Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Geoscience and Petroleum

Ștefan Cătălin Crăciun

Integrated Boundary Conditions and HPC schemes for enhanced Viscoacoustic Modeling

Master's thesis in Petroleum Geophysics
Supervisor: Børge Arnsten
July 2023

NTNU
Norwegian University of Science and Technology

Ștefan Cătălin Crăciun

Integrated Boundary Conditions and HPC schemes for enhanced Viscoacoustic Modeling

Master’s thesis in Petroleum Geophysics
Supervisor: Børge Arnsten
July 2023

Norwegian University of Science and Technology
Faculty of Engineering
Department of Geoscience and Petroleum

Norwegian University of
Science and Technology

Abstract

This thesis presents a comprehensive investigation of the viscoacoustic wave equation, focusing on Absorbing Boundary Conditions and High Performance Computing.

We show that the C-PML boundary conditions are equivalent to a Standard Linear Solid viscoelastic kernel that incorporates a time-dependent density and bulk modulus. We implement a finite difference scheme that uses the same wave equation both inside and outside the absorbing zone, thus it is only the model parameters that change to effectively absorb the wavefield.

Two different viscoelastic models are implemented: the Standard Linear Solid and Maxwell mechanisms. The former was shown to better simulate realistic wave attenuation, while the latter provides frequency-independent attenuation, which might be more desirable for an absorbing boundary.

The thesis also investigates a series of optimization strategies aimed at improving the performance of seismic experiments, making even a personal laptop capable of running complex simulations. These different optimization schemes are implemented on a single-core CPU, multiple cores, or the GPU using Python, C, and Julia programming languages. The results have shown that Python, when equipped with the right modules and packages, can be a powerful tool for optimization, capable of rivaling C and CUDA in terms of performance. We also explored the capabilities of Julia, a high-level language designed for scientific computing and demonstrate its potential to outperform C.

Acknowledgements

First of all, I want to thank my supervisor, Prof. Børge Arnsten. I am deeply grateful for his guidance, patience, and insight. His expertise has been invaluable throughout this process, and I am fortunate to have been able to learn from him.

I also want to convey my sincere thanks to my girlfriend, Xiao. She has been there for me during all the tough times, always ready to lend a hand with difficult coding problems. When I found myself stuck in code debugging for endless hours, it was her support and motivation that helped me push through.

Lastly, I express my gratitude to my family for their support. They granted me the opportunity to attend NTNU, a decision that has opened numerous doors and led to many wonderful experiences.

In conclusion, this project has been a remarkable journey, one filled with challenges, learning, and growth. The experiences and knowledge I've gathered during this process are invaluable. To all those who have contributed to this journey in one way or another, thank you.

Table of Contents

List of Figures **vii**

List of Tables **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Boundary Conditions 2

1.3 High Performance Computing 3

1.4 Objectives 4

2 Viscoacoustic Wave Theory **5**

2.1 Wave Equation 5

2.2 The Finite-Difference Method 7

2.2.1 Derivatives 7

2.2.2 Higher Order Operators 8

2.2.3 Discretization and Staggered Grids 12

2.2.4 Analytical Solution 14

2.2.5 Numerical Stability, Dispersion and Anisotropy 14

2.3 Boundary Conditions 18

2.3.1 Viscoelastic Media 19

2.3.2 Time-Dependent Density 20

2.3.3 Standard Linear Solid Model 21

2.3.4 Maxwell Model 23

2.3.5 Viscoelastic Equations of Motion 25

2.3.6 Discretization and Staggered Grids 29

2.3.7 Absorbing Boundaries and Tapering 32

2.3.8 Comparison with C-PML 34

3 Numerical Implementation and Optimization **36**

3.1 Saxpy 36

3.1.1 Single Core CPU 36

3.1.2 Multiple Core CPU 44

3.1.3 GPU 47

3.1.4 Apple Silicon Chip 54

3.2 Viscoacoustic Finite Difference Modelling 56

4 Results 59
4.1 Viscoacoustic Modelling . . . 59
4.1.1 Comparison with the Analytical Solution . . . 59
4.1.2 Absorbing Boundaries . . . 61
4.1.3 Standard Linear Solid vs Maxwell . . . 65
4.2 Benchmarks . . . 72
4.2.1 Performance Metrics . . . 72
4.2.2 Test Systems . . . 72
4.2.3 Saxpi . . . 73
4.2.4 Viscoacoustic Modelling . . . 74
5 Conclusion and Discussion 78
Bibliography 80

List of Figures

1 Snapshot of horizontal ground movement modeled using a finite difference technique. The depicted scenario is a simulation of the M5.3 Roermond earthquake that occurred in the Cologne region of Germany in 1992, using a 3D representation of the sedimentary basin. The red and blue colors represent horizontal ground velocities that are positive and negative, respectively. The lower wave velocity found in the sedimentary basin enhances seismic motion relative to the adjacent bedrock, resulting in a significant extension of the trembling duration. 1

2 The Frontier Supercomputer, which is the world's fastest supercomputer as of July 2023, managing to achieve a performance of 1.102 exaFLOPS (10^{18} FLOPS). . . . 3

3 3D Cartesian coordinate system in which the domain Ω resides 5

4 Visual representation of Taylor operators weights for the central derivative ($f^c(x)$). The derivative is computed for the red circle. The OX axis defines the grid point location in relation to the derivative location (x[0]). It can be noted how the magnitude of the weights decreases as the distance from the derivation point increases. In this case the operator length is 8, meaning that the interpolation radius of the derivative is 8 grid points. In total, this derivation stencil includes 16 points. Also, it can be seen that the weights are anti-symmetric in relation to x[0]. 10

5 First derivative for a Ricker Wavelet with a central frequency of 25Hz. The time step used for the derivative is 5ms (dt = 5ms). The black lines represents the analytical solution; the blue dotted line is the forward difference ($O(dx)$); the orange dotted line is the backward difference ($O(dx)$); the green dotted line is the central difference of order 2 ($O(dx^2)$) and the red dotted line is the central difference of order 4 ($O(dx^4)$). The operators that have a higher order are closer to the analytical solution. 11

6 It can be seen how the high order finite-difference schemes converge more rapidly to the correct derivative on a regular grid, if one reduces the grid spacing dx. 11

7 Staggered Grid schematic; σ_{xx}, σ_{zz} and K are defined on the regular grid vertexes, while v_x, v_z, ρ_x and ρ_z are defined on the staggered grid, which is displaced by half a grid point. 13

8 Explosion of the finite-difference solution represented in trace form. The blue line denotes the finite-difference approximation of the wavefield. It can be seen how the solution explodes at around 0.18 seconds, as the numerical errors get accumulated over time. 15

9 Wavefield propagation in the subsurface showcasing the explosion of the finite-difference solution at different time steps (Nt = 25, 50, 75 and 100). It can be seen how the wavefield disintegrates as time progresses. 16

10 Dispersion in the finite-difference method. The blue line denotes the finite-difference approximation of the wavefield, while the red line shows the analytical solution. It can be seen how the wavelet becomes dispersive and disintegrates. 17

11 Numerical Anisotropy in the finite-difference method. It can be seen that the dispersion is the strongest at 0° and 90° with respect to the grid and the smallest at 45°. 17

12 Wavefield propagation in the subsurface showcasing the Dirichlet boundary conditions. It can be seen how the wavefield is perfectly reflected back into the medium as it reaches the rigid boundaries. The polarity is also reversed. 18

13 Stress–strain curves for an elastic material (left) and a viscoelastic material (right). The red shaded area represents the hysteresis loop and shows the amount of energy lost (in the form of heat dissipation) in the loading and unloading cycle. 19

14 Standard Linear Solid Model. The mechanical model consists of two elastic springs (with the elastic modulus k) and a dashpot (where η is the viscosity of the material). σ is the applied stress and ϵ is the deformation. 21

15 Maxwell Model, consisting of one elastic spring (with the elastic modulus k) and a dashpot (where η is the viscosity of the material) connected in series. σ is the applied stress and ϵ is the deformation. 24

16 Staggered Grid for the Viscoacoustic wave equation schematic; σ_{xx}, σ_{xx}, γ, K_u and the α parameters are defined on the regular grid vertexes, while v_x, v_z, ρ_{ux}, ρ_{uz}, θ and the η parameters are defined on the staggered grid, which is displaced by half a grid point. 30

17 The subsurface model enclosed by an absorptive medium. The Acoustic Medium is obtained for a high quality factor ($Qmax = 10^4$). The Absorptive medium has a decreasing quality factor given by a tapering function, starting from the inner edge (the red dotted line) until it reaches the Qmin value ($Qmin = 1.1$), the black outer line. 33

18 Example of the Q-model tapering, where $Qmax = 10^4$, $Qmin = 1.1$ and $Nb = 15$. . 33

19 Basic elements of a Single Core CPU; The core contains the ALU (Arithmetic Logic Unit), which performs arithmetic and logic operations; The Control Unit tells the ALU what operation to perform and in which order to perform them; The L1, L2 and L3 caches are very fast and expensive memory that store temporary data that's actively being used by the CPU; The DRAM (Dynamic Random Access Memory) is the main memory used by the computers. While it can be very large, it is much slower than the Cache memory. The DRAM is not part of the CPU itself but is an external component. The CPU communicates with the DRAM through a memory bus. 37

20 The Saxpy computation implemented using Cython; the yellow lines show all interactions with Python. 40

21 Basic elements of a Multiple Core CPU; Each core has its own memory cache, but all the cores also have access to shared memory. Tasks are sent to the CPU, and then they are distributed across the cores for simultaneous processing. 44

22 The Cython implementation using Multithreading; a lighter yellow line indicates fewer interactions with Python. 45

23 Basic elements of an Nvidia GPU; Each green square could represent a CUDA core. These cores have a lower clock rate than their CPU counterpart, but they can number in the thousands. The GPU has its own private memory, usually called GDDR RAM (Graphics Double Data Rate Random Access Memory). This is a type of synchronous memory that is specifically designed for GPUs. 47

24 Example of heterogeneous computing when executing a script. The computationally heavy areas are performed by the GPU (red), while the less demanding serial code is performed by the CPU (blue). 48

25 Example of Thread hierarchy, showcasing a 2D grid containing 2D blocks. 49

26 Example of Thread hierarchy, where the threads are organized in 2D blocks. 49

27 The Unified memory architecture found on all Apple M-series devices. The CPU and GPU are integrated in the same chip. 54

28 The execution flow of the Main file. The computationally heavy areas are denoted with Red, while the less demanding code blocks are blue. 56

29 The execution flow of the AC2D file. The computationally heavy areas are denoted with Red, while the less demanding code blocks is blue. 57

30 Comparison of the Acoustic analytical solution (red) with the simulated Viscoacoustic solution (blue). It can be seen how these two overlap, thus proving that the Viscoacoustic solution is reduced to the Acoustic case when a large Quality factor is chosen. 60

31 Comparison of the Acoustic analytical solution (red) with the simulated Viscoacoustic solution (blue). It can be seen how the numerical solution is attenuated when choosing a small Quality factor ($Q = 1.5$). 60

32 Snapshots from the Viscoacoustic simulation using the SLS model. Parameters: 201x201 square mesh, $dx = 10m$, $dt = 0.5ms$, $Nt = 2000$ (giving a total simulation time of $1s$). The differentiator length is 8. The model is homogeneous, with a P wave velocity of $2000m/s$, a density of $2000kg/m^3$ and a quality factor of 10^5. The Q model is tapered at the borders, reaching a Qmin value of 1.1. The source pulse is a Ricker Wavelet with a dominant frequency of $25Hz$, which is placed in the middle of the model ($x = 1000m$, $z = 1000m$). $Nb = 20$ (Number of grid points for the absorbing boundary). 61

33 Snapshots from the Viscoacoustic simulation using the Maxwell model. Parameters: 201x201 square mesh, $dx = 10m$, $dt = 0.5ms$, $Nt = 4000$ (giving a total simulation time of $2s$). The differentiator length is 8. The three layered model has an increasing velocity/density with depth. The quality factor is 10^5. The Q model is tapered at the borders, reaching a Qmin value of 1.1. The source pulse is a Ricker Wavelet with a dominant frequency of $25Hz$, which is placed in the middle of the first layer ($x = 1000m$, $z = 350m$). $Nb = 30$ (Number of grid points for the absorbing boundary). 62

34 Marmousi velocity model (top) and density model (bottom). The mesh is 500x174 with $dx = 20m$. 63

35 Snapshots from the Viscoacoustic simulation using the Marmousi model. Parameters: 500x174 mesh, $dx = 20m$, $dt = 0.5ms$, $Nt = 8000$ (giving a total simulation time of $4s$). The differentiator length is 8. The quality factor is 10^5. The Q model is tapered at the borders, reaching a Qmin value of 1.1. The source pulse is a Ricker Wavelet with a dominant frequency of $10Hz$. $Nb = 30$. 64

36 The Seismogram obtained from the simulation by placing a line of hydrophones just below the water-air interface. Here, the absorbing boundary area is cut from the seismogram 65

37 Edge Artifacts from the SLS mechanism. The source is a Ricker wavelet with a dominant frequency of $15Hz$. The absorbing boundary width is 30 grid points. . . 66

38 The reflection artifacts for the Standard Linear Solid mechanism (top) and Maxwell mechanism (bottom). For each case, we show the results for a dominant frequency of $15Hz$ (left) and $25Hz$ (right). 67

39 Trace comparison between the Maxwell and SLS mechanisms for a wavelet with a dominant frequency of 15Hz. 68

40 Trace comparison between the Maxwell and SLS mechanisms for a wavelet with a dominant frequency of 25Hz 68

41 Frequency versus attenuation response for the Maxwell and SLS mechanisms. The dominant frequency of the propagating wavefield is $15Hz$ ($f_0 = 15Hz$). The Quality factor is 1.1, $c_0 = 2000m/s$ and $z = 150m$. 71

42 Julia serial CPU (black), Multithread (blue), and GPU (red) Saxpy implementations with varying problem sizes. The OY ans OX axis are logarithmic here. The runtime is given in milliseconds. 73

43 Serial CPU implementations for one shot using the viscoacoustic finite-difference method with varying problem size: Python For Loop (orange), NumPy (green), Numba (black), Cython (purple), C (blue) and Julia (red). The runtime is given in seconds. 74

44 Multithread implementations for one shot using the viscoacoustic finite difference method with varying problem size: Numba (black), Cython (purple), C (blue) and Julia (red). The number of threads is set at 16. The runtime is given in seconds. . 75

45 GPU implementations for one shot using the viscoacoustic finite difference method with varying problem size: Numba CuPy (black), CUDA C (blue), Julia CUDA (red) and Julia Metal M1 (purple). The blocksize is 16x16. The runtime is given in seconds. 76

46 CPU, Multithread and GPU implementations using Numba: Python Numba serial CPU (black), Multithread Numba (blue) and Numba CuPy GPU (green). 76

47 The relative speedup of the different implementations when compared to the Numba serial CPU version. 77

List of Tables

1 Finite-Difference Coefficients for a Staggered Grid as defined by Holberg 1987. l is the length of the differentiator (for a given l value, the total number of neighboring points used in the numerical derivation is $2 * l$). 12

2 Hardware Characteristics 72

List of Source Codes

1 The variable declaration for Saxpy implemented using Python: a is a $float32$ scalar and **x** and **y** are NumPy arrays filled with $float32$ random values (between 0 and 10). 38
2 The Saxpy computation implemented using a simple for loop. 38
3 The Saxpy computation implemented using NumPy vectorization. 38
4 The Saxpy computation implemented using Numba. 39
5 The variable declaration for Saxpy implemented using C: a is a $float32$ scalar and **x** and **y** are dynamically allocated arrays filled with $float32$ random values (between 0 and 10). 41
6 The Saxpy computation implemented in C. 41
7 The interface file for SWIG 42
8 The variable declaration for Saxpy implemented using Julia: a is a $float32$ scalar and **x** and **y** are arrays filled with $float32$ random values (between 0 and 10). . . . 43
9 The Saxpy computation implemented using a for loop in Julia 43
10 The vectorized Saxpy computation in Julia 44
11 The Multithreaded Saxpy computation implemented using Numba. 45
12 The C implementation using Multithreading 46
13 The Julia Multithreaded implementation of Saxpy 46
14 The Python GPU implementation of Saxpy 50
15 The CUDA GPU implementation of Saxpy 51
16 The Julia vectorized GPU implementation of Saxpy 52
17 The Julia kernel GPU implementation of Saxpy 53
18 The Julia Metal vectorized GPU implementation of Saxpy 55
19 The Julia Metal kernel GPU implementation of Saxpy 55
20 The Ac2dvx GPU kernel implemented in Julia; Here we use a 2D grid and block. . 58

1 Introduction

1.1 Motivation

Seismic modeling plays an important role when it comes to understanding and exploring the subsurface structures and features of the Earth. Several disciplines, which include the exploration of natural resources, seismic hazard assessment, and the understanding of the Earth's internal structure, benefit from using this method.

Seismic modeling can identify prospective areas for the exploration of natural resources such as oil, natural gas, and minerals by providing high-resolution images of subsurface geological formations. This serves to optimize the extraction process, reduce risks, and enhance the resource management overall. Moreover, it is essential for assessing seismic risks, predicting potential earthquakes, and comprehending their impact (Figure 1). It enables scientists to simulate seismic wave propagation, aiding them in estimating the magnitude and potential damage caused by earthquakes in various regions. Furthermore, it provides geoscientists with a valuable instrument for studying the Earth's crust and deeper structures, thereby shedding light on the planet's geological history.

Several numerical modeling techniques, such as: finite difference, finite element, and spectral methods, can be used for seismic wave propagation (Igel 2017). In order to derive the wavefield solution, the elastic properties for a grid-based geological model are required. The media in question can be acoustic, elastic, or viscoelastic (in each case the materials being isotropic or anisotropic).

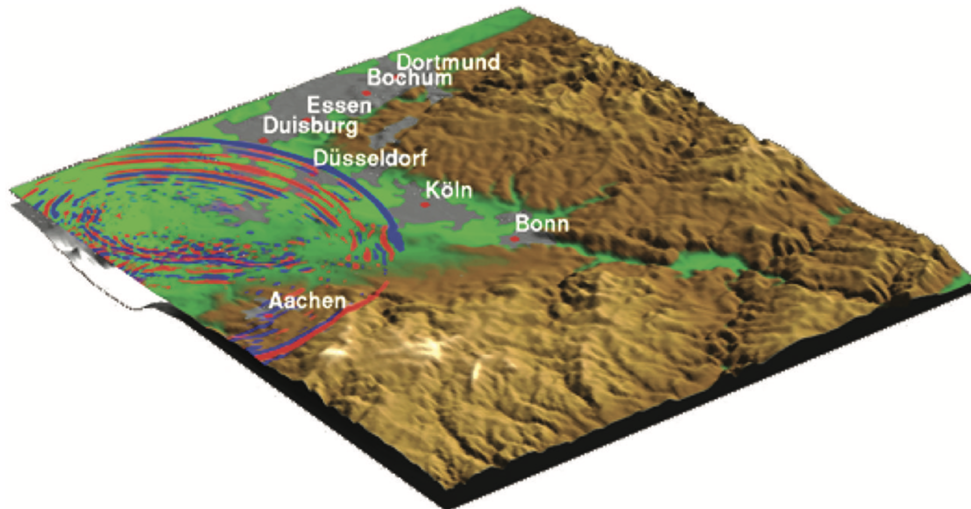


Figure 1: Snapshot of horizontal ground movement modeled using a finite difference technique. The depicted scenario is a simulation of the M5.3 Roermond earthquake that occurred in the Cologne region of Germany in 1992, using a 3D representation of the sedimentary basin. The red and blue colors represent horizontal ground velocities that are positive and negative, respectively. The lower wave velocity found in the sedimentary basin enhances seismic motion relative to the adjacent bedrock, resulting in a significant extension of the trembling duration.

Source: Igel and Stupazzini 2015

As computational power increased exponentially with time, it allowed for bigger and finer models to be tackled. Combining this with advances in the computational algorithms, this has allowed for numerous cutting-edge subsurface imaging techniques, such as diffraction and wavefront tomography (Devaney and Oristaglio 1984; Wu and Toksöz 1987; Pratt and Worthington 1988; Gan et al. 1995; Gelius 1995; Xie et al. 2018), reverse time migration (Baysal et al. 1983; McMechan 1983; Dai et al. 2011; Xu et al. 2011; Y. Zhang et al. 2013), and full-waveform inversion (Tarantola 1984; Pratt and Worthington 1990; Sambridge et al. 1991; Virieux and Operto 2009; Vigh et al.

2014; Li and Demanet 2016). All of these methods rely heavily on numerical seismic wave modeling in order to generate high-resolution images of the subsurface.

There have been great incentives to find ways to improve the methods described above. Finite difference has been one of the core methods utilized for the simulation of seismic wave propagation, due to its relative simplicity, ease of implementation on computers, but also for its sturdiness. The work presented in this thesis will focus on two main areas, namely: Absorbing Boundary Conditions and High Performance Computing.

1.2 Boundary Conditions

The seismic wave equations are mathematically described by partial differential equations, which pertain to the continuous domain. But, due to the limited capacity of computer memory, it is essential to use a 2-D or 3-D grid of finite size in numerical modeling. An infinite computational domain cannot be modeled, thus when using computers to simulate seismic waves, a transition from the continuous to the discrete domain must be made. The edges of the chosen subsurface model act as boundaries, and if the boundary conditions are not tackled effectively they produce unwanted reflections, called artifacts, or even instability. The boundary reflections are undesirable because they do not correspond to any actual boundaries in the simulated real-world scenarios, thus distorting the numerically simulated data. As a result, it is essential to reduce or eradicate as best as possible these artificial edge effects, as they superimpose over the true wavefield solutions and distort it.

Several numerical strategies have been developed over the years to reduce or eliminate the artificial effects that arise at the model's boundaries. Clayton and Engquist 1977 and Reynolds 1978 introduced a non-reflecting boundary condition technique where the wave equation at the model edges is replaced with the one-way wave equations, which has been shown to be highly effective at absorbing any artificial reflections caused by wave propagation perpendicular (or close to) to the model boundaries. This method only allows wave transmission at the boundaries, thus effectively suppressing reflections. Higdon 1991 carried this concept one step further by incorporating the 2D time-domain acoustic and elastic wave modeling for these absorbing boundary conditions. However, the main problem with this method is that it is unable to effectively suppress the artificial reflections caused by waves traveling at small or oblique angles to the model's borders. In order to tackle this challenge, Cerjan et al. 1985 proposed a solution in the form of a gradational attenuation scheme in the time-domain, accomplished by incorporating absorbing zones around the model's edges. Their method demonstrated that this gradational attenuation scheme is simpler than the aforementioned absorbing boundary conditions, and it was effective for all time-domain wave modeling problems. Then, Sochacki et al. 1987 and Serón et al. 1996 introduced additional damping terms into the wave equation, which facilitated better attenuation within the absorbing zone.

The introduction of the perfectly matched layer (PML) method in electromagnetic wave modeling by Berenger 1994 marked a significant advance in eliminating artificial reflections. Since then, numerous researchers have effectively adapted this method for acoustic and elastic wave modeling (Francis and Tsogka 2001; Komatitsch and Tromp 2003; Komatitsch and Martin 2007; W. Zhang and Shen 2010). Drossaert and Giannopoulos 2007 introduced an improved version of the PML, called convolutional perfect matched layer (C-PML). This method offers improvements by increasing absorption and reducing dispersion in the absorbing layer at the cost of increased computational complexity. The universal applicability of the PML method to any first-order wave equation in arbitrary media (W. Zhang and Shen 2010) has made it the method of choice for seismic wave modeling problems. One of the main disadvantages of the PML method is that for all first-order spatial derivatives in the time domain, additional recursive calculations are required. In addition, its application to second-order wave equations is not as straightforward (Komatitsch and Tromp 2003). Moreover, the effectiveness of the PML method is highly dependent on selecting the proper parameters for the absorbing zones in relation to the area or volume of interest (W. Zhang and Shen 2010).

The main idea behind PML is represented by a complex coordinate transformation which leads

to replacing derivatives in the wave equation (Komatitsch and Martin 2007; Carcione and Kosloff 2013). Since the theory behind this method was originally derived from Maxwell's equations, the mathematical equations behind it are complex, abstract and it is hard to find correlations with seismic waves and the physical properties of rocks. Carcione and Kosloff 2013 managed to re-interpret the C-PML absorbing boundaries in terms of mechanical models and have shown their relationships in the time and frequency domain. Thus, the C-PML boundary conditions can be described by a wave equation with time-dependent bulk modulus and density.

1.3 High Performance Computing

Moore's Law predicts that the number of transistors in an integrated circuit doubles about every two years. What could have been achieved decades ago only by complex and expensive supercomputers nowadays can be achieved by regular smartphone devices. The limits of performance and computational power are pushed further up every year (Figure 2), enabling scientists to develop new technologies for energy, medicine, materials, and so forth (Oak Ridge National Laboratory 2023). Today, seismic simulations require complex computations over millions or even billions of grid points and entail large datasets. Using supercomputers and HPC schemes, these extensive simulations can be completed in a reasonable period of time.

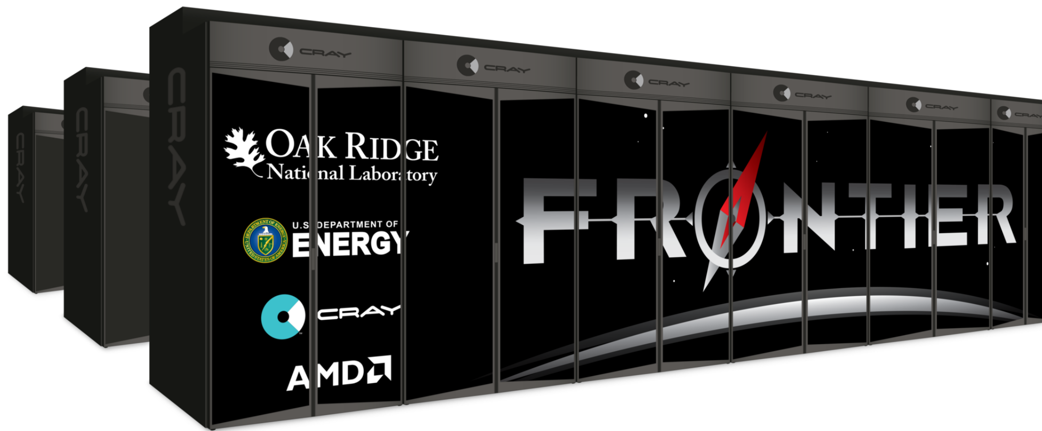


Figure 2: The Frontier Supercomputer, which is the world's fastest supercomputer as of July 2023, managing to achieve a performance of 1.102 exaFLOPS (10^{18} FLOPS).

