, the vast majority of the authors have considered homogeneous adhesion interfaces in theoretical as well as in experimental works. Considering the interfaces as homogeneous simplifies enormously the wave propagation modeling. However, it implies that the bond is intact or uniformly degraded, which is not a realistic assumption. It is more realistic to consider locally degraded bonds, i.e., that only a small portion of the bond is degraded. This spatial inhomogeneity gives rise to the scattering of ultrasonic waves, bringing new challenges to the modeling.

Leiderman et al [26] have implemented a recursive algorithm based on an adaptation of the invariant embedding technique to numerically simulate the interaction between ultrasonic waves and defective interfaces. The algorithm is numerically stable, even for evanescent wave components at high frequencies. They used a quasi-static approximation (QSA) to model adhesion interfaces and the perturbation method to account for nonuniform defects. However, as it is common for this method, they have shown that the resulting series will converge for some value of the parameters, but will diverge for others. In Ref. [31], Nakagawa et al have studied a similar problem, solving directly a discretization of the resulting convolution integral equation of an inhomogeneous fracture between two identical isotropic half-spaces. However, his formulation is overwhelmingly complex and becomes intractable for the case of layered medium. Later, Leiderman and Castello have presented in Ref. [27] a whole new approach to simulate the interaction between ultrasonic waves and heterogeneous interfaces. The QSA approximation was still used to model the adhesion interfaces, but the direct scattering problem was now formulated as least-squares problem that was then solved accordingly. The developed method is well suited for anisotropic as well as for isotropic layers and does not suffer from any of the pitfalls of the two other mentioned works, being relatively simple to implement, numerically stable and convergent for any case. In that sense, we stand that it is the best method to modeling the interaction between ultrasonic waves and defective interfaces. We would like to emphasize that, to the best of our knowledge, only these two authors

have treated non-homogeneous interfaces.

In Ref. [28], the authors have utilized the Particle Swarm Optimization (PSO) technique combined with the same algorithm presented in [27] to solve the correlated inverse problem. That is, they have shown how to recover the stiffness distribution at an internal interface of a layered composite from the reflected ultrasonic field measured at its top. Since the PSO is an iterative search method that works heuristically, the associated direct problem of modeling the interaction between ultrasonic waves and defective interfaces have to be solved hundreds (or even thousands!) of times to find the best individual (each individual corresponds to a different interfacial stiffness distribution).

The implementation related to the work presented in Ref. [27] was developed using MATLAB and takes around a minute to solve the direct problem. Therefore, it makes it non feasible to solve the inverse problem in real time, since this requires the direct problem to be solved hundreds - or thousands - of times, as discussed above, making it impractical to be used in industrial ultrasonic inspection systems, such as the one schematically depicted in Figure 1.2. In that sense, in the present work we revisit the implementation of the algorithm presented in Ref. [27], aiming to optimize it and, therefore, to reduce considerably the execution time. In order to achieve this goal, we divide the algorithm into distinct major stages, each one having its particular tasks and processes. We then analyze each stage individually to understand which one(s) takes longer to run. As we find which stages/tasks are critical in the processing, we model the problem for a SIMT (Single Instruction Multiple Thread) parallelism strategy, suitable for a GPU parallel implementation, capable to accelerate the performance of the bottlenecks. Last, we perform simulations of a few case studies to compare the execution time of our new enhanced code with the previous implementations. Roughly speaking, as will be seen later, our new implementation has reduced the execution time by a factor of 25.



Figure 1.2: Inspecting scheme.

In what follows, we present the mathematical formulation used for modeling the interaction between ultrasonic inspecting waves and defective interfaces. Then, in Chapter 3, we describe the algorithm proposed in Ref. [27] and find the bottlenecks in execution based in theoretical complexity analysis and computational tests, and in Chapter 4 we show the proposed optimization for the execution of this bottleneck. In Chapter 5, we present and discuss the results obtained with our new implementation. Finally, we conclude in Chapter 6, where we also propose future works.

Chapter 2

Theoretical Formulation

The scenario considered in the present work is schematically represented in Figure 2.1. As it is observed, there are two solid elastic half-spaces and, in addition, we assume that the thin interface between these half-spaces is actually an interfacial layer with its own constitutive properties.



Figure 2.1: scenario considered in the present work. \mathbf{u}_2^+ is the incident displacement field, \mathbf{u}_1^+ is the reflected displacement field and \mathbf{u}_1^- is the transmitted displacement field.

2.1 Elastic half-spaces

First, we should say that we assume that all the field variables are time-harmonic dependent, i.e., the temporal dependence is represented by $e^{-i\omega t}$. Following, we define some field variables crucial to the development of this work:

$$\mathbf{u} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix}^T \tag{2.1}$$

is the displacement vector field defined at each point of the domain. u_x is the component in the x direction, u_y is the component in the y direction and u_z is the component in the z direction. Each component is a function of coordinates x, y and z. Notice that they may be complex values.

$$\mathbf{t} = \begin{bmatrix} \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{bmatrix}^T.$$
(2.2)

is the traction vector containing the stress tensor components acting at the plane xy. Each component is a function of coordinates x, y and z and it may be a complex value.

Based on the invariant embedding technique [8], our formulation is developed, firstly, decomposing the displacement \mathbf{u} and the traction \mathbf{t} into upgoing and downgoing fields:

$$\mathbf{u} = \mathbf{u}_1 + \mathbf{u}_2, \tag{2.3}$$

$$\mathbf{t} = \mathbf{t}_1 + \mathbf{t}_2. \tag{2.4}$$

In the equations 2.3 and 2.4 the subscript 1 is related to upgoing fields while subscript 2 refers to downgoing fields. In a practical example, \mathbf{u}_1 is the displacement field traveling in the positive direction of z. On the other hand, \mathbf{t}_2 is the traction field associated with the wave traveling in the negative vertical direction z.

From the exact solution of the elastodynamic equations of motion, we may define the local impedance tensors \mathbf{Z}_1 and \mathbf{Z}_2 that relate up and downgoing tractions with the corresponding displacement, or velocity, fields:

$$\bar{\mathbf{t}}_{\alpha} = -i\omega \, \mathbf{Z}_{\alpha} \, \bar{\mathbf{u}}_{\alpha}, \qquad \alpha = 1, 2. \tag{2.5}$$

These tensors are 3-by-3 matrix operators. Further details and expressions for tensors \mathbf{Z}_1 and \mathbf{Z}_2 for isotropic medium can be found at the appendix of this work. For the case of anisotropic medium, it can be found in Refs [26, 25]. The bar over the fields variables in Equation 2.5 represent that these fields are no longer evaluated on the Cartesian coordinates, but, in fact, a Fourier Transform has been applied at each component in order to transform the x spatial direction into the x component of the wavenumber vector (k_x) .

The wavenumber vector **k** always point into the propagation direction and its modulus k is related with the velocity of propagation c and the angular frequency ω by equation

$$k = \frac{\omega}{c}.\tag{2.6}$$

The x component of the wavenumber vector (k_x) is the projection of vector **k** on the x-axis, such as

$$k_x = k \sin\theta. \tag{2.7}$$

Figure 2.2 illustrates this decomposition. Notice that a simple application of the Pythagoras' theorem is sufficient to find k_x assuming that both the wavenumber k and θ , which is the propagation angle, are known. It means that k_x is indicative of the propagation angle and vice-versa.



Figure 2.2: Decomposition of wavenumber vector k.

2.2 Adhesion Interfaces

We consider the Quasi Static Approximation (QSA) for interfaces. This approximation was apparently first proposed by Baik and Thompson in Ref. [5]. The QSA is valid when the inspecting wavelength is much larger than the interfacial layer thickness (the interface is considered as infinitesimally thick), so that it is possible to replace it by a set of tangential and normal springs as schematically represented in Figure 2.3.



Figure 2.3: The Quasistatic approximation (QSA). We replace the interface by a set of tangential and normal springs.

In the QSA, the springs connect the two substrates and enforce continuity of the traction and (approximately) displacement vectors. It gives us the following boundary conditions:

$$\mathbf{K}(\mathbf{u}^+ - \mathbf{u}^-) = \mathbf{t}^+, \tag{2.8}$$

$$\mathbf{t}^{+} = \mathbf{t}^{-}.\tag{2.9}$$

The superscript "+" indicates the values of the field variables immediately above the interface, while the superscript "-" indicates those immediately below. **K** is the spring diagonal matrix that represents the effective interfacial stiffness. Its entries are normal and tangential spring constants:

$$\mathbf{K} = \begin{bmatrix} K_{xx} & 0 & 0\\ 0 & K_{yy} & 0\\ 0 & 0 & K_{zz} \end{bmatrix}.$$
 (2.10)

The QSA has been broadly used in theoretical and experimental works to model adhesive bonding, fractured embedded in solid medium and rough contact interfaces between solids (see, e.g., [35, 39, 21, 29, 33, 34, 18, 31, 24, 36]). The precise values of the spring constants can be written in terms of the elastic properties and nominal thickness of the considered interfacial layer [35]. In practice, values of \mathbf{K} can be experimentally obtained with the aid of inverse methods.

Although the QSA is valid when the inspecting wavelength is much larger than the interface thickness, it is possible to give different interpretations to boundary conditions defined in Eqs 2.8 and 2.9, depending on the frequency range of interest. At higher frequencies, where the wavelength is shorter or comparable to the adhesive layer thickness, the interface between the adhesive layer and the adherent are modeled by Eqs. 2.8 and 2.9, with the matrix \mathbf{K} being interpreted as the adhesion interface stiffness. In this case, the adhesive layer itself is modeled as an entire separate elastic layer of finite thickness. On the other hand, at lower frequencies, the wavelength is much larger than the thickness of the adhesive layer, so that the entire adhesive layer is modeled by Eqs. 2.8 and 2.9 and the matrix \mathbf{K} portrays the adhesion stiffness. Examples of the usage of Eqs. 2.8 and 2.9 to model the rough contact interface between two solids can be found in [39, 31, 24, 36]. In this case, the matrix \mathbf{K} is proportional to the interfacial contact area between the solids.

In the QSA context, adhesion defects are represented in the reduction of the springs constants. It is based on the notion that bond strength and bond stiffness are correlated. The vast majority of authors have considered uniform adhesion imperfections, so that along the whole extension of the interface it is observed uniformly reduced stiffness. However, this approach is not realistic. In more realistic scenarios, localized flaws in defective adhesive bonds are observed and, within the QSA approximation, these local imperfections should be modeled by corresponding local reductions in the spring stiffness constants in **K**. Accordingly, in the present work, imperfections are supposed to be localized, implying that $\mathbf{K} = \mathbf{K}(x)$, i.e., the interfacial stiffness is allowed to vary in the x direction.

This spatial heterogeneity breaks the problem's translational invariance upon which depend analytical methods, and gives rise for the scattering effect. In the present context, the scattering effect is characterized by the presence of a broad range of k_x components in the field's spectrum. For instance, if a plane wave impinges the interface with a defined incident angle, it will be reflected and transmitted in several different directions (different transmission and reflection angles). This is in contrast to the geometric reflection/transmission where a plane wave will be reflected and transmitted as a single plane wave only. This effect is schematically represented in Figure 2.4.



Figure 2.4: The scattering effect observed.

2.3 Scattering from imperfectly bonded half-spaces

Based on the mathematical basis introduced above, it is possible to develop the formulation for the situation depicted in figure 2.1 to compute the resulting scattered field \mathbf{u}_1^+ . We should say that we took the main idea from the field of parameter estimation and reformulate the scattering problem as an inverse problem. Accordingly, our first step is to take an initial guess for $\bar{\mathbf{u}}_1^+$ (we recall that the incident field \mathbf{u}_2^+ is known). Then, we calculate $\bar{\mathbf{t}}^+$ with the aid of Eqs. 2.4 and 2.5, and use Eq. 2.5 again in conjunction with the Boundary Condition 2.9 to solve for $\bar{\mathbf{u}}_2^-$ (we recall that there are only downpropagating waves in the lower halfspace - radiation condition, i.e, $\mathbf{u}_2^- = \mathbf{u}^-$):

$$\bar{\mathbf{u}}_{2}^{-} = [\mathbf{Z}_{2}^{-}]^{-1} (-i\omega)^{-1} \bar{\mathbf{t}}^{+}.$$
(2.11)

Finally, we apply the inverse Fourier Transform to find \mathbf{u}_2^- .

As it will be shown in later sections, this result can be used in conjunction with \mathbf{u}_2^+ to write a cost function based on Eq. 2.8, whose minimization parameter is $\bar{\mathbf{u}}_1^+$ so that the value assumed by this function is zero when the value assumed by the minimization parameter satisfies the Boundary Condition 2.8. In that sense, \mathbf{u}_1^+ responsible for minimizing this cost function is also the solution for the current scattering problem.

First, we define \mathbf{r}_j as the residual vector:

$$\mathbf{r}_j = \mathbf{K}_j (\mathbf{u}_j^+ - \mathbf{u}_j^-) - \mathbf{t}_j^+.$$
(2.12)

Then, the cost function can be written simply as the ordinary least-squares norm as:

$$\sum_{j=1}^{M} [real \{\mathbf{r}_j\}]^T [real \{\mathbf{r}_j\}] + [imag \{\mathbf{r}_j\}]^T [imag \{\mathbf{r}_j\}], \qquad (2.13)$$

where M is the total number of points at the interface in which the fields variables are evaluated and $\mathbf{u}_j(\mathbf{t}_j, \mathbf{K}_j)$ is the value of the field variable at the j_{th} point in the interface. The minimum number of points at the interface where the field variables have to be evaluated in order to get accurate results and ensure the solution uniqueness is still an open issue. So far, the same number of points that was used to perform the DFT have been used.

As it is shown in equation 2.13, the real and imaginary parts of the residual \mathbf{r}_j must be treated independently. In that sense, the minimization parameter vector \mathbf{p} can be written as:

$$\mathbf{p} = \begin{bmatrix} [real \{\bar{\mathbf{u}}_{11}^+\}]^T, & [imag \{\bar{\mathbf{u}}_{11}^+\}]^T, & [real \{\bar{\mathbf{u}}_{12}^+\}]^T, & [imag \{\bar{\mathbf{u}}_{12}^+\}]^T, \\ \dots & [real \{\bar{\mathbf{u}}_{1N}^+\}]^T, & [imag \{\bar{\mathbf{u}}_{1N}^+\}]^T, \end{bmatrix}$$
(2.14)

where

$$\bar{\mathbf{u}}_{1\beta}^{+} = \bar{\mathbf{u}}_{1}^{+}(k_{x\beta}), \qquad \beta = 1, 2, 3, ..., N$$
 (2.15)

and $k_{x\beta}$ are the k_x wave number components present in the harmonic wave spectrum. Notice that N is the total number of wave number components. So, if the most general case is to be considered, the length of **p** is $N \times 2 \times 3 = 6N$, where 2 represents the real and imaginary parts of minimization parameters, while 3 is related with all the 3 possible spatial directions.