Source: Oak Ridge National Laboratory 2023

One of the main challenges for students, Ph.D. candidates, researchers, or even professors that work with seismic modeling is that not everyone has access to such supercomputers, or the access is limited, as it is a shared resource that many people desire to use. Even more, it is not that straightforward to write code and work with these supercomputers, if your coding experience is limited. Sometimes, processing data locally is required. When debugging code, running tests, implementing new methodologies, or when you are in the field you do not have access to a supercomputer or workstation. But, with the recent advances, you can turn your own personal laptop into a computing machine that can process complicated models in a decent amount of time, if done properly.

The programming languages that are most commonly used in this area of scientific computing are C, C++, and Fortran. Most students nowadays that pursue geophysics do not have a coding background or are not familiar with these languages. That is not to say that programming is not an essential skill that every geophysicist should have, but even at a university level, it is preferred to use high level, easy to use programming languages such as Python. Python has made programming more accessible, reducing the learning curve traditionally associated with scientific computing, due to its simple syntax and extensive libraries. Maybe the most important aspect is that it has a vast

amount of free and open-source libraries. Packages such as NumPy, SciPy, Matplotlib, Pandas, or PyTorch are easy to use and allow students to process, view and interpret their data with relative ease. But, the major problem with Python is that it is very slow from a computational point of view.

1.4 Objectives

The first main objective of this thesis is to try to make the theory behind C-PML easy to understand, ground it in terms that are more related to rock properties and get more insight into the boundary condition problem in order to see what aspects can be improved in the future. We will test a modified version of the C-PML method with a time dependent density and bulk modulus that uses the same wave equations both inside and outside the absorbing boundaries, thus it is only the model/material parameters that change.

The second main objective of the thesis is to test whether nowadays a personal laptop is enough to perform seismic modeling and if it is still necessary to use languages such as C or C++ when implementing simulations or if there are ways to easily speed up your performance even when using Python or other similar languages.

In order to achieve these objectives, we will use the 2D acoustic/viscoacoustic wave equation implemented using a finite difference scheme. Chapter 2 will present the basic theory behind Finite Difference and the Boundary Conditions that are going to be implemented. Chapter 3 will showcase the methodology for computing in an efficient manner by using a simple problem (Saxpy - Single-Precision A·X Plus Y) and test the performance using different programming languages and implementing the code both on the CPU (Single Core and Multithread) and the GPU. In Chapter 4, we show how the new boundary conditions perform, while also showcasing the time performance of the different HPC implementations. The final chapter draws the conclusions in the work done in this thesis and proposes what further improvements can be made for future work.

2 Viscoacoustic Wave Theory

In this chapter, the theory required to numerically solve the 2D acoustic wave equation using the finite difference method is presented. Also, we show how the viscoacoustic wave equation with the new absorbing boundary conditions are derived and implemented. In the end, the equivalence between the C-PML method and viscoelasticity is made.

2.1 Wave Equation

The seismic wave equation is typically depicted by a hyperbolic partial differential equation. This equation is derived from two fundamental principles: Newton's second law, which provides the equation of motion, and Hooke's law, which provides the constitutive relation.

Let us consider an elastic medium Ω represented using a 3D Cartesian coordinate system. The domain is characterized by the density $\rho(\boldsymbol{x})$ and Lamé parameters $\lambda(\boldsymbol{x})$ and $\mu(\boldsymbol{x})$ at each spatial point $\boldsymbol{x}$, where $\boldsymbol{x} = (x, y, z)$ is any point in the domain Ω shown in Figure 3.

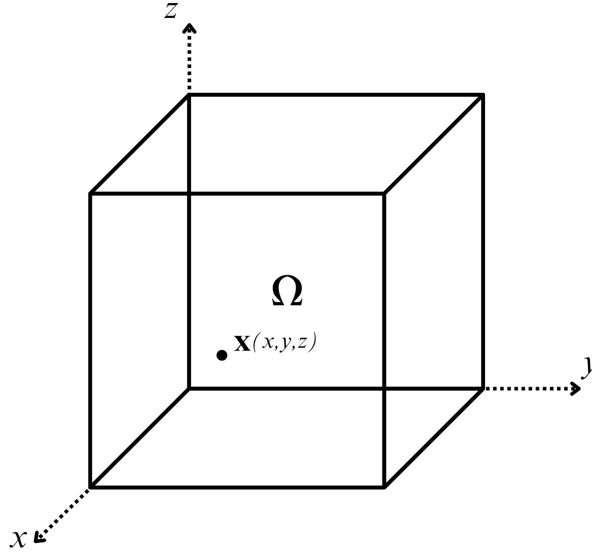


Figure 3: 3D Cartesian coordinate system in which the domain Ω resides

According to Ikelle and Amundsen 2005, the equation of motion for an elastic medium is given by:

$$\rho(\boldsymbol{x})\partial_t^2 u_i(\boldsymbol{x}, t) = \partial_j \sigma_{ij}(\boldsymbol{x}, t) + f_i(\boldsymbol{x}, t) \tag{1}$$

where $u_i(\boldsymbol{x})$ is the particle displacement in the i direction, t is the time, σ_{ij} is the stress tensor and f_i is a driving force (representing any external force applied to the material) in the i direction, where $i, j = (x, y, z)$.

Hooke's law provides the constitutive relationship between the stress tensor and particle displacement:

$$\sigma_{ij}(\boldsymbol{x}, t) = c_{ijkl} e_{kl} + q_{ij} \tag{2}$$

where c_{ijkl} is the elastic tensor (Hooke's tensor), e_{kl} is the strain tensor and q_{ij} is the driving stress (any external stress applied to the material).

For an isotropic material, the elastic tensor takes the form (Hudson 1981):

$$c_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu\left(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right) \tag{3}$$

where δ is the Kronecker delta function, which is 1 if its indices are equal and 0 if they are not (Ikelle and Amundsen 2005):

$$\delta_{ij} = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases} \quad i, j = x, y, z \tag{4}$$

Thus, we can rewrite (1) and (2) as:

$$\rho(\boldsymbol{x})\partial_t^2 u_i(\boldsymbol{x}, t) = \partial_j \sigma_{ij}(\boldsymbol{x}, t) + f_i(\boldsymbol{x}, t), \tag{5}$$

$$\sigma_{ij}(\boldsymbol{x}, t) = \lambda(\boldsymbol{x}) e_{kk}\delta_{ij} + 2\mu e_{ij} + q_{ij}. \tag{6}$$

where the cubic dilatation is given by $e_{kk} = e_{xx} + e_{yy} + e_{zz}$.

By considering the expression for strain as a function of particle displacement (Ikelle and Amundsen 2005), the strain tensor e_{ij} can be rewritten as:

$$e_{ij} = \frac{1}{2}\left[\partial_i u_j(\boldsymbol{x}, t) + \partial_j u_i(\boldsymbol{x}, t)\right] \tag{7}$$

By expanding the individual terms of the equations (5) and (6), we obtain the 3D isotropic elastic wave equations:

$$\begin{aligned}
\rho\partial_t^2 u_x &= \partial_x\sigma_{xx} + \partial_y\sigma_{xy} + \partial_z\sigma_{xz} + f_x, \\
\rho\partial_t^2 u_y &= \partial_x\sigma_{yx} + \partial_y\sigma_{yy} + \partial_z\sigma_{yz} + f_y, \\
\rho\partial_t^2 u_z &= \partial_x\sigma_{zx} + \partial_y\sigma_{zy} + \partial_z\sigma_{zz} + f_z, \\
\sigma_{xx} &= \lambda\left(e_{xx} + e_{yy} + e_{zz}\right) + 2\mu e_{xx} + q_{xx}, \\
\sigma_{yy} &= \lambda\left(e_{xx} + e_{yy} + e_{zz}\right) + 2\mu e_{yy} + q_{yy}, \\
\sigma_{zz} &= \lambda\left(e_{xx} + e_{yy} + e_{zz}\right) + 2\mu e_{zz} + q_{zz}, \\
\sigma_{xy} &= 2\mu e_{xy} + q_{xy}, \\
\sigma_{xz} &= 2\mu e_{xz} + q_{xz}, \\
\sigma_{yz} &= 2\mu e_{yz} + q_{yz}.
\end{aligned} \tag{8}$$

The 2D set of equations can be derived by assuming non-zero particle displacements only in the xz plane, where x denotes the horizontal distance and z is the depth. Also, in order to get the velocity-stress formulation, we consider that $\partial_t u_i = v_i$. Thus, the 2D velocity stress formulation for the elastic wave equations can be described by the following system of partial differential equations:

$$\begin{aligned}
\rho\partial_t v_x &= \partial_x\sigma_{xx} + \partial_z\sigma_{xz} + f_x, \\
\rho\partial_t v_z &= \partial_x\sigma_{zx} + \partial_z\sigma_{zz} + f_z, \\
\partial_t\sigma_{xx} &= \left(\lambda + 2\mu\right)\partial_x v_x + \lambda\partial_z v_z + \dot{q_{xx}}, \\
\partial_t\sigma_{zz} &= \lambda\partial_x v_x + \left(\lambda + 2\mu\right)\partial_z v_z + \dot{q_{zz}}, \\
\partial_t\sigma_{xz} &= \mu\left(\partial_z v_x + \partial_x v_z\right) + \dot{q_{xz}}.
\end{aligned} \tag{9}$$

Finally, to obtain the 2D acoustic wave equation, we set the second Lamé parameter (the shear modulus), represented by μ, to zero. The first Lamé parameter can be written in terms of Bulk and Shear modulus as 10, so the first Lamé parameter is equivalent to the bulk modulus in the acoustic case.

$$\lambda = K - \frac{2\mu}{3} \tag{10}$$

Thus, the velocity strain formulation for the 2D isotropic acoustic wave equation is given by:

$$\begin{aligned}
\rho \partial_t v_x &= \partial_x \sigma_{xx} + f_x, \\
\rho \partial_t v_z &= \partial_z \sigma_{zz} + f_z, \\
\partial_t \sigma_{xx} &= K \left(\partial_x v_x + \partial_z v_z\right) + \dot{q_{xx}}, \\
\partial_t \sigma_{zz} &= K \left(\partial_x v_x + \partial_z v_z\right) + \dot{q_{zz}}.
\end{aligned} \tag{11}$$

Both the density and the bulk modulus are spatially dependent ($\rho = \rho(\boldsymbol{x})$, $K = K(\boldsymbol{x})$), but time independent.

2.2 The Finite-Difference Method

The previously examined equations describe the behavior of seismic waves in a continuous domain. Unfortunately, there are no analytical solutions available in order to quickly solve these equations in complex heterogeneous media and predict how the wavefield will look like in a certain medium at a certain time. As a result, we utilize computers that simulate the wavefield using numerical approximations of these equations.

The Finite-Difference method is one of the most popular and successful numerical techniques used. Its mathematical simplicity and adaptability make it so that an algorithm can be quickly fitted to the problem at hand (Igel 2017). The basic principle of the method is that the derivatives in the differential wave equations are replaced by numerical approximations. This method can be implemented either using an explicit or implicit scheme. This thesis will focus on the explicit method, which consists of predicting the future wavefield at a certain time and spatial location in terms of its known value at the current and previous time step and by knowing the present neighboring location values. Thus, the wavefield is solved recursively time step by time step (Ikelle and Amundsen 2005).

The first applications of the finite-difference method applied to elastic wave propagation started in the 1970s (Alterman and Karal 1968, Boore 1970, Alford et al. 1974 and Kelly et al. 1976). One of the most popular implementations of the finite difference explicit scheme for the wave equation is the staggered grid method, first used by Madariaga 1976 and Virieux and Madariaga 1982. Later, Virieux adapted the method for SH and P-SV 2D wave propagation (Virieux 1984 and Virieux 1986). In order to improve the accuracy of the numerical approximations, high-order operators were introduced (Levander 1988 and Graves 1996).

2.2.1 Derivatives

In the simplest form, the derivative of a function $f(x)$ is the difference between two values of the function, which forms the slope as the limit approaches zero. The three most common ways to express this concept are the forward derivative (12), backward derivative (13) and centered derivative (14). In these equations, we consider $f(x)$ to be a continuous function.

$$\partial_x f(x) = \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx} \tag{12}$$

$$\partial_x f(x) = \lim_{dx \to 0} \frac{f(x) - f(x-dx)}{dx} \tag{13}$$

$$\partial_x f(x) = \lim_{dx \to 0} \frac{f(x+dx) - f(x-dx)}{2dx} \tag{14}$$

In the finite difference method, the subsurface model parameters are defined on a grid. Thus, for the equations written above, instead of using the limit, we use a finite distance between the grid points (dx), that gives a numerical approximation. By using the Taylor expansion, we can obtain both the finite approximations and the accuracy order for the forward, backward and central derivatives, respectively.

The fundamental principle underlying the Taylor series is that it approximates a function close to a given point. The Taylor series of the function $f(x+dx)$ is defined as follows:

$$f(x+dx) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!} dx^n \tag{15}$$

This is an infinite sum, which can be truncated after a few terms. If we unroll equation 15, we obtain:

$$f(x+dx) = f(x) + f'(x)dx + \frac{1}{2!} f''(x)dx^2 + \frac{1}{3!} f'''(x)dx^3 + ... \tag{16}$$

Since in this example case, we are interested in the first derivative only, all the higher order derivatives, which denote the higher order terms are truncated. In the same manner, the Taylor expansion of $f(x-dx)$, $f(x+2dx)$, $f(x-2dx)$, and so on, can be done. If we rearrange the terms in equation 16, we obtain the forward derivative formulation on a grid (17). Using a similar approach the backward and central derivatives can be obtained (18, 19):

$$\partial_x f^+ = \frac{f(x+dx) - f(x)}{dx} + O(dx) \tag{17}$$

$$\partial_x f^- = \frac{f(x) - f(x-dx)}{dx} + O(dx) \tag{18}$$

$$\partial_x f^c = \frac{f(x+dx) - f(x-dx)}{2dx} + O(dx^2) \tag{19}$$

where $O(dx^m)$ denotes the accuracy of the numerical approximation of the derivative. For the forward and backward derivative, m is 1, while for the central derivative m is 2. So, the central difference formula has an extra order of accuracy, even though it has the same computational cost.

2.2.2 Higher Order Operators

For the forward derivative (17), the weights for $f(x+dx)$ and $f(x)$ are 1 and -1 respectively. A derivative can be expressed as a weighted sum of function values at different neighboring points. By including more neighboring points, the accuracy of the derivative increases. A systematic way to determine the weights that have to be multiplied with the function values at different locations in order to obtain the finite derivative approximations can be done by solving a system of equations.

As an example, the central derivative formula will be considered. The goal is to determine the first derivative of the function $f(x)$, based on the function values at four neighboring points $f(x+dx)$ $f(x+2dx)$, $f(x-dx)$ and $f(x-2dx)$:

$$\frac{\partial f(x)}{\partial x} \approx a\ f(x+dx) + b\ f(x-dx) + c\ f(x+2dx) + d\ f(x-2dx) \tag{20}$$

These terms can be expressed as a system of four equations as follows:

$$
\begin{aligned}
a\ f(x+dx) &= a\ \left[\ f(x)+f^{'}(x)dx+\frac{1}{2!}f^{''}(x)dx^2+\frac{1}{3!}f^{'''}(x)dx^3+\dots\ \right] \\
b\ f(x-dx) &= b\ \left[\ f(x)-f^{'}(x)dx+\frac{1}{2!}f^{''}(x)dx^2-\frac{1}{3!}f^{'''}(x)dx^3+\dots\ \right] \\
c\ f(x+2dx) &= c\ \left[\ f(x)+2f^{'}(x)dx+2^2\frac{1}{2!}f^{''}(x)dx^2+2^3\frac{1}{3!}f^{'''}(x)dx^3+\dots\ \right] \\
d\ f(x-2dx) &= d\ \left[\ f(x)-2f^{'}(x)dx+2^2\frac{1}{2!}f^{''}(x)dx^2-2^3\frac{1}{3!}f^{'''}(x)dx^3+\dots\ \right]
\end{aligned}
\tag{21}
$$

where a, b, c, and d denote the weights. In order to find the weights that allow us to calculate the first derivative of the function, the equations are summed up and the higher order terms are eliminated:

$$
\begin{aligned}
af(x+dx)+bf(x-dx)+cf(x+2dx)+df(x-2dx) \approx \\
f(x)[a+b+c+d]+ \\
+dxf'(x)[a-b+2c-2d]+ \\
+\frac{1}{2!}dx^2f''(x)[a+b+4c+4d]+ \\
+\frac{1}{3!}dx^3f'''(x)[a-b+8c-8d]
\end{aligned}
\tag{22}
$$

A system of linear equations can be formed:

$$
\begin{aligned}
a+b+c+d &= 0 \\
a-b+2c-2d &= \frac{1}{dx} \\
a+b+4c+4d &= 0 \\
a-b+8c-8d &= 0
\end{aligned}
\tag{23}
$$

This system can be expressed in matrix form as:

$$
\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 2 & -2 \\ 1 & 1 & 4 & 4 \\ 1 & -1 & 8 & -8 \end{pmatrix}
\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} =
\begin{pmatrix} 0 \\ \frac{1}{dx} \\ 0 \\ 0 \end{pmatrix}
\tag{24}
$$

Solving this system using matrix inversion will provide the weights a, b, c and d, corresponding to $f(x+dx)$, $f(x-dx)$, $f(x+2dx)$ and $f(x-2dx)$, respectively.

$$
\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} =
\begin{pmatrix} \frac{8}{12\mathrm{d}x} \\ -\frac{8}{12\mathrm{d}x} \\ -\frac{1}{12\mathrm{d}x} \\ \frac{1}{12\mathrm{d}x} \end{pmatrix}
\tag{25}
$$

Inserting the weights back into equation 20 gives use the 4$^{\text{th}}$ order finite difference approximation of the derivative:

$$
f^{'}(x) = \frac{f(x-2dx)-8f(x-dx)+8f(x+dx)-f(x+2dx)}{12\mathrm{d}x} + O(dx^4)
\tag{26}
$$

It can be noticed that the modulus of the weights closer to the function location where we want to compute the derivative are bigger than the ones further away from it. Using the same algorithm, the weights for any random operator length can be computed (Figure 4).

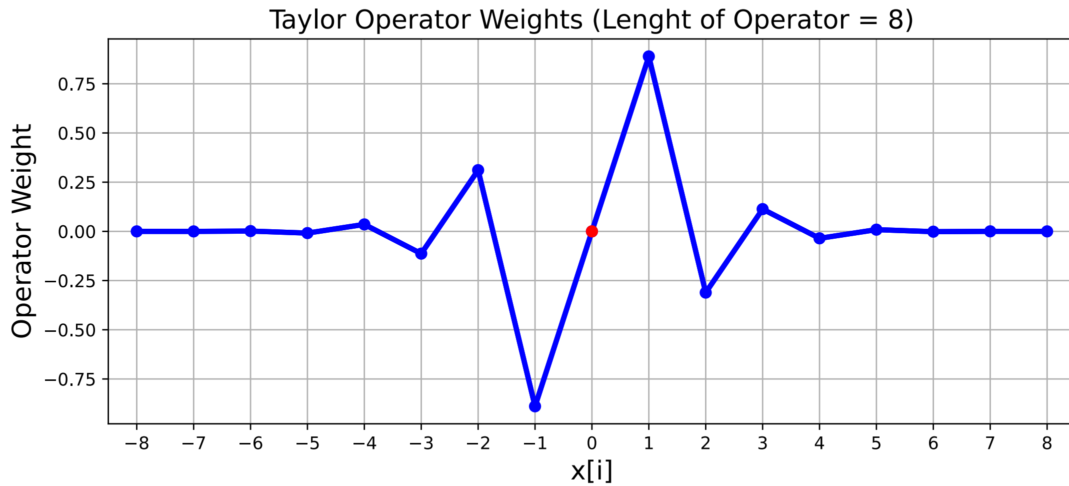


Figure 4: Visual representation of Taylor operators weights for the central derivative ($f^c(x)$). The derivative is computed for the red circle. The OX axis defines the grid point location in relation to the derivative location (x[0]). It can be noted how the magnitude of the weights decreases as the distance from the derivation point increases. In this case the operator length is 8, meaning that the interpolation radius of the derivative is 8 grid points. In total, this derivation stencil includes 16 points. Also, it can be seen that the weights are anti-symmetric in relation to x[0].

This algorithm can be adapted for higher order forward and backward derivatives also. The same principle can also be applied for the second or third derivative.

In order to visualize and verify the accuracy of the different numerical approximations of the derivative we can take a signal source in the form of a Ricker wavelet and compare the analytical solution with the different finite difference operators, as shown in Figure 5. Another way to compare the accuracy of these different methods is to see how quick they converge to the analytical solution by varying the size of the grid increment (Figure 6).

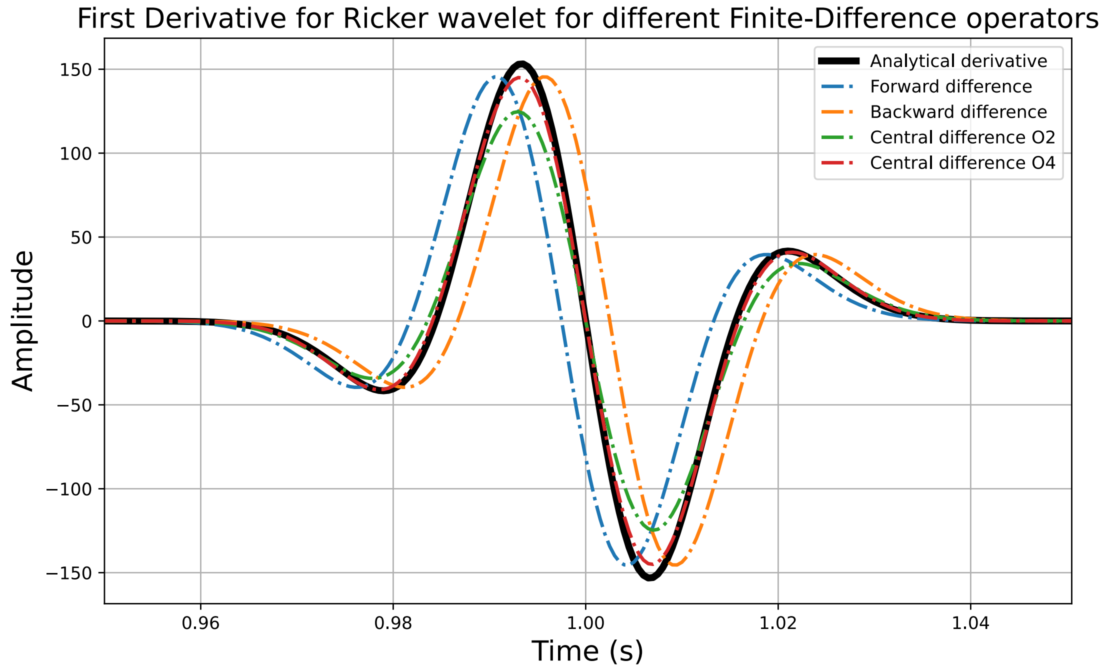


Figure 5: First derivative for a Ricker Wavelet with a central frequency of 25Hz. The time step used for the derivative is 5ms (dt = 5ms). The black lines represents the analytical solution; the blue dotted line is the forward difference ($O(dx)$); the orange dotted line is the backward difference ($O(dx)$); the green dotted line is the central difference of order 2 ($O(dx^2)$) and the red dotted line is the central difference of order 4 ($O(dx^4)$). The operators that have a higher order are closer to the analytical solution.

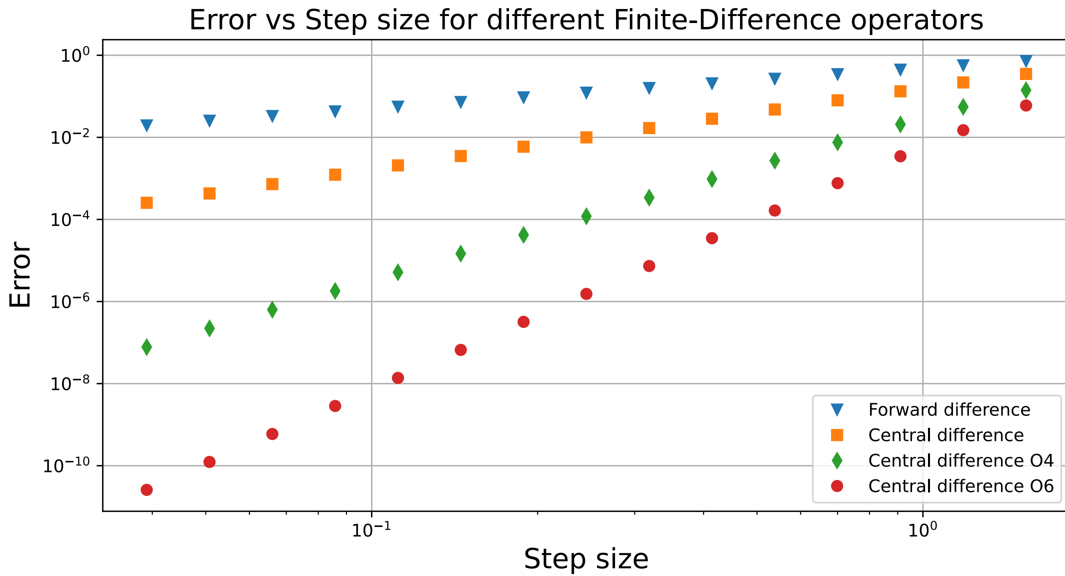


Figure 6: It can be seen how the high order finite-difference schemes converge more rapidly to the correct derivative on a regular grid, if one reduces the grid spacing dx.

The weights can be computed in many different ways, the Taylor expansion being just one of the methods available.

In this thesis, the coefficients are found through an optimization process described by Holberg 1987. This method concentrates on finding the right weights that minimize the errors in the numerical

simulations rather than minimizing the error in terms of the higher order derivatives in the Taylor expansion. The coefficients that are going to be used in our numerical simulations are shown in Table 1.

l	$\alpha = 1$	$\alpha = 2$	$\alpha = 3$	$\alpha = 4$	$\alpha = 5$	$\alpha = 6$	$\alpha = 7$	$\alpha = 8$
1	1.0021							
2	1.1452	-0.0492						
3	1.2036	-0.0833	0.0097					
4	1.2316	-0.1041	0.0206	-0.0035				
5	1.2463	-0.1163	0.0290	-0.0080	0.0018			
6	1.2542	-0.1213	0.0344	-0.0170	0.0038	-0.0011		
7	1.2593	-0.1280	0.0384	-0.0147	0.0059	-0.0022	0.0007	
8	1.2626	-0.1312	0.0412	-0.0170	0.0076	-0.0034	0.0014	-0.0005

Table 1: Finite-Difference Coefficients for a Staggered Grid as defined by Holberg 1987. l is the length of the differentiator (for a given l value, the total number of neighboring points used in the numerical derivation is $2 * l$).