It is possible to conclude that the cost function Equation 2.13 is quadratic with respect to the minimization parameters because each residual \mathbf{r}_j represented in the cost function is a linear function of the minimization parameter vector represented in Equation 2.14. Since the cost function is quadratic, it only has one (global) minimum, and the Jacobian (sensitivity) matrix associated to the problem is not a function of the minimization parameters. In this particular case, the minimization can be performed in a single iteration, as shown in [3]:

$$\mathbf{p}^* = \mathbf{p}^{(0)} - [\mathbf{J}^T \mathbf{J}]^{-1} \mathbf{J}^T \mathbf{r}(\mathbf{p}^{(0)}), \qquad (2.16)$$

where \mathbf{p}^* is an array that contains the solution for the scattering problem, $\mathbf{p}^{(0)}$ is the minimization parameter vector that contains the initial guess for the scattered field, \mathbf{J} is the Jacobian matrix computed, and $\mathbf{r}(\mathbf{p}^0)$ is the residual column vector evaluated for the initial guess $\mathbf{p}^{(0)}$. In the most general case, the length of \mathbf{r} is $M \times 2 \times 3 = 6M$ (using the same considerations mentioned above for the length of \mathbf{p}). The components of the Jacobian matrix are determined using the finite differences. If the forward difference is used, it can be represented as:

$$J_{ij} = \frac{r_i(p_1^{(0)}, p_2^{(0)}, ..., p_j^{(0)} + \varepsilon, ..., p_N^{(0)}) - r_i(p_1^{(0)}, p_2^{(0)}, ..., p_j^{(0)}, ..., p_N^{(0)})}{\varepsilon}.$$
 (2.17)

In Equation 2.17, ε is a small parameter. It is noticeable that **J** is a matrix with dimensions $6M \times 6N$ and that the j_{th} column of **J** can be assembled from the j_{th} minimization parameter perturbation. It is important to recall that \mathbf{u}^- is a function of the minimization parameter in Equation 2.12 and has to be computed with Eq. 2.11.

Another point worth mentioning is that if the same points in the x-axis are to be used to compute the residual vector and to perform the FFT, then the resulting Jacobian matrix \mathbf{J} is square. For the case of square \mathbf{J} , it is possible to simplify Equation 2.16 to

$$\mathbf{p}^* = \mathbf{p}^{(0)} - \mathbf{J}^{-1} \mathbf{r}(\mathbf{p}^{(0)}).$$
(2.18)

At this point, we should say that in the present work we perform the discrete version of Fourier transform (FFT), where the frequency components represented in the harmonic x-wave number spectrum are linked to the x axis discretization. Further, it is intrinsic to the formulation presented here that first we have to guess the wave number spectrum's frequency content (or, conversely, discretize the x axis) to choose the initial guess accordingly. For the discretization, two issues have to be taken into account:

i) As it was discussed earlier (and will be seen in the numerical results presented in Chapter 5), the scattered field spectrum is expected to be broader than the specular field because this broadening effect is intimately related with the scattering of the incident field by the defective interface. In that sense, it is difficult to know a priori which is the largest frequency component that has to be represented in the spectrum. For the solution to be

considered valid, Δx must be small enough to capture this largest value, i.e, it must be small enough to avoid aliasing. Anyway, it is always possible to identify in the scattered field spectrum the cases where aliasing had occurred. For this cases, a finer discretization of the x-axis is required in order to consider larger k_x components.

ii) Δk_x must be small enough for all the excited modes that appear in the wave number spectrum in the form of "spikes" (as seen in the numerical results presented in Chapter 5) to be represented accurately. In fact, in general, the larger is the angular frequency (ω) of the inspecting incident field, the thinner becomes that "spikes". We recall that Δk_x is related to the extent of the represented x-axis. The larger the x-axis representation, the smaller Δk_x .

In the sense of what is discussed in the two items above, it can be seen that for several cases a fine x-axis discretization will be needed. Besides that, we will need to represent a large extent of the x-axis. Thus, for these cases, we will have a large number of minimization parameters and a relevant bottleneck for fast and real time solving strategies. In this work we will introduce novel modelling and solution representations, turning the method much more faster and close to real time simulation.

Chapter 3

Serial Implementation and Bottleneck Analysis

On this chapter, we present and discuss the serial implementation of the algorithm to compute the scattered field resulting from the interaction of a time harmonic ultrasonic incident wave with a plane arbitrarily heterogeneous interface located between two halfspaces described and formulated in Chapter 2. For a better comprehension, a pseudocode of this algorithm will be presented.

After presenting and discussing the algorithm, we will detail it in minor tasks and processes in order to analyze meticulously the time spent by each stage and its relevance for the global time spent by the entire application. As it will be shown, since the algorithm is composed mostly of dense linear algebra operations such as matrix/vector multiplications and solving linear systems, and application of Fourier transforms, a theoretical time analysis is made in order to predict possible bottlenecks through the workflow. Ultimately, some simulations will be made to illustrate and confirm those assumptions.

The method to compute the scattered displacement field consists of a series of calculations based on the mathematical formulation presented in Chapter 2. From now on, the integer **n** represents the total number of points considered in the discretization of the interface. As input, it receives numerical data that characterizes the upper and lower halfspaces, the distribution of the stiffness of springs along the defective interface, and the incident displacement field which is composed of 3-by-**n** complex arrays, where 3 stands for each one of the spatial directions x, y and z.

As the phase speed of an elastic wave can be considered a property of the medium, we use the *P*-wave speed and *S*-wave speed as well as the density ρ as constants to characterize the half-spaces. From these values, it is possible to calculate Lamé first and second parameters, respectively, λ and μ (also known as the shear modulus) that will be used later in the program. All of these parameters that characterizes the media are to be considered as real scalar values.

A summary of the computational procedures corresponding to the Scattering Algorithm based in the theoretical formulation is described as follows and can be schematically seen in Figure 3.1.

Algorithm 1: Scattering Algorithm.				
input : 3-by-n complex arrays \mathbf{u}_2^+ , 3-by-n complex arrays \mathbf{u}_1^+ , 3-by-3-by-n real				
matrix K , real scalar constants ρ , \mathbf{v}_p and \mathbf{v}_s for each medium.				
output: 3-by- n complex arrays \mathbf{u}_{scat}				

- 1 Compute tensors \mathbf{Z}_1 and \mathbf{Z}_2 for each wavenumber $\mathbf{k}_{\beta x}$ component;
- ${\bf 2}$ Compute residual vector ${\bf r}({\bf p}^0)$ for the initial guess for ${\bf u}_1^+$;
- ${\bf 3}$ Assembly of Jacobian matrix ${\bf J}$;
- 4 Compute \mathbf{p}^* and assemble the scattered displacement field $\bar{\mathbf{u}}_{scat}$;



"Scattering Algorithm" flow chart

Figure 3.1: Flow chart of the Scattering Algorithm.

As depicted in the Scattering Algorithm and Figure 3.1, the program must sequentially accomplish four different stages (each one dependent of the previous) in order to compute correctly the 3-by-**n** complex array \mathbf{u}_{scat} that represents the scattered displacement field.

First, it must compute the local impedance tensor \mathbf{Z}_1 and \mathbf{Z}_2 for each component of the wavenumber vector x domain discretization. These tensors are computationally treated as 3-by-3-by-**n** complex matrices and will be needed at the next stage, when the 6**n**-size real array $\mathbf{r}(\mathbf{p}^0)$ representing the residual column vector will be calculated based on the Eq. 2.12 for an arbitrary 3-by-**n** complex array \mathbf{u}_1^+ that stands for the initial guess for the solution of the problem. The third stage consists on the assembly of the Jacobian matrix, a 6**n**-by-6**n** real matrix, using the same calculations as before, but with the subtle difference that emerges from perturbing each component of the 6**n**-size real array $\mathbf{p}^{(0)}$ at a time. Finally, we compute 6**n**-size real array \mathbf{p}^* by solving Equation 2.18 and assemble the 3-by-**n** complex matrix \mathbf{u}_{scat} that is the solution of the scattering problem. Each one of these stages will be better described in the next sections.

3.1 Computing Z_1 and Z_2

As algorithms 2 and 3 shows, the computation of 3-by-3 complex matrices \mathbf{Z}_1 and \mathbf{Z}_2 for a k_x component basically consists of a few value attributions, the calculation of the inverse of matrix \mathbf{A} and its multiplication by matrix \mathbf{L} (both \mathbf{A} and \mathbf{L} are defined in the algorithm itself). Notice that those matrices are square and of order 3 so this computation can be performed very fast. However, as mentioned in Algorithm 1, it must be done for each value of k_x .

Algorithm 2: Compute Z_1

input : real scalar constants μ , ρ and λ , real value \mathbf{k}_x and ω . **output:** 3-by-3 complex matrix \mathbf{Z}_1 Compute velocity of propagation of a P-type wave 1 $\mathbf{C}_P \leftarrow \mathrm{SQRT}(\lambda + 2\mu/\rho)$; Compute velocity of propagation of a S-type wave 3 $\mathbf{C}_S \leftarrow \mathrm{SQRT}(\mu/\rho)$; $\mathbf{k}_P \leftarrow \omega / \mathbf{C}_P$; $\mathbf{k}_S \leftarrow \omega / \mathbf{C}_S$; $\mathbf{k}_{zP} \leftarrow \mathrm{SQRT}(\mathbf{k}_P^2/\mathbf{k}_x^2)$; s $\mathbf{k}_{zS} \leftarrow \text{SQRT}(\mathbf{k}_S^2/\mathbf{k}_x^2)$; $\sin(\theta_P) \leftarrow \mathbf{k}_x/\mathbf{k}_P$; $\cos(\theta_P) \leftarrow \mathbf{k}_{zP}/\mathbf{k}_P$; $\sin(\theta_S) \leftarrow \mathbf{k}_x/\mathbf{k}_S$; $\cos(\theta_P) \leftarrow \mathbf{k}_{zS}/\mathbf{k}_S$; Setting matrix A_1 13 $A_1[0][0] \leftarrow \sin(\theta_P)$; $A_1[0][1] \leftarrow -\cos(\theta_S)$; $A_1[0][2] \leftarrow 0.0$; $A_1[1][0] \leftarrow 0.0$; $A_1[1][1] \leftarrow 0.0$; $A_1[1][2] \leftarrow 1.0$; $A_1[2][0] \leftarrow \cos(\theta_P)$; $A_1[2][1] \leftarrow \sin(\theta_S)$; $A_1[2][2] \leftarrow 0.0$; Setting matrix L_1 $\mathbf{23}$ $L_1[0][0] \leftarrow -\mu * (\mathbf{k}_x * \cos(\theta_P) + \mathbf{k}_{zP} * \sin(\theta_P));$ $L_1[0][1] \leftarrow \mu * (\mathbf{k}_{zS} * \mathbf{cos}(\theta_S) - \mathbf{k}_x * \mathbf{sin}(\theta_S));$ $L_1[0][2] \leftarrow 0.0$; $L_1[1][0] \leftarrow 0.0$; $L_1[1][1] \leftarrow 0.0$; $L_1[1][2] \leftarrow -\mu * \mathbf{k}_{zS}$; $L_1[2][0] \leftarrow -\lambda * (\mathbf{k}_{zP} * \mathbf{cos}(\theta_P) + \mathbf{k}_x * \mathbf{sin}(\theta_P)) - 2\mu * \mathbf{k}_{zP} * \mathbf{cos}(\theta_P);$ $L_1[2][1] \leftarrow \lambda * (\mathbf{k}_x * \mathbf{cos}(\theta_S) - \mathbf{k}_{zS} * \mathbf{sin}(\theta_S)) - 2\mu * \mathbf{k}_{zS} * \mathbf{cos}(\theta_S);$ $L_1[2][2] \leftarrow 0.0$; Multiplying matrices 33 $\mathbf{Z}_1(\mathbf{k}_x) \leftarrow (L_1 * A_1^{-1})/\omega$; return $\mathbf{Z}_1(\mathbf{k}_x)$;

Algorithm 3: Compute \mathbf{Z}_2

input : real scalar constants μ , ρ and λ , real values \mathbf{k}_x and ω . output: 3-by-3 complex matrix \mathbf{Z}_2 . 1 Compute velocity of propagation of a P-type wave $\mathbf{C}_P \leftarrow \mathrm{SQRT}(\lambda + 2\mu/\rho)$; Compute velocity of propagation of a S-type wave 3 $\mathbf{C}_S \leftarrow \mathrm{SQRT}(\mu/\rho)$; $\mathbf{k}_P \leftarrow \omega / \mathbf{C}_P$; $\mathbf{k}_S \leftarrow \omega / \mathbf{C}_S$; $\mathbf{k}_{zP} \leftarrow \mathrm{SQRT}(\mathbf{k}_P^2/\mathbf{k}_x^2)$; s $\mathbf{k}_{zS} \leftarrow \text{SQRT}(\mathbf{k}_{S}^{2}/\mathbf{k}_{r}^{2})$; $\sin(\theta_P) \leftarrow \mathbf{k}_x/\mathbf{k}_P$; $\cos(\theta_P) \leftarrow \mathbf{k}_{zP}/\mathbf{k}_P$; $\sin(\theta_S) \leftarrow \mathbf{k}_x/\mathbf{k}_S$; $\cos(\theta_P) \leftarrow \mathbf{k}_{zS}/\mathbf{k}_S$; Setting matrix A_2 13 $A_2[0][0] \leftarrow \sin(\theta_P)$; $A_2[0][1] \leftarrow \cos(\theta_S)$; $A_2[0][2] \leftarrow 0.0$; $A_2[1][0] \leftarrow 0.0$; $A_2[1][1] \leftarrow 0.0$; $A_2[1][2] \leftarrow 1.0$; $A_2[2][0] \leftarrow -\cos(\theta_P)$; $A_2[2][1] \leftarrow \sin(\theta_S)$; $A_2[2][2] \leftarrow 0.0$; Setting matrix L_2 23 $L_2[0][0] \leftarrow \mu * (\mathbf{k}_x * \mathbf{cos}(\theta_P) + \mathbf{k}_{zP} * \mathbf{sin}(\theta_P));$ $L_2[0][1] \leftarrow \mu * (\mathbf{k}_{zS} * \mathbf{cos}(\theta_S) - \mathbf{k}_x * \mathbf{sin}(\theta_S));$ $L_2[0][2] \leftarrow 0.0$; $L_2[1][0] \leftarrow 0.0$; $L_2[1][1] \leftarrow 0.0$; $L_2[1][2] \leftarrow \mu * \mathbf{k}_{zS}$; $L_2[2][0] \leftarrow -\lambda * (\mathbf{k}_{zP} * \mathbf{cos}(\theta_P) + \mathbf{k}_x * \mathbf{sin}(\theta_P)) - 2\mu * \mathbf{k}_{zP} * \mathbf{cos}(\theta_P);$ $L_2[2][1] \leftarrow -\lambda * (\mathbf{k}_x * \cos(\theta_S) - \mathbf{k}_{zS} * \sin(\theta_S)) + 2\mu * \mathbf{k}_{zS} * \cos(\theta_S);$ $L_2[2][2] \leftarrow 0.0$; Multiplying matrices 33 $\mathbf{Z}_2(\mathbf{k}_x) \leftarrow (L_2 * A_2^{-1})/\omega$; return $\mathbf{Z}_2(\mathbf{k}_x)$;

As mentioned before, further details and expressions for tensors \mathbf{A} , \mathbf{L} and \mathbf{Z} for isotropic media can be found in the appendix of this work while references [26, 25] treats the case for anisotropic media.

3.2 Computing the residual column vector

The 6n-length real array $\mathbf{r}(\mathbf{p}^0)$ is calculated based on Eqs. 2.9, 2.11 and 2.12 of the mathematical formulation. It is noticeable that the 3-by-3-by-n matrices \mathbf{Z}_1 and \mathbf{Z}_2 computed in the previous stage will be needed for this calculation.

Algorithm 4: Compute residual column vector $\mathbf{r}(p^0)$ input : 3-by-n complex arrays \mathbf{u}_2^+ , 3-by-n complex arrays \mathbf{u}_1^+ , 3-by-3-by-n matrix **K**, 3-by-3-by-**n** complex matrices \mathbf{Z}_1 and \mathbf{Z}_2 . output: 6n-size real array $\mathbf{r}(\mathbf{p}^0)$ 1 for i = 1 to 3 do $\bar{\mathbf{u}}_2^+(i) \leftarrow FFT(\mathbf{u}_2^+(i));$ $\mathbf{2}$ $\mathbf{\bar{u}}_1^+(i) \leftarrow FFT(\mathbf{u}_1^+(i));$ 3 4 for j = 1 to n do $\left| \quad \mathbf{\bar{t}}^+ \leftarrow (-i\omega) * (\mathbf{Z}_1^+ * \mathbf{\bar{u}}_1^+ + \mathbf{Z}_2^+ * \mathbf{\bar{u}}_2^+) \right|;$ $\bar{\mathbf{u}}^- = (\mathbf{Z}_2^-)^{-1} * (-i\omega) * \bar{\mathbf{t}}^+;$ 7 for i = 1 to 3 do $\mathbf{t}^+(i) \leftarrow IFFT(\bar{\mathbf{t}}^+(i));$ $\mathbf{u}^{-}(i) \leftarrow IFFT(\bar{\mathbf{u}}^{-}(i));$ 9 Since $\mathbf{u}^+ = \mathbf{u}_1^+ + \mathbf{u}_2^+$ 10 11 for j = 1 to n do 12**13** cont1 \leftarrow 0 ; 14 for j = 1 to n do $\mathbf{r}(p^0)(cont1) \leftarrow REAL(\mathbf{r}(1,j));$ 15 $\mathbf{r}(p^0)(cont1+1) \leftarrow REAL(\mathbf{r}(2,j));$ 16 $\mathbf{r}(p^0)(cont1+2) \leftarrow REAL(\mathbf{r}(3,j));$ $\mathbf{17}$ $\mathbf{r}(p^0)(cont1+3) \leftarrow IMAG(\mathbf{r}(1,j));$ 18 $\mathbf{r}(p^0)(cont1+4) \leftarrow IMAG(\mathbf{r}(2,j));$ 19 $\mathbf{r}(p^0)(cont1+5) \leftarrow IMAG(\mathbf{r}(3,j));$ 20 $cont1 \leftarrow cont1 + 6$; $\mathbf{21}$ **22** return $\mathbf{r}(\mathbf{p}^0)$;

Notice that register i (generally going from 1 to 3) is considered for calculations at each spatial direction, while register j computes at each point of the discretization (from 1 to **n**).

As described in Algorithm 4, at this stage we are required to apply Fourier transformations (direct or inverse) at the fields variables arrays in order to transform domain from spatial to wavenumber coordinates. This procedure is explicit in lines 1 - 3 and 7 - 9. Notice that these transforms are applied for each direction independently. Lines 4 - 6 describes the calculation of 3-by-**n** complex arrays $\bar{\mathbf{t}}^+$ and $\bar{\mathbf{u}}_2^-$ for each point of the wavenumber domain, while lines 14 - 21 counts for the 6**n** real array $\mathbf{r}(\mathbf{p}^0)$ computation and assembly. It is yet worth mentioning that as the final residual column vector $\mathbf{r}(\mathbf{p}^0)$ contain only real values, complex numbers must be broken in real and imaginary parts independently (functions *REAL* and *IMAG* in the algorithm).

3.3 Computation and assembly of the Jacobian matrix

The computation and assembly of the Jacobian matrix \mathbf{J} are described in Algorithm 5. At this stage, the algorithm fills each column of \mathbf{J} at a time repeating the same procedure: first, it computes a new residual column vector \mathbf{r}_{guess} that results from the application of a small perturbation ε at one of the minimization parameters contained in \mathbf{p}^0 array. Then, it uses the finite difference (see Equation 2.17) of \mathbf{r}_{guess} and the original residual column vector $\mathbf{r}(\mathbf{p}^0)$ computed at the previous stage to fill the corresponding column of \mathbf{J} . Notice that the output matrix is supposed to be square and of order 6**n** filled with real values.

Algorithm 5: Computation and assembly of Jacobian matrix J						
input : 3-by- n complex arrays \mathbf{u}_2^+ , 3-by- <i>n</i> complex arrays \mathbf{u}_1^+ , 3-by-3-by- n real						
matrix K , 3-by-3-by- n complex matrices \mathbf{Z}_1 and \mathbf{Z}_2 , real scalar ε						
output: $6n$ -by- $6n$ real matrix J						
1 $\bar{\mathbf{u}}_{guess} \leftarrow \bar{\mathbf{u}}_1^+$;						
2 for each column (col) of J do						
3 $point \leftarrow \lfloor col/6 \rfloor$;						
4 $coord \leftarrow 2 * \lfloor (col \% 6)/2 \rfloor$;						
$5 type \leftarrow ((col \% 6) \% 2) ;$						
$6 if(type == 1) \qquad type \leftarrow I \qquad (\text{imaginary part})$						
$7 if(type == 0) \qquad type \leftarrow 1 \qquad (real part)$						
$\mathbf{s} [\bar{\mathbf{u}}_{guess}[coord][point] \leftarrow \bar{\mathbf{u}}_{guess}[coord][point] + type * \epsilon ;$						
9 =====================================						
10 Use Algorithm 4 to compute residual vector \mathbf{r}_{guess} for $\bar{\mathbf{u}}_1^+ = \bar{\mathbf{u}}_{guess}$;						
11 ====================================						
12 for $j = 1$ to $6n$ do						
13 $ [\mathbf{J}[j][col] \leftarrow (\mathbf{r}_{guess}(j) - \mathbf{r}(\mathbf{p}^0)(j))/\epsilon ; $						
14 return J :						

Lines 3-7 identifies and relate the current column of the iteration with the point and coordinate of \mathbf{u}_{guess} upon which perturbation ϵ may be applied (line 8). Line 10 refers to a call for the same procedure of Algorithm 4, but this time, the initial guess utilized for calculation is updated with the small perturbation ε . Algorithm 4 will return a different residual vector \mathbf{r}_{guess} at each column computation. At line 13 each element of the column of matrix \mathbf{J} is computed from a forward finite difference scheme as mentioned. Finally, line 14 of the pseudocode deduct the perturbation from the related coordinated of \mathbf{u}_{guess} so that it is once again equivalent to the initial guess before another column is computed.

3.4 Computation and assembly of the scattered displacement field

Finally, at the last stage of the algorithm, the 3-by-**n** complex array \mathbf{u}_{scat} (solution of the scattering problem), is computed and assembled. The 6**n**-size real array \mathbf{p}^* that contains the solution is computed based on Eq. 2.18. This computation involves the matrix-vector multiplication of the inverse of the 6**n**-by-6**n** real matrix **J** by the 6**n**-size real array $\mathbf{r}(p^0)$. However, it is computationally advantageous to reframe this multiplication in a simple algebraic problem of solving a liner system. Doing so we avoid the difficult work of calculating the inverse of such a large matrix.

Suppose \mathbf{A} is a square matrix of coefficients, \mathbf{x} is a solution vector and \mathbf{b} is a right-hand side of the generic linear system:

$$\mathbf{A}\mathbf{x} = \mathbf{b}.\tag{3.1}$$

If \mathbf{A} is invertible, it is possible to multiply both sides of equality by the inverse of matrix \mathbf{A} , obtaining:

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b},\tag{3.2}$$

where \mathbf{A}^{-1} represents the inverse of \mathbf{A} .

Notice that the multiplication of any matrix by its inverse results in the identity matrix of adequate order. For this case, as consequence, the solution vector equals the multiplication of the inverse of matrix \mathbf{A} and the right-hand side vector \mathbf{b} .

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.\tag{3.3}$$

This means that it is possible to reformulate Eq. 2.18 to

$$\mathbf{p}^* = \mathbf{p}^{(0)} - \mathbf{y},\tag{3.4}$$

where \mathbf{x}' is the solution vector of the linear system $\mathbf{J} \cdot \mathbf{y} = \mathbf{r}(\mathbf{p}^{(0)})$.

Algorithm 6: Compute $\bar{\mathbf{u}}_{scat}$				
input : 6 <i>n</i> -by-6 <i>n</i> real jacobian matrix J , 6 <i>n</i> real residual column vector $\mathbf{r}(p^0)$				
output: 3-by- <i>n</i> complex arrays $\bar{\mathbf{u}}_{scat}$				
1 $\mathbf{p}^* \leftarrow \mathbf{p}^0 - \mathbf{J}^{-1} * \mathbf{r}(\mathbf{p}^{(0)})$;				
2 Assembly of $\bar{\mathbf{u}}_{scat}$				
$\mathbf{s} \ cont1 \leftarrow 0$;				
4 for $j = 1$ to n do				
5 $\bar{\mathbf{u}}_{scat}[0][j] \leftarrow \mathbf{p}^*(cont1) + i * \mathbf{p}^*(cont1+1);$				
$6 [\mathbf{\bar{u}}_{scat}[1][j] \leftarrow \mathbf{p}^*(cont1+2) + i * \mathbf{p}^*(cont1+3)$				
$\mathbf{\tau} \bar{\mathbf{u}}_{scat}[2][j] \leftarrow \mathbf{p}^*(cont1+4) + i * \mathbf{p}^*(cont1+5) ;$				
$\mathbf{s} _ cont1 \leftarrow cont1 + 6 ;$				
9 return $\bar{\mathbf{u}}_{scat}$;				

In the pseudocode for Algorithm 6, lines 3 - 8 represent the assembly of the 3-by-**n** complex array $\bar{\mathbf{u}}_{scat}$ from the 6**n**-size real array \mathbf{p}^* computed in line 1. Notice that this solution 3-by-**n** complex array is in the k_x domain.

3.5 Finding the bottleneck of the Scattering Algorithm

The algorithm described above was proposed by Leiderman and Castello in Ref. [27]. Later, the same authors have used its formulation again to solve the correlated inverse scattering problem in Ref. [28]. For better organization of our development, it will be analyzed based on the division of stages made in the previous sections. In this division, the whole model is compounded by four major stages: i) computation of the 3-by-3-by-**n** complex matrices \mathbf{Z}_1 and \mathbf{Z}_2 ; ii) computation of the 6**n** real array $\mathbf{r}(\mathbf{p}^0)$; iii) computation and assembly of the square matrix **J** of order 6**n**; iv) computation of the 3-by-**n** complex array \mathbf{u}_{scat} . We will analyze each one of these stages individually in order to find the most expensive tasks in terms of time complexity. This analysis gives a good intuition of where the bottleneck is during execution. Later, we will perform a few experimental simulations to validate this theoretical analysis.

In the first stage (Algorithms 2 and 3), the routine used to compute 3-by-3-by-**n** complex matrices \mathbf{Z}_1 and \mathbf{Z}_2 is composed basically of a few value attributions. These attributions are executed a constant number of times and it is independent to the size of the discretization. Some real value constants that characterize the half-space are used to compute the elements of matrices **A** and **L**. Notice that these matrices are square and of order 3, so multiplication and inversion are also performed with a constant number of operations. Therefore, its complexity is of order O(1). As these calculations are executed for every component of the domain, this routine is called **n** times during execution of the entire program. Thus, the total complexity of the first stage of the Scattering Algorithm is of order O(n).

At the second stage (computation of the residual column vector), there are tasks of very different nature being accomplished. Using the same observations made in the previous stage, we can stand that the complexity assigned for tasks that involve arithmetic operations with matrices and vectors of a pre-fixed low order are O(1). This is valid for the lines that denote calculation of fields variables such as traction and displacement fields and the $\mathbf{r}(\mathbf{p}^0)$ array. For each component of the discretization, values for $\mathbf{\bar{t}}$ and $\bar{\mathbf{u}}$ are computed based in matrix-vector operations, so the total cost of these tasks is of order O(n). In addition, the break of vector \mathbf{r}_j for attribution of vector $\mathbf{r}(\mathbf{p}^0)$ is of complexity O(n) as well, since the same quantity of operations are done to separate the real and imaginary parts of the complex vector \mathbf{r}_i for each component. The task within the second stage that has the largest complexity and, therefore, the associated higher cost, is the Fourier Transform over complex arrays $\bar{\mathbf{t}}_2^+$, $\bar{\mathbf{u}}_2^+$, $\bar{\mathbf{u}}_-$. In our implementation, for both direct and inverse transform, we use the same algorithm known as Fast Fourier Transform (FFT). This is a recursive algorithm based on the divide-and-conquer strategy that takes advantage upon the intrinsic symmetry of the problem to execute it in time $O(n \log(n))$. As it is explicited in Algorithm 4, each array representing fields variable $\bar{\mathbf{t}}_2^+$, $\bar{\mathbf{u}}_2^+, \, \bar{\mathbf{u}}^-$ execute the FFT for each spatial direction independently. Thus, in terms of order of magnitude and complexity, the group of tasks related to Fourier transforms hold the higher associated cost $(O(n \log(n)))$ in comparison with other tasks at this second stage (O(n)).

In the routine that computes and assembles the Jacobian matrix, first, it is made an

attribution of values for a 3-by-**n** complex array called $\bar{\mathbf{u}}_{guess}$ equivalent to the initial guess array $\bar{\mathbf{u}}_1^+$ of same size and dimension. This attribution task costs O(n) because it basically consists of copying information from each of the **n** points of domain from the former array. After this copy, each column of the Jacobian matrix is computed. The number of steps remains the same for each column and, therefore, to obtain the total cost of the entire routine, we multiply the number of columns (6n) by the resulting cost of a complexity analysis of the operations performed for each column. Inside this procedure, there is a call for the Algorithm 4 in order to compute a residual column vector related to the case when the initial guess array $\mathbf{p}^{(0)}$ suffers a small perturbation ε at one of its components. We can use the previous analysis to conclude that this call costs $O(n \log(n))$ since its most expensive task corresponds to the application of the FFT. This way, the total complexity of Stage 3 is of order $O(n^2 \log(n))$. Other tasks in the loop do not have complexity equal or greater to this value. As an example, the associated cost to identifying the coordinate and domain point for application of the perturbation (lines 3-7) is of order O(n) since it is made by a constant number of operations for each column. Additionally, the computation of each element of the column of the Jacobian matrix J (line 13) costs O(n) as for each of the 6n elements it is made a simple arithmetic operation of subtraction and division. Hence, in the total sum of all iterations, this step has cost of order $O(n^2)$.

Lastly, we analyze the final stage of the implementation. Eq. 2.18 is used to compute 6n-length real array p^* . As discussed earlier, instead of computing the inverse of the Jacobian matrix **J** and multiply it by the $\mathbf{r}(\mathbf{p}^0)$ array, it is much more interesting in the computational context to solve the implicit linear system. The solution of this linear system can be made by many possible direct or iterative methods. We will explain in more details in the next chapter of this work but, in resume, the best and most used method is to solve it with a LU factorization, since the Jacobian matrix **J** is assumed to be any generic square matrix.

The LU factorization method has two main steps: factorizing the coefficient matrix A into the product of a lower (L) triangular matrix by an upper (U) triangular matrix, and solving the resulting triangular linear system. The factorization step costs $O(n^3)$ while the solution of the resulting triangular linear systems is of order $O(n^2)$. Therefore, the cost associated with the computation of the 6**n**-size array **p**^{*} via Equation 2.18 is of order $O(n^3)$ and, therefore, it is the most expensive task of the final stage of the implementation. Other steps in the forth stage consists in the attribution and assembly of the 3-by-**n** complex array $\bar{\mathbf{u}}_{scat}$ (lines 3 - 8), executed in time O(n). Table 3.1 summarizes the conclusions made by the analysis of each stage of the implementation.

Table 3.1: Summary of theoretical complexity analysis for each stage of the Scattering Algorithm.