2.2.3 Discretization and Staggered Grids

In order to run a simulation of the seismic wave propagation using finite differences, the subsurface model properties (acoustic case: Density and Bulk Modulus) and the wavefield characteristics (particle velocity and stresses) must be discretized both in space and in time.

Let us consider a regular grid defined as follows:

$$\begin{aligned} x &= i\Delta x, & i &= 0, 1, 2, \ldots, N_x, \\ z &= k\Delta z, & k &= 0, 1, 2, \ldots, N_z, \\ t &= n\Delta t, & n &= 0, 1, 2, \ldots, N_t. \end{aligned} \tag{27}$$

where N_x and N_z are the number of grid points in the x and z directions. N_t is the total number of time steps. Each location in the grid can be accessed through the position vector $\boldsymbol{x}$ defined as $\boldsymbol{x} = (x, z)$. This is considered the reference grid.

In addition, a second grid that is offset or staggered relative to the regular grid will be defined. In the staggered-grid technique, not all quantities in Equation 11 are located at the same grid points. Certain parameters are defined as being half a grid point displaced in relation to the reference grid (Figure 7).

The stresses and the bulk modulus are defined on a regular grid as follows:

$$\begin{aligned} \sigma_{xx}(\boldsymbol{x}, t) &= \sigma_{xx}(x, z, t), \\ \sigma_{zz}(\boldsymbol{x}, t) &= \sigma_{zz}(x, z, t), \\ K(\boldsymbol{x}) &= K(x, z). \end{aligned} \tag{28}$$

The particle velocities v_x and v_z and the density ρ are defined on staggered grids as follows:

$$\begin{aligned} v_x(\boldsymbol{x}, t) &= v_x(x + \Delta x/2, z, t), \\ v_z(\boldsymbol{x}, t) &= v_z(x, z + \Delta z/2, t), \\ \rho_x(\boldsymbol{x}) &= \rho(x + \Delta x/2, z), \\ \rho_z(\boldsymbol{x}) &= \rho(x, z + \Delta z/2). \end{aligned} \tag{29}$$

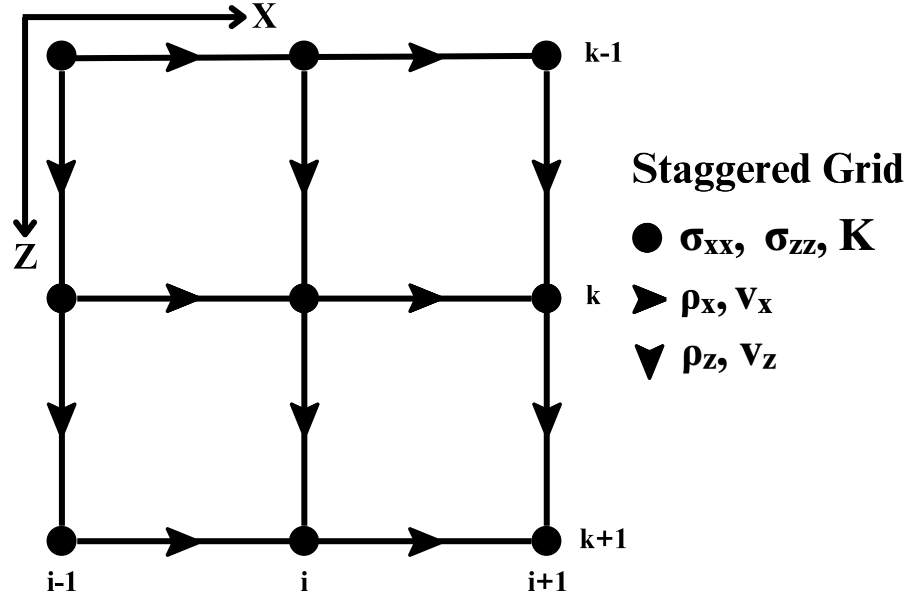


Figure 7: Staggered Grid schematic; σ_{xx}, σ_{zz} and K are defined on the regular grid vertexes, while v_x, v_z, ρ_x and ρ_z are defined on the staggered grid, which is displaced by half a grid point.

In Figure 6, it was shown that the error for a finite difference approximation of a derivative depends on the size of the spatial increment dx. By using the staggered grid technique, we are able to reduce the grid size by a factor of two, thus enhancing the accuracy. We can do this by first substituting the differentiations in equation 11, namely ∂_x and ∂_z, with the numerical operators d_x^+, d_x^-, d_z^+, and d_z^-.

These operators establish the connection between the reference and the staggered grid. Let us consider a function $f(x)$. The derivative of this function in the x direction can be approximated at $f(x + \Delta x/2)$ and at $f(x - \Delta x/2)$ by:

$$\begin{aligned} f'(x + \Delta x/2) &= d_x^+ f(x), \\ f'(x - \Delta x/2) &= d_x^- f(x) \end{aligned} \tag{30}$$

The d^+ and d^- differentiators are given by (Holberg 1987):

$$\begin{aligned} \partial^+ &= \frac{1}{\Delta x} \sum_{l=1}^{L} \alpha_l [f(x + l\Delta x) - f(x - (l-1)\Delta x)] \\ \partial^- &= \frac{1}{\Delta x} \sum_{l=1}^{L} \alpha_l [f(x + (l-1)\Delta x) - f(x - l\Delta x)] \end{aligned} \tag{31}$$

where Δx is the grid size increment, L is the operator length, α_l are the coefficients defined in Table 1. The same principle can be applied for the z-direction.

Thus, the numerical approximation of equation 11 can be written as:

$$\begin{aligned} \partial_t v_x &= \rho_x^{-1} d_x^+ \sigma_{xx} + f_x, \\ \partial_t v_z &= \rho_z^{-1} d_z^+ \sigma_{zz} + f_z, \\ \partial_t \sigma_{xx} &= K \left(d_x^- v_x + d_z^- v_z \right) + \dot{q_{xx}}, \\ \partial_t \sigma_{zz} &= K \left(d_z^- v_z + d_x^- v_x \right) + \dot{q_{zz}}. \end{aligned} \tag{32}$$

With regards to time, the Leapfrog method (explicit time-stepping) will be implemented and the

time derivatives will be approximated using the central difference formulation:

$$f'(t) = \frac{f(t + \Delta t/2) - f(t - \Delta t/2)}{\Delta t} \tag{33}$$

where Δt is the time step increment.

Algorithm for the Two-Dimensional Case

In order to formulate the complete numerical solution of the 2D finite difference acoustic equations, we will first consider the pseudo-stress σ, defined as:

$$\sigma = \frac{1}{2}(\sigma_{xx} + \sigma_{zz}) \tag{34}$$

Using the equation for the numerical approximation of the time derivative (33), the particle velocity components and the stresses from (32) can be computed as:

$$\begin{aligned} v_x(t + \Delta t/2) &= \Delta t[\rho_x^{-1} d_x^+ \sigma_{xx}(t) + \rho_x^{-1} f_x(t)] + v_x(t - \Delta t/2), \\ v_z(t + \Delta t/2) &= \Delta t[\rho_z^{-1} d_z^+ \sigma_{zz}(t) + \rho_z^{-1} f_z(t)] + v_z(t - \Delta t/2), \\ \sigma(t + \Delta t) &= \Delta t K \left(d_x^- v_x(t + \Delta t/2) + d_z^- v_z(t + \Delta t/2) \right) + \Delta t \dot{q} + \sigma(t). \end{aligned} \tag{35}$$

2.2.4 Analytical Solution

In order to verify the accuracy of the computed numerical model, it is useful to compare it against an analytical solution, which is possible for a homogeneous medium. The solution is given by Green's function, which for the 2D homogeneous acoustic problem takes the following form:

$$G(x, z, t) = \frac{1}{2\pi {V_p}^2} \frac{H\left((t - t_s) - \frac{|r|}{V_p}\right)}{\sqrt{(t - t_s)^2 - \frac{r^2}{{V_p}^2}}} \tag{36}$$

where H denotes the Heaviside function:

$$H(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{37}$$

G is the Green's function, (x, z) are the spatial coordinates, the source is located at (x_s, z_s), (t) is time, V_p is the constant P-wave velocity in the medium, t_s is the time at which the source is active and r is the source-receiver distance (offset), calculated as $r = \sqrt{(x - x_s)^2 + (z - z_s)^2}$.

From equation 36, it can be seen that the analytical solution is a damped Heaviside function, caused by the geometrical spreading of the wave.

2.2.5 Numerical Stability, Dispersion and Anisotropy

For the set of equations defined in 35, the wave motion parameters (v_x, v_z and σ) are computed recursively for each time step. In order to calculate the particle-velocity components at time step $(t + \Delta t/2)$ and the stress components at time step $(t + \Delta t)$, the particle-velocity components from

the previous time step ($t - \Delta t/2$) and the stress components from the current time step (t) are required.

This process of recursive calculations can result in numerical instability. Each error that is introduced in the numerical solution, even if it is relatively small, can be amplified over successive time steps. For an inappropriate choice of time sampling interval, spatial sampling interval and velocity the numerical solution may produce instabilities, anomalies and can even explode. In the most simple form for the 2D case, the stability/CFL (Courant–Friedrichs–Lewy) criterion takes the form:

$$Vp_{max} * \zeta * \frac{\Delta t}{dx} \leq \frac{1}{\sqrt{2}} \tag{38}$$

where Vp_{max} is the maximum P wave velocity in the model, Δt is the time sampling interval and dx is the spatial grid interval. The factor ζ depends on the length of the differentiator operator, dimension of the problem (1D, 2D, 3D) and the overall algorithm.

An example of the explosion of the finite difference solution is shown in Figure 8 and Figure 9, respectively. In this case, a homogeneous medium was used ($Vp = 2000 m/s$, $\rho = 2000 kg/m^3$). The source is represented by a Ricker wavelet with a dominant frequency of 25Hz, the grid interval is 10m, the time sampling interval is 2.74 ms and the Length of the differentiator is 3.

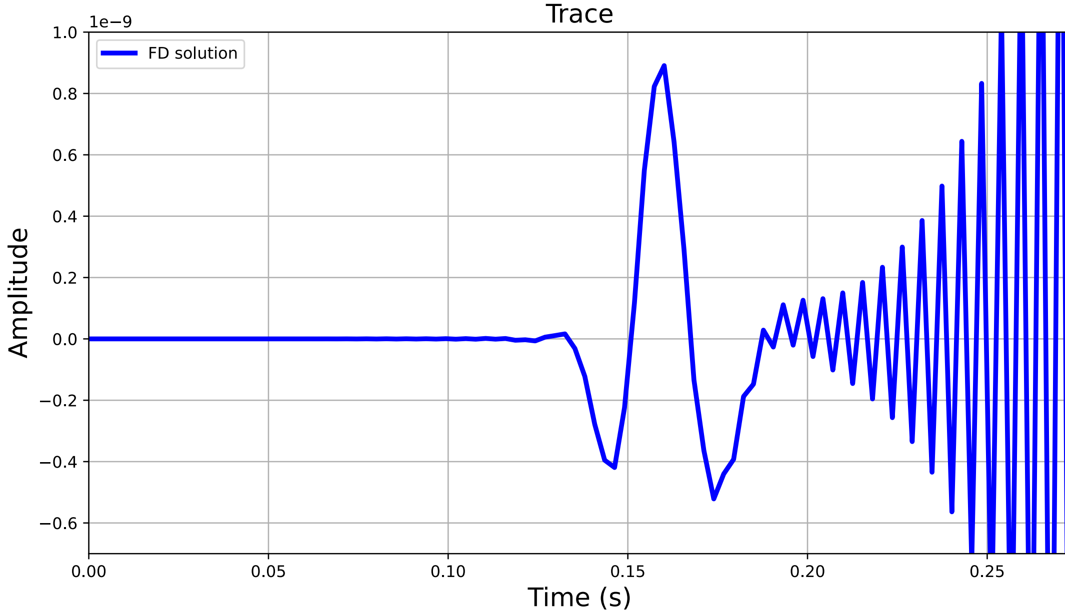


Figure 8: Explosion of the finite-difference solution represented in trace form. The blue line denotes the finite-difference approximation of the wavefield. It can be seen how the solution explodes at around 0.18 seconds, as the numerical errors get accumulated over time.

By using the CFL criterion, we can derive the optimum time sampling interval for a given velocity and spatial grid. But a stable solution is not necessarily an accurate one. Numerical dispersion is the phenomenon in which the wave speed of a numerical solution varies with frequency, resulting in the dispersion of the wave packets as they propagate through a medium. This error is caused by the truncation that occurs when spatial derivatives are approximated. Dispersion becomes more pronounced for higher frequency components. In order to reduce this phenomena, the wavelength must be sampled with an adequate number of grid points. According to Virieux 1986, a derivative approximation of second order accuracy (O2) requires a minimum of ten grid points per wavelength, while Levander 1988 shows that a fourth-order approximation requires a minimum of five grid points per dominant wavelength. An example of wave dispersion is shown in Figure 10. The disintegration

of the pulse is really strong and there is a significant difference between the simulation response and the analytical solution.

The last major aspect to consider is numerical anisotropy, which refers to the dependence of the accuracy of the numerical solution on the direction of wave propagation. An example of this is shown in Figure 11. The wave propagation is least precise along the grid directions (typically along the coordinate axes), while accuracy increases in other directions. This effect is especially pronounced for derivation stencils with fewer points or when using high central frequencies for the source pulse.

Both dispersion and anisotropy can be suppressed by utilizing stencils with a higher order accuracy and using a spatial grid that offers a large enough number of grid points per wavelength.

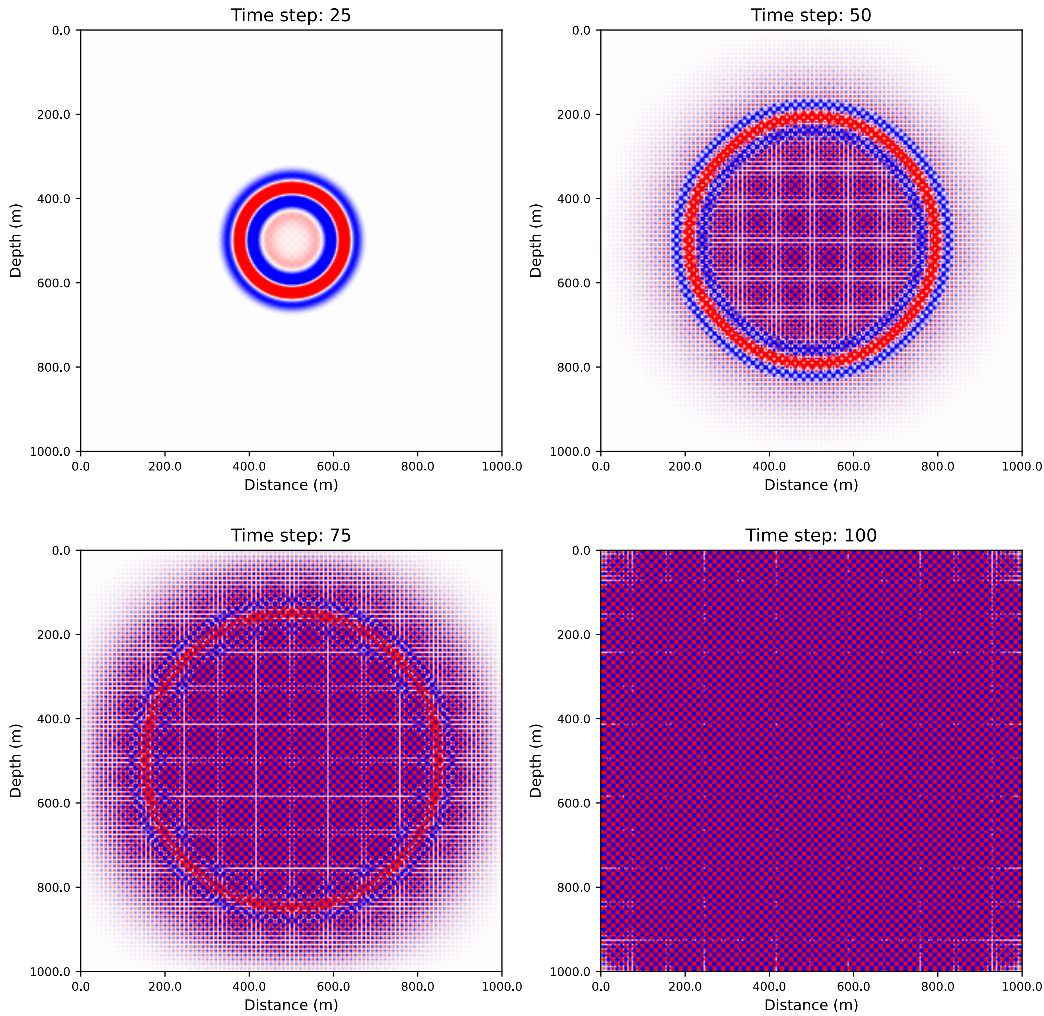


Figure 9: Wavefield propagation in the subsurface showcasing the explosion of the finite-difference solution at different time steps (Nt = 25, 50, 75 and 100). It can be seen how the wavefield disintegrates as time progresses.

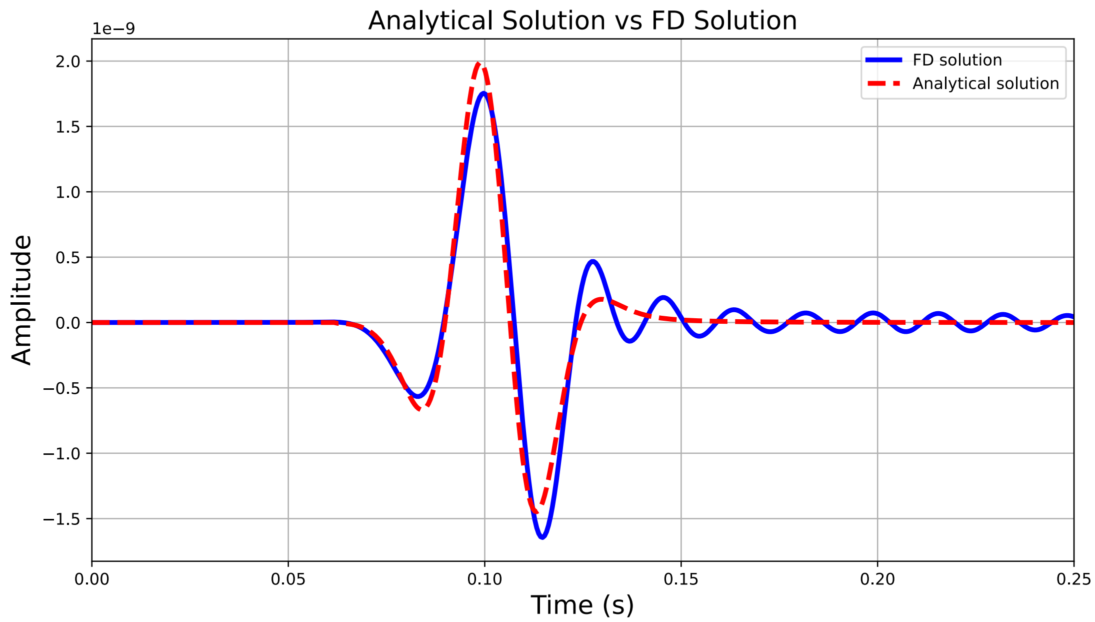


Figure 10: Dispersion in the finite-difference method. The blue line denotes the finite-difference approximation of the wavefield, while the red line shows the analytical solution. It can be seen how the wavelet becomes dispersive and disintegrates.

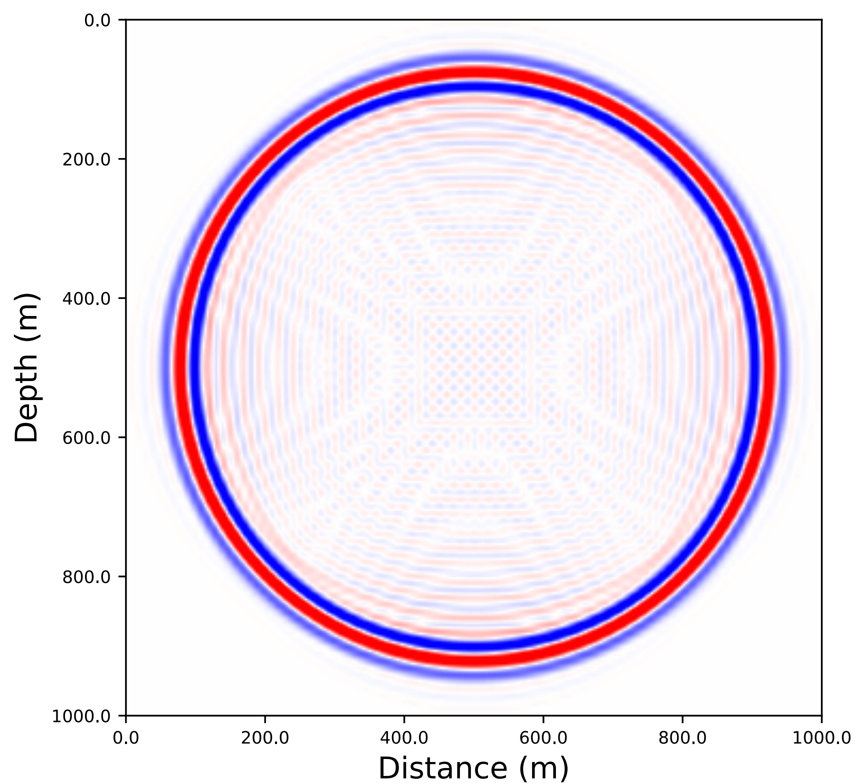


Figure 11: Numerical Anisotropy in the finite-difference method. It can be seen that the dispersion is the strongest at 0° and 90° with respect to the grid and the smallest at 45°.

2.3 Boundary Conditions

The last aspect of finite difference modeling that has not been addressed are the boundary conditions. These convey how the wavefield behaves at the edges of the subsurface model.

In the most simple form, they are given by the Dirichlet boundary conditions as:

$$\begin{aligned} \text{Left boundary } (x=0): &\quad v_x(t+\Delta t/2)=0, \\ \text{Right boundary } (x=N_x): &\quad v_x(t+\Delta t/2)=0, \\ \text{Top boundary } (z=0): &\quad v_z(t+\Delta t/2)=0, \\ \text{Bottom boundary } (z=N_z): &\quad v_z(t+\Delta t/2)=0. \end{aligned} \tag{39}$$

These conditions ensure that there is no movement of the medium at the boundaries at any given time, thus simulating a perfectly rigid boundary where waves are completely reflected (Figure 12).

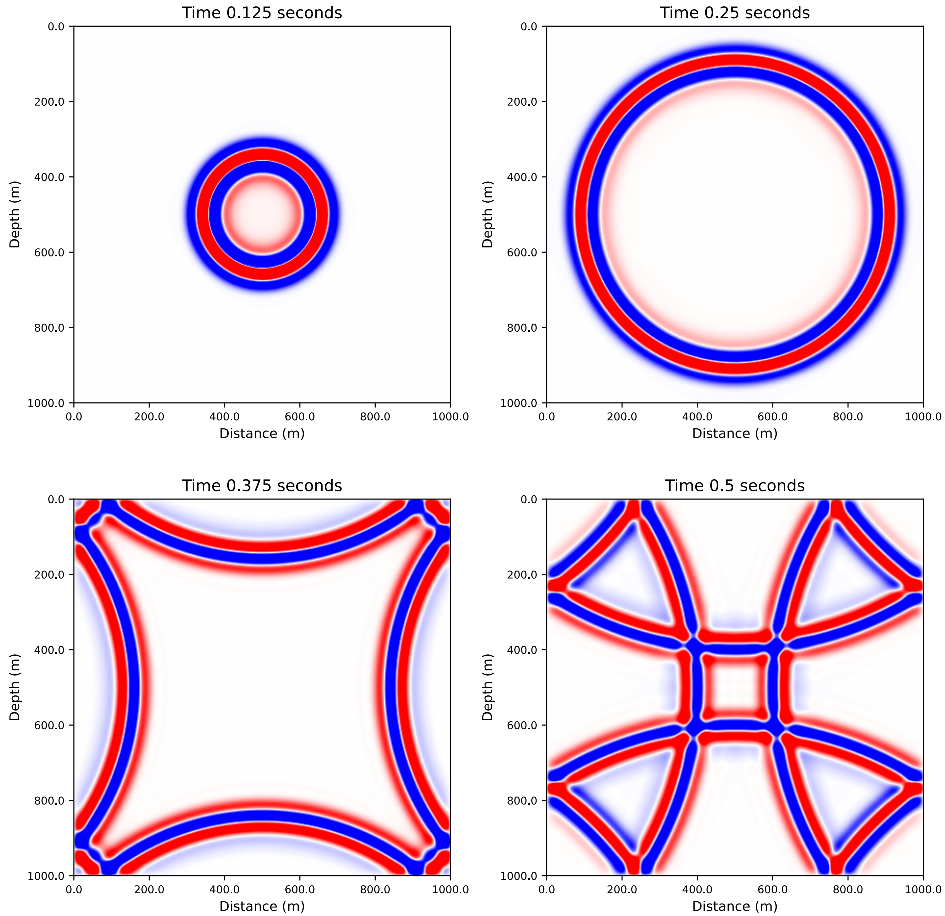


Figure 12: Wavefield propagation in the subsurface showcasing the Dirichlet boundary conditions. It can be seen how the wavefield is perfectly reflected back into the medium as it reaches the rigid boundaries. The polarity is also reversed.

While this might be desirable for the top boundary, as it can simulate a Water-Air or Land-Air interface, for the other boundaries it is needed to employ a method that absorbs the energy of the waves as they approach these artificial boundaries. From these, C-PML is one of the most successful and used methods that are deployed.

In the following, we will derive a visco-acoustic implementation of the equations in 11, by introducing a time dependent bulk modulus and density, and then show how these equations are equivalent to C-PML.

2.3.1 Viscoelastic Media

For an elastic medium, the stress only depends on the local strain at each point in space and time and the total energy of the system is conserved. For a linear viscoelastic medium, the stress at a given point in space and in time depends on the whole time history of the strain at that point (Hudson 1981). Thus, viscoelastic materials exhibit a time-dependent response to stress. This means that when these materials undergo a change in deformation as a result of applied stress, the reaction is not instantaneous but rather develops gradually. This response is depicted by a time-dependent function that characterizes the behavior of the material. This function contains the stress or strain history of the viscoelastic material (Carcione and Casula 1992). These types of media are said to have 'memory'. This implies that the present response of the material to stress is greater influenced by events in the recent past and as time passes, they diminish. The main implication of this is that the total energy of the system is not conserved anymore, but a part of it is dissipated (usually in the form of heat loss). So, waves that propagate in such materials are damped.