Stage	Most Expensive Task	Computational Cost
1	n O(1) order-3 matrix computations	O(n)
2	O(1) FFT of n -size arrays	$O(n \cdot log(n))$
3	O(n) FFTs of n -size arrays	$O(n^2 \cdot log(n))$
4	Solve linear system $\mathbf{J}\mathbf{y} = \mathbf{r}(\mathbf{p}^{(0)})$	$O(n^3)$
Overall	Solve linear system $\mathbf{J}\mathbf{y} = \mathbf{r}(\mathbf{p}^{(0)})$	$O(n^3)$

As conclusion, we found out that the bottleneck of the current implementation lies on solving the linear system $\mathbf{Jy} = \mathbf{r}(\mathbf{p}^{(0)})$. Table 3.2 shows the results from a few simulations that were made to confirm the theoretical analysis. These simulations use an entirely serial implementation which applies implementations from the FFTW[14] library to perform the FFT and a LU factorization from LAPACK[2] library to solve the bottleneck dense linear system. Details of this implementation, including environment aspects, half-spaces and defect characterization are discussed in the chapter 5 of this work.

				/	v	0		
	Stage 1		Stage 2		Stage 3		Stage 4	
Points	time (s)	%	time (s)	%	time (s)	%	time (s)	%
400	0.0011	0.1194	0.0003	0.0284	0.1971	21.5061	0.7179	78.3461
1200	0.0033	0.0149	0.0007	0.0032	1.9705	8.8671	20.2485	91.1149
2400	0.0066	0.0035	0.0015	0.0008	9.3440	4.9998	177.5339	94.9959
4800	0.0132	0.0008	0.0029	0.0002	40.8540	2.6009	1529.9221	97.3981

Table 3.2: Time/cost analysis for each stage.

As Table 3.2 illustrates, Stage 4 really represents the bottleneck of the serial implementation. Our theoretical analysis foresaw the trend observed in practice that, for larger domains, the linear system solution becomes dominant in time consumption. As an example, in a 400-point it took 0.72 seconds (approximately 78.35%) while in a 4800-point simulation, it took 1529.92 seconds (over 25 minutes), 97.39% of the entire execution time. Therefore, our main goal is to propose strategies to reduce the time taken to perform this particular computation.
Chapter 4

Using GPU solution in the critical path

Based on the complexity analysis, we propose a novel formulation for the problem, suitable for a SIMT (Single Instruction Multiple Thread) parallel paradigm. As mentioned in section 3.5 and detailed by Tables 3.1 and 3.2, the main bottleneck of the algorithm consist of finding the solution of a linear system $\mathbf{A}x = \mathbf{b}$. The Jacobian matrix \mathbf{J} takes the role as the coefficient matrix \mathbf{A} and the residual column vector $\mathbf{r}(\mathbf{p}^0)$ as the righthand side \mathbf{b} . We present a brief discussion to justify the method chosen to solve this linear system and explain its basic functioning before showing how it is executed in the parallel context, where large and highly parallelable operations are benefited from the computation being executed through GPU accelerators.

Additionally, we propose another optimization for the Scattering Algorithm based solely on the analysis of the initial guess for the reflected displacement field \mathbf{u}_1^+ . As it will be shown, if a null initial guess is chosen, the execution time of a simulation is relatively reduced (especially for discretizations with a reduced number of points) as a lot of unnecessary computation is avoided.

4.1 Solving a Dense Linear System

It is possible to computationally solve a linear system with direct or iterative methods. Theoretically, direct methods solve the system in such a way that the exact solution is found in a finite number of steps. This is accomplished through operations that modify the original coefficients matrix A to the point that this solution can be found by a simple backward substitution of the unknown variables. On the other hand, iterative methods such as Gauss-Seidel or the Conjugate Gradient reformulates the linear system so that, at each iteration, the initial guess for the system's solution is updated until it reaches

a value that is close enough to the original solution. Each method, whether direct or iterative, has its own advantages and inconveniences. While direct methods can be slower for many cases and hold round-off error associated with the arithmetic being used that may compromises the final solution, iterative methods need a preliminary analysis over convergence. In case this convergence can not be guaranteed, the method is inadequate as it may never find a solution that satisfies the problem.

In general, for systems of higher order (and, therefore, with larger coefficient matrices), it is assumed that direct methods are more suitable for dense matrices, while iterative methods works better for sparse ones. A linear system of equations is supposed to be dense if the entries of the coefficient matrix A are mostly composed of non-zero elements. Likewise, sparse linear systems are the ones in which the matrix A presents a high percentage of null elements. Computationally, large order sparse matrices have the big advantage of low memory occupancy, once it is not necessary to save the null entries of the matrix, although it is additionally required to save the data location. On the other hand, large and dense matrices demand large memory availability in order to store information.

In our particular problem, the Jacobian matrix assembled is dense because the vast majority of its elements are non-zero. In fact, we expect that all the elements are non-zero.

The most conventional direct method to solve a linear system of equations is the Gaussian Elimination. It consists, basically, of two steps: triangularization of the the system of equations and a backward substitution to solve the resulting system. The triangularization of the coefficient matrix A modifies the original matrix so that its elements under the main diagonal become zeros through operations of rows exchange and sum of entire rows with a multiple of another one.

$$(E_i) \leftrightarrow (E_j), \tag{4.1}$$

where E_i represents the entire *i*-th row,

$$(E_j + m_{j,i} Ei) \leftrightarrow (E_j) \tag{4.2}$$

and

$$m_{j,i} = \frac{a_{ji}^{(i)}}{a_{ii}^{(i)}}.$$
(4.3)

Notice that application of these operations does not change the original solution of the linear system as the right-hand side vector \mathbf{b} is also subject for the same operations performed for the matrix A.

Once the system is properly triangulated, it assumes the shape illustrated below,

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

in such a way that the solution can be found through a simple backward substitution.

$$x_n = \frac{b_n}{a_{nn}}$$
 and $x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{j=i+1}^n a_{ij} x_j \right].$ (4.4)

For our application, at least at first, it is not possible to predict the characteristics of the Jacobian matrix, so it is fundamental for our algorithm the ability to deal with a big possibility of distinct scenarios of both defective interfaces and incident wave fields. With that in mind, the model must adopt the method that can solve the linear system for generic matrices, i.e., matrices that are not of a special type such as symmetric, positive definite, diagonally dominant etc. In general, for these generic matrices the LU decomposition is the most indicated method, in opposition to others optimized implementations available in different libraries, such as QR, Cholesky and Bunch-Kaufman (LDL) factorizations. Each one of these factorizations is based in different principles to decompose the original coefficient matrix A into a product of matrices with very specific particularities.

Ref. [4] discuss that, as a viable alternative, the QR factorization is also suitable for this problem. However, despite the fact it has the great advantage of being numerically stable intrinsically, it is usually more time-expensive and its application is more adequate for solving least-squares problems where the number of unknowns exceeds the number of equations.

On the other hand, in order to achieve numerical stability, it is very common to add pivoting techniques to the LU method, although it increases the risk of the entire factorization to run slower than the alternative QR method. It is observed that, for most cases, the simple adding of this technique does not represent a significant increase in time cost and its application is proved worthy.

4.2 LU factorization

It takes $O(n^3/3)$ operations to transform the original coefficient matrix A into a triangular one, while to solve the resulting linear system by a backward substitution takes $O(n^2)$ operations. The number of operations required to solve a lower-triangular system is similar. In order to explore this discrepancy in time cost, it is very common to reconfigure the problem factorizing the matrix A into the product of a lower triangular matrix L and an upper triangular matrix U (A = LU). Then, we can solve for x more easily by using a two-step process.

- 1. Let y = Ux. Solve the lower triangular system Ly = b for y. Since L is triangular, determining y from this equation requires $O(n^2)$ operations.
- 2. Once y is known, the upper triangular system Ux = y requires an additional $O(n^2)$ operations to determine the solution x.

Solving the linear system Ax = b in factored form results in a reduction in operations from $O(n^3/3)$ to $O(2n^2)$ comparing with the usual Gaussian Elimination method. However, this reduction comes with a cost; it takes $O(n^3/3)$ operations to determine the specific lower and upper triangular matrices L and U. But once this factorization is determined, we can solve the linear system for the original matrix A by this simplified manner for any number of vectors b.

To perform the LU factorization, the Gaussian elimination should be performed on the system Ax = b without row interchanges. This observation is equivalent to having nonzero pivot elements at each step of the triangularization. At the end of the triangularization steps, the modified matrix $A^{(k)}$ is upper triangular. If we consider this modified matrix to be our desired matrix U, it is possible to determine the lower triangular matrix L considering the operations applied at the original matrix A.

The first step in the Gaussian elimination process consists of performing at the first element the operations described in Eq. 4.2 with i = 1, for each row below. All the entries in the first column below the diagonal are assumed as zero. The system of operations proposed can be viewed in another way as a multiplication of the original matrix A on the left by the elementary matrix $l^{(1)}$ resulting in $A^{(2)}$:

$$l^{(1)} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -m_{21} & 1 & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ -m_{n1} & 0 & \cdots & 0 & 1 \end{bmatrix}$$

$$l^{(1)} \cdot A = A^{(2)} \tag{4.5}$$

$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ -m_{21} & 1 & \ddots & & \vdots \\ \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ -m_{n1} & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ a_{11}^{(1)} & a_{22}^{(1)} & & \vdots \\ \vdots & & \ddots & & \vdots \\ a_{11}^{(1)} & \cdots & \cdots & a_{nn}^{(1)} \end{bmatrix} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & & a_{2n}^{(2)} \\ \vdots & & \ddots & \vdots \\ 0 & & & \ddots & \vdots \\ 0 & & a_{n2}^{(2)} & \cdots & \cdots & a_{nn}^{(2)} \end{bmatrix}.$$

In a similar manner, we build $l^{(2)}$, which is the identity matrix with the entries below the diagonal in the second column replaced by the negatives of the multipliers $m_{j,2}$. Multiplying $A^{(2)}$ on the left by $l^{(2)}$ will result in $A^{(3)}$ upon which every element below the first two elements of the diagonal are zeros. In general, for $A^{(k)}x = b(k)$, we will have $l^{(k)}$ so that

The process ends with the formation of $A^{(n)}x = b^{(n)}$, where $A^{(n)}$ is the upper triangular matrix U

$$A^{(n)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ddots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & a_{(n-1)n}^{(n-1)} \\ 0 & \cdots & \cdots & 0 & a_{nn}^{(n)} \end{bmatrix},$$

given by

$$l^{(n-1)} \cdot l^{(n-2)} \cdots l^{(1)} \cdot A = A^{(n)}.$$
(4.6)

Assuming M to be the resulting matrix of the multiplication of all the $l^{(k)}$ elementary matrices, we have that MA = U. Multiplying on the left by the inverse (assuming it is invertible), we have that

$$A = M^{-1}U. (4.7)$$

Since M is compounded by the product of matrices $l^{(k)}$, We can assume that its inverse is the also the product of the inverse of each particular elementary matrix l in the opposite order.

$$M^{-1} = (l^{(1)})^{-1} \cdot (l^{(2)})^{-1} \cdots (l^{(n-1)})^{-1}.$$
(4.8)

The inverse of each matrix $l^{(k)}$ is simple to be calculated:

$$(l^{(k)})^{-1} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & & & & \vdots \\ \vdots & & & & \vdots & m_{k+1,k} & \ddots & \ddots & & \vdots \\ \vdots & & & & \vdots & \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & & & & & \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & m_{n,k} & 0 & \cdots & 0 & 1 \end{bmatrix}$$

Therefore, the product of the inverse of these elementary matrices is our desired lower

triangular matrix L, that is:

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ m_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ m_{n1} & \cdots & m_{n,n-1} & 1 \end{bmatrix}$$

4.2.1 Permutation matrices

From a practical standpoint, the LU factorization presented is useful only when row interchanges are not required to control the round-off error that results from the used finitedigit arithmetic . Luckily, many systems actually can be solved without this important step, but we must consider the modifications that must be made when row interchanges are required.

An $n \times n$ permutation matrix $P = [p_{ij}]$ is a matrix obtained by rearranging the rows of I_n , the identity matrix. This gives a matrix with precisely one nonzero entry in each column and row (always being a 1). As an illustration, the matrix P presented below is a 3 × 3 permutation matrix.

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

If a generic 3×3 matrix A is multiplied on the left by P, we observe the effect of interchanging the second and third rows of A. If the multiplication was made on the right, the effect observed would be a column interchange in the matrix A.

$$PA = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

Two properties of permutations matrices are related to Gaussian elimination. The first, already described, shows that if we have a permutation of the integers 1, ..., n such as $k_1, ..., k_n$, the permutation matrix is defined as follows. The second is that P is invertible and $P^{-1} = P^T$

$$p_{ij} = \begin{cases} 1, & if \quad j = k_i \\ 0, & otherwise \end{cases}$$

The product PA permutes the rows of A

$$PA = \begin{bmatrix} a_{k11} & a_{k12} & \cdots & a_{k1n} \\ a_{k21} & a_{k22} & \cdots & a_{k2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{kn1} & a_{kn2} & \cdots & a_{knn} \end{bmatrix}.$$

This way, if we knew the row interchanges required to solve the linear system by Gaussian Elimination, then we could arrange the original equations in such a way that no row interchanges would be needed since all the pivots were always well placed at each step. Hence, there is a rearrangement of the equations in the system that permits Gaussian elimination to proceed without row interchanges. That is, for any non-singular matrix A, a permutation matrix P exists for which the system PAx = Pb can be solved without row interchanges and, as a consequence, this matrix PA can be factored into PA = LU, where L is lower triangular and U is upper triangular.

Multiplying both sides of the equality by P^{-1} (or equivalently P^T) produces the factorization

$$A = (P^T L)U. (4.9)$$

Notice that the matrix U is still upper triangular, but $P^T L$ is no longer triangular unless P = I.

Further details for the formulation presented in this section can be found in reference [11].

4.3 Parallel LU factorization

Parallelism in current computing architectures is omnipresent. This includes not only systems such as laptops and supercomputers of higher performance in the world, ranked by top500 list [38], but also smartphones, watches and other small portable devices. In fact, the scientific high performance computing community has faced drastic hardware changes since the arising of multicore architectures, in such a way that the level of heterogeneity in modern computing systems grows gradually with parallelism. In this context, CPUs multicore are usually combined with high performance accelerators such as graphic processing units (GPUs) and coprocessors like Intel's Xeon Phi. Extracting maximum performance from these systems' heterogeneity makes parallel programming of computational problems extremely challenging. This occurs because of the disparity between systems with this new architectures (many cores with reduced bandwidth and memory available per core) and the software traditionally associated with scientific applications, such as numerical libraries, upon which the High Performance Computing (HPC) community have relied for their performance and accuracy for a long time. Moreover, considering the future of research and development on parallel programming models, Ref. [13] observes that it is essential to take notice of two aspects: possible projected changes in systems architectures that will define environments on which HPC software will execute in the near future, and the nature of applications that this software will have to serve eventually. For the first case, it is possible to predict future architectures examining roadmaps of major hardware vendors and the next generations of platforms of supercomputers being developed. For the applications, it is reasonable to extract trends from the topics and subjects of study that today are already considered of highest interest like big-data analytics, deep learning etc.

As discussed, a traditional way of exploring parallelism without parallel programming de facto is through the use of implementations of already parallelized HPC libraries. Among them, linear algebra libraries and, in particular, DLA are of fundamental importance in a vast range of scientific and engineering applications. Thus, applications will not perform well if these libraries do not perform well.

Generally, a efficient parallel algorithm will present two important characteristics, as discussed in Ref.[12]:

1. Fine granularity:

Cores are associated with relatively small local memories. This mean that operations are splitted into tasks that cover small portions of data so that bus traffic is reduced as data locality is improved.

2. Asynchronicity:

Synchronization points in a parallel execution severely affects the efficiency of an algorithm as the degree of thread level parallelism (TLP) grows and granularity becomes smaller. On the other hand, the use of asynchronous execution usually hides the latency of access to memory.