An example of how an elastic versus a viscoelastic stress-strain relation over time might look like is shown in Figure 13. It can be seen that in the elastic case, both the stress and strain changes over time are in phase, while in the viscoelastic case, stress and strain are out of phase during the loading and unloading cycle.

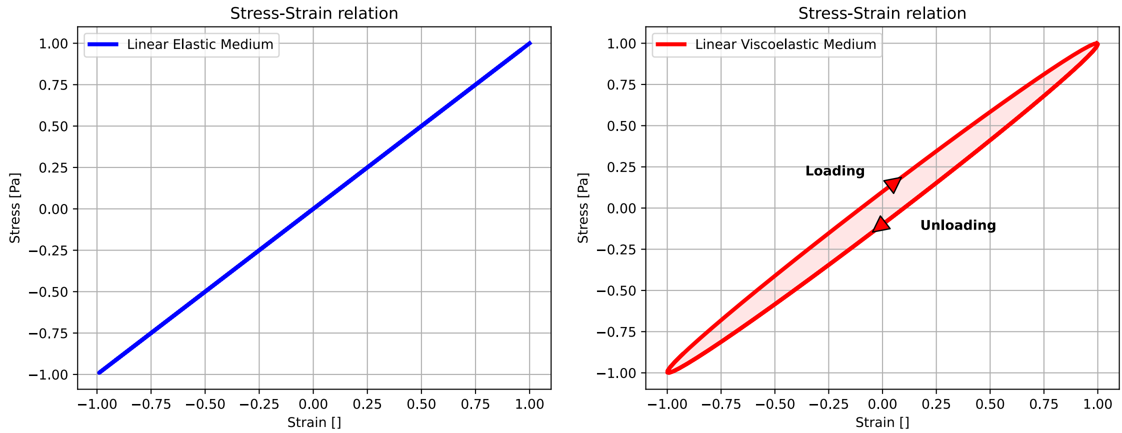


Figure 13: Stress–strain curves for an elastic material (left) and a viscoelastic material (right). The red shaded area represents the hysteresis loop and shows the amount of energy lost (in the form of heat dissipation) in the loading and unloading cycle.

The stress and the strain history are linearly related up to a given point in time. The strain resulting from any increase in stress is cumulative, adding to the strain already present in the material due to previous stress events. Boltzmann's generalization of Hooke's law to the viscoelastic case is given by (Hudson 1981) as:

$$\sigma_{ij} = \psi_{ijkl} * \dot{e}_{kl} \tag{40}$$

where σ_{ij} represents the components of the stress tensor, ψ_{ijkl} is the relaxation tensor, that describes how the material responds to stress over time and $\dot{e}_{kl}$ denotes the strain rate tensor.

The "$*$" denotes the convolution operation. For two arbitrary functions $a(t)$ and $b(t)$, the convolution operation is defined as:

$$a(t) * b(t) = \int_0^t a(t-\tau)b(\tau)d\tau \tag{41}$$

where τ, in our case, will represent the relaxation time.

By integrating by parts equation 40 and setting the initial condition $e(t=0)=0$ we get:

$$\sigma_{ij}(t) = \psi(0)_{ijkl} e_{kl}(t) + \int_{0+}^{t} \dot{\psi}_{ijkl}(t-\tau) e_{kl}(\tau) d\tau \tag{42}$$

If we consider that $\psi(0) = c_{ijkl}$ (Hudson 1981), where c_{ijkl} is the appropriate stiffness tensor for the elastic case (unrelaxed elastic modulus), we can rewrite equation 42 as:

$$\sigma_{ij}(t) = c_{ijkl} e_{kl}(t) + \int_{0+}^{t} \dot{\psi}_{ijkl}(t-\tau) e_{kl}(\tau) d\tau \tag{43}$$

This equation shows that the stress at any given time (t), depends on the unrelaxed elastic modulus, the current strain $e_{kl}(t)$, and a relaxation term. It can be seen that the case of perfect elasticity can be obtained by making the relaxation term (the time integral) equal to zero.

We define the time derivative of the relaxation function as:

$$\dot{\psi} = \phi(t) \tag{44}$$

Using this formulation of the rate of change of the relaxation function, we can rewrite equation 43 as:

$$\sigma_{ij}(t) = c_{ijkl} e_{kl} + \int_{0+}^{t} \phi_{ijkl}(t-\tau) e_{kl}(\tau) d\tau \tag{45}$$

This can be rewritten as a convolution of the stiffness of the material and strain, integrated over time:

$$\sigma_{ij}(t) = c_{ijkl}(t) * e_{kl}(t) \tag{46}$$

where c_{ijkl} is given by:

$$c_{ijkl}(t) = \psi(0)_{ijkl} \delta(t) + \phi_{ijkl}(t) \tag{47}$$

where $\psi(t=0)$ corresponds to the unrelaxed modulus.

A similar equation can be written for the time-dependent viscoelastic Lamé parameters ($\phi_\lambda(t)$ and $\phi_\mu(t)$), if an isotropic medium is considered.

$$\begin{aligned} \lambda(t) &= \lambda_u \delta(t) + \phi_\lambda(t), \\ \mu(t) &= \mu_u \delta(t) + \phi_\mu(t). \end{aligned} \tag{48}$$

The Lamé parameters are expressed as a sum of their unrelaxed components (λ_u, μ_u), and time-dependent parts ($\phi_\lambda(t)$ and $\phi_\mu(t)$).

2.3.2 Time-Dependent Density

A time relaxation process for the density can be introduced using a similar approach as for the Lamé parameters (equation 48). We relate the inverse of the effective density of the material to a function of time ($\rho_{eff}^{-1}(t)$) as follows:

$$\rho_{eff}^{-1}(t) = \rho_{eff}^{-1}(0)\delta(t) + \chi(t) \tag{49}$$

Substituting $\rho_{eff}^{-1}(0)$ into the equation with the initial (unrelaxed) density ρ_u^{-1} gives:

$$\rho_{eff}^{-1}(t) = \rho_u \delta(t) + \chi(t) \tag{50}$$

Thus, the density can be expressed as the sum of the unrelaxed component (ρ_u^{-1}) and time-dependent part, $\chi(t)$.

Now, we need a mechanical model in order to model the behavior of our viscoelastic material. In other words, to describe the relaxation function $\psi(t)$. In this thesis, we will consider the Standard Linear Solid Model and the Maxwell model.

2.3.3 Standard Linear Solid Model

There are numerous models that have been derived in order to represent a viscoelastic media. The following equations can be derived regardless of the chosen model. According to Carcione and Casula 1992, the Standard Linear Solid Model can represent processes such as grain boundary relaxations (Figure 14).

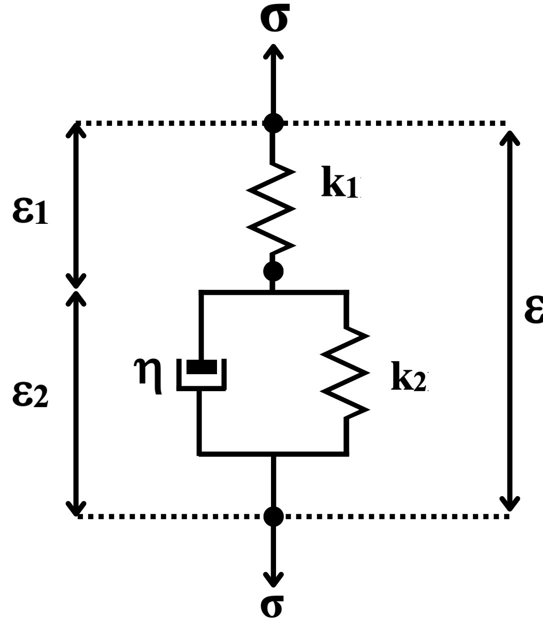


Figure 14: Standard Linear Solid Model. The mechanical model consists of two elastic springs (with the elastic modulus k) and a dashpot (where η is the viscosity of the material). σ is the applied stress and ϵ is the deformation.

Source: Modified after Carcione and Casula 1992

Thus, the components of the relaxation tensor ψ_{ijkl} are given by:

$$\psi(t) = K_r \left[1 - \left(1 - \frac{\tau_\epsilon}{\tau_\sigma}\right) \exp\left(-t/\tau_\sigma\right)\right] H(t) \tag{51}$$

where K_r represents the relaxed elastic modulus. The SLS (standard linear solid) element is characterized by two relaxation times, τ_σ and τ_e, for strain and stress, respectively. $H(t)$ is the Heaviside step function, given as:

$$H(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 & \text{if } t \geq 0 \end{cases} \tag{52}$$

This ensures the causality principle , meaning that the current values of stress do not depend on the future values of stress (Hudson 1981).

The time derivative of the relaxation function ($\phi(t) = \dot{\psi}$) becomes:

$$\phi(t) = \left[\left(\frac{K_r}{\tau_\sigma} \right) \left(1 - \frac{\tau_\epsilon}{\tau_\sigma} \right) \exp\left(-t/\tau_\sigma\right) \right] H(t). \tag{53}$$

The unrelaxed modulus ($\psi(t = 0)$), K_u, is related to the relaxed modulus K_r through equation 54:

$$K_u = \frac{\tau_\epsilon}{\tau_\sigma} K_r \tag{54}$$

Rearranging the terms, we can write K_r as:

$$K_r = \frac{K_u}{\frac{\tau_\epsilon}{\tau_\sigma}} \tag{55}$$

Finally, the ϕ function can then be expressed in terms of the unrelaxed modulus as follows:

$$\phi(t) = \left(\frac{\exp\left(-t/\tau_\sigma\right)}{\tau_\epsilon} \right) K_u \left(1 - \frac{\tau_\epsilon}{\tau_\sigma} \right) \tag{56}$$

We rewrite the equation as:

$$\phi(t) = \left(\frac{\exp\left(-t/\tau_\sigma\right)}{\tau_\epsilon} \right) \Delta K \tag{57}$$

In the ΔK term, the extent of the bulk modulus relaxation time is included.

$$\Delta K = K_u \left(1 - \frac{\tau_\epsilon}{\tau_\sigma} \right) \tag{58}$$

A similar equation can be written if we consider the Lamé parameters. The $\phi_\lambda(t)$ and $\phi_\mu(t)$ terms are defined in a similar way to equation 57, but each of them has separate relaxation times for the first and second Lamé parameters (τ_σ^λ, τ_σ^μ, τ_ϵ^λ, and τ_ϵ^μ):

$$\phi_\lambda(t) = \left(\frac{\exp\left(-t/\tau_\sigma^\lambda\right)}{\tau_\epsilon^\lambda} \right) \Delta\lambda \tag{59}$$

$$\phi_\mu(t) = \left(\frac{\exp\left(-t/\tau_\sigma^\mu\right)}{\tau_\epsilon^\mu} \right) \Delta\mu \tag{60}$$

where $\Delta\lambda$ and $\Delta\mu$ are:

$$\begin{aligned} \Delta\lambda &= \lambda_u \left(1 - \frac{\tau_\epsilon^\lambda}{\tau_\sigma^\lambda} \right) \\ \Delta\mu &= \mu_u \left(1 - \frac{\tau_\epsilon^\mu}{\tau_\sigma^\mu} \right) \end{aligned} \tag{61}$$

Time-Dependent Density

For density, the time-dependent part, $\chi(t)$, is given as:

$$\chi(t) = \left(\frac{\exp\left(-t/\tau_\sigma^\rho\right)}{\tau_\epsilon^\rho} \right) \Delta\rho \tag{62}$$

where $\Delta\rho$ is:

$$\Delta\rho = \rho_u \left(1 - \frac{\tau_\epsilon^\rho}{\tau_\sigma^\rho} \right) \tag{63}$$

Q-model Parametrization

The last step is to link the relaxation times (τ_σ and τ_ϵ) to a more relatable and intuitive subsurface parameter, the quality factor (Q). This is a dimensionless parameter that characterizes the energy dissipation caused by anelastic (viscoelastic) processes in the Earth's subsurface. Waves traveling in a medium with a high Q-factor will experience less damping, whereas, in a medium with a low Q-factor, waves attenuate rapidly.

According to Carcione and Casula 1992, for a standard linear solid model with a single element the Q-factor is defined as:

$$Q(\omega) = Q_0 \frac{1 + \omega^2 \tau_0^2}{2\omega\tau_0} \tag{64}$$

where ω is the angular frequency and Q_0 and τ_0 are given as:

$$\begin{aligned} Q_0 &= \frac{2\tau_0}{\tau_\epsilon - \tau_\sigma}, \\ \tau_0^2 &= \tau_\epsilon \tau_\sigma. \end{aligned} \tag{65}$$

$\omega = 1/\tau_0$ denotes the frequency for which the Q-factor reaches its minimum value, corresponding to the absorption peak.

Thus, if we give a Q-model and a chosen frequency as input parameters for the finite difference simulation, the relaxation times τ_σ and τ_ϵ can be computed as:

$$\begin{aligned} \tau_\epsilon &= \frac{\tau_0}{Q_0} \left[\sqrt{Q_0^2 + 1} + 1 \right], \\ \tau_\sigma &= \frac{\tau_0}{Q_0} \left[\sqrt{Q_0^2 + 1} - 1 \right]. \end{aligned} \tag{66}$$

2.3.4 Maxwell Model

A similar approach to the one described for the Standard Linear Solid can be followed to derive the equations for the Maxwell Model. According to Carcione and Casula 1992, the Maxwell Model can be associated with the attenuation caused by a viscoelastic fluid (Figure 15).

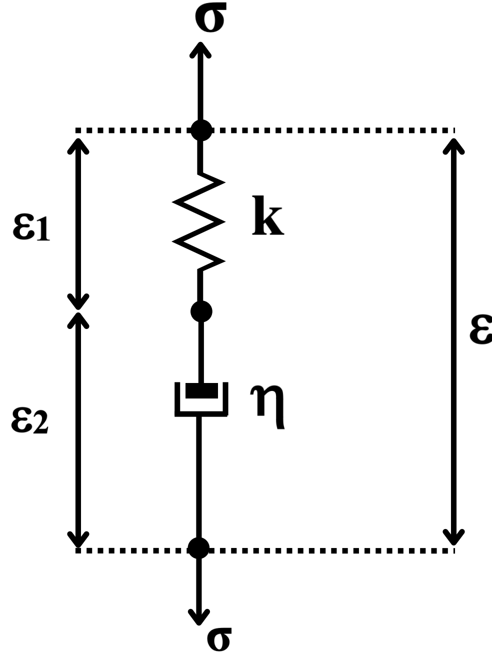


Figure 15: Maxwell Model, consisting of one elastic spring (with the elastic modulus k) and a dashpot (where η is the viscosity of the material) connected in series. σ is the applied stress and ϵ is the deformation.

Source: Modified after Carcione and Casula 1992

In this case, the relaxation function ψ takes the form:

$$\psi(t) = K_u \exp\left(-t/\tau_0\right) H(t) \tag{67}$$

where K_u represents the unrelaxed elastic modulus.

The time derivative of the relaxation function ($\phi(t) = \dot{\psi}$):

$$\phi(t) = -\Delta K \frac{1}{\tau_0} \exp\left(-t/\tau_0\right) H(t) \tag{68}$$

where $\Delta K = K_u$.

A similar equation can be written if we consider the Lamé parameters and density.

$$\begin{aligned} \phi_\lambda(t) &= -\Delta\lambda \frac{1}{\tau_0^\lambda} \exp\left(-t/\tau_0^\lambda\right) \\ \phi_\mu(t) &= -\Delta\mu \frac{1}{\tau_0^\mu} \exp\left(-t/\tau_0^\mu\right) \\ \chi(t) &= -\Delta\rho \frac{1}{\tau_0^\rho} \exp\left(-t/\tau_0^\rho\right) \end{aligned} \tag{69}$$

where $\Delta\lambda = \lambda_u$, $\Delta\mu = \mu_u$ and $\Delta\rho = \rho_u$.

The Q-value is related to τ_0 by:

$$\tau_0 = Q(\omega)/\omega \tag{70}$$

2.3.5 Viscoelastic Equations of Motion

In the previous section, it was shown that for a viscoelastic medium, an extra term that encompasses the time relaxation for the Lamé parameters and density needs to be added to the equations.

Thus, we can write the viscoelastic equations of motion and the constitutive relation, while also accommodating a time-dependent density. This introduces a relaxation process for the density that is similar to that of a viscoelastic medium. The motion equations and constitutive relation given in (5) and (6) can be rewritten as follows:

$$\begin{aligned}\partial_t^2 u_i(\boldsymbol{x},t) &= \rho_u^{-1}(\boldsymbol{x})\partial_j\sigma_{ij}(\boldsymbol{x},t) + f_i(\boldsymbol{x},t)\\ &+ \chi(\boldsymbol{x},t) * \partial_j\sigma_{ij}(\boldsymbol{x},t)\\ \sigma_{ij}(\boldsymbol{x},t) &= \lambda_u(\boldsymbol{x},t)e_{kk}\delta_{ij} + 2\mu_u(\boldsymbol{x},t)e_{ij} + q_{ij}(\boldsymbol{x},t)\\ &+ \delta_{ij}\phi_\lambda(\boldsymbol{x},t) * e_{mm} + 2\phi_\mu(\boldsymbol{x},t) * e_{ij}\end{aligned} \tag{71}$$

where χ, ϕ_λ and ϕ_μ are the time derivatives of the relaxation function for density, the first and second Lamé parameters, respectively. ρ_u, λ_u and μ_u are the unrelaxed density, first and second Lamé parameters, respectively.

For the 2D (non-zero particle displacements only in the xz plane) acoustic case, we set the second Lamé parameter to zero. These equations unfold to the following individual components:

$$\begin{aligned}\partial_t^2 u_x &= \rho_u^{-1}\partial_x\sigma_{xx} + f_x + \chi * \partial_x\sigma_{xx},\\ \partial_t^2 u_z &= \rho_u^{-1}\partial_z\sigma_{zz} + f_z + \chi * \partial_z\sigma_{zz}.\end{aligned} \tag{72}$$

And the stress-strain relations become:

$$\begin{aligned}\sigma_{xx} &= \lambda_u\left(e_{xx} + e_{zz}\right) + q_{xx}\\ &+ \phi_\lambda * [e_{xx} + e_{zz}]\\ \sigma_{zz} &= \lambda_u\left(e_{xx} + e_{zz}\right) + q_{zz}\\ &+ \phi_\lambda * [e_{xx} + e_{zz}]\end{aligned} \tag{73}$$

Velocity-Stress Formulation

Given the velocity $v_i = \dot{u}_i$ and the property that $(f * g)' = f' * g = f * g'$, one gets:

$$\begin{aligned}\partial_t v_x &= \rho_u^{-1}\partial_x\sigma_{xx} + f_x\\ &+ \chi * \partial_x\sigma_{xx}\\ \partial_t v_z &= \rho_u^{-1}\partial_z\sigma_{zz} + f_z\\ &+ \chi * \partial_z\sigma_{zz}\end{aligned} \tag{74}$$

The stress-strain relations in this case becomes:

$$\begin{aligned}\dot\sigma_{xx} &= \lambda_u\left(\dot e_{xx} + \dot e_{zz}\right) + \dot q_{xx}\\ &+ \phi_\lambda * [\dot e_{xx} + \dot e_{zz}]\\ \dot\sigma_{zz} &= \lambda_u\left(\dot e_{xx} + \dot e_{zz}\right) + \dot q_{zz}\\ &+ \phi_\lambda * [\dot e_{xx} + \dot e_{zz}]\end{aligned} \tag{75}$$

Memory Functions

In this format, the convolution operation adds a significant computational cost. It can be eliminated by defining some memory variables and including the time convolution into one set of variables. This implies a recursive computation of the memory functions.

$$\begin{aligned} \gamma_\lambda(t) &= \frac{1}{\Delta\lambda}\phi_\lambda * [\dot{e}_{xx} + \dot{e}_{zz}], \\ \theta_{kij}(t) &= \frac{1}{\Delta\rho^{-1}}\chi * \partial_k [\sigma_{ij}]. \end{aligned} \tag{76}$$

where $\Delta\lambda$ and $\Delta\rho$ are decided based on the viscoelastic model used.

Standard Linear Solid

For the Standard linear solid the expressions for the γ and θ functions are given as:

$$\begin{aligned} \gamma_\lambda &= \left[\frac{\exp\left(-t/\tau_\sigma^\lambda\right)}{\tau_\epsilon^\lambda}\right] * [\dot{e}_{xx} + \dot{e}_{zz}], \\ &= \left[\frac{\exp\left(-t/\tau_\sigma^\lambda\right)}{\tau_\epsilon^\lambda}\right] * \dot{e}_{xx} + \left[\frac{\exp\left(-t/\tau_\sigma^\lambda\right)}{\tau_\epsilon^\lambda}\right] * \dot{e}_{zz}, \\ &= \gamma_x^\lambda + \gamma_z^\lambda \\ \theta_{kij} &= \left[\frac{\exp\left(-t/\tau_\sigma^\rho\right)}{\tau_\epsilon^\rho}\right] * \partial_k \sigma_{ij}. \end{aligned} \tag{77}$$

We will take γ_x^λ as an example to show how we can replace the convolution operation with a recursive relation. By considering:

$$\begin{aligned} \gamma_x^\lambda(t+\Delta t) &= \int_0^{t+\Delta t} \frac{1}{\tau_\epsilon^\lambda} \exp\left(-\frac{t+\Delta t-\tau}{\tau_\sigma^\lambda}\right) \dot{e}_{xx}(\tau) d\tau \\ \gamma_x^\lambda(t+\Delta t) &= \frac{1}{\tau_\epsilon^\lambda} \exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right) \int_0^{t+\Delta t} \exp\left(-\frac{t-\tau}{\tau_\sigma^\lambda}\right) \dot{e}_{xx}(\tau) d\tau \\ \gamma_x^\lambda(t+\Delta t) &= \frac{1}{\tau_\epsilon^\lambda} \exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right) \int_0^{t} \exp\left(-\frac{t-\tau}{\tau_\sigma^\lambda}\right) \dot{e}_{xx}(\tau) d\tau \\ &+ \frac{1}{\tau_\epsilon^\lambda} \exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right) \int_t^{t+\Delta t} \exp\left(-\frac{t-\tau}{\tau_\sigma^\lambda}\right) \dot{e}_{xx}(\tau) d\tau. \end{aligned} \tag{78}$$

By considering that $\dot{e}_{xx}(t)$ is constant in the interval t to $t+\Delta t$, the second integral is approximated as following:

$$\begin{aligned} \gamma_x^\lambda(t+\Delta t) &= \frac{1}{\tau_\epsilon^\lambda} \exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right) \int_0^{t} \exp\left(-\frac{t-\tau}{\tau_\sigma^\lambda}\right) \dot{e}_{xx}(\tau) d\tau \\ &+ \frac{1}{\tau_\epsilon^\lambda} \exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right) \dot{e}_{xx}(t) \int_t^{t+\Delta t} \exp\left(-\frac{t-\tau}{\tau_\sigma^\lambda}\right) d\tau \end{aligned} \tag{79}$$

After solving the integral we get:

$$
\begin{aligned}
\gamma_x^\lambda(t+\Delta t) = \gamma_x^\lambda(t)\exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right) \\
+\frac{\tau_\sigma^\lambda}{\tau_\epsilon^\lambda}\left[1-\exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right)\right]\dot{e}_{xx}(t)
\end{aligned}
\tag{80}
$$

By assuming that the time step interval is very small ($\Delta t \ll 1$) the previous equation can be further simplified:

$$
\begin{aligned}
\gamma_x^\lambda(t+\Delta t) = \gamma_x^\lambda(t)\exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right) \\
+\frac{\Delta t}{\tau_\epsilon^\lambda}\dot{e}_{xx}(t)
\end{aligned}
\tag{81}
$$

In the same manner the equations for γ_z^λ and θ can be written.

Maxwell Solid

For the Maxwell solid the expressions for the γ and θ functions are given as:

$$
\begin{aligned}
\gamma_\lambda &= \left[-\frac{1}{\tau_0^\lambda}\exp\left(-t/\tau_0^\lambda\right)\right] * [\dot{e}_{xx}+\dot{e}_{zz}] \\
&= \left[-\frac{1}{\tau_0^\lambda}\exp\left(-t/\tau_0^\lambda\right)\right] * \dot{e}_{xx} + \left[-\frac{1}{\tau_0^\lambda}\exp\left(-t/\tau_0^\lambda\right)\right] * \dot{e}_{zz} \\
&= \gamma_x^\lambda + \gamma_z^\lambda \\
\theta_{kij} &= \left[-\frac{1}{\tau_0^\rho}\exp\left(-t/\tau_0^\rho\right)\right] * \partial_k\sigma_{ij}
\end{aligned}
\tag{82}
$$

As in the previous case, we take γ_x^λ as an example to show how we can replace the convolution operation with a recursive relation. By considering:

$$
\begin{aligned}
\gamma_x^\lambda(t+\Delta t) &= -\int_0^{t+\Delta t}\frac{1}{\tau_0^\lambda}\exp\left(-\frac{t+\Delta t-\tau}{\tau_0^\lambda}\right)\dot{e}_{xx}(\tau)d\tau \\
\gamma_x^\lambda(t+\Delta t) &= -\frac{1}{\tau_0^\lambda}\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right)\int_0^{t+\Delta t}\exp\left(-\frac{t-\tau}{\tau_0^\lambda}\right)\dot{e}_{xx}(\tau)d\tau \\
\gamma_x^\lambda(t+\Delta t) &= -\frac{1}{\tau_0^\lambda}\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right)\int_0^{t}\exp\left(-\frac{t-\tau}{\tau_0^\lambda}\right)\dot{e}_{xx}(\tau)d\tau \\
&\quad -\frac{1}{\tau_0^\lambda}\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right)\int_t^{t+\Delta t}\exp\left(-\frac{t-\tau}{\tau_0^\lambda}\right)\dot{e}_{xx}(\tau)d\tau
\end{aligned}
\tag{83}
$$

By considering that $\dot{e}_{xx}(t)$ is constant in the interval t to $t+\Delta t$, the second integral is approximated as following:

$$
\begin{aligned}
\gamma_x^\lambda(t+\Delta t) &= -\frac{1}{\tau_0^\lambda}\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right)\int_0^{t}\exp\left(-\frac{t-\tau}{\tau_0^\lambda}\right)\dot{e}_{xx}(\tau)d\tau \\
&\quad -\frac{1}{\tau_0^\lambda}\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right)\dot{e}_{xx}(t)\int_t^{t+\Delta t}\exp\left(-\frac{t-\tau}{\tau_0^\lambda}\right)d\tau
\end{aligned}
\tag{84}
$$

After solving the integral we get:

$$\gamma_x^\lambda(t+\Delta t) = \gamma_x^\lambda(t)\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right) + \left[1-\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right)\right]\dot{e}_{xx}(t) \tag{85}$$

By assuming that the time step interval is very small ($\Delta t \ll 1$) the previous equation can be further simplified:

$$\gamma_x^\lambda(t+\Delta t) = \gamma_x^\lambda(t)\exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right) + \frac{\Delta t}{\tau_0^\lambda}\dot{e}_{xx}(t) \tag{86}$$

In the same manner, the equations for γ_z^λ and θ can be written.