The high performance dense linear algebra software libraries (i.e, LAPACK [2]) provides a wide set of linear algebra operations aiming to accomplish high performance in systems equipped with memory hierarchies. The algorithms implemented leverage the idea of blocking in order to limit the amount of bus traffic, making it possible for data to be reused. In general, LAPACK's factorization algorithms can be depicted as the repetition of two fundamental steps:

- 1. **Panel factorization**: depending of the DLA operation being performed, a number of transformations are computed for a small submatrix of the original problem (the panel). These transformations are typically computed by means of Level-2 BLAS operations (matrix-vector operations). They can be accumulated, although the way this accumulation occurs depends entirely on the operation performed.
- 2. Trailing submatrix update: all the transformations that have been accumulated in the panel factorization are to be applied to the rest of the matrix (the trailing submatrix). This update can be computed by means of Level-3 BLAS operations (matrix-matrix).

As follows, we present briefly the fundamental strategy behind the parallel LU factorization.

The algorithm has as input the coefficient square matrix A of order n and aims to compute a lower-triangular matrix L and an upper-triangular U so that A = LU. Matrix A is divided in submatrices A_{00} , A_{01} , A_{10} , A_{11} as illustrated in Figure 4.1. We assume that the panel is of order b so that A_{00} is of order bxb, A_{01} is of order bx(n-b), A_{10} is of order (n-b)xb and A_{11} is of order (n-b)x(n-b).

To make visualization clearer, we also divide matrices L and U, keeping in mind their properties: since L is lower-triangular and U is upper-triangular, all the elements above or below the diagonal are zeros (for U or L, respectively). This results that L_{00} and L_{11} are also lower-triangular and L_{01} is composed entirely of zero entries. Analogously, U_{00} and U_{11} are upper-triangular while U_{10} is zeroed. Since A = LU, we have that:





Figure 4.1: Parallel LU factorization

- I) $A_{00} = L_{00} * U_{00}$;
- II) $A_{01} = L_{00} * U_{01};$
- III) $A_{10} = L_{10} * U_{00} ;$
- IV) $A_{11} = L_{10} * U_{01} + L_{11} * U_{11}$.

I) In the first step of the algorithm, the panel factorization is performed at the A_{00} submatrix. Since this factorization is a difficult-to-parallelize task, a LU factorization is performed sequentially at this small portion of the original coefficients matrix A. As a result, L_{00} and U_{00} are computed.

2) Since L_{00} is computed in the first step, it is possible to compute U_{01} by a simple backward substitution for each column. Usually, sub-matrix U_{01} is divided in bxb matrices so the BLAS routine "trsm" can be performed as illustrated by Figure 4.2.

3) Analogously, L_{10} can also be computed by application of the "trsm" routine because U_{00} were also computed in the first step and it is a lower triangular matrix.



Figure 4.2: Illustrations of computation of submatrices U_{01} and L_{10} .

Steps 2 and 3 can occur independently, since there is no data dependencies between them. This means they are well suited to occur in parallel. Moreover, since matrices U_{01} and L_{10} were divided in *bxb* blocks, it is also possible to execute computation for each block independently in parallel. Figure 4.2 illustrates this procedure.

4) At this step, sub-matrix A_{11} needs to be update to get A_{11} so that:

 $A_{11}' = L_{11} * U_{11}$, hence

 $A_{11}' = A_{11} - L_{10} * U_{01}.$

This update can be made by a simple matrix multiplication computed in previous steps. Since $L_{10} * U_{01}$ is of order (n - b)x(n - b), this multiplication costs $O((n - b)^3)$ and it is the bottleneck of the algorithm. Still, as it is basically a Level-3 BLAS operation, it has very efficient parallel implementations that can be used to optimize execution time considerably.

After matrix A_{11}' is updated, the algorithm performs the same steps in the remainder matrix recursively until it reaches a bxb matrix $A_{11}^{(k)}$. At this point, a sequential LU factorization is applied and the whole matrix is factorized.



Figure 4.3: Scheme that illustrates how matrix A is factorized in the parallel LU algorithm. Darker areas of matrix A means factorization has been already performed.

It is yet worth mentioning that it is not entirely necessary to allocate two matrices to

store L and U coefficients. It can be written over the coefficients of the original matrix A as it is being factorized. Elements above the diagonal will refer to matrix U (including the elements of the diagonal), while elements below will refer to matrix L. It is only necessary to allocate another matrix if some technique of pivoting is to be applied. Even for those cases, it is not usually necessary to allocate another matrix of order nxn.

It is clear that in the algorithm described above, steps 2, 3 and 4 correspond to the "trailing submatrix update" mentioned earlier in this section. This two-phase division is typical of LAPACK's block algorithms. It consists in organizing the linear algebra algorithm in such a way that only a small portion of the total computation is accomplished at the "panel phase", while most part is performed in the "update phase". Doing so, the panel factorization is identified as a sequential task that represents a small fractions of the total number of flops: only $O(n^2)$ are in the "panel" while the total computation sums up to $O(n^3)$ flops. This first step is referred as a sequence of operations that can not be easily parallelized and, as an example, for the LU factorization, this applies to the "getf2" BLAS routine. As discussed in Ref. [6], parallelism in factorization algorithms are to be explored in Level-3 BLAs routines that are generally only present in the "trailing submatrix update" phase. As long as the majority of flops computed are in this phase, it is reasonable to expect that in a well made implementation the global performance of the algorithm is similar to the performance obtained by the Level-3 BLAS routine used in the update phase. This methodology implies the "fork-join" parallelism model in which the execution flow of factorization represents a sequence of sequential operations (panel factorizations) interleaved by parallel update ones. It is important to notice that each factorization algorithm and implementation may vary, modifying the pattern shown in Figure 4.3.

However, LAPACK have shown some limitations on multicore heterogeneous systems. The performance relies on the use of standard Level-3 Basic Linear Algebra Subprograms (BLAS), since these are the kind of operation that LAPACK can parallelize. This follows to the expensive "fork-join" paradigm, and have made prudent to revisit or redesign existing numerical linear algorithms to be better suited for such hardware.

The PLASMA project (Parallel Linear Algebra for Scalable Multicore Architectures) [20] uses tiled algorithms in order to obtain high performance. These algorithms can be represented as Directed Acyclic Graphs (DAGs) where nodes are tasks of fine granularity while edges represents data dependencies between the tasks. Thus, a runtime environment is used to schedule efficiently all the tasks in multicore platforms. This schedule provides very efficient algorithms capable of executing tasks of fine granularity asynchronously, therefore removing the expensive synchronizations associated with the "fork-join" between large tasks (which typically happens when BLAS is executed in parallel). However, this asynchronous scheduling have presented overhead of execution in tasks of fine granularity in many multicore architectures like current GPUs and Xeon Phi coprocessors.

A very popular alternative considered the state-of-the-art in terms of performance are the hybrid algorithms. Typically, in this approach, small or memory bounds tasks in the critical path of the algorithm are allocated to the CPU, while large data-parallel tasks are assigned to the GPU. This is the case for the hybrid algorithms implemented by the MAGMA[37] library. It makes use of high level tasks of coarse granularity that are split hierarchically into fine grain data-parallel subtasks through parallel BLAS implementations.

Although usually working fine, the strategy of scheduling the difficult-to-parallelize tasks on CPUs can present drawbacks if the balance between processors and accelerators is not well made, as pointed out in Ref. [19]. As an example, slow CPUs, even after tuning, can make a fast GPU idle. This effect can be even aggravated if the system presents a slow CPU-to-GPU communication bandwidth. Moreover, from an energy perspective, the hybrid approach consumes power in both hardwares (processor and accelerator) since they are operating simultaneously. Typically, the power efficiency of CPUs (measured in flops/watt) is lower than GPUs, so it is natural to expect hybrid algorithms to be less efficient in this particular context. On the other hand, implementations that use only the GPU during computation leave the CPU in idle state, where energy consumption is lower. This is one of the factors that have motivated the development of DLA algorithms that uses only the accelerator to perform the computation. Additionally, GPU-only codes are appealing because they remove the expensive cost related to the CPU-to-GPU communication and the associated challenge of tuning slow bandwidths. In fact, it is observed that in the most modern GPUs, GPU-only codes can achieve higher performance than the hybrid MAGMA algorithms when the difficult-to-parallelize CPU tasks and communications cannot be overlapped entirely by the GPU computations.

Development of GPU-only dense linear algebra algorithms have been avoided in the past because of two main reasons: i) the implementation and optimization of difficult-toparallelize computations could be avoided through the use of hybrid algorithm and; ii) as it was evident at that time, hybrid algorithms were usually faster. However, at least at first, it was necessary to develop algorithms of this type to cover cases when the size of the problem was so small that the total computation at the GPU was not enough to overlap the expensive CPU-to-GPU communication. The evolution of those algorithms dedicated entirely to GPUs made it possible to envisage efficient implementations for even larger problems.

Therefore, as indicated in the following chapter, we opt to use a GPU-only implementation of the LU factorization.

4.4 "Null Guess" optimization

The simple assumption that the initial guess for the reflected displacement field \mathbf{u}_1^+ is null, makes it possible to save operations and resources at the third stage (Algorithm 5). Computationally, this implies that \mathbf{u}_1^+ is a 3-by-**n** complex array completely filled with zeroes entries. In that manner, as calculations related with the fields variables of displacement and traction at the superior and inferior half-spaces involve multiplications of square matrix by vector (both of order 3) at each point of the domain, and as we have assumed that this array represents for each component a 3-dimensional vector that has only zero entries, we conclude that, in fact, the program might be doing a lot of unnecessary arithmetic operations, once it is known beforehand that the results are all going to be zero. Since our minimization method described in chapter 2 does not specify a initial guess for \mathbf{u}_1^+ , it is valid to assume it is null.

Within the assembly of the Jacobian matrix, for the computation of each column, a small perturbation ε is applied to the real or imaginary part of only one coordinate of one of the **n** components of $\bar{\mathbf{u}}_{guess}$. For any other component of this array, we assume that the zero value remains. Therefore, for each column being assembled only one matrix-vector multiplication is required, which is located at the component upon which the perturbation was applied. This way, the traction at the upper half-space ($\bar{\mathbf{t}}^+$) and the displacement at the lower half-space ($\bar{\mathbf{u}}^-$), both in the k_x domain, can be computed doing $(6n) \times (n-1)$ algebraic matrix-vector operations less than in the original implementation. Figure 4.4 illustrates the computation being spared: perturbation ε is being applied at component j = m. This means that for all other components ($j \neq m$) the computation of $\bar{\mathbf{t}}_1^+$ is unnecessary.



Figure 4.4: Illustration of the "Null Guess" optimization.

With that in mind, a small change has to be made at the previous implementation, in such a way that within the loop that assembles **J** only in the particular point where the perturbation ε is applied, the computation of the mentioned traction is performed. This requires a minor adaptation in Algorithm 4 when it is called from Algorithm 5 (Assembly of the Jacobian matrix) to prevent unnecessary calculations.

Algorithm 7: Adaptation of Algorithm 4 proposed for optimization.

input : 3-by-n complex arrays $\bar{\mathbf{u}}_2^+$, $\bar{\mathbf{u}}_{quess}$ and $\bar{\mathbf{t}}^+$, 3-by-3-by-n matrix K, 3-by-3-by-**n** complex matrices \mathbf{Z}_1 and \mathbf{Z}_2 , integer point. output: 6n real array \mathbf{r}_{guess} 1 $\bar{\mathbf{t}}_{guess} \leftarrow (-i\omega) * \mathbf{Z}_1^+(point) * \bar{\mathbf{u}}_{guess}(point) ;$ 2 $\bar{\mathbf{t}}^+(point) \leftarrow \bar{\mathbf{t}}^+(point) + \bar{\mathbf{t}}_{quess}$; s for j=1 to n do 4 $| \bar{\mathbf{u}}^- = (\mathbf{Z}_2^-)^{-1} * (-1\omega) * \bar{\mathbf{t}}^+$; 5 for *i*=1 to 3 do $\mathbf{t}^+(i) \leftarrow IFFT(\bar{\mathbf{t}}^+(i));$ 6 $\mathbf{u}_2^+(i) \leftarrow IFFT(\bar{\mathbf{u}}_2^+(i));$ 7 $\mathbf{u}^{-}(i) \leftarrow IFFT(\bar{\mathbf{u}}^{-}(i));$ 8 9 for j = 1 to n do $| \mathbf{r}(j) \leftarrow \mathbf{K}(j) * (\mathbf{u}^+(j) - \mathbf{u}^-(j)) - \mathbf{t}^+(j) ;$ 10 11 $cont1 \leftarrow 0$; 12 for j=1 to n do $\mathbf{r}_{quess}(cont1) \leftarrow REAL(\mathbf{r}(1,j));$ 13 $\mathbf{r}_{auess}(cont1+1) \leftarrow REAL(\mathbf{r}(2,j));$ 14 $\mathbf{r}_{quess}(cont1+2) \leftarrow REAL(\mathbf{r}(3,j));$ 15 $\mathbf{r}_{quess}(cont1+3) \leftarrow IMAG(\mathbf{r}(1,j));$ 16 $\mathbf{r}_{quess}(cont1+4) \leftarrow IMAG(\mathbf{r}(2,j));$ $\mathbf{17}$ $\mathbf{r}_{quess}(cont1+5) \leftarrow IMAG(\mathbf{r}(3,j));$ 18 $cont1 \leftarrow cont1 + 6$; 19 20 return \mathbf{r}_{guess} ;

Simulations with reduced domains can present considerable gains in terms of execution time for the third stage of the algorithm with the application of this simple proposed optimization. In the Chapter 5 we will show that, in the context of a relatively small number of minimization parameters, improvements at the third stage can be considered meaningful for the whole method since its relevance is bigger for these cases.

Chapter 5

Results and discussion

In this chapter, we compare all the implementations developed in the context of the present work. They are: i) a MATLAB [30] implementation proposed and implemented in Ref. [27]; ii) a C-language implementation, where the linear system bottleneck is solved with the serial implementation of LU factorization given in LAPACK library collection; iii) an implementation that solves the same linear system with the GPU-only LU factorization using the cuSOLVER library [32] included in NVIDIA's CUDA Toolkit. Simulations have been performed using two distinct systems with characteristics of CPUs and GPUs described in Table 5.1. System 1 has Ubuntu 17.04 as operational system and is composed of a processor Intel Core i7-7500U (2.70GHz) with 4 cores and 8 GB of memory RAM and a GeForce 940MX (Maxwell architecture) with 4 GB of memory RAM and 384 CUDA cores as GPU. System 2 operates in Ubuntu 16.04, with processors Intel Xeon E5-2698 (2.20 GHz) in a cluster with 80 cores and distributed 528 GB of memory RAM. In this system, only the cuSOLVER implementation was performed in a Tesla P100-SXM2 (Pascal architecture) with 16 GB of memory RAM and 3584 cuda cores. In a nutshell: implementation i), ii), and iii) were performed in an ordinary personal computer (System 1) while implementation iii) was performed also in the GPU cluster (System 2). As presented later, for each simulation, we considered a different defect.

CPU HOST RAM GPUGPU RAM SystemCPU cores memory GPU cores memory 1 Core i7-7500U GeForce 4 8 GB384 4 GB $(2.70 \, \text{GHz})$ 940MX 2Intel Xeon E5-2698 80 528 GBTesla 3584 16 GB(2.20 GHz)P100-SXM2

Table 5.1: Characteristics of system environments used in simulations.

As a validation, the implementations were tested for the scenario with no defect at the interface. For this particular case, the interface presents homogeneous stiffness along its whole extension and the problem can be solved analytically. The comparison between analytic and numeric results have validated the computational implementations. In addition, we compared the obtained numerical results with the numerical results obtained by the method introduced in Ref [26]. Furthermore, it is always possible to check if the obtained numerical results satisfy all the boundary conditions (zeroing the residual vector), confirming that it worked properly.