Viscoelastic Equations

This gives the final form of the viscoelastic equations (87).

$$\begin{aligned}
\partial_t v_x &= \rho_u^{-1}\partial_x\sigma_{xx} + f_x + \theta_x\Delta\rho^{-1}, \\
\partial_t v_z &= \rho_u^{-1}\partial_z\sigma_{zz} + f_z + \theta_z\Delta\rho^{-1}, \\
\dot{\sigma}_{xx} &= \lambda_u\left(\dot{e}_{xx}+\dot{e}_{zz}\right) + \dot{q}_{xx} + \gamma_\lambda\Delta\lambda, \\
\dot{\sigma}_{zz} &= \lambda_u\left(\dot{e}_{zz}+\dot{e}_{xx}\right) + \dot{q}_{zz} + \gamma_\lambda\Delta\lambda.
\end{aligned} \tag{87}$$

where

$$\begin{aligned}
\dot{e}_{xx} &= \partial_x v_x, \\
\dot{e}_{zz} &= \partial_z v_z.
\end{aligned} \tag{88}$$

The memory functions given in equation 76 can be rewritten in the recursive form as:

$$\begin{aligned}
\gamma_\lambda(t) &= \alpha_1\gamma_\lambda(t-\Delta t) + \alpha_2\left(\dot{e}_{xx}+\dot{e}_{zz}\right), \\
\theta_{kij}(t) &= \eta_1\theta_{kij}(t-\Delta t) + \eta_2\partial_k\sigma_{ij}.
\end{aligned} \tag{89}$$

where the α_1, α_2, η_1 and η_2 coefficients depend on the chosen viscoelastic model.

For the Standard Linear Solid mechanism they are defined as:

$$\begin{aligned}
\alpha_1 &= \exp\left(-\frac{\Delta t}{\tau_\sigma^\lambda}\right), \\
\alpha_2 &= \frac{\Delta t}{\tau_\epsilon^\lambda}, \\
\eta_1 &= \exp\left(-\frac{\Delta t}{\tau_\sigma^\rho}\right), \\
\eta_2 &= \frac{\Delta t}{\tau_\epsilon^\rho}.
\end{aligned} \tag{90}$$

While for the Maxwell mechanism they are defined as:

$$
\begin{aligned}
\alpha_1 &= -\frac{1}{\tau_0^\lambda} \exp\left(-\frac{\Delta t}{\tau_0^\lambda}\right), \\
\alpha_2 &= \frac{\Delta t}{\tau_0^\lambda}, \\
\eta_1 &= -\frac{1}{\tau_0^\rho} \exp\left(-\frac{\Delta t}{\tau_0^\rho}\right), \\
\eta_2 &= \frac{\Delta t}{\tau_0^\rho}.
\end{aligned} \tag{91}
$$

2.3.6 Discretization and Staggered Grids

In addition to the discretization scheme defined for the acoustic case, we need to add the memory functions (θ and γ) and the α and η coefficients on several regular and staggered grids (Figure 16).

Thus, the new discretization scheme is defined as follows. The stresses, γ function and the unrelaxed bulk modulus are defined on a regular grid as follows:

$$
\begin{aligned}
\sigma_{xx}(\boldsymbol{x}, t) &= \sigma_{xx}(x, z, t), \\
\sigma_{zz}(\boldsymbol{x}, t) &= \sigma_{zz}(x, z, t), \\
\gamma(\boldsymbol{x}, t) &= \gamma(x, z, t), \\
K_u(\boldsymbol{x}, t) &= K_u(x, z, t).
\end{aligned} \tag{92}
$$

Since in the acoustic case $\gamma_\mu = 0$, we will consider that $\gamma_\lambda = \gamma$. λ and K are used interchangeably as they are equivalent in the acoustic case.

The particle velocities (v_x and v_z), θ functions and the density are defined on staggered grids as follows:

$$
\begin{aligned}
v_x(\boldsymbol{x}, t) &= v_x(x + \Delta x/2, z, t), \\
v_z(\boldsymbol{x}, t) &= v_z(x, z + \Delta z/2, t), \\
\theta_x(\boldsymbol{x}, t) &= \theta_x(x + \Delta x/2, z, t), \\
\theta_z(\boldsymbol{x}, t) &= \theta_z(x, z + \Delta z/2, t), \\
\rho_{ux}(\boldsymbol{x}) &= \rho_{ux}(x + \Delta x/2, z), \\
\rho_{uz}(\boldsymbol{x}) &= \rho_{uz}(x, z + \Delta z/2).
\end{aligned} \tag{93}
$$

The visco-elastic coefficients α_1, α_2 are defines on a regular grid, while η_1 and η_2 are defined on a staggered grid:

$$
\begin{aligned}
\alpha_{1x}(\boldsymbol{x}) &= \alpha_{1x}(x, z), \\
\alpha_{2x}(\boldsymbol{x}) &= \alpha_{2x}(x, z), \\
\alpha_{1z}(\boldsymbol{x}) &= \alpha_{1z}(x, z), \\
\alpha_{2z}(\boldsymbol{x}) &= \alpha_{2z}(x, z), \\
\eta_{1x}(\boldsymbol{x}) &= \eta_{1x}(x + \Delta x/2, z), \\
\eta_{2x}(\boldsymbol{x}) &= \eta_{2x}(x + \Delta x/2, z), \\
\eta_{1z}(\boldsymbol{x}) &= \eta_{1z}(x, z + \Delta z/2), \\
\eta_{2z}(\boldsymbol{x}) &= \eta_{2z}(x, z + \Delta z/2).
\end{aligned} \tag{94}
$$

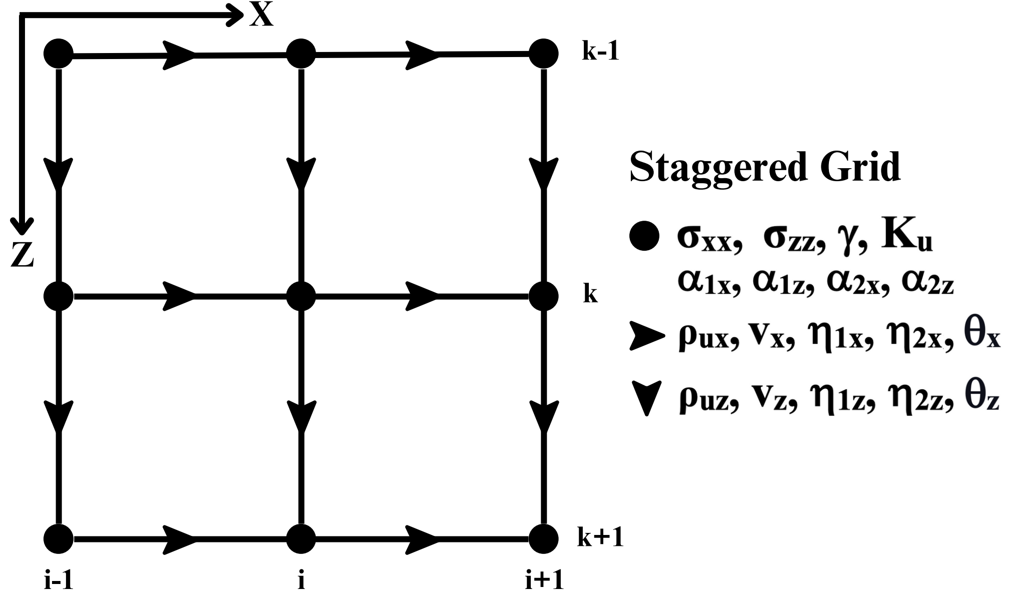


Figure 16: Staggered Grid for the Viscoacoustic wave equation schematic; σ_{xx}, σ_{xx}, γ, K_u and the α parameters are defined on the regular grid vertexes, while v_x, v_z, ρ_{ux}, ρ_{uz}, θ and the η parameters are defined on the staggered grid, which is displaced by half a grid point.

Thus, the numerical approximation of equation 87 can be written as:

$$\begin{aligned}
\partial_t v_x &= \rho_{ux}^{-1} d_x^{+} \sigma_{xx} + f_x + \theta_x \Delta \rho_x^{-1}, \\
\partial_t v_z &= \rho_{uz}^{-1} d_z^{+} \sigma_{zz} + f_z + \theta_z \Delta \rho_z^{-1}, \\
\dot{\sigma}_{xx} &= K_u \left(d_x^{-} v_x + d_z^{-} v_z \right) + \dot{q}_{xx} + \gamma \Delta \lambda, \\
\dot{\sigma}_{zz} &= K_u \left(d_x^{-} v_x + d_z^{-} v_z \right) + \dot{q}_{zz} + \gamma \Delta \lambda.
\end{aligned} \tag{95}$$

Algorithm for the Two-Dimensional Case

For the numerical solution of the 2D finite difference viscoacoustic equation, again, we will first consider the pseudo-stress (σ) defined as:

$$\sigma = \frac{1}{2}(\sigma_{xx} + \sigma_{zz}) \tag{96}$$

The particle velocity components and the stresses can be computed as:

$$\begin{aligned}
v_x(t + \Delta t/2) &= \Delta t[\rho_{ux}^{-1} d_x^{+} \sigma_{xx}(t) + f_x(t)] + \\
&+ \theta_x(t) \Delta \rho_x^{-1} \Delta t + v_x(t - \Delta t/2), \\
v_z(t + \Delta t/2) &= \Delta t[\rho_{uz}^{-1} d_z^{+} \sigma_{zz}(t) + f_z(t)] + \\
&+ \theta_z(t) \Delta \rho_z^{-1} \Delta t + v_z(t - \Delta t/2), \\
\sigma(t + \Delta t) &= \Delta t K_u \left[d_x^{-} v_x(t + \Delta t/2) + d_z^{-} v_z(t + \Delta t/2) \right] + \Delta t \dot{q} + \\
&+ \gamma(t + \Delta t/2) \Delta K \Delta t + \sigma(t).
\end{aligned} \tag{97}$$

We now split the γ function into two parts γ_x and γ_z as follows:

$$\begin{aligned}\sigma(t+\Delta t) = \Delta t K_u \left[d_x^- v_x(t+\Delta t/2) + d_z^- v_z(t+\Delta t/2)\right] + \Delta t \dot{q} \\ + \Delta t \left[\gamma_x(t+\Delta t/2)\Delta K + \gamma_z(t+\Delta t/2)\Delta K\right] + \sigma(t).\end{aligned} \tag{98}$$

The θ functions are updated as:

$$\begin{aligned}\theta_x(t+\Delta t) &= \eta_{1x}\theta_x(t) + \eta_{2x}\partial_x\sigma(t), \\ \theta_z(t+\Delta t) &= \eta_{1z}\theta_z(t) + \eta_{2z}\partial_z\sigma(t).\end{aligned} \tag{99}$$

The γ functions are given by:

$$\begin{aligned}\gamma_x(t+3/2\Delta t) &= \alpha_{1x}\gamma_x(t+\Delta t/2) + \alpha_{2x}d_x^- v_x(t+\Delta t/2) \\ \gamma_z(t+3/2\Delta t) &= \alpha_{1z}\gamma_z(t+\Delta t/2) + \alpha_{2z}d_z^- v_z(t+\Delta t/2)\end{aligned} \tag{100}$$

The α and η coefficients are computed based on the chosen viscoelastic model. Also, λ and K are used interchangeably as they are equivalent in the acoustic case.

Standard Linear Solid

The coefficients are given as:

$$\begin{aligned}\alpha_{1x} &= \exp\left(-\frac{d_x(x)\Delta t}{\tau_\sigma^\lambda}\right), \\ \alpha_{2x} &= \frac{d_x(x)\Delta t}{\tau_\epsilon^\lambda}, \\ \alpha_{1z} &= \exp\left(-\frac{d_z(z)\Delta t}{\tau_\sigma^\lambda}\right), \\ \alpha_{2z} &= \frac{d_z(z)\Delta t}{\tau_\epsilon^\lambda}, \\ \eta_{1x} &= \exp\left(-\frac{d_x(x)\Delta t}{\tau_\sigma^\rho}\right), \\ \eta_{2x} &= \frac{d_x(x)\Delta t}{\tau_\epsilon^\rho}, \\ \eta_{1z} &= \exp\left(-\frac{d_z(z)\Delta t}{\tau_\sigma^\rho}\right), \\ \eta_{2z} &= \frac{d_z(z)\Delta t}{\tau_\epsilon^\rho}.\end{aligned} \tag{101}$$

$\Delta\lambda$ and $\Delta\rho^{-1}$ are given as:

$$\begin{aligned}\Delta\lambda &= \lambda_u\left(1 - \frac{\tau_\epsilon^\lambda}{\tau_\sigma^\lambda}\right), \\ \Delta\rho^{-1} &= \rho_u^{-1}\left(1 - \frac{\tau_\epsilon^\rho}{\tau_\sigma^\rho}\right).\end{aligned} \tag{102}$$

Maxwell

The coefficients are given as:

$$
\begin{aligned}
\alpha_{1x} &= -\frac{1}{\tau_0^\lambda}\exp\left(-\frac{d_x(x)\Delta t}{\tau_0^\lambda}\right),\\
\alpha_{2x} &= \frac{d_x(x)\Delta t}{\tau_0^\lambda},\\
\alpha_{1z} &= -\frac{1}{\tau_0^\lambda}\exp\left(-\frac{d_z(z)\Delta t}{\tau_0^\lambda}\right),\\
\alpha_{2z} &= \frac{d_z(z)\Delta t}{\tau_0^\lambda},\\
\eta_{1x} &= -\frac{1}{\tau_0^\rho}\exp\left(-\frac{d_x(x)\Delta t}{\tau_0^\rho}\right),\\
\eta_{2x} &= \frac{d_x(x)\Delta t}{\tau_0^\rho},\\
\eta_{1z} &= -\frac{1}{\tau_0^\rho}\exp\left(-\frac{d_z(z)\Delta t}{\tau_0^\rho}\right),\\
\eta_{2z} &= \frac{d_z(z)\Delta t}{\tau_0^\rho}.
\end{aligned}
\tag{103}
$$

$\Delta\lambda$ and $\Delta\rho^{-1}$ are given as:

$$
\begin{aligned}
\Delta\lambda &= \lambda_u,\\
\Delta\rho^{-1} &= \rho_u^{-1}.
\end{aligned}
\tag{104}
$$

For both the Standard Linear Solid and Maxwell models, the profile functions d_x and d_z have the form:

$$
\begin{aligned}
d_x(x) &= (x/L)^2,\\
d_z(y) &= (z/L)^2.
\end{aligned}
\tag{105}
$$

where L is the length of the absorbing layer.

2.3.7 Absorbing Boundaries and Tapering

The absorbing boundary conditions can be implemented using the viscoacoustic set of equations by defining a strongly absorbing medium in a border zone with width Nb (or L). Such a subsurface model is shown in Figure 17. In addition to the Velocity and Density model, we must define a subsurface Quality factor model (Q-model). For the medium itself, we can simulate the acoustic case by setting a very high Quality factor ($Q = 10^4$). This will result in relaxation times that are close to zero, thus the relaxation terms in equation 97 will cancel out and the equation simplifies to the acoustic case. A more realistic attenuation can also be simulated ($10 \leq Q \leq 200$). In order to dampen the seismic waves in the border zone of the medium, we set a Q value of Qmax at the inner boundary (taken from the Q-model) and then the Q value is gradually reduced to Qmin at the outer boundary. The manner in which the Q values decrease from Qmax to Qmin is given by a tapering function.

The tapering (profile) functions $d_x(x)$ and $d_z(z)$ given in (105) ensure that the damping is gradually increased from the interior of the absorbing layer to the outer boundary, which allows for a smooth transition that minimizes unwanted reflections at the boundaries. In our case, the Q-model varies proportionally with the square of the distance from the inner border:

$$d_x(x) = \left(\frac{x}{Nb}\right)^2 \tag{106}$$

Where Nb is the number of grid points in the absorbing boundary and x is the distance in the absorbing zone. Such a tapering profile for the Q-model is shown in Figure 18.

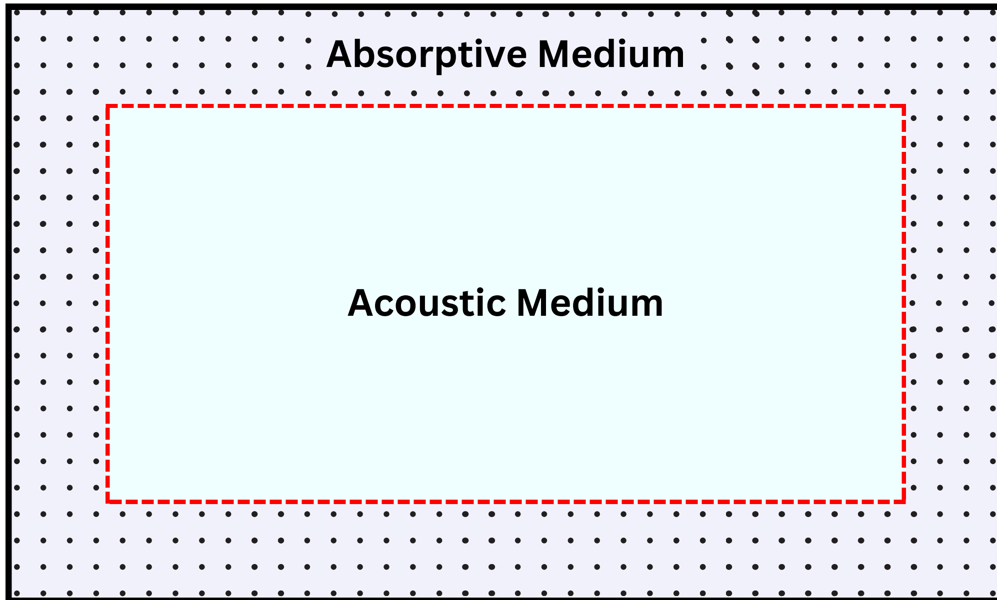


Figure 17: The subsurface model enclosed by an absorptive medium. The Acoustic Medium is obtained for a high quality factor ($Qmax = 10^4$). The Absorptive medium has a decreasing quality factor given by a tapering function, starting from the inner edge (the red dotted line) until it reaches the Qmin value ($Qmin = 1.1$), the black outer line.

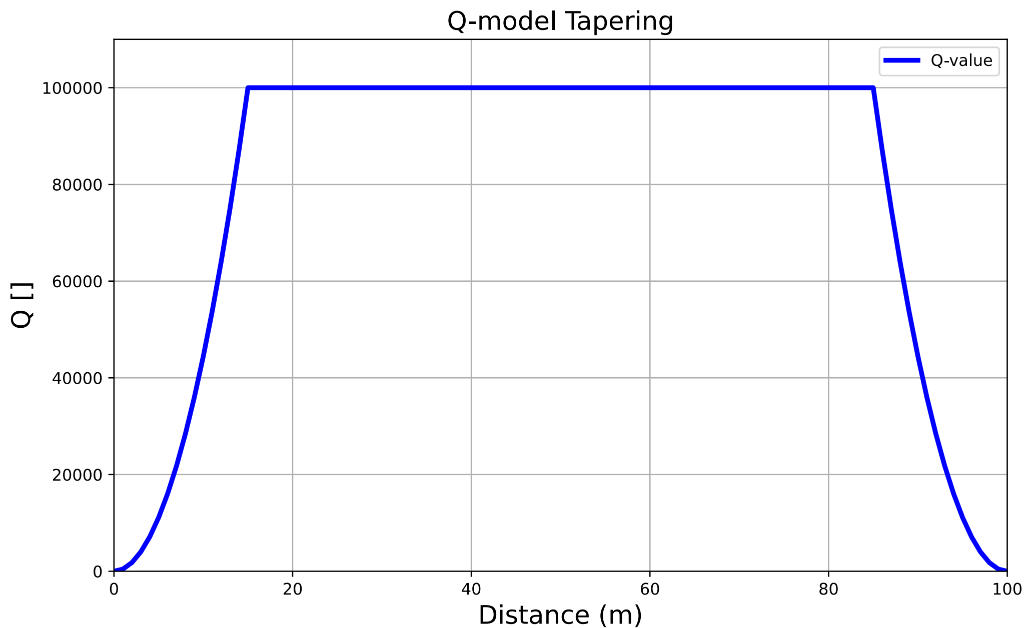


Figure 18: Example of the Q-model tapering, where $Qmax = 10^4$, $Qmin = 1.1$ and $Nb = 15$.

2.3.8 Comparison with C-PML

In order to implement the C-PML method, each spatial derivative within the attenuation region is replaced by a time convolution (Komatitsch and Martin 2007). For simplicity, we consider the 1D case:

$$\partial_i f \to s * \partial_i f, \quad i = x, z; \tag{107}$$

where f is either a stress component or a particle velocity.

$$s(t) = \frac{\delta(t)}{\epsilon} + a \exp(-bt) H(t). \tag{108}$$

The absorbing parameters (constants) are given by ϵ, a and b.

Thus the C-PML implementation for the acoustic wave equation defined in 11 modified for the 1D case is:

$$\begin{aligned} \dot{v}(x,t) &= s(t) * \left[\rho^{-1}(x)\partial_x \sigma(x,t)\right] \\ \dot{\sigma}(x,t) &= s(t) * \left[K(x)\partial_x v(x,t)\right] + \dot{q}(x,t) \end{aligned} \tag{109}$$

where s is given by equation 108.

Carcione and Kosloff 2013 points out that these C-PML boundary conditions are actually the kernel of a viscoelastic model, where both the time dependent Bulk Modulus and the Density can be given by the Standard Linear Solid mechanism.

If $s(t) * \rho^{-1}(x) \to \rho^{-1}(x,t)$ and $s(t) * \kappa(x) \to \kappa(x,t)$ then:

$$\begin{aligned} \dot{v}(x,t) &= \rho^{-1}(x,t) * \partial_x \sigma(x,t), \\ \dot{\sigma}(x,t) &= K(x,t) * \partial_x v(x,t) + \dot{q}(x,t). \end{aligned} \tag{110}$$

which is the stress-strain relation for a visco-elastic medium (46). The kernel simplifies to the acoustic case when $s(t) = \delta(t)$.

We can determine the equivalence between the relaxation times (τ_σ and τ_ϵ) in the viscoacoustic wave equation based on the standard linear solid mechanism and the relaxation parameters (ϵ, a and b) in the C-PML method by comparing the C-PML equation with our SLS equation (48, 56):

$$\begin{aligned} s(t) &= \frac{\delta(t)}{\epsilon} + a \exp(-bt) H(t), \\ \lambda(t) &= \lambda_u \delta(t) + \lambda_u \frac{1}{\tau_\epsilon}\left(1 - \frac{\tau_\epsilon}{\tau_\sigma}\right) \exp\left(-t/\tau_\sigma\right) H(t). \end{aligned} \tag{111}$$

Comparing the equations we see that:

$$\begin{aligned} \lambda_u &= \frac{1}{\epsilon}, \\ \lambda_u \frac{1}{\tau_\epsilon}\left(1 - \frac{\tau_\epsilon}{\tau_\sigma}\right) &= a, \\ \frac{1}{\tau_\sigma} &= b. \end{aligned} \tag{112}$$

Solving for τ_ϵ, τ_σ and λ_u:

$$\begin{aligned} \lambda_u &= \frac{1}{\epsilon}, \\ \tau_\epsilon &= \frac{1}{b + a\epsilon}, \\ \tau_\sigma &= \frac{1}{b}. \end{aligned} \tag{113}$$

3 Numerical Implementation and Optimization

The term High Performance Computing (HPC) usually refers to the use of supercomputers and parallel processing techniques for solving complex computational problems (Landro and Amundsen 2018). The supercomputer part will be neglected, the thesis being focused on the regular user that wishes to run seismic modeling experiments on his personal machine.

In this chapter, the code implementation of the viscoacoustic wave equation, with a concentration on leveraging HPC techniques to maximize computational efficiency, will be addressed. Based on the piece of hardware that performs the computations, the methods that are going to be benchmarked can be divided into three main branches: Single Core (Serial) CPU, Multiple Core (Multithread) CPU, and Graphics Card (GPU). Each of these will be implemented using C, Python, and Julia (also making use of different libraries and compilers that can accelerate the code). In addition, the capabilities of the Apple silicon M1, which integrates the CPU and GPU onto a single chip, will be shown.

We will showcase and exemplify these various techniques using a simple and easy to implement problem, namely SAXPY (Single-precision AX Plus Y), a function that is frequently used as a performance metric. The full code for all the Saxpy and Finite Difference implementations can be viewed in the attached GitHub repository (see Appendix).

3.1 Saxpy

Saxpy is a basic linear algebra operation that takes the simple form of $\mathbf{y} = a\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors of 32-bit floats with N elements and a is a 32-bit float scalar. In expanded form, the operation is given as:

$$\begin{aligned} y[1] &= a \cdot x[1] + y[1] \\ y[2] &= a \cdot x[2] + y[2] \\ y[3] &= a \cdot x[3] + y[3] \\ &\vdots \\ y[N] &= a \cdot x[N] + y[N] \end{aligned} \tag{114}$$

In other words, each element of the $\mathbf{x}$ vector is multiplied by the scalar a, and then added to the $\mathbf{y}$ vector, the final result either being overwritten over the original elements of the $\mathbf{y}$ vector (in order to save memory space), or saved in a new vector $\mathbf{z}$.

3.1.1 Single Core CPU

Modern CPUs (Central Processing Units) are typically designed with multiple cores, each core being capable of independently executing instructions. This enables the execution of multiple processes concurrently, a concept known as parallelism, which substantially boosts computational speed and efficiency.

In the conventional method of computation, when writing and then executing regular scripts, only a single core from the CPU is utilized. This is known as a serial implementation of the code, in which a single operation is performed on a single piece of data at a time (Figure 19). This bears the name of Single Instruction Single Data (SISD). Here, instructions are executed sequentially, one after the other (Cheng et al. 2014). While single core CPUs can attain high clock speeds, the serial nature of their processing imposes inherent limitations on the execution time of large computation problems.

Unfortunately, the basic code syntax of most programming languages target a single core, and if we want to make use of more, we need to use specific libraries and make modifications to the code itself to enable parallelization where it is possible.

Nonetheless, good performance can be attained with a single core also, if done properly, and the size of the problem is relatively small.

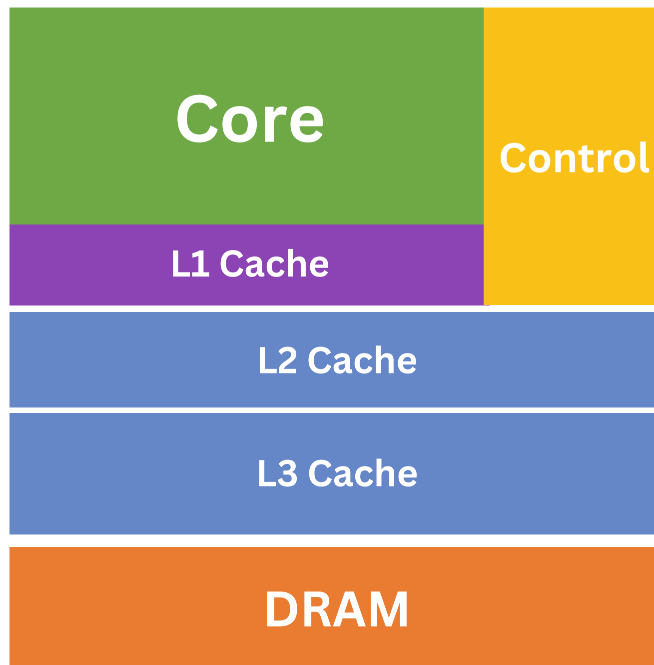


Figure 19: Basic elements of a Single Core CPU; The core contains the ALU (Arithmetic Logic Unit), which performs arithmetic and logic operations; The Control Unit tells the ALU what operation to perform and in which order to perform them; The L1, L2 and L3 caches are very fast and expensive memory that store temporary data that's actively being used by the CPU; The DRAM (Dynamic Random Access Memory) is the main memory used by the computers. While it can be very large, it is much slower than the Cache memory. The DRAM is not part of the CPU itself but is an external component. The CPU communicates with the DRAM through a memory bus.

Source: Modified after Cheng et al. 2014

Python

Python is a high-level, interpreted programming language with a focus on code readability and productivity. It supports procedural, object-oriented, and functional programming paradigms (Python Documentation 2023). Python's extensive collection of open source libraries and modules makes it suitable for a variety of tasks. In the past years, it has seen significant improvements in the field of scientific computation, allowing for a variety of options for those who wish to use it as a replacement to C/C++. Python's syntax is clear and straightforward, making it an excellent choice for beginners. However, the biggest disadvantage is that being an interpreted language, it is significantly slower than compiled languages such as C. Nonetheless, when execution speed is crucial, Python can be integrated with C or be optimized with tools and modules, which can help boost the speed in a significant way.

For all the Python implementations, a is defined as a $float32$ variable ($a = 3.1415$), while the $\mathbf{x}$ and $\mathbf{y}$ arrays are populated with random $float32$ numbers between 0 and 10. The $size$ variable indicates the number of elements in the arrays (Listing 1).

```python
import numpy as np

size = 1e6              # No. of elements in arrays
a = np.float32(3.1415) # scalar value (np.float32)
x = 10*np.random.rand(size).astype(np.float32) # array
y = 10*np.random.rand(size).astype(np.float32) # array
```

Listing 1: The variable declaration for Saxpy implemented using Python: a is a $float32$ scalar and $\mathbf{x}$ and $\mathbf{y}$ are NumPy arrays filled with $float32$ random values (between 0 and 10).

NumPy (Numerical Python) is an open-source library that provides support for large multidimensional arrays and matrices, together with a collection of functions to operate on these arrays, including mathematical, logical, shape manipulation, sorting, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation, and so forth. NumPy's functions are written in C, which provides performance comparable to conventional compiled languages while retaining Python's flexibility and ease of use (NumPy Developers 2023).

Saxpy Python For Loop CPU

The most straightforward and rudimentary approach to perform the Saxpy operation is by using a simple for loop that executes the operation on each element of the $\mathbf{x}$ and $\mathbf{y}$ arrays (Listing 2).

This method is not the most efficient way to conduct the Saxpy function, particularly for large vectors, despite its simplicity and ease of implementation. The Python interpreter introduces a significant latency when interpreting the for loop. In contrast to compiled languages, Python scripts must be interpreted line-by-line during execution, resulting in a performance penalty (Python Documentation 2023). This overhead is accentuated in the context of a for loop, where a single operation (in this instance, the Saxpy operation) is continually executed multiple times. Also, Python employs dynamic typing, which enables a great deal of flexibility, but can result in additional runtime time spent verifying and converting data types. Even though we have expressly defined these arrays as 32-bit floats, Python's dynamic type verification may still incur additional performance overhead.

```python
for i in range(size):
    y[i] = a * x[i] + y[i]
```

Listing 2: The Saxpy computation implemented using a simple for loop.

Saxpy Python NumPy CPU

As it was mentioned before, NumPy has an extensive library that allows to execute calculations using pre-compiled and optimized C functions, but using a for loop does not take advantage of this aspect. Instead, we must make use of vectorization, which enables us to conduct computations on entire arrays concurrently rather than iterating over them (NumPy Developers 2023). As shown in Listing 3, the Saxpy operation can be implemented with a single line of Python code, with the entire operation performed element by element on the NumPy arrays $\mathbf{x}$ and $\mathbf{y}$.

```python
y = a * x + y
```

Listing 3: The Saxpy computation implemented using NumPy vectorization.

NumPy's array-based operations substantially reduces the overhead imposed by the Python interpreter and enables us to take advantage of the underlying C-based implementation when iterating over the *for* loop. Dynamic typing may incur some delay, but the effect is less pronounced due to NumPy's efficient array operations management.

Using NumPy's vectorized operations is intuitive and also reduces the lines of code needed. This is a major benefit in terms of code readability and maintenance. The main drawback with this approach arises when dealing with nested loops and non-elementwise operations, such as a stencil, as these sometimes cannot be easily vectorized.

Saxpy Python Numba CPU

Numba is a just-in-time compiler for Python that can generate machine code from a subset of Python and NumPy code. It can be adapted and implemented for CPU (serial and multithread) and GPU, substantially enhancing the speed of numerical and scientific computations. It translates Python functions into optimized machine code at runtime using the LLVM compiler library, thus greatly enhancing the performance (Numba Documentation 2023). JIT compilation is the process of compiling source code during execution (just-in-time), as opposed to ahead-of-time (AOT) compilation, in which the source code is compiled before execution. In the most simple form, in order to optimize a function, you simply attach the `@jit` decorator ahead of it, and Numba will try to optimize the function automatically as best as possible (Listing 4).

```
from numba import jit

@jit(nopython=True)
def saxpy_numba(a, x, y):
    size = x.shape[0]
    for i in range(size):
        y[i] = a * x[i] + y[i]
    return y

y = saxpy_numba(a, x, y)
```

Listing 4: The Saxpy computation implemented using Numba.

The `@jit` decorator instructs Numba to compile the function in "no Python" mode (without using the Python interpreter). This mode is optimized for performance and is most effective when the function uses only NumPy arrays and loops. Unfortunately, this does not work with Python objects.

Numba provides a balance between the simplicity and readability of Python and the efficacy of lower-level programming languages. The main disadvantage of using Numba is that it introduces an additional layer of complexity to the code. It requires knowledge of how the JIT compiler works, how to take advantage of it, and what its limitations are. Thus, one needs to adapt the code accordingly.

Saxpy Python Cython CPU

The Cython programming language is a superset of Python that combines the simplicity of Python with the performance of C by permitting calls to C functions and the declaration of C types. This enables the compiler to produce highly efficient C code from Cython code. Once generated, the C code compiles with all main C/C++ compilers (Cython Documentation 2023). After the translation from Cython to C, we obtain a module that can be imported to Python, allowing for fast computations and integration with the rest of the Python code.

All the Cython functions need to be contained in a *.pyx* file, which will be later compiled. The

Saxpy operation is implemented using a for loop structure. Static typing is used, which can substantially accelerate the execution of code. Also, we use `@cython.boundscheck (False)`, `@cython.wraparound (False)`, and `@cython.cdivision (True)` to disable bounds checking, negative indexing, and C-style division, respectively. This will further increase the performance (Figure 20).

```
Generated by Cython 0.29.32

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

+01: cimport cython
 02: @cython.boundscheck(False)
 03: @cython.wraparound(False)
 04: @cython.cdivision(True)
+05: cpdef inline float[::1] saxpy_cython_V2(float alpha, float[::1] x, float[::1] y):
 06:     cdef int i
+07:     cdef int n = x.shape[0]
+08:     for i in range(n):
+09:         y[i] = alpha * x[i] + y[i]
+10:     return y
```

Figure 20: The Saxpy computation implemented using Cython; the yellow lines show all interactions with Python.

Cython also provides an annotated view of the functions you write that highlights all of its interactions with Python. By minimizing these interactions as much as possible, we can further improve the efficiency. In the case of our function, the main interaction is the conversion of NumPy variable arguments to C variable arguments.

To call the Cython function within a Python script, you first need to import the compiled module, similar to how you would import any other Python module: `from saxpy_cython import *`. Then you can call the Cython Saxpy function as: `y = saxpy_cython(a, x, y)`.

Again, as with Numba, the primary disadvantage of Cython is that it adds complexity and requires additional knowledge for effective use.

C

C is a general-purpose, compiled programming language (the C code is transformed into machine code by a compiler before it is run) that provides low-level memory access and is highly efficient. It is a procedural language that has a static type system. Its syntax is more complex than Python's, making it less beginner friendly. Also, memory management needs to be done manually (no automatic garbage collection) and it does not support Object Oriented Programming. However, its speed makes it the preferred programming language for high performance computing (which is also the case in the present). As a compiled language, C is significantly faster than Python and other interpreted languages.

For all the C implementations, a is defined as a $float32$ variable ($a = 3.1414$), while $\mathbf{x}$ and $\mathbf{y}$ are pointers to memory blocks populated with random $float32$ numbers between 0 and 10 (Listing 5). Pointers are variables that store the memory address of other variables (in our case, the memory address for the arrays). By using pointers, we are able to work directly with the system's memory and use it more efficiently, which is important when working with large data.

The `malloc()` function is utilized to dynamically allocate a memory block of the specified byte size. It is important to note that these allocated memory blocks are contiguous, so array elements are stored in adjacent memory locations. This allows for efficient access to array elements. However,

because the is no automatic memory management, the arrays must be freed when they are no longer required in order to prevent memory breaches.

While C code is very fast, this comes with some compromises. It lacks a lot of features present in languages like Python, such as: array index bound checking, automatic garbage collection, runtime type checking, exception management, and so forth. Thus, it can become cumbersome to use for someone that does not have a background in programming.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {

    long long size = 1e6; // No. of elements in arrays
    float a = 3.1415f;    // scalar value
    float* x = (float*)malloc(size * sizeof(float)); // pointer
    float* y = (float*)malloc(size * sizeof(float)); // pointer

    srand(time(NULL));
    for (long long i = 0; i < size; i++) {
        x[i] = ((float)rand()/(float)(RAND_MAX/10));
        y[i] = ((float)rand()/(float)(RAND_MAX/10));
    }

    // Free allocated memory!
    free(x);
    free(y);

    return 0;
}
```

Listing 5: The variable declaration for Saxpy implemented using C: a is a $float32$ scalar and **x** and **y** are dynamically allocated arrays filled with $float32$ random values (between 0 and 10).

Saxpy C For Loop CPU

The Saxpy operation is implemented in C using a for loop that iterates over the arrays (Listing 6). As the memory blocks allocated for the arrays are contiguous, accessing array elements is done efficiently.

```
void saxpy(float* x, float* y, float a, long long size) {
    for (long long i = 0; i < size; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

Listing 6: The Saxpy computation implemented in C.

Saxpy SWIG Wrapper

The Simplified Wrapper and Interface Generator (SWIG) is an interface compiler that links C and C++ programs to a number of high-level programming languages. By using the information from the C header files, it generates the linking code required for languages, such as Python, to access the underlying C code. One of the main advantages is that it requires minimal or no changes to the existing C code (SWIG Documentation 2023).

For the Saxpy operation, the C function we want to wrap can be, for instance, the one used in the previous section. We only need the function header file (*.h* file). Then, we construct a SWIG interface file (*.i* file) containing instructions on how to encapsulate our C function for SWIG. This interface file instructs SWIG how to convert the C function arguments to Python arguments. One of the main advantages is that NumPy arrays are also supported. An example of a code block that may be included in a SWIG interface file for the Saxpy function is shown in Listing 7.

```
%module wrapper
%{
  #define SWIG_FILE_WITH_INIT
  #include "wrapper.h"
%}

%include "numpy.i"
%init %{
import_array();
%}

%apply (float* INPLACE_ARRAY1, int DIM1) {(float* x, int sizex),(float* y, int
↪  sizey)}

%include "wrapper.h"
%clear (float* INPLACE_ARRAY1, int DIM1);
```

Listing 7: The interface file for SWIG

In order to generate the wrapper code and then compile it as a Python module, we also need a *setup.py* file that specifies the module name and the compilation arguments. Then, we can generate the module using the terminal command: `swig -python wrapper.i && python setup.py build_ext --inplace`. Now, the wrapper can be imported into a Python script as any other module and the Saxpy wrapper function can be called with: `y = wrapper.saxpy(a, x, y)`.

SWIG's primary benefit is that it enables the user to access C's speed for computation heavy problems while still being able to use Python and its extensive library ecosystem. But, it adds extra complexity, as you still need to be familiar with C and also learn all the caveats of SWIG for effective use.

Julia

Julia is a high-level, high-performance, dynamic programming language that was designed for the purpose of scientific and numerical computing, with performance comparable to statically typed languages, such as C. Julia utilizes just-in-time (JIT) compilation, which is powered by the LLVM compiler, and type inference in order to translate the Julia code into efficient machine code at runtime (Julia Documentation 2023). Julia is dynamically typed, but type inference is used to determine the types of variables and expressions. This enables it to generate more efficient

machine code, as it can optimize for particular types. Julia also supports multiple dispatch, which boosts performance by enabling the compiler to optimize code execution based on the types of the function parameters. It also has extensive libraries for: plotting, benchmarking your code, optimization modules and parallel computing both on the CPU and GPU. The JIT compilation, type inference and multiple dispatch all contribute to a boost in speed that makes Julia a great alternative to C.

Some disadvantages with Julia are related to the fact that it is a relatively young programming language (around ten years of development), thus it has a smaller community and is still maturing. Also, it has a 1-based indexing, which can be problematic and cause confusion sometimes, as 0-based indexing is the norm. Lastly, the JIT compilation can introduce overhead the first time a function is called (JIT latency), which sometimes can be significant (Julia Documentation 2023). In other words, the first time you execute a script/function, the compiler has to translate the Julia code into machine code, which takes some time. However, once the function has been compiled, subsequent queries to the function will be executed quickly because the machine code is cached. This reduces the impact of JIT latency on large time intensive computational problems, where the initial compilation time is just a small fraction of the total execution time.

For all the Julia implementations, a is defined as a $float32$ constant ($a = 3.1414$), while the $\mathbf{x}$ and $\mathbf{y}$ arrays are populated with random $float32$ numbers between 0 and 10 (Listing 8).

```
size = 1e6                    # No. of elements in arrays
const a = Float32(3.1415)     # scalar value (float32)
x = 10*rand(Float32, size)    # array
y = 10*rand(Float32, size)    # array
```

Listing 8: The variable declaration for Saxpy implemented using Julia: a is a $float32$ scalar and $\mathbf{x}$ and $\mathbf{y}$ are arrays filled with $float32$ random values (between 0 and 10).

Saxpy Julia CPU

In Julia, the Saxpy operation can be implemented in different ways, all of them showcasing good performance. We can iterate over each element with a for loop as shown in Listing 9. Here we can use the `BenchmarkTools` module to measure the performance of the function. The `@benchmark` macro executes the operation multiple times and returns some statistics, such as the mean and median execution time, that can be utilized for performance analysis. Julia facilitates all the necessary tools in order to easily test your code. The first execution of a function will include the extra compilation time, but subsequent calls will be faster because Julia caches the compiled code.

```
using BenchmarkTools

function saxpy(a,x,y)
    for i in 1:length(x)
        y[i] = a * x[i] + y[i]
    end
end

@benchmark saxpy(a,x,y)
```

Listing 9: The Saxpy computation implemented using a for loop in Julia

Julia also supports vectorization similar to NumPy, allowing for effective element-wise operations involving arrays (addition, subtraction, multiplication, etc). To implement this, we must make use of the dot syntax (.), which needs to be added before any mathematical operands (Listing (10)).

```
function saxpy(a,x,y)
    y .= a .* x .+ y
end
```

Listing 10: The vectorized Saxpy computation in Julia

3.1.2 Multiple Core CPU

Multithreading refers to a script that can take advantage of the multiple cores of the CPU by making the cores work on the same task at the same time. Thus, a single instruction set is applied to the dataset, the data being distributed across the cores. An example of this is an array multiplied by a scalar value. If two cores would be used, then the workload will be 'cut' in two, each half being given to a different core. This is possible because no information exchange is required between the cores to perform this computation. Thus, the available computing capacity is increased, and the time required to solve the problem will be halved.

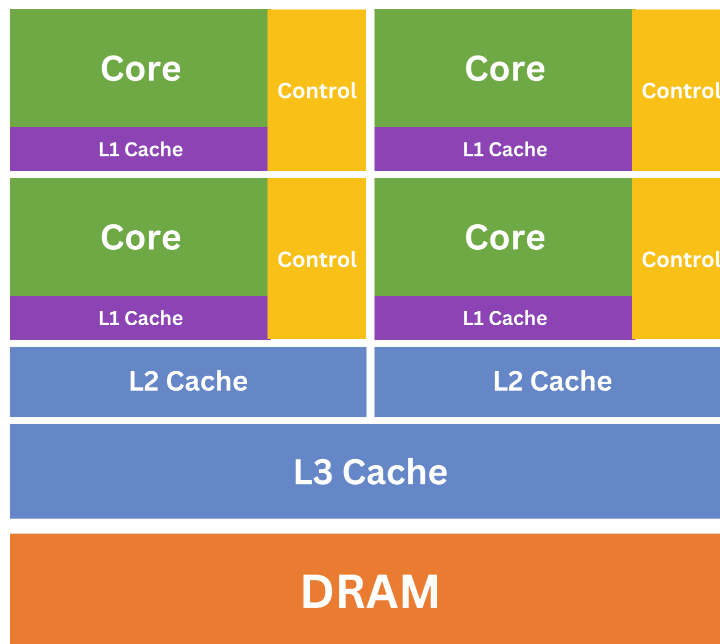


Figure 21: Basic elements of a Multiple Core CPU; Each core has its own memory cache, but all the cores also have access to shared memory. Tasks are sent to the CPU, and then they are distributed across the cores for simultaneous processing.

Source: Modified after Cheng et al. 2014

Saxpy Python Numba Multithread CPU

Numba also has Multithreading capabilities. The `@jit` decorator can automate the execution of NumPy array expressions across multiple CPU cores, making it easy to write parallel loops (Numba Documentation 2023).

To create a parallelized version of the Saxpy function using Numba, we just need to add the @jit decorator with the following arguments: nopython=True, parallel=True. For a further speed up, we can also add fastmath=True.

```
@jit(nopython=True,parallel=True,fastmath=True)
def saxpy_numba(a, x, y):
    size = x.shape[0]
    for i in numba.prange(size):
        y[i] = a * x[i] + y[i]
    return y
```

Listing 11: The Multithreaded Saxpy computation implemented using Numba.

The parallel=True argument enables automatic parallelization. In this mode, Numba will identify any loops that can be parallelized and then alter them to execute concurrently across multiple threads. You can indicate which loops are to be parallelized by using the numba.prange function. The fastmath=True argument enables mathematical optimizations that may result in quicker execution times at the expense of precision.

In order to set the number of threads that are going to be used in the execution of the code, we can use the following line: numba.set_num_threads(alpha), where *alpha* is the number of threads.

Saxpy Python Cython Multithread CPU

To use Cython's support for multithreading, we use the prange function to parallelize the for loop, thereby improving performance by distributing computation across multiple CPU cores (Figure 22). The desired number of threads to be used is given as one of the function arguments.

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
+01: cimport cython
 02: from cython.parallel import parallel, prange
 03: @cython.boundscheck(False)
 04: @cython.wraparound(False)
 05: @cython.cdivision(True)
+06: cpdef inline float[::1] saxpy_cython_multithread(float alpha, float[::1] x, float[::1] y, int threads):
 07:     cdef int i
+08:     cdef int n = x.shape[0]
+09:     for i in prange(n, nogil=True, num_threads=threads):
+10:         y[i] = alpha * x[i] + y[i]
+11:     return y
```

Figure 22: The Cython implementation using Multithreading; a lighter yellow line indicates fewer interactions with Python.

Python's Global Interpreter Lock (GIL) is a mechanism that prevents concurrent execution, allowing only one thread to execute at a time. This is primarily used to facilitate memory management, but it can be a performance bottleneck. By adding nogil=True to the Cython function, will cause the GIL to be deactivated for that section of code, allowing for parallel execution on multiple threads.

It is important that in the compilation of the *.pyx* file, the "$-fopenmp$" flag is included. OpenMP is an API that supports multiplatform shared-memory parallel programming in C, C++, and Fortran (OpenMP Documentation 2023). It is used to facilitate parallel computations in Cython by spawning multiple execution threads within a single process.

Saxpy C Multithread CPU

In C, the most straightforward way to implement multithreaded operations is by using the OpenMP library. For the Saxpy function, we can use the OpenMP directive `#pragma omp parallel` to parallelize the *for* loop (Listing 12).

```
#include <omp.h>
void saxpy(float* x, float* y, float a, long long size) {
    long long i;
    #pragma omp parallel for default(none) private(i) shared(a, x, y, size)
    for (i = 0; i < size; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

Listing 12: The C implementation using Multithreading

The $default(none)$ clause informs the compiler that no variables are shared between threads by default. This is a best practice for ensuring that we have total control over which variables are shared between threads and which are not.

The clause $private(i)$ instructs the compiler to consider the i variable as thread-private. Each thread will have its own copy of i, and modifications to i in one thread will not impact its value in another.

Lastly, $shared(a, x, y, size)$ specifies that these variables are shared between all threads (all threads can read and write to these variables). The number of threads to be utilized can be set with `omp_set_num_threads(alpha);` where *alpha* is the number of threads.

Saxpy Julia Multithread CPU

In order to use Multithreading in Julia, no external module has to be imported, as the $Threads$ module is part of Julia's standard library. Thus, it is simple to write multithreaded code in Julia, as the module provides several tools for managing threads and writing multithreaded code.

The number of threads to be used has to be specified before executing the script. This is done by setting the following environment variable: `JULIA_NUM_THREADS = alpha`, where alpha is the thread count, prior to launching Julia. Modifying the number of threads while Julia is operating cannot be done.

For the Saxpy function, we can automatically parallelize the for loop by using the `Threads.@threads` macro (Listing 13).

```
function saxpy(a,x,y)
    Threads.@threads for i in eachindex(x)
        y[i] = a * x[i] + y[i]
    end
end
```

Listing 13: The Julia Multithreaded implementation of Saxpy

The `eachindex(x)` function generates an iterator that traverses each index of the array, and the `Threads.@threads` macro distributes these iterations across multiple threads (Julia Documentation 2023).

3.1.3 GPU

The use of Graphics Processing Units (GPUs) for scientific computing is arguably one of the most important developments in the field of High Performance Computing. GPUs were initially designed for rendering graphics and their main use was in the gaming industry, but they have since evolved into structures capable of conducting multiple mathematical operations simultaneously, making them ideally adapted for large scale data processing. For the following section, we will consider the architecture of an Nvidia GPU, which could be found in a regular workstation.

Modern GPUs can be thought of as containing thousands of lesser CPUs (Figure 23). This immense parallel architecture enables them to efficiently handle tasks that can be divided into multiple smaller tasks, such as the computations typically required in numerical simulations.

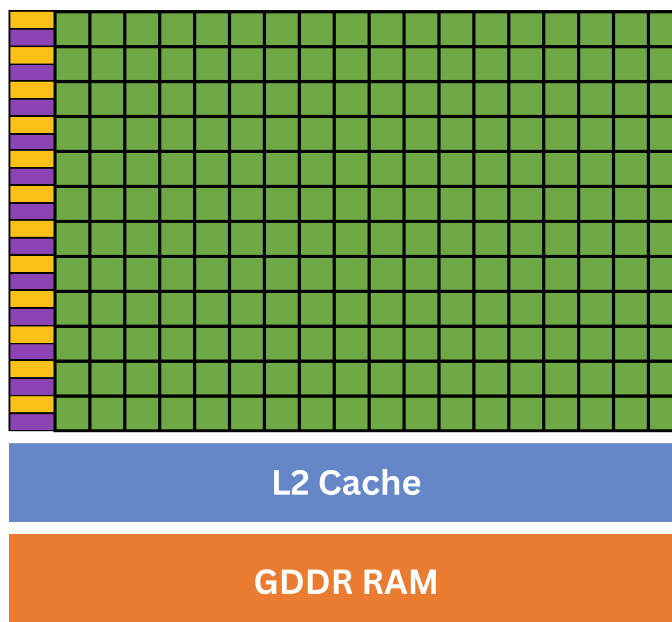


Figure 23: Basic elements of an Nvidia GPU; Each green square could represent a CUDA core. These cores have a lower clock rate than their CPU counterpart, but they can number in the thousands. The GPU has its own private memory, usually called GDDR RAM (Graphics Double Data Rate Random Access Memory). This is a type of synchronous memory that is specifically designed for GPUs.

Source: Modified after Cheng et al. 2014

The GPU is not meant to replace the CPU. CPUs are optimized for complex tasks and logic problems, while GPUs, with their immense parallelism, excel at image rendering, machine learning, and numerical simulations. This is where the paradigm of Heterogeneous computing is introduced (Figure 24).

This refers to the concept of building software and scripts that use both the CPU and the GPU (Cheng et al. 2014). The sequential, less computationally intensive parts are executed on the CPU, while the intensive parallels part can be executed on the GPU, thus allowing for optimal performance.

The "host" and "device" are terms that are usually used in this context. Typically, "host" refers to the CPU and its memory, whereas "device" refers to the GPU and its memory. It is important to note that the current norm is that the CPU and GPU both have their own respective memory. Thus, "host code" refers to the portions of code that are executed on the CPU, whereas "device code" refers to the portions of code executed on the GPU.