It is important to notice that for all simulations presented here it was assumed plain strain state (every field is constant in the y direction and $\frac{\partial}{\partial y} = 0$). It means that, for example, a simulation using a discretization of **n** points at the interface (and consequently **n** x-wavenumber components) will lead to 4**n** minimization parameters (**n**x2x2). This results in a Jacobian square matrix of order 4**n** with $(4\mathbf{n})^2$ elements.

In all the simulations we have assumed the ultrasonic inspection of two bonded halfspaces. The upper half-space is made of copper and the lower half-space is made of aluminum. Additionally, there is an epoxy layer with nominal thickness of 200 μm between these half-spaces that acts as the adhesive layer. The considered structure is illustrated in Figure 5.1. Table 5.2 shows the mechanical properties of the constituent materials.



Figure 5.1: Material structure considered in the numerical simulations.

	Density	P-wave speed	S-wave speed
Material	(kg/m^3)	(m/s)	(m/s)
Aluminum	2700	6320	3130
Copper	8930	4660	2160
Epoxy	1200	1480	1030

Table 5.2: Mechanical properties of layers represented in the model.

The incident field is a 100 kHz time harmonic P-wave with an angle of incidence of 60°. These values were selected based on the strategy introduced in Ref. [26], in order to enhance the scattering effect. For this inspecting frequency, the whole epoxy layer can be approximated by an equivalent interfacial stiffness, as discussed in the fourth paragraph of Section 2.2 (Adhesion Interfaces). Ref. [35] shows how to obtain the spring matrix corresponding to the QSA from the mechanical properties and geometry of the considered interfacial adhesive layer. For the present case, when the interface is intact, it is:

$$\mathbf{K} = \begin{bmatrix} 0.6389 & 0 & 0\\ 0 & 0.6389 & 0\\ 0 & 0 & 2.7685 \end{bmatrix} \times 10^{13} \, Pa/m.$$

As discussed earlier, to simulate locally degraded interfaces, we have assumed that matrix \mathbf{K} depends upon the position at the interface. More specifically, in defective regions, we reduced the original values in the diagonal of \mathbf{K} .

We recall here that, although we have considered the same structure for all simulations, our algorithm is suitable for all sorts of materials, frequencies, angles of incidence and defect geometries.

5.1 First case

Here, we assumed that the interface is defective with the zz component of **K** being characterized by a Gaussian with beam waist about 40 cm large with minimum value equal to 10% of the original interfacial stiffness. I.e., only 10% of the original stiffness remains at the peak of the defect. For the present simulation we have represented a portion of the structure extending about 2.14 m, corresponding to $x \approx -1.07$ m to $x \approx 1.07$ m, as shown in Figure 5.2.



Figure 5.2: Interfacial stiffness considered in example 1.

In order to perform the DFT and analyze the implementations performance, we discretized the interface with 400, 1000, 2500 and 5000 points, dividing our 2.14 m portion accordingly. Therefore, for the 400-points simulation, $\Delta x \approx 0.0054 \, m$. In terms of the wave number harmonic spectrum, it corresponds to $k_{xmin} \approx -583.84 \, m^{-1}$, $k_{xmax} \approx 580.92 \, m^{-1}$ and $\Delta k_x \approx 2.92 \, m^{-1}$. We recall that we work in the plane strain state. In that sense, 400 wave number components leads to 1600 minimizations parameters and a Jacobian matrix with 2560000 elements (1600 x 1600). Table 5.3 condenses this information for all performed simulations.

Table 0.9. Trumerical information of binduations for example 1.							
Points	x_{min}	(m)	$x_{max}(m)$	$\Delta x(m)$	$k_{xmin}(m^{-1})$	$k_{xmax}(m^{-1})$	$\Delta k_x(m^{-1})$
400	-1.0'	762	1.0708	0.0054	-583.8410	580.9218	2.9192
1000	-1.0	762	1.0708	0.0022	-1459.6024	1456.6832	2.9192
2500	-1.0'	762	1.0708	0.0008	-3649.0062	3646.0870	2.9192
5000	-1.0	762	1.0708	0.0004	-7298.0124	7295.0932	2.9192
			Minii	nization	Elements	in the	
		Poir	nts para	imeters	Jacobian mat	$rix (\times 10^6)$	
		40	0 1	.600	2.5	6	
		100)0 4	000	16.0	C	
		250)0 1	0000	100.	0	
		500)0 2	0000	400.	0	

Table 5.3: Numerical information of simulations for example 1.

For each simulation, the resulting spectrum of the scattered displacement field in the z direction is plotted in Figure 5.3. Table 5.4 shows the corresponding execution time.



Figure 5.3: The spectrum of the scattered displacement field in the z direction for simmulations of case 1.

	MATLAB	LAPACK	CUSOLVER	CUSOLVER
Points	(s)	(s)	*PC (s)	*DGX (s)
400	10,2829	$1,\!1575$	0,7336	0,9986
1000	$59,\!2131$	$12,\!9763$	4,0165	2,4446
2500	$465,\!8981$	$208,\!3537$	39,0873	$12,\!5161$
5000	$2444,\!8990$	$1757,\!2441$	$268,\!2733$	$51,\!2595$

Table 5.4: Execution time of simulations for example 1 in different implementations.

The plots presented in Figure 5.3 as well as Table 5.4 deserve some comments. First, notice that the resulting spectrum illustrates the broadening effect associated with the scattering of the incident field by the nonuniform interface. More specifically, the obtained scattered reflected field spectrum is broader than the incident field spectrum that is composed by only one k_x component (an incident plane wave). This broadening effect becomes harder to visualize as the number of points (number of considered k_x components) increases, but it is very clear in Figures 5.3(a) & (b). Second, for this particular example, it is obvious that we have considered much more components in the k_x spectrum than it is necessary (since there are only a few components that have significant amplitudes). It means that we could have performed the same simulation with much lesser points (a coarser discretization) and even then we would have valid results. However, this may be not the case where other scenarios are considered, as for example the case of layered structures (see, e.g., Ref. [27]), or when either higher frequencies or sharper defects are considered, as discussed at the end of Section 2.3. For these scenarios, a wider range of relevant k_x components are to be expected in the spectrum. This is partially seen in the next presented examples, where there is more than one mode (spike) present in the scattered reflected spectrum. As we intend to extend the present work to these scenarios too, it makes sense to work here with a large number of k_x components and, therefore, a large number of minimization parameters.

Second, as discussed in Chapter 1, in Ref. [28] the authors solved the related inverse problem of recovering the stiffness distribution at an internal interface of layered composite heuristically. In doing so, for each inverse problem they had to solve hundreds of times the direct problem (the current scattering problem), each direct problem corresponding to a different stiffness distribution. In that context, the results shown in Table 5.4 indicate that for the simulation with a x-axis discretization of 1000 points (a typical simulation size), our proposed implementation using cuSOLVER took around 2,44 seconds to be executed in system 2. In comparison with the MATLAB implementation that took around 59,21 seconds to execute, our new implementation has improved performance by a factor of around 25 times. As it will be discussed later in Chapter 6, this opens the possibility for the inverse problem to be solved in real time. Actually, as our discretization becomes finer, and, therefore, more points of the interface are considered, the bigger is the improvement, including here the improvement over the implementation using LAPACK that is around 35 times slower in simulations with a discretization of 5000 points.

5.2 Second case

For the second case, we assumed that the interface is defective with the zz components of **K** being characterized as a Gaussian with beam waist about 20 cm large with minimum value equal to 10% of the original interfacial stiffness. I.e, only 10% of the original stiffness remains at the peak of the defect. Differently from the previous case, for the present simulation we adopted a fixed value for Δx of 0.002 m. In that sense, an increase in the number of discretization points corresponds to an increase of the represented interface length (that corresponds, therefore, to a decrease in Δk_x). Figure 5.4 illustrates the stiffness distribution for all simulations performed in this example.



Figure 5.4: Interfacial stiffness considered in example 2.

In order to perform the DFT and analyze the performance of our implementation, we considered discretizations with 400, 1200, 2500 and 5000 points. As an example, for the 1200-point simulation, a portion of about 2, 4m was considered. In terms of the wave number harmonic spectrum, it corresponded to $k_{xmin} \cong -1570.79 \, m^{-1}$, $k_{xmax} \cong 1568.18 \, m^{-1}$ and $\Delta k_x \cong 0.42 \, m^{-1}$. We recall that we work in the plane strain state. Therefore, 1200 k_x components leads to 4800 minimization parameters and a Jacobian matrix with 23040000 elements (4800 x 4800). Table 5.5 condenses this information for all performed simulations.

Table 5.5: Numerical information of simulations for example 2.						
Points	$x_{min} m$	$x_{max} m$	$\Delta x m$	$k_{xmin} m^{-1}$	$k_{xmax} m^{-1}$	$\Delta k_x m^{-1}$
400	-0.400	0.3980	0.0020	-1570.7963	1568.1783	1.2500
1200	-1.200	1.1980	0.0020	-1570.7963	1568.1788	0.4167
2500	-2.5000	2.4980	0.0020	-1570.7963	1568.1788	0.2000
5000	-5.0000	4.9980	0.0020	-1570.7963	1568.1788	0.0999
		Minim	ization	Elements	s in the	
	Points	s paran	neters	Jacobian ma	trix $(\times 10^6)$	
	400	16	600	2.5	6	
	1200	48	00	23.0	04	
	2500	100	000	100	0.0	
	5000	20	000	400.0		

Table 5.5: Numerical information of simulations for example 2.

For each simulation, the resulting spectrum of the scattered displacement field in the z direction is plotted in Figure 5.5. Table 5.6 shows the corresponding execution time.



Figure 5.5: The spectrum of the scattered displacement field in the z direction in the upper half-space for simulations of example 2.

	MATLAB	LAPACK	cuSOLVER	CUSOLVER
Points	(s)	(s)	*PC (s)	*DGX (s)
400	9,2754	0,9821	0,8206	1,0809
1200	88,8934	22.4578	5.8723	3.5213
2500	467,0685	$207,\!5607$	39,0850	12,5161
5000	$2446,\!4900$	$1757,\!9672$	268,2178	$51,\!2595$

Table 5.6: Execution time of simulations for example 2 in different implementations.

Regarding the resulting spectrum of the scattered displacement field plotted in Figure 5.5, the same considerations made in the previous example still applies. In addition, it is clear in this example that there is more than one mode being excited by the interaction between the inspecting field and the defective interface (the extra two spikes present in the spectrum). Moreover, significant amplitude for negative k_x values evidenced the backscattering phenomenon characterized by reflected waves propagating in the negative x-direction.

Results shown in Table 5.6 indicate that for the simulation with x-axis discretization of 1200 points (a typical simulation size), our proposed implementation using cuSOLVER took around 3.52 seconds to be executed in *system* 2. In comparison with the MATLAB implementation that took around 88.89 seconds to execute, our new implementation has improved performance by a factor of around 25 times, that is similar to the previous example. As the discretization becomes finer, and, therefore, more points of the interface are considered, the bigger is the improvement.

5.3 Third case

Here, we assumed that the interface is defective with the zz components of the stiffness matrix **K** being characterized by a Gaussian with beam waist about only 2 cm large with minimum value equal to 10% of the original interface stiffness. I.e., only 10% of the original stiffness remains at the peak of the defect. For the present simulation we have represented a portion of the structure extending about 2.14 m, corresponding to $x \approx -1.07 m$ to $x \approx 1.07 m$, as shown in Figure 5.6.



Figure 5.6: Interfacial stiffness considered in example 3.

In order to perform the DFT and analyze the implementation performance, we considered samples of 400, 1000, 2500 and 5000 points, dividing our 2.14 m portion accordingly. Since we choose to repeat the discretizations adopted in the first case, Table 5.3 still condenses the information of the execution time for the present simulations.

For each simulation, the resulting spectrum of the scattered displacement field in the z-direction is plotted in Figure 5.7. Table 5.7 shows the corresponding execution time.



Figure 5.7: The spectrum of the scattered displacement field in the z direction in the upper half-space for simmulations of example 3.

	MATLAB	LAPACK	cuSOLVER	cuSOLVER
Points	(s)	(s)	*PC (s)	*DGX (s)
400	9,0128	0,9846	0,7660	1,0820
1000	$59,\!1961$	$12,\!9831$	$3,\!9078$	2,4209
2500	466,3943	$220,\!6346$	39,0041	$12,\!9574$
5000	2469,5021	$1758,\!6686$	$268,\!4357$	$54,\!5730$

Table 5.7: Execution time of simulations for example 3 in different implementations.

The same considerations made in the two final paragraphs of Section 5.1 and in the penultimate paragraph of Section 5.2 are also valid for the present case, i.e., it is possible to visualize in Figure 5.7 the broadening associated to the scattering effect, 3 different excited modes, and the backscattering phenomenon.

Results shown in Table 5.7 show that for the simulation with x-axis discretization of 1000 points (a typical simulation size), our proposed implementation using cuSOLVER took around 2.42 seconds to be executed in *system* 2. In comparison with the MATLAB implementation that took around 59.19 seconds to execute, our new implementation has, again, improved the performance by a factor of around 25 times. As before, as the discretization becomes finer, and, therefore, more points of the interface are considered, the bigger is the improvement.

5.4 Forth case

For the forth case, we assume that the interface is defective with the zz components of **K** reduced to 10% of its original value in a portion of the bond extending about 30 cm. I.e, for this simulation we have represented a *square* defect in a portion of the structure extending about 2.14 m, corresponding to $x \approx -1.07 m$ to $x \approx 1.07 m$, as shown in Figure 5.8.



Figure 5.8: Interfacial stiffness considered in example 4.

In order to perform the DFT and analyze our implementation performance, we discretized the interface with 400, 1000, 2500 and 5000 points, dividing our 2.14 m portion accordingly. Since we choose to repeat the discretization adopted in the first case, Table 5.3 condenses this information for all performed simulations.

For each simulation, the resulting spectrum of the scattered displacement field in the z direction is plotted in Figure 5.9. Table 5.8 shows the corresponding execution time.



Figure 5.9: The spectrum of the scattered displacement field in the z direction in the upper half-space for simmulations of example 4.

	MATLAB	LAPACK	cuSOLVER	cuSOLVER
Points	(s)	(s)	*PC (s)	*DGX (s)
400	9,3042	1,1515	0,7811	1,0789
1000	$59,\!9050$	14,7500	$3,\!8657$	2,4694
2500	466,7717	$233,\!9072$	39,0930	$13,\!6291$
5000	$2431,\!6890$	$1815,\!6481$	268,3206	52,7353

Table 5.8: Execution time of simulations for example 4 in different implementations.

The same considerations made in the two final paragraphs of Sections 5.1 and in the penultimate paragraph of 5.2 are also valid for the present case, i.e., it is possible to visualize in Figure 5.9 the broadening effect, 3 different excited modes and the backscattering phenomena in the spectrum.

Results shown in Table 5.7 indicate that for the simulation with x-axis discretization of 1000 points (a typical simulation size), our proposed implementation using cuSOLVER took around 2.46 seconds to be executed in *system* 2. In comparison with the MATLAB implementation that took around 59.90 seconds to execute, our new implementation has, again, improved performance by a factor of around 25 times. As the discretization becomes finer, and, therefore, more points of the interface are considered, the bigger is the improvement.

5.5 Fifth case

For the fifth and last case, we assume that the interface is defective with the zz components of the **K** reduced to 10% of its original value in a portion of the bond extending about 5 cm. For the present simulation, it was once again adopted a fixed value for Δx of 0.002 m. As discussed in the second case, this augment in the discretization reflects in a larger portion of the structure being represented. Figure 5.10 illustrates the stiffness distribution for all performed simulations.



Figure 5.10: Interfacial stiffness considered in example 5.

In order to perform the DFT and analyze the performance of our implementation, we considered discretizations with 400, 1200, 2500 and 5000 points. Since we choose to repeat the discretizations adopted for the second case, Table 5.5 condenses this information for all performed simulations.