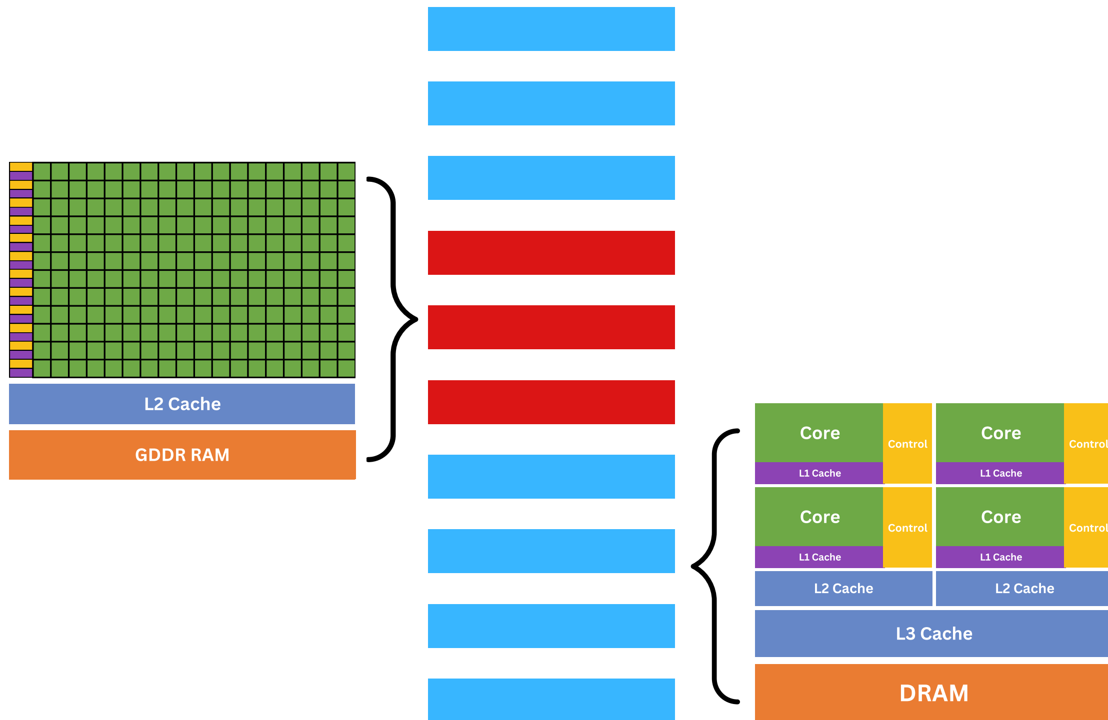


Figure 24: Example of heterogeneous computing when executing a script. The computationally heavy areas are performed by the GPU (red), while the less demanding serial code is performed by the CPU (blue).

Source: Modified after Cheng et al. 2014

In the context of GPU programming, the functions are replaced with kernels. In other words, a function that is written for parallel execution on the GPU. When a kernel is called, it is executed in parallel by each core of the GPU concurrently.

One important aspect to consider in GPU programming is that the execution model is hierarchical and consists of several components that have a specific terminology (Figure 25).

The **Device** refers to the GPU itself, which executes the kernels that are sent. The **Grid** represents the problem space in its entirety. In other words, all threads spawned by a single kernel launch are collectively called a grid (Cheng et al. 2014). Also, all the threads within a grid share the same global memory (Figure 23). The grid is divided into an array of blocks that all utilizes the same kernel. Pertaining to each case scenario, 1D, 2D, or 3D grids can be defined.

A **Block** is a collection of threads that can cooperate which each other and exchange data through shared memory. A block can be 1D, 2D, or 3D, similar to a grid. All threads within a block can synchronize their execution.

Threads are the smallest element of the hierarchy (Figure 26). Each thread executes a single instance of the kernel function. Threads from different blocks cannot cooperate with each other. Each thread has a unique ID, that can be accessed using two parameters: the $blockID$, giving the block index within a grid, and the $threadID$, the thread index within a block.

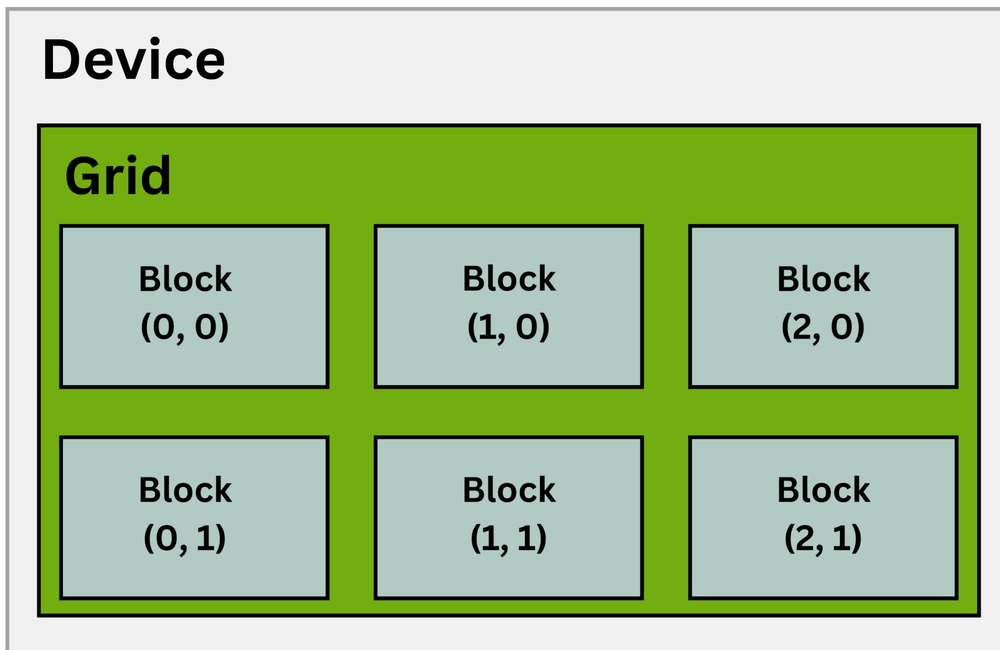


Figure 25: Example of Thread hierarchy, showcasing a 2D grid containing 2D blocks.

Source: Modified after Cheng et al. 2014

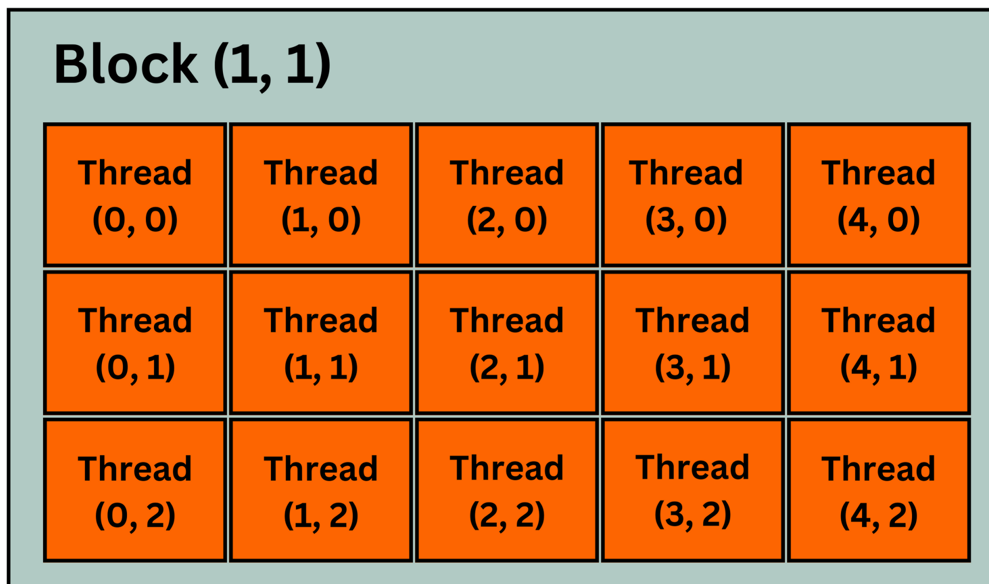


Figure 26: Example of Thread hierarchy, where the threads are organized in 2D blocks.

Source: Modified after Cheng et al. 2014

The dimension choice for the blocks and grids can have a direct effect on the efficiency and performance of the kernel. At the moment, the maximum number of threads that can be allocated per block for a regular Nvidia GPU is 1024. Following this hierarchy scheme, the GPU (device) can execute kernels across a grid of blocks, with multiple threads executing the kernel in each block.

Saxpy Python CuPy Numba GPU

For the Python GPU computation, we will consider the implementation on an Nvidia device and, at its core, will make use of the CUDA toolkit. CUDA (Compute Unified Device Architecture) is an NVIDIA-developed parallel computing platform and application programming interface (API) model (NVIDIA CUDA Documentation 2023). CUDA's central concept is to utilize the massive parallel processing capacity of Nvidia GPUs to execute computations. Its syntax is similar to the C language.

The integration of CUDA into Python can be done easily by making use of the Numba and CuPy libraries. CuPy is an open-source library that has a similar interface to NumPy for creating and manipulating GPU arrays. It allows users to write GPU accelerated code in a similar way you would use NumPy, resulting in a significant array performance increase (CuPy Documentation 2023). Numba also provides GPU capabilities and allows the user to write CUDA kernels in Python syntax in a straightforward way and then execute them on the GPU without needing to know CUDA or C (Numba Documentation 2023).

Combining Numba and CuPy is a powerful tool that allows to create kernels by only modifying only a few lines of code. Numba gives the user a fine control when creating the CUDA kernels, while CuPy provides a GPU accelerated version of the NumPy interface, making it easier to create arrays, offload them to the GPU and manipulate the data.

The Saxpy operation is implemented using a Numba kernel as shown in Listing 14. The kernel is declared by adding the `@cuda.jit` decorator, which compiles the function on the GPU in a JIT fashion.

```
import cupy as cp
import numba.cuda as cuda

@cuda.jit
def saxpy_kernel(a, x, y):
    i = cuda.grid(1)
    if i < x.size:
        y[i] = a * x[i] + y[i]

x = 10 * cp.random.rand(size).astype(cp.float32)
y = 10 * cp.random.rand(size).astype(cp.float32)

# Set up the grid and block dimensions
threads_per_block = 1024
blocks_per_grid = (x.size + (threads_per_block - 1)) // threads_per_block

saxpy_kernel[blocks_per_grid, threads_per_block](a, x, y)
cp.cuda.Device().synchronize()
```

Listing 14: The Python GPU implementation of Saxpy

By using the `i = cuda.grid(1)` function, a unique index for each thread (i) across the grid is calculated. The operand 1, denotes the dimension of the problem. The Saxpy operation is executed if the thread's index falls within the range of the size of the input vectors. This check is essential for preventing out of bounds accesses, as the total number of threads (determined by the grid and block dimensions) may exceed the size of the vector.

The **x** and **y** arrays are constructed similarly to NumPy, but using the *cp* (CuPy) annotation instead. The primary difference between NumPy and CuPy is where the arrays are stored. CuPy creates arrays directly in the GPU's device memory, whereas NumPy generates arrays that are

stored in the DRAM memory, which is accessed by the CPU. These arrays can also be created with NumPy on the CPU and then transferred, using CuPy, to the GPU.

The Grid and Block dimensions are established using a 1D layout. The number of threads per block is set to 1024 (the maximum possible value), and the number of blocks per grid is calculated to cover all elements of **x** and **y**. As CUDA operations are typically asynchronous, the `cp.cuda.Device().synchronize()` line is used to ensure that all CUDA operations have been completed before the program continues.

Saxpy C CUDA GPU

In order to use Nvidia's CUDA capabilities, we can directly write the kernels within a C script, but save the file with a *.cu* extension. Listing 15, illustrates how to implement this on the Saxpy operation.

```
#include <cuda.h>

__global__ void saxpy(float *x, float *y, float a, long long size )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size)
        y[i] = a * x[i] + y[i];
}

int main() {

    // Allocate memory for the arrays on the GPU
    float* d_x, *d_y;
    cudaMalloc(&d_x, size * sizeof(float));
    cudaMalloc(&d_y, size * sizeof(float));

    // Copy the data from the CPU to the GPU
    cudaMemcpy(d_x, x, size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, size * sizeof(float), cudaMemcpyHostToDevice);

    // Set the Block and Grid dimensions
    int threadsPerBlock = 1024;
    int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

    // Execute kernel
    saxpy<<<blocksPerGrid, threadsPerBlock>>>(d_x, d_y, a, size);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    // Copy the data from the GPU to the CPU
    cudaMemcpy(y, d_y, size * sizeof(float), cudaMemcpyDeviceToHost);

    // Free memory
    cudaFree(d_x);
    cudaFree(d_y);

    return 0;
}
```

Listing 15: The CUDA GPU implementation of Saxpy

The kernel is defined using the `__global__` function, indicating that it will be executed on the GPU. The unique thread index i is computed by using the block index (`blockIdx.x`), the number of threads in each block (`blockDim.x`), and the thread index within the block (`threadIdx.x`). This index is used to access the corresponding elements in each array, and while the index resides within the array dimensions, the operation is executed.

In order to allocate memory on the GPU for the arrays, the `cudaMalloc` command is used. The data is then copied from the host to the device using the `cudaMemcpy` function and the `cudaMemcpyHostToDevice` argument.

The block and grid dimensions are then defined, with the number of threads per block set to the maximum of 1024 and the number of blocks per grid calculated to ensure that all **x** and **y** elements are covered.

After the kernel function is executed, we use the `cudaDeviceSynchronize` command to ensure all the GPU computations are finished before proceeding. After, the data is sent back from device to host, the GPU memory is liberated using `cudaFree`.

This combination of kernel execution, memory management, and data transfer forms the basis of the majority of CUDA C programs, allowing the GPU's capability to be utilized directly within the C programming language.

Saxpy Julia GPU

For the GPU implementation in Julia, the CUDA.jl module is utilized. This package provides a high-level interface for CUDA programming, allowing GPU kernels to be written directly in Julia. This can be done both by using a high or low level of abstraction.

The high-level, vectorized approach is easily implemented and resembles the NumPy API, and is almost the same as the Julia CPU implementation, the only difference being given by how the arrays are defined in the first place (Listing 16).

```
using BenchmarkTools
using CUDA

function saxpy(a,x,y)
    y .= a .* x .+ y
end

x = CUDA.rand(Float32, size)
y = CUDA.rand(Float32, size)

@benchmark CUDA.@sync saxpy(a,x,y)
```

Listing 16: The Julia vectorized GPU implementation of Saxpy

Using the `CUDA.` annotation before the arrays ensures that they are created on the GPU directly and the operations on them are also performed by the GPU. Utilizing the `CUDA.@sync` macro guarantees that all GPU computations are completed prior to benchmarking. The arrays can also be created on the host and transferred later to the device. The main advantage with this approach is that kernel and threading hierarchy are implemented automatically. The garbage collection is done automatically, so the memory does not have to be freed manually.

On the other hand, the low-level, manual kernel approach offers greater control over the computation's execution and optimization (Listing 17).

```
function saxpy_gpu_kernel!(a,x,y)
    i = (blockIdx().x - 1)* blockDim().x + threadIdx().x
    if i <= length(y)
        @inbounds y[i] = a * x[i] + y[i]
    end
end

nthreads = CUDA.attribute(device(),CUDA.DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK)
nblocks = cld(size, nthreads)

@benchmark CUDA.@sync @cuda(
threads = nthreads,
blocks = nblocks,
saxpy_gpu_kernel!(a,x,y)
)
```

Listing 17: The Julia kernel GPU implementation of Saxpy

In this case, the kernel is defined manually. The index of each thread is calculated in a similar fashion to C and Numba. The `@cuda` macro is used to execute the kernel function and the `@inbounds` macro is used to disable bounds checking. This allows to create more complex custom kernels and manually set the thread hierarchy.

3.1.4 Apple Silicon Chip

In 2020, Apple started producing their own set of chips for their working stations, switching from the Intel $x86_64$ architecture to Apple Silicon ARM. The M1 chip, the first of this generation, has introduced a unified memory architecture, presenting a new approach to memory management that facilitates faster computation. Moreover, the CPU and GPU are integrated into the same chip.

The Unified Memory Architecture enables the CPU and GPU to share the same physical memory system (Figure 27). The paradigm was that the CPU and GPU have their own separate memory, requiring data transfer between them, which added time transfer delays. On M1 devices, the CPU and GPU cores can read and write to the same memory space, eliminating the need for synchronization protocols and data copying between distinct CPU and GPU memory banks (Apple Developer Documentation 2023). This results in significant performance increases when dealing with applications that use Heterogeneous computing. From a coding perspective, managing the data transfers between the CPU and GPU can be complicated. Using the unified memory architecture, data management becomes easier and reduces the complexity of the code.

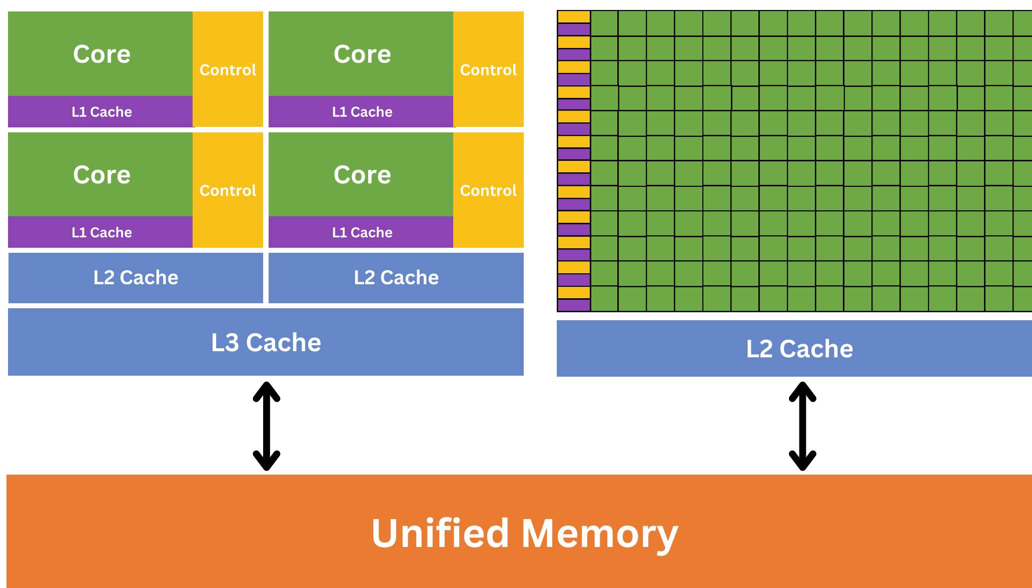


Figure 27: The Unified memory architecture found on all Apple M-series devices. The CPU and GPU are integrated in the same chip.

In order to utilize the capabilities of the Apple silicon chip, the Metal Shading Language (based on C++), also created by Apple, is required. But using it to develop scripts can be quite cumbersome. By using Julia and the Metal.jl module, users are able to utilize Apple's M1 GPUs capabilities by providing an interface to the Metal framework. The package is based on the same foundations as CUDA.jl and it uses the same syntax and principles (Julia Metal 2023).

Just like CUDA.jl, Metal.jl provides both high and low level abstractions when implementing the Saxpy function. In the vectorized version, we define the arrays using the `MtlArrays` function (Listing 18). It is important to note that here the arrays are stored in the unified memory and the `MtlArrays` annotation only instructs that the operations involving them are to be executed on the GPU.

```
using BenchmarkTools
using Metal

function saxpy(a,x,y)
    y .= a .* x .+ y
end

x = MtlArray(rand(Float32, size))
y = MtlArray(rand(Float32, size))

@benchmark saxpy(a,x,y)
```

Listing 18: The Julia Metal vectorized GPU implementation of Saxpy

The low-level approach permits greater control over the execution and optimization of computations by creating the kernel manually and setting the threading hierarchy. As shown in Listing 19, the @metal macro is used to execute the kernel function. In this package, the synchronization is done automatically.

```
using BenchmarkTools
using Metal

function saxpy_kernel(a,x,y)
    i = thread_position_in_grid_1d()
    if i <= length(x)
      @inbounds y[i] = a * x[i] + y[i]
    end
    return nothing
  end

x = MtlArray(rand(Float32, size))
y = MtlArray(rand(Float32, size))

threads = 1024
groups = cld(size, threads)

@benchmark @metal threads=threads groups=groups saxpy_kernel(a,x,y)
```

Listing 19: The Julia Metal kernel GPU implementation of Saxpy

3.2 Viscoacoustic Finite Difference Modelling

In the last subchapter, we examined several Saxpy implementation strategies, as it is easy to showcase the basis of each method with just a few lines of code. Now, we will be concentrating on incorporating these techniques into the Viscoacoustic finite difference modeling script. As the implementation of this is quite extensive and it spans over several files, we will not showcase it in detail but rather describe the main structure of it. At their core, the computation heavy parts can be implemented using the same principles as shown with Saxpy, but with an added layer of complexity.

In order to have a more detailed overview of how each method is implemented, all the source code for both the Saxpy and Viscoacoustic modeling can be found in the GitHub repository attached (see Appendix). Pertaining to the finite difference modeling script, the Python_CPU_For_Loop folder also contains a detailed explanation of the code.

All the 2D viscoacoustic libraries, whether of the programming language used or CPU/GPU implementation, have a *Main* file, where all the modeling parameters are set (Velocity, Density, Q-model, absorbing boundaries, relaxation times, etc.), the source wavelet/receivers and their coordinates are defined and the Solver, where all the computations are taking place, is called. The general execution flow in the Main file is shown in Figure 28.

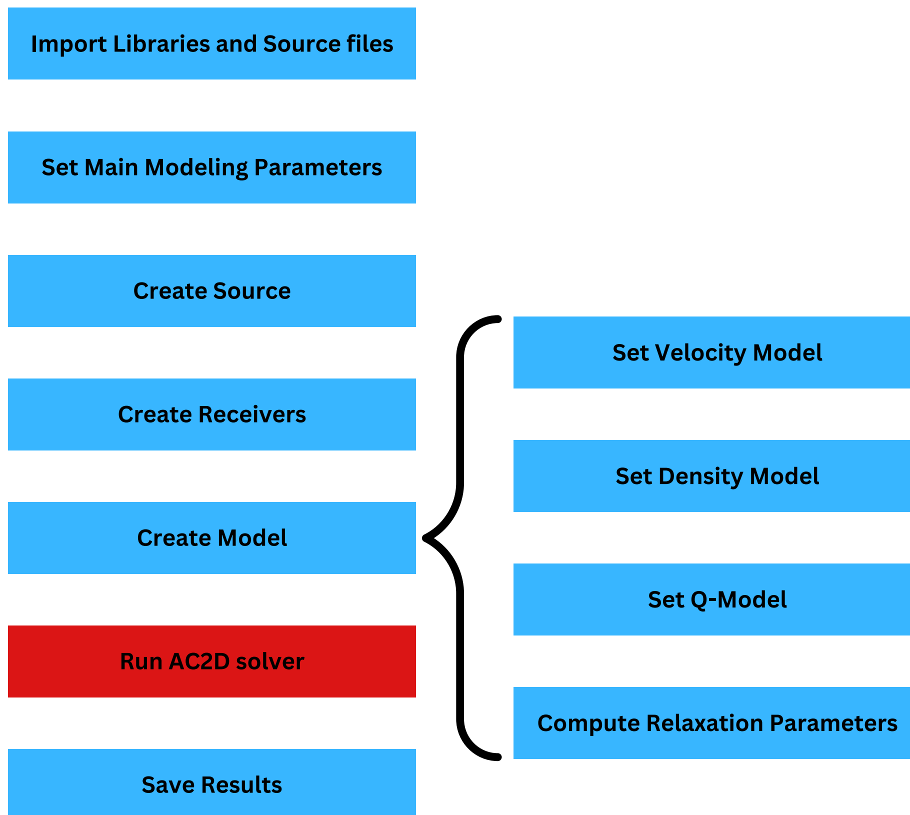


Figure 28: The execution flow of the Main file. The computationally heavy areas are denoted with Red, while the less demanding code blocks are blue.

This strategy promotes a modular approach and code reuse, making the code simpler to manage, debug and implement using the different optimization techniques. The source files that are imported include methods for: the source, receiver, model, AC2D solver, differentiator and visualization. The Model module computes the appropriate relaxation parameters based on the chosen viscoelastic mechanism (Standard Linear Solid or Maxwell).

In Python, in order to encapsulate the data and functions related to these items we use classes. In the C implementations, we use structs, which are user-defined data types that enable us to store various data types together. This is especially beneficial when we want to combine variables that are related. For instance, a struct may comprise all the variables required for the receivers. Even though you cannot explicitly associate functions with structs, the functions themselves can accept structs as parameters. In Julia, mutable structs are used, which enable us to modify their fields.

From the showcased execution flow, the most important code block is represented by the AC2D solver, where the finite difference computations based on the equations defined in 97 are executed.

The general flow is shown in Figure 29.

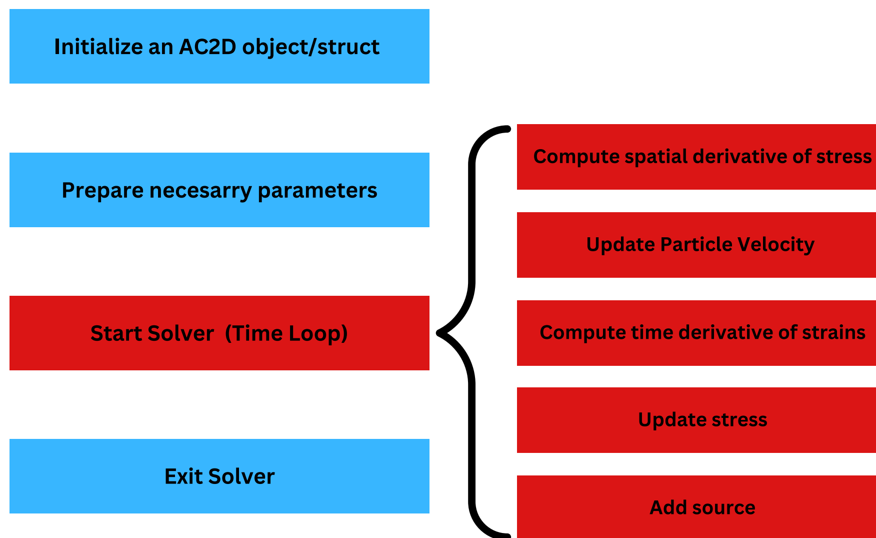


Figure 29: The execution flow of the AC2D file. The computationally heavy areas are denoted with Red, while the less demanding code blocks is blue.

In the figure, the computationally intensive portions are highlighted in red. Thus, it is here where the optimization methods can be implemented most effectively. It can be seen that the AC2D solver, which conducts the finite difference computations, is where the majority of these intensive computations take place. This function accounts almost for the entirety of the execution time of the simulation. Thus, the different optimization techniques will be implemented on the functions that are called in the solver.