For each simulation, the resulting spectrum of the scattered displacement field in the z direction is plotted in Figure 5.11. Table 5.9 shows the corresponding execution time.



Figure 5.11: The spectrum of the scattered displacement field in the z direction in the upper half-space for simmulations of example 5.
	MATLAB	LAPACK	cuSOLVER	cuSOLVER
Points	(s)	(s)	*PC (s)	*DGX (s)
400	8,9944	1,1617	0,8085	1,1078
1200	90,8279	24.4255	6.0201	3.6142
2500	466, 3943	$220,\!6346$	39,0041	12,9574
5000	$2469,\!5021$	$1758,\!6686$	$268,\!4357$	$54,\!5730$

Table 5.9: Execution time of simulations for example 5 in different implementations.

The same considerations made in the two final paragraphs of Sections 5.1 and in the penultimate paragraph of 5.2 are also valid for the present case, i.e., it is possible to visualize in Figure 5.9 the broadening effect, 3 different excited modes and the backscattering phenomena in the spectrum.

Results shown in Table 5.7 indicate that for the simulation with x-axis discretization of 1200 points (a typical simulation size), our proposed implementation using cuSOLVER took around 3.61 seconds to be executed in *system* 2. In comparison with the MATLAB implementation that took around 90.83 seconds to execute, our new implementation has improved performance by a factor of around 25 times. Actually, as the discretization becomes finer, and, therefore, more points of the interface are considered, the bigger is the improvement.

5.6 Profiling the GPU

In order to have a better view of how the GPU is behaving during the execution of our implementation, especially during the stage identified as the bottleneck (when the GPU is really called into action), we measured the time spent in each data transfer and kernel execution. We could record a CUDA event before and after each transfer or kernel execution, and use cudaEventElapsedTime(), but we can get the elapsed transfer time without instrumenting the source code with CUDA events by using nvprof, a commandline CUDA profiler included with the CUDA Toolkit. Moreover, later, we analyzed the output data file with NVIDIA Visual Profiler, also included in CUDA Toolkit. This is a very useful tool to optimize and tune CUDA codes with respect to the hardware specificities.

We choose to run and present the profile of simulations with a discretization of 5000 points in *System* 2 since it was the example upon which we observed greater improvement in performance. However, we recall here that, since our implementation uses a library

provided by NVIDIA, we expect that much of the code implementation (including kernel concurrent invocations and overlaps in execution) is already optimized and, therefore, there is very little we could possibly do to obtain even more improvements.

Table 5.10 shows the total time spent with the three main actions while GPU is active to perform the computation: i) transfer data from host (CPU) to device (GPU); ii) execute kernels to perform the computations; and iii) transfer the resulting data from device (GPU) to host (CPU).

Table 5.10: . Duration of actions performed in GPU during a 5000-points simulation

Action	Duration (s)
Memory copy (Host to Device)	0.3761
Memory copy (Device to Host)	0.24×10^{-3}
Kernel execution	1.8346

Results in Table 5.10 show that the amount of time spent during computation (1.83 seconds summing all 7542 kernel invocations) is about 5 times the amount spent during data transfer (totalizing around 0.3761 seconds summing the time spent copying memory from host to device and from device to host). In addition, the reason why most of the time spent in data transferring applies only to "Host to Device" is because we are only expected to transfer the large Jacobian matrix for computation purposes. We are not interested in obtaining the resulting factorized matrix, only the solution vector. It means that, for the simulation with a discretization of 5000 points, we do not need to transfer 3.2 GB of data back from the device to the host. This important economy saves a lot of data traffic in the GPU. In comparison, the solution vector data sizes only 160 KB, so this memory copy is performed much faster.

A very common strategy to optimize the performance of CUDA programs consists of overlapping memory copy of data between the CPU and the GPU with kernel invocation and execution, minimizing the time spent with memory traffic. However, it cannot be done in our implementation, since we are not able to overlap any memory copy with a kernel execution because the cuSOLVER library demands from the developer a pre-allocation of all the data involved in computation before any function from its collection are called from the code. More specifically, we need to transfer all the data related to the Jacobian matrix and the residual column vector and guarantee that the device is synchronized before calling the parallel LU solver. On the other hand, it would be interesting to use more than one copy engine to overlap the transfer of the Jacobian matrix data from host to device. The 3.2 GB data could be sent in multiple batches concurrently, possibly reducing considerably the time spent by our simulation. This approach could be applied in future optimizations, but it is restricted to more robust GPUs since it would depend entirely if the GPU available contains more than one "Host to Device" copy engine. Our implementation could only use this strategy if executed in *System* 2 because the Tesla 100-SXM2 has 5 copy engines integrated, while the GeForce 940MX has only one.

During the computation phase, the cuSOLVER library is capable of invoking and execute kernels concurrently. This is accomplished using two distinct CUDA streams that run a lot of the time concurrently. This is done optimally, taking advantage of operations that do not have data dependency to occur at the same time. Table 5.11 specifies the kernels executed during the factorization procedure highlighting their relevance (in percentage) and the number of times each one of them is invoked.

Table 5.11: . Kernels					
Kernel	Invocations	Relevance $(\%)$			
maxwell_dgemm_128x64_nn	214	40.8			
$getf2_domino$	1250	38.7			
laswp	2620	13.3			
$maxwell_dgemm_64x64_nn$	910	4.0			
$trsm_l_mul32$	1503	1.1			
$trsm_left$	83	1.1			
others (relevance < 1.0 %)	935	1.0			

As discussed in Setion 4.3, the critical path in the parallel LU algorithm corresponds to matrix multiplications in the "trailing submatrix update" phase. This fact is explicit in Table 5.11, as the maxwell_dgemm kernels (generic matrix-matrix multiplication in double precision BLAS routines) sum 44.8% relevance with 1124 invocations. The second most important kernel with 38.7% relevance and 1250 invocations is the getf2 routine, related to the "panel factorization" phase where small sub-portions of the original matrix is factorized in lower and upper submatrices. Since this routine is a difficult-to-parallelize task, it is executed sequentially in the GPU justifying its great relevance in the computation time. The laswp kernel, third in the relevance hierarchy (13.3%) with 2620 invocations, is responsible for making entire line swaps of the matrix during the factorization procedure. It is related to the pivoting technique used here to guarantee numerical stability. The trsm kernels are also part of the trailing phase, as discussed in the previous Chapter. It consists of the solution of small triangular sub systems generated during the factorization and sums 1586 invocations, but as it can be performed very fast, the total account is of only 2.2% relevance.

5.7 Other Simulations

Our proposed implementation using the cuSOLVER's GPU-only LU factorization implementation for solving the linear system was able to enhance performance significantly in comparison with the previous MATLAB implementation, and the serial LAPACK implementation tested (even though we assume that MATLAB has already well-optimized methods). Results show that the larger the number of points considered in the interface, larger is the absolute difference between performance of those implementations, although the *ratio* of this improvement can decay because of hardware limitations (especially for *system* 1). Moreover, executions in *system* 2 evidence that more modern and powerful GPU architectures are capable of really impressive performance. For simulations with a 5000-points discretization, the proposed method has obtained results executed around 46 times faster than the MATLAB implementation, 33 times faster than LAPACK serial implementation and even 5 times faster than the cuSOLVER implementation executed in *system* 1.

At this point, we should say that simulations with a very large number of points, especially seen in Table 5.14, find a very useful application for cases with ultrasonic waves of higher frequencies, where each adhesion interface is modeled individually and the adhesive is treated as a layer. In Refs. [26], this problem was solved with the perturbation method for discretizations with a number of points of order 10^4 . In this case, the resulting spectrum of the scattered displacement field presents many spikes (related to the modes excited) in a phenomenon called leaky waves, as shown in Fig. 14 of Ref. [26].

Based on these discussions, we executed simulations with the implementations proposed in this work increasing the number of points of the discretization from 400 points to the supported memory limit. Here, we choose to add 400 points to the discretization of the next simulation (400, 800, 1200 etc). Since the GPU in *system* 2 possess more memory (16 GB), it was possible to execute simulations with up to 11200 points in this system. Here, in order to avoid redundancy, we opt to omit the plotting of the displacement field spectrum since it is very similar to the spectra presented in the five previous cases. Yet, Tables 5.12, 5.13 and 5.14 shows the execution time of each simulation, highlighting the percentage computational time of each one of the four major stages of the algorithm described in Chapter 3.

LAPACK implementation in System 1						
	Stage 1	Stage 2	Stage 3	Stage 4	TOTAL	
Points	%	%	%	%	time (s)	
400	0.1194	0.0284	21.5081	78.3461	0.9164	
800	0.0321	0.0074	12.2072	87.7533	6.8417	
1200	0.0149	0.0032	8.8671	91.1149	22.2231	
1600	0.0045	0.0010	3.9616	96.0329	96.7085	
2000	0.0053	0.0012	6.0311	93.9624	103.0902	
2400	0.0035	0.0008	4.9998	94.9959	186.8859	
2800	0.0025	0.0006	4.2570	95.7398	298.3853	
3200	0.0019	0.0004	3.4916	96.5060	463.7924	
3600	0.0015	0.0003	3.3875	96.6107	648.3082	
4000	0.0012	0.0003	3.1075	96.8910	895.6887	
4400	0.0011	0.0002	3.1378	96.8608	1149.3483	
4800	0.0008	0.0002	2.6009	97.3981	1570.7922	
5200	0.0007	0.0002	2.6133	97.3858	1990.3980	

Table 5.12: Execution time of implementation using LAPACK library to solve the linear system in Stage 4.

Table 5.13: Execution time of implementation using cuSOLVER library to solve the linear system in Stage 4.

cuSOLVER implementation in System 1						
	Stage 1	Stage 2	Stage 3	Stage 4	TOTAL	
Points	%	%	%	%	time (s)	
400	0.0926	0.0124	14.9692	84.9258	1.3324	
800	0.1023	0.0148	36.0367	63.8462	2.3114	
1200	0.0625	0.0088	33.8514	66.0772	5.8039	
1600	0.0038	0.0055	30.9318	69.0589	12.3509	
2000	0.0271	0.0042	28.2072	71.7616	21.6047	
2400	0.0203	0.0030	24.9696	75.0071	35.1234	
2800	0.0154	0.0024	23.6908	76.2914	53.4647	
3200	0.0123	0.0018	21.0309	78.9549	77.3740	
3600	0.0096	0.0015	21.6331	78.3558	108.5451	
4000	0.0082	0.0013	19.3320	80.6586	143.1311	
4400	0.0068	0.0011	19.1327	80.8594	187.8935	
4800	0.0059	0.0009	17.2719	82.7213	238.6193	
5200	0.0051	0.0008	17.2397	82.7544	299.6355	

cuSOLVER implementation in System 2							
	Stage 1	Stage 2	Stage 3	Stage 4	TOTAL		
Points	%	%	%	%	time (s)		
400	0.2020	0.0271	23.7061	76.0648	0.9441		
800	0.1650	0.0271	53.9266	45.8813	1.7837		
1200	0.1433	0.0180	69.7378	30.1009	3.3464		
1600	0.1178	0.0144	76.0201	23.8477	5.3763		
2000	0.0913	0.0131	80.1465	19.7492	8.4394		
2400	0.0829	0.0108	82.0016	17.9046	11.8280		
2800	0.0642	0.0089	83.3254	16.6016	16.2091		
3200	0.0575	0.0076	83.5864	16.3486	21.7117		
3600	0.0486	0.0068	83.9558	15.9889	27.1398		
4000	0.0471	0.0066	84.1242	15.8222	33.1237		
4400	0.0432	0.0058	85.0645	14.8865	40.6586		
4800	0.0360	0.0050	84.6189	15.3402	49.2561		
5200	0.0336	0.0058	84.6426	15.3181	59.7855		
5600	0.0326	0.0047	84.6169	15.3458	63.1926		
6000	0.0220	0.0032	84.7944	15.1705	71.8268		
6400	0.0271	0.0039	83.0346	16.9344	86.6642		
6800	0.0244	0.0038	85.8317	14.5737	103.0061		
7200	0.0250	0.0036	84.3876	15.5838	105.5999		
7600	0.0216	0.0035	85.6401	14.3349	128.8178		
8000	0.2118	0.0031	83.6238	16.1613	140.7402		
8400	0.0213	0.0036	84.2743	15.7009	147.9404		
8800	0.0190	0.0033	84.7451	15.2326	171.7939		
9200	0.0174	0.0031	85.6389	14.3406	198.1747		
9600	0.0173	0.0028	82.0421	17.9377	203.6605		
10000	0.0174	0.0030	84.0997	15.8799	216.7895		
10400	0.0160	0.0027	84.4404	15.5409	244.1156		
10800	0.0161	0.0027	83.7137	16.2676	250.9999		
11200	0.0147	0.0025	82.9171	17.0658	283.9708		

Table 5.14: Execution time of implementation using cuSOLVER library to solve the linear system in Stage 4.

In system 1, it was not possible to execute simulations for much larger domains than 5200 points due to memory limitations. As the number of points grows, the Jacobian matrix becomes very heavy in terms of data storage; in a 5000-points simulation in double precision (8 bytes needed for each value stored), the square Jacobian matrix has 20000 columns and consequently takes up to 3.2 Gigabytes. This opens the door for future implementations of the present method in distributed memory systems to overcome this limitation.

Moreover, results presented in Table 5.14 show that the execution in *system* 2 was capable of accelerate the solution of the linear system that characterizes stage 4 of the algorithm in such a way that it is no longer the bottleneck of the problem. This occurs for discretizations of 800 points or larger, and the bottleneck becomes the third stage, i.e., the computation and assembly of the Jacobian matrix. This important result allows future studies of new possible optimizations of this implementation focused on improving the form this particular computation is done, including parallel approaches to perform this assembly.

5.8 Final consideration

Since the main goal was achieved, we can only expect that this new proposed implementation may be incorporated to enhance performance of the algorithm that solves the scattering problem in multilayered structures, first introduced in Ref. [27]. We expect the obtained gain to be transferred since the solution of the scattering problem in multilayered structures is based on a very similar formulation, and took around the same amount of time to be solved if compared to the problem with two halfspaces and one defective interface treated in the present work.

Moreover, the optimization achieved opens the door to its incorporation to the correlated inverse scattering problem, since it usually solves the direct problem hundred of times. Additionally, as the stiffness distribution at the interface is to be found heuristically by the Particle Swarm Optimization (in this method, each generation is compounded of several individuals, where each individual is a possible solution for the inverse scattering problem), we expect that each generation can be parallelized because the set of individuals at each generation can be computed concurrently, i.e., with no data dependencies. In the previous work (see, Ref. [28]), the solution was usually found in at least 6 generations, with each of them compounded of 20 individuals. If we assume that each direct problem was solved in one minute, and the whole method was implemented serially, it took around 200 minutes to find the stiffness distribution. On the other hand, we estimate that our new improved implementation to the direct problem (usually taking around 2,4 seconds) added to this parallel approach application in the PSO itself could lead to the solution of the inverse problem being computed in around 15 seconds. If we are to be pragmatic, we could say that the solution previously taking a few hours can be potentially found in a matter of seconds and, thus, be used in a real-time ultrasound inspection system.

Chapter 6

Conclusion and future works

In a previous work, an algorithm to numerically simulate the interaction between ultrasonic waves and localized adhesion defects in adhesive bonds was developed and implemented. The implementation was developed using MATLAB and takes around a minute to run, making it non feasible to solve the correlated inverse problem in real time. In the present work, we developed a novel GPU parallel implementation, aiming to reduce considerably the execution time. We then presented several simulations, comparing the execution time of our new enhanced code with the previous implementations. Roughly speaking, our new implementation has reduced the execution time by a factor of 25, opening the possibility for solving the inverse problem in real time, providing future developments in the adhesive bond industrial ultrasonic inspecting systems.

Results and discussions presented in this work envisage a list of possible future works related to the theme studied:

i) Applying the same strategy to enhance the performance of the direct scattering problem in multilayered structures, i.e., structures with two or more adhesion interfaces;

ii) Incorporate the implementation proposed to enhance the performance of the correlated inverse scattering problem;

iii) Since results with modern GPU architectures indicates that the third stage of implementation could be a new bottleneck, apply parallel computing strategies to obtain better performance in the computation and assembly of the Jacobian matrix;

iv) Make use of methods present in libraries with hybrid CPU-GPU implementations (notably MAGMA) to solve the linear system of stage 4. This approach could lead to even better performance results than the ones obtained using cuSOLVER's GPU-only approach.

References

- ALBIEZ, M., VALLÉE, T., UMMENHOFER, T. Adhesively bonded steel tubes Part II: Numerical modelling and strength prediction. *International Journal of Adhesion* and Adhesives 90 (2019), 211 – 224.
- [2] ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L. S., DEMMEL, J. W., DONGARRA, J. J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKEN-NEY, A., SORENSEN, D. LAPACK users' guide, 1992. SIAM, Philadelphia, PA. Avaiable at http://www.netlib.org/lapack/lug.
- [3] ASTER, R. C., BORCHER, B., THURBER, C. H. Parameter Estimation and Inverse Problems. Elsevier Academic Press, 2005.
- [4] BABOULIN, M., DONGARRA, J., TOMOV, S. Some issues in dense linear algebra for multicore and special purpose architectures.
- [5] BAIK, J.-M., THOMPSON, R. B. Ultrasonic scattering from imperfect interfaces: A quasi-static model. Journal of Nondestructive Evaluation 4, 3 (Dec 1984), 177–196.
- [6] BARRACHINA MIR, S., CASTILLO, M., IGUAL, F., MAYO, R., QUINTANA-ORTI, E. S. Solving dense linear systems on graphics processors. p. 739–748.
- BEALL, F. Monitoring of in-situ curing of various wood-bonding adhesives using acousto-ultrasonic transmission. International Journal of Adhesion and Adhesives 9 (1989), 21 – 25.
- [8] BELLMAN, R., KALABA, R. Functional equations, wave propagation and invariant imbedding. *Journal of Mathematics and Mechanics* 8, 5 (1959), 683–704.
- [9] BOCKENHEIMER, C., FATA, D., POSSART, W., ROTHENFUSSER, M., NETZEL-MANN, U., SCHAEFER, H. The method of non-linear ultrasound as a tool for the non-destructive inspection of structural epoxy-metal bonds—a résumé. *International Journal of Adhesion and Adhesives 22*, 3 (2002), 227 – 233.
- [10] BUDZIK, M., MASCARO, B., JUMEL, J., CASTAINGS, M., SHANAHAN, M. Monitoring of crosslinking of a dgeba-pamam adhesive in composite/aluminium bonded joint using mechanical and ultra-sound techniques. *International Journal of Adhesion* and Adhesives 35 (2012), 120 – 128.
- [11] BURDEN, R. L., FAIRES, D. Numerical Analysis. Cengage Learning, 2010.
- [12] BUTTARI, A., LANGOU, J., KURZAK, J., DONGARRA, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (janeiro de 2009), 38–53.

- [13] DONGARRA, J., ABALENKOVS, M., ABDELFATTAH, A., GATES, M., HAIDAR, A., KURZAK, J., LUSZCZEK, P., TOMOV, S., YAMAZAKI, I., YARKHAN, A. Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomput. Front. Innov.: Int. J.* 2, 4 (mar de 2015), 67–86.
- [14] FRIGO, M., JOHNSON, S. G. The design and implementation of FFTW3. Proceedings of the IEEE 93, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".
- [15] GALY, J., MOYSAN, J., MAHI, A. E., YLLA, N., MASSACRET, N. Controlled reduced-strength epoxy-aluminium joints validated by ultrasonic and mechanical measurements. *International Journal of Adhesion and Adhesives* 72 (2017), 139 – 146.
- [16] GAUTHIER, C., EL-KETTANI, M. E.-C., GALY, J., PREDOI, M., LEDUC, D., IZBICKI, J.-L. Lamb waves characterization of adhesion levels in aluminum/epoxy bi-layers with different cohesive and adhesive properties. *International Journal of* Adhesion and Adhesives 74 (2017), 15 – 20.
- [17] GAUTHIER, C., GALY, J., EL-KETTANI, M. E.-C., LEDUC, D., IZBICKI, J.-L. Evaluation of epoxy crosslinking using ultrasonic lamb waves. *International Journal* of Adhesion and Adhesives 80 (2018), 1 – 6.
- [18] GOLUB, M. Propagation of elastic waves in layered composites with microdefect concentration zones and their simulation with spring boundary conditions. *Acoustical Physics 56* (04 2010), 848–855.
- [19] HAIDAR, A., ABDELFATAH, A., TOMOV, S., DONGARRA, J. High-performance cholesky factorization for gpu-only execution. In *Proceedings of the General Purpose GPUs* (2017), GPGPU-10, p. 42–52.
- [20] INNOVATIVE COMPUTING LABORATORY, UNIVERSITY OF TENESSEE 2010. PLASMA User's Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3, 2019. Avaiable at http://icl.cs.utk.edu/projectsfiles/ plasma/pdf/users_guide.pdf/.
- [21] KARPUR, P., KUNDU, T., DITRI, J. Adhesive Joint Evaluation Using Lamb Wave Modes with Appropriate Displacement, Stress, and Energy Distribution Profiles, vol. 18. 01 1999, p. 1533–1542.
- [22] KUMAR, R. V., BHAT, M., MURTHY, C. Evaluation of kissing bond in composite adhesive lap joints using digital image correlation: Preliminary studies. *International Journal of Adhesion and Adhesives* 42 (2013), 60 – 68.
- [23] LAVRENTYEV, A. I., ROKHLIN, S. I. Models for ultrasonic characterization of environmental degradation of interfaces in adhesive joints. *Journal of Applied Physics* 76, 8 (1994), 4643–4650.
- [24] LAVRENTYEV, A. I., ROKHLIN, S. I. Ultrasonic spectroscopy of imperfect contact interfaces between a layer and two solids. *The Journal of the Acoustical Society of America 103*, 2 (1998), 657–664.

- [25] LEIDERMAN, R., BARBONE, P. E., BRAGA, A. M. B. Reconstructing the adhesion stiffness distribution in a laminated elastic plate: Exact and approximate inverse scattering solutions. *The Journal of the Acoustical Society of America* 122, 4 (2007), 1906–1916.
- [26] LEIDERMAN, R., BRAGA, A. M. B., BARBONE, P. E. Scattering of ultrasonic waves by defective adhesion interfaces in submerged laminated plates. *The Journal* of the Acoustical Society of America 118, 4 (2005), 2154–2166.
- [27] LEIDERMAN, R., CASTELLO, D. Scattering of ultrasonic waves by heterogeneous interfaces: Formulating the direct scattering problem as a least-squares problem. *The Journal of the Acoustical Society of America* 135, 1 (2014), 5–16.
- [28] LEIDERMAN, R., CASTELLO, D. Detecting and classifying interfacial defects by inverse ultrasound scattering analysis. *Wave Motion* 65 (05 2016).
- [29] LI, B., HEFETZ, M., ROKHLIN, S. I. Ultrasonic evaluation of environmentally degraded adhesive joints. In *Review of Progress in Quantitative Nondestructive Evaluation* (1992), D. O. Thompson and D. E. Chimenti, Eds., vol. 11, p. 1221–1228.
- [30] MATHWORKS. MATLAB, 2019. Avaiable at http://www.mathworks.com/ products/matlab.html/.
- [31] NAKAGAWA, S., NIHEI, T., MEYER, L. R. Plane wave solution for elastic wave scattering by a heterogeneous fracture. J. Acoust. Soc. Am 115 (2004), 2761–2772.
- [32] NVIDIA. cuSOLVER 10.0, 2019. Avaiable at http://docs.nvidia.com/cuda/ cusolver/.
- [33] PIALUCHA, T., CAWLEY, P. The detection of a weak adhesive/adherend interface in bonded joints by ultrasonic reflection measurements. In *Review of Progress in Quantitative Nondestructive Evaluation. Vol. 11B* (1992), D. O. Thompson and D. E. Chimenti, Eds., vol. 11, p. 1261–1266.
- [34] RAJABI, M., M. HASHEMINEJAD, S. Acoustic resonance scattering from a multilayred cylindrical shell with imperfect bonding. *Ultrasonics* 49 (12 2009), 682–695.
- [35] ROKHLIN, S. I., HUANG, W. Ultrasonic wave interaction with a thin anisotropic layer between two anisotropic solids: Exact and asymptotic-boundary-condition methods. *The Journal of the Acoustical Society of America* 92, 3 (1992), 1729–1742.
- [36] THOMAS, R., W DRINKWATER, B., LIAPTSIS, D. The reflection of ultrasound from partially contacting rough surfaces. *The Journal of the Acoustical Society of America* 117 (03 2005), 638–45.
- [37] TOMOV, S., DONGARRA, J., BABOULIN, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 36, 5-6 (2010), 232–240.
- [38] TOP500. Supercomputer Sites, (1993–2019). Avaiable at http://www.top500.org/.
- [39] W. DRINKWATER, B., DWYER-JOYCE, R., M. ROBINSON, A. The Use of Ultrasound to Investigate Rough Surface Contact Phenomena, vol. 18. 01 1999, p. 1455– 1462.

APPENDIX A – Impedance Tensors in Isotropic Media

Consider an isotropic and homogenous elastic solid medium, subjected to small deformations. Supposing that we have two plane waves, given by the superposition of P, SV and SH waves, propagating up and down the z direction and that we are solving the problem in the time frequency domain. Assuming x=0, since we are only interested in propagation on the z-axis, we have that the displacement and traction vectors can be writen as described in Chaper 2 as:

$$\mathbf{u} = \begin{bmatrix} u_x & u_y & u_z \end{bmatrix}^T, \tag{A.1}$$

and

$$\mathbf{t} = \begin{bmatrix} \tau_{zx} & \tau_{zy} & \sigma_{zz} \end{bmatrix}^T. \tag{A.2}$$

It is possible to decompose these vectors into up- and downgoing fields with the aid of the invariant imbedding technique (Ref. [8]):

$$\mathbf{u} = \mathbf{u}_1 + \mathbf{u}_2 \tag{A.3}$$

and

$$\mathbf{t} = \mathbf{t}_1 + \mathbf{t}_2. \tag{A.4}$$

In addition, after transforming these vector to the k_x domain, they can be rewritten in a decomposition of matrices:

$$\bar{\mathbf{u}}_{\alpha}(z) = \mathbf{A}_{\alpha} \Phi_{\alpha}(z) \mathbf{C}_{\alpha}, \qquad \alpha = 1, 2$$
 (A.5)

$$\bar{\mathbf{t}}_{\alpha}(z) = -i\mathbf{L}_{\alpha}\Phi_{\alpha}(z)\mathbf{C}_{\alpha}, \qquad \alpha = 1,2$$
(A.6)

 C_1 and C_2 are the amplitude vectors defined as:

$$\mathbf{C}_1 = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \tag{A.7}$$

and

$$\mathbf{C}_2 = \begin{bmatrix} D \\ E \\ F \end{bmatrix}$$
(A.8)

where A is the amplitude of the upgoing P-wave, B is the amplitude of the upgoing SVwave, C is the amplitude of the upgoing SH-wave, D is the amplitude of the downgoing P-wave, E is the amplitude of the downgoing SV-wave and F is the amplitude of the downgoing SH-wave.

Operators A_1 , A_2 are given by:

$$\mathbf{A}_{1} = \begin{bmatrix} \sin(\theta_{1}) & -\cos(\theta_{2}) & 0\\ 0 & 0 & 1\\ \cos(\theta_{1}) & \sin(\theta_{2}) & 0 \end{bmatrix}$$
(A.9)

and

$$\mathbf{A}_{2} = \begin{bmatrix} \sin(\theta_{4}) & \cos(\theta_{5}) & 0\\ 0 & 0 & 1\\ -\cos(\theta_{4}) & \sin(\theta_{5}) & 0 \end{bmatrix}$$
(A.10)

where θ_1 is the propagation angle of the upgoing P-wave, θ_2 is the propagation angle of the upgoing SV-wave, θ_4 is the propagation angle of the downgoing P-wave and θ_5 is the propagation angle of the downgoing SV-wave.

Operators \mathbf{L}_1 , \mathbf{L}_2 , $\Phi_1(z)$ and $\Phi_2(z)$ are defined as:

$$\mathbf{L}_{1} = \begin{bmatrix} -\mu(\sin(\theta_{1})k_{z_{1}} + \cos\theta_{1}k_{x}) & \mu(\cos(\theta_{2})k_{z_{2}} - \sin(\theta_{2})k_{x}) & 0\\ 0 & 0 & -\mu k_{z_{3}}\\ -\lambda(\cos(\theta_{1})k_{z_{1}} + \sin(\theta_{1})k_{x}) - 2\mu\cos(\theta_{1})k_{z_{1}} & \lambda(-\sin(\theta_{2})k_{z_{2}} + \cos(\theta_{2})k_{x}) - 2\mu\sin(\theta_{2})k_{z_{2}} & 0\\ (A.11) \end{bmatrix}$$

$$\mathbf{L}_{2} = \begin{bmatrix} \mu(\sin(\theta_{4})k_{z_{4}} + \cos\theta_{4}k_{x}) & \mu(\cos(\theta_{5})k_{z_{5}} - \sin(\theta_{5})k_{x}) & 0\\ 0 & 0 & \mu k_{z_{6}}\\ -\lambda(\cos(\theta_{4})k_{z_{4}} + \sin(\theta_{4})k_{x}) - 2\mu\cos(\theta_{4})k_{z_{4}} & \lambda(\sin(\theta_{5})k_{z_{5}} - \cos(\theta_{5})k_{x}) + 2\mu\sin(\theta_{5})k_{z_{5}} & 0\\ & (A.12) \end{bmatrix}$$

$$\Phi_{1}(z) = \begin{bmatrix} e^{ik_{z_{1}}z} & 0 & 0\\ 0 & e^{ik_{z_{2}}z} & 0\\ 0 & 0 & e^{ik_{z_{3}}z} \end{bmatrix},$$
(A.13)
$$\Phi_{2}(z) = \begin{bmatrix} e^{-ik_{z_{4}}z} & 0 & 0\\ 0 & e^{-ik_{z_{5}}z} & 0\\ 0 & 0 & e^{-ik_{z_{6}}z} \end{bmatrix},$$
(A.14)

$$\begin{bmatrix} 0 & 0 & e^{-s_0} \end{bmatrix}$$

Where k_x and k_{z_β} are the projections of the wave number vector **k** in the x and z direction.
Additionally, λ and μ are, respectively, Lamé first and second parameters and are both constant
for isotropic media.

The traction vectors can also be written as a function of the matrix operators $\mathbf{Z}_1, \mathbf{Z}_2$.

$$\bar{\mathbf{t}}_{\alpha}(z) = -i\omega \mathbf{Z}_{\alpha} \bar{\mathbf{u}}_{\alpha}(z), \qquad \alpha = 1, 2 \tag{A.15}$$

These operators \mathbf{Z}_1 and \mathbf{Z}_2 are the impedance tensors, which relate the tractions to the displacement fields and depend only on the material. Using Eq. A.15 in conjunction with Eqs. A.5 and A.6, we can compute these operators as:

$$\mathbf{Z}_{\alpha} = \frac{1}{\omega} \mathbf{L}_{\alpha} \cdot [\mathbf{A}_{\alpha}]^{-1}, \alpha = 1, 2$$
(A.16)

,