An example of how one of these kernels might look like is shown in Listing 20, where the particle velocity computation in the x directions using the CUDA.jl module is implemented. All other case scenarios are executed in a similar fashion (Note: for all the GPU implementations, a 2D Grid and Block size were chosen).

```
function Ac2dvx_kernel!(vx_gpu, exx_gpu, thetax_gpu, rho_gpu, drhox_gpu,
↪  eta1x_gpu, eta2x_gpu, dt, nx, ny)
    x = threadIdx().x + blockDim().x * (blockIdx().x - 1)
    y = threadIdx().y + blockDim().y * (blockIdx().y - 1)

    if x <= nx && y <= ny
        @inbounds vx_gpu[x, y] = dt * (1.0 / rho_gpu[x, y]) * exx_gpu[x, y] +
        ↪  vx_gpu[x, y] + dt * thetax_gpu[x, y] * drhox_gpu[x, y]
        @inbounds thetax_gpu[x, y] = eta1x_gpu[x, y] * thetax_gpu[x, y] +
        ↪  eta2x_gpu[x, y] * exx_gpu[x, y]
    end
    return
end
```

Listing 20: The Ac2dvx GPU kernel implemented in Julia; Here we use a 2D grid and block.

4 Results

In the first section of this chapter, we will show the effectiveness of the absorbing boundary conditions implemented using viscoelastic kernels and discuss both the SLS and Maxwell models. Then, in the second part, the benchmark results of the different code implementations for viscoacoustic modeling are presented.

4.1 Viscoacoustic Modelling

We will show that by using the discussed finite-difference implementation, we can simulate an acoustic/viscoacoustic medium and implement absorbing boundary conditions using the same set of equations. The edge artifacts generated by this method will then be then analyzed. For all cases, when we refer to the Standard Linear Solid and Maxwell mechanisms, we assume that the density is time-dependent.

4.1.1 Comparison with the Analytical Solution

In order to showcase that the Viscoacoustic equations given in (87) are reduced to the acoustic case when we choose a sufficiently large Quality factor for the model, we can compare the results from the simulation with the analytical solution of the acoustic case for a homogeneous isotropic medium given in equation 36.

In this test, a 101x101 square mesh with a grid spacing interval of $10m$ was used, giving a subsurface model size of $1km$ by $1km$. The time sampling interval is $0.5ms$ and the total number of time steps is 550 (giving a recording length of $0.275s$). A differentiator length (radius) of 8 was chosen.

The model is homogeneous, with a P wave velocity of $2000m/s$ and a density of $2000kg/m^3$. The Quality factor is set at 10^5 and the Standard Linear Solid mechanism is chosen. The source pulse is a Ricker Wavelet with a dominant frequency of $25Hz$. The absorbing boundaries are disregarded in this case, as the modeling parameters were chosen in such a way that the wavefield does not reach the boundaries of the model.

The source is placed in the middle of the model ($x = 500m$, $z = 500m$) and we record the wavefield with a receiver placed at ($x = 650$, $z = 500m$). The result of the comparison between the analytical solution and the simulated solution is shown in Figure 30. It can be seen that by using a large Quality factor, the relaxation terms in the viscoacoustic equation are canceled out and we are left with the acoustic case.

If we want to simulate a more realistic attenuation, then we just have to set the according Quality factor for different types of lithologies ($10 \leq Q \leq 200$). In Figure 31, we compare the analytical solution to a numerical simulation where the Quality factor was set to 1.5 (this is a very low value that was chosen just to exaggerate the attenuation of the wavefield for visualization purposes). As we will show later, the Standard Linear Solid mechanism is the more appropriate model to simulate realistic wave attenuation as it is frequency dependent (higher frequencies are more attenuated).

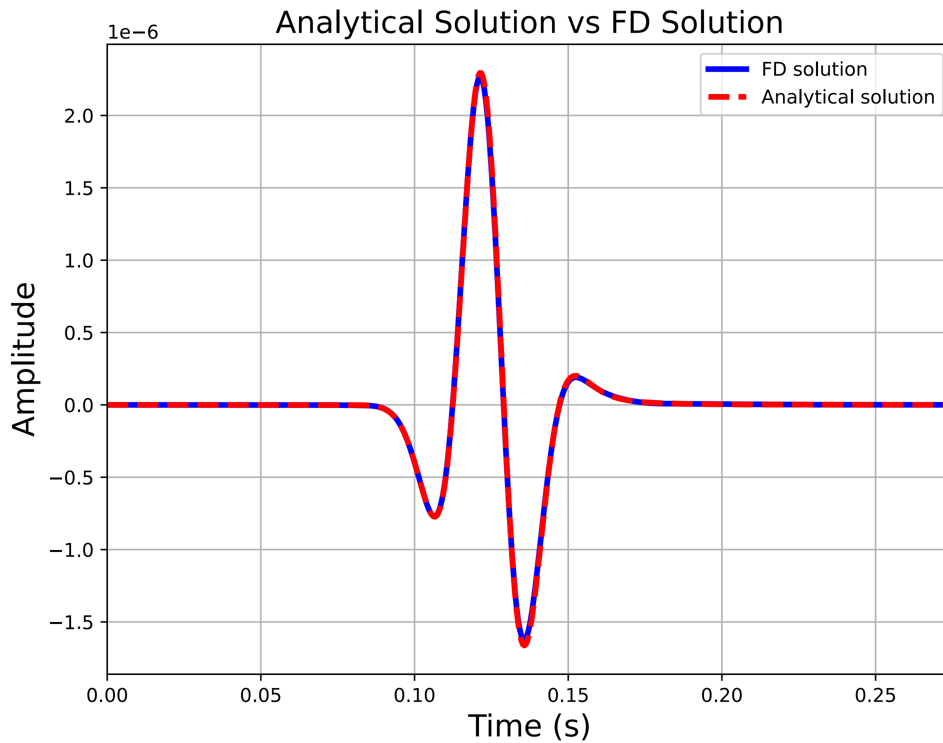


Figure 30: Comparison of the Acoustic analytical solution (red) with the simulated Viscoacoustic solution (blue). It can be seen how these two overlap, thus proving that the Viscoacoustic solution is reduced to the Acoustic case when a large Quality factor is chosen.

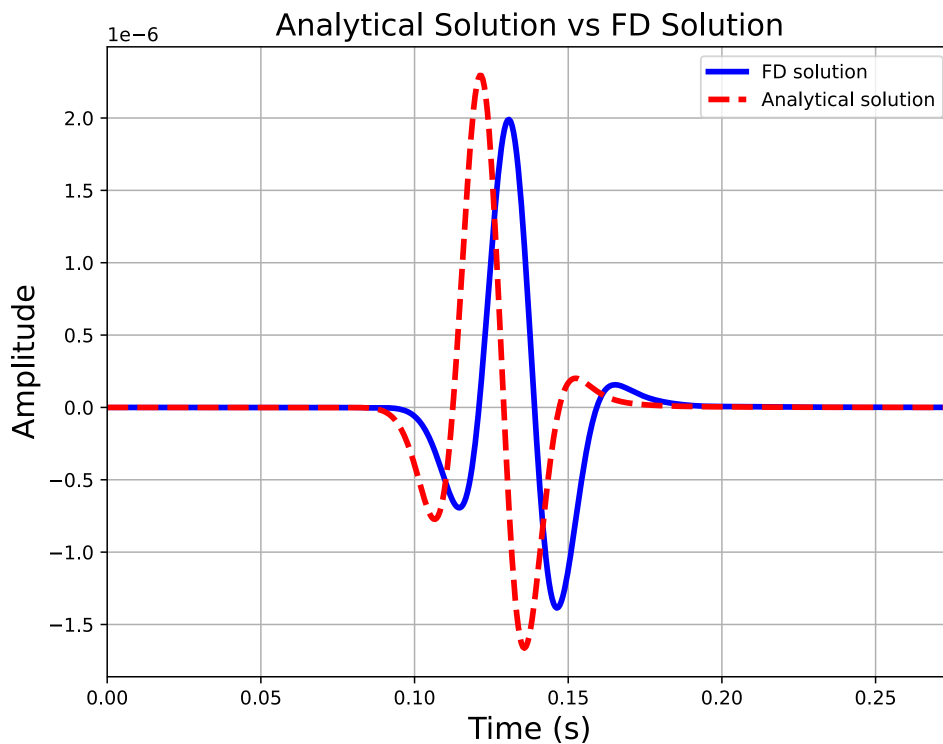


Figure 31: Comparison of the Acoustic analytical solution (red) with the simulated Viscoacoustic solution (blue). It can be seen how the numerical solution is attenuated when choosing a small Quality factor ($Q = 1.5$).

4.1.2 Absorbing Boundaries

The absorbing boundary conditions can be implemented just by tapering the Q-model at the model edges, as illustrated in Figures 17 and 18. Thus, the same viscoacoustic equations can be used both inside the model itself and in the absorbing boundaries.

Homogeneous Model

The results for the Standard Linear Solid mechanism with a time-dependent density (equivalent to C-PML) for a homogeneous model are presented in Figure 32. A similar result is obtained if the Maxwell model is chosen. It can be seen that the wavefield is highly damped is it enters the absorbing boundary area until it is completely dissipated. In this case scenario, the absorbing boundary has a width of 20 grid points and completely envelops the model.

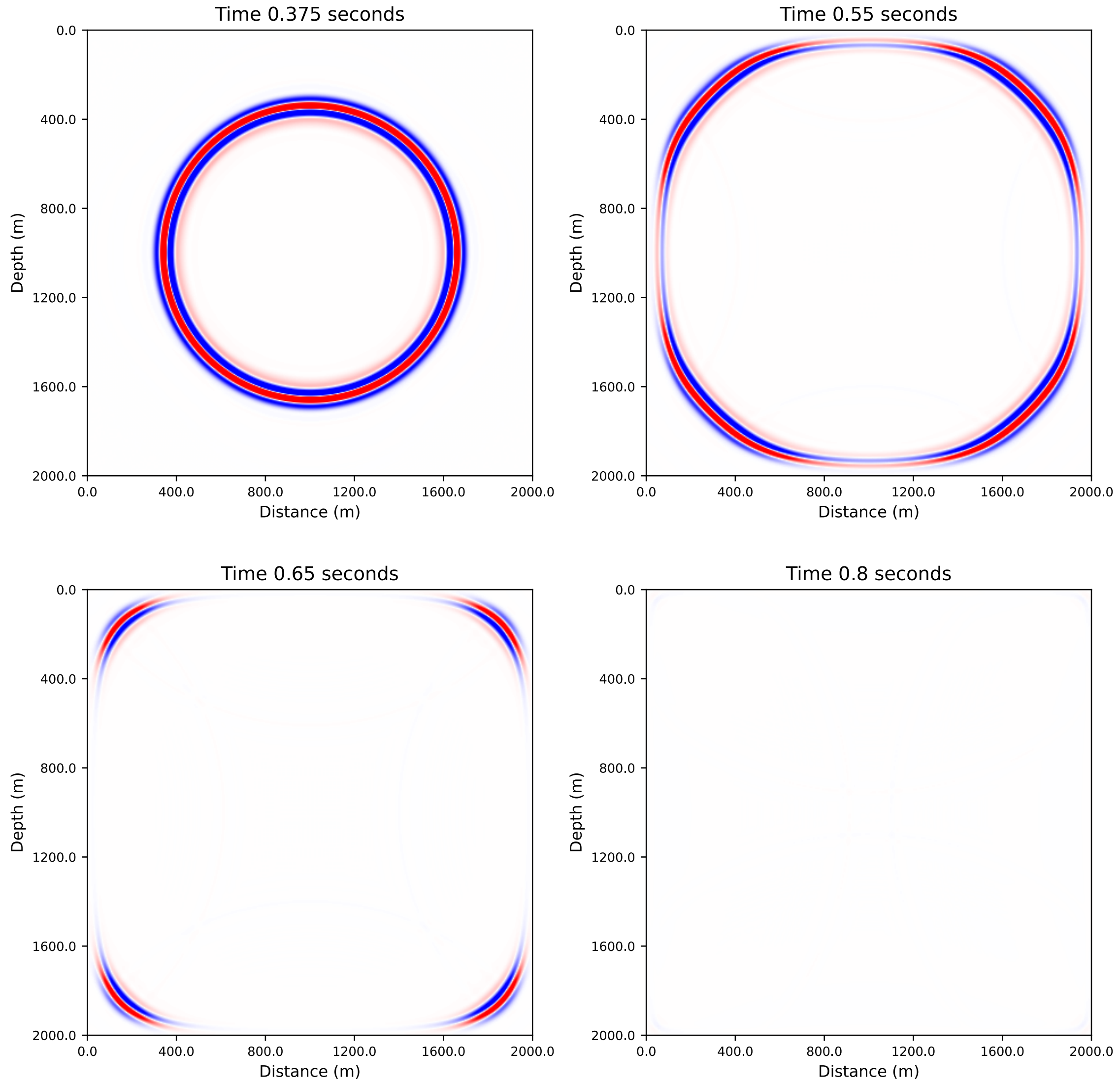


Figure 32: Snapshots from the Viscoacoustic simulation using the SLS model. Parameters: 201x201 square mesh, $dx = 10m$, $dt = 0.5ms$, $Nt = 2000$ (giving a total simulation time of $1s$). The differentiator length is 8. The model is homogeneous, with a P wave velocity of $2000m/s$, a density of $2000kg/m^3$ and a quality factor of 10^5. The Q model is tapered at the borders, reaching a Qmin value of 1.1. The source pulse is a Ricker Wavelet with a dominant frequency of $25Hz$, which is placed in the middle of the model ($x = 1000m$, $z = 1000m$). $Nb = 20$ (Number of grid points for the absorbing boundary).

Layered Model

The absorbing boundaries also perform well in heterogeneous cases. Figure 33 depicts the response for a layered model using the Maxwell mechanism. The modeling parameters are the same as with the homogeneous case, but we introduce 3 layers that have P wave velocities of 2000, 2500, and $3000 m/s$, respectively. Each layer also corresponds to a respective density of 2000, 2500, and $3000 kg/m^3$. The velocities and densities increase with depth. The source is positioned in the middle of the first layer. In this case scenario, the absorbing boundary has a width of 30 grid points and completely envelops the model.

The results clearly show that the wavefield is completely absorbed once it reaches the boundary area.

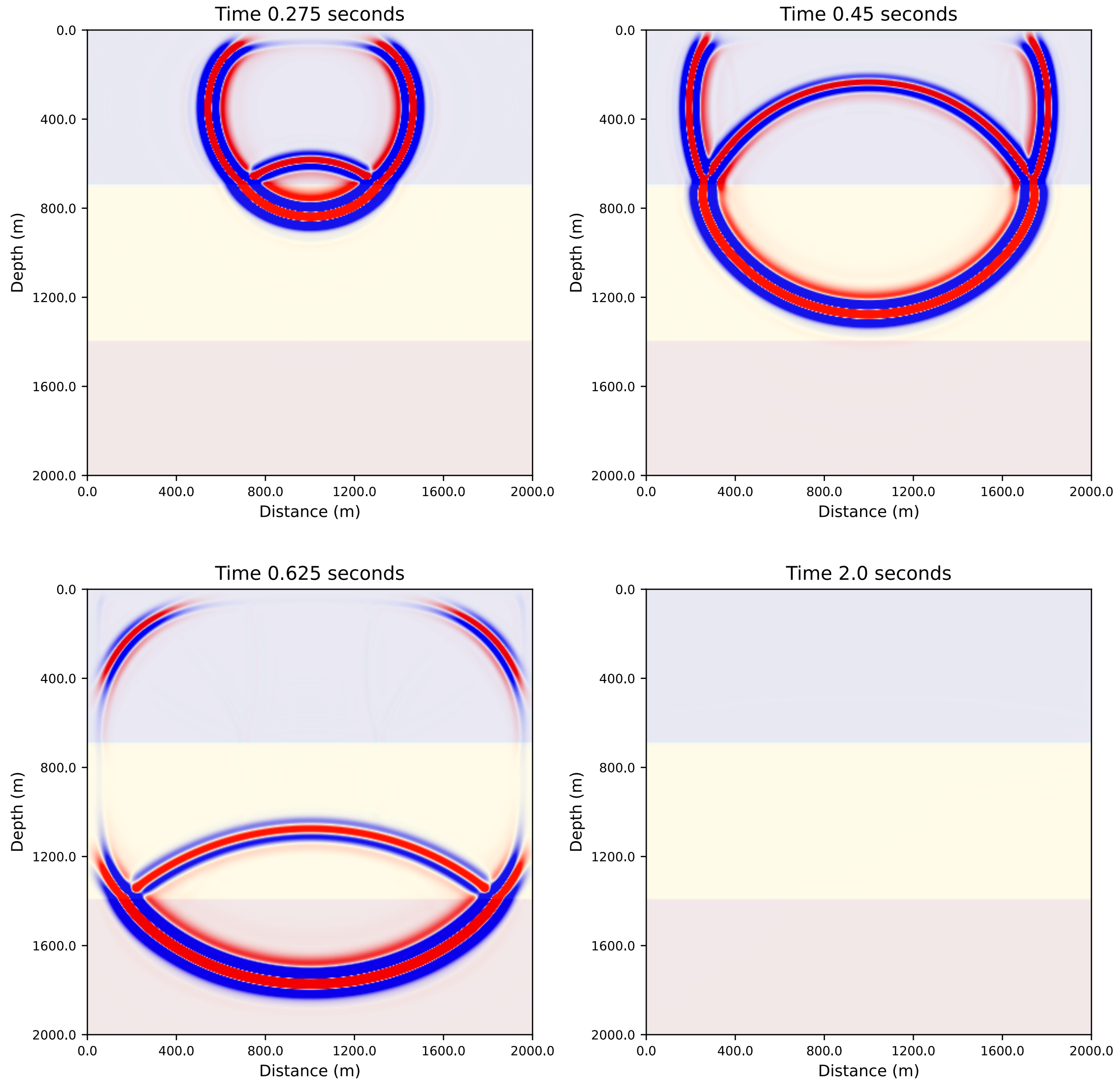


Figure 33: Snapshots from the Viscoacoustic simulation using the Maxwell model. Parameters: 201x201 square mesh, $dx = 10m$, $dt = 0.5ms$, $Nt = 4000$ (giving a total simulation time of $2s$). The differentiator length is 8. The three layered model has an increasing velocity/density with depth. The quality factor is 10^5. The Q model is tapered at the borders, reaching a Qmin value of 1.1. The source pulse is a Ricker Wavelet with a dominant frequency of $25Hz$, which is placed in the middle of the first layer ($x = 1000m$, $z = 350m$). $Nb = 30$ (Number of grid points for the absorbing boundary).

Marmousi Model

In this section, we present the results of applying our method to a highly complex subsurface model. For this case scenario, we consider the Marmousi model. The top boundary is considered perfectly reflective, and the Maxwell mechanism is utilized. Also, we place the source in the middle of the model just below the water-air interface.

Figure 34 shows the Marmousi velocity and density models utilized. The model is rectangular (500x174) with a grid spacing interval of 20m.

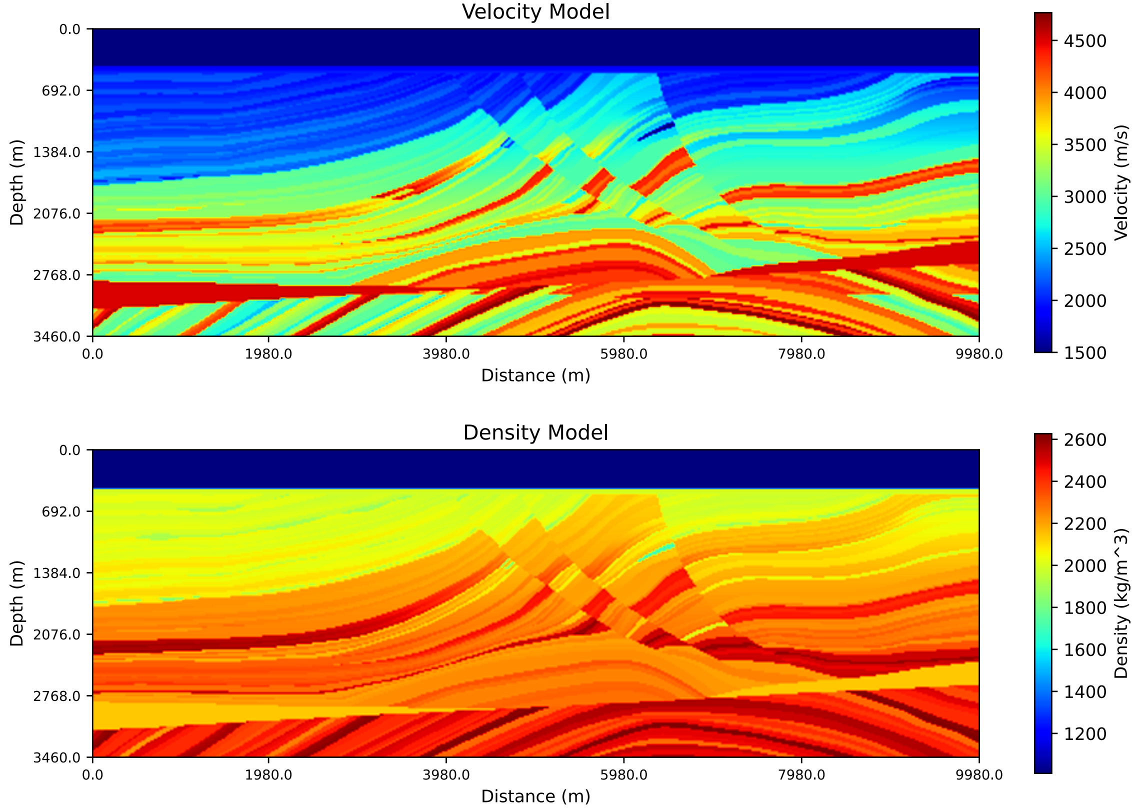


Figure 34: Marmousi velocity model (top) and density model (bottom). The mesh is 500x174 with $dx = 20m$.

For the simulation, we use a time sampling interval of $0.5ms$ and a total simulation time of $4s$ ($Nt = 8000$). The differentiator length (radius) is 8. The Quality factor is set at 10^5. The source pulse is a Ricker Wavelet with a dominant frequency of $10Hz$. The absorbing boundary width is 30. Several snapshots of the simulated wavefield are shown in Figure 35.

The corresponding seismogram is shown in Figure 36. The receivers are placed just below the topside of the model (below the water surface) and cover the whole length of the model.

From the results, it can be seen that as the wavefield enters the absorbing edges, it is properly damped, regardless of the incidence angle.

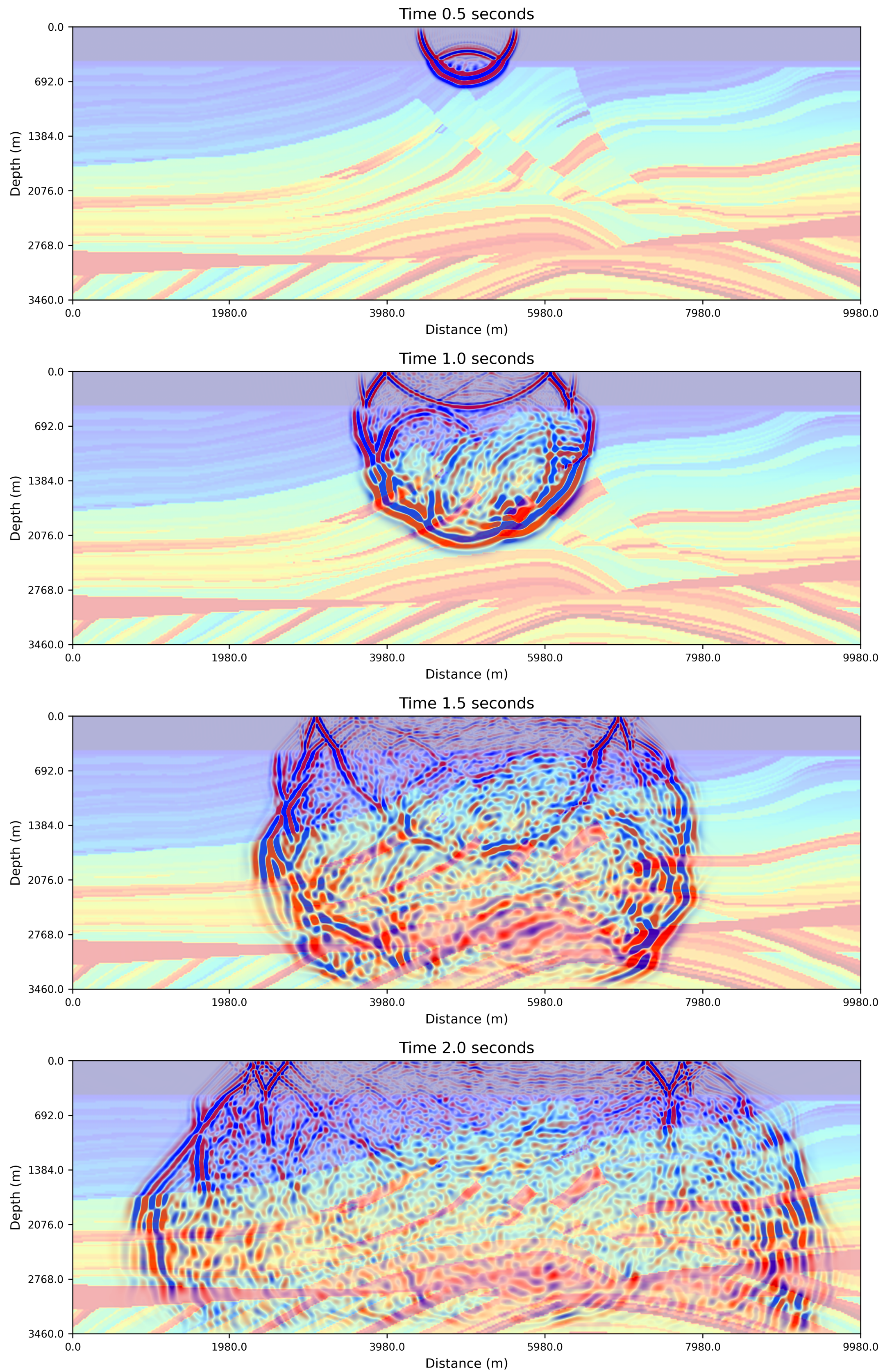


Figure 35: Snapshots from the Viscoacoustic simulation using the Marmousi model. Parameters: 500x174 mesh, $dx = 20m$, $dt = 0.5ms$, $Nt = 8000$ (giving a total simulation time of $4s$). The differentiator length is 8. The quality factor is 10^5. The Q model is tapered at the borders, reaching a Qmin value of 1.1. The source pulse is a Ricker Wavelet with a dominant frequency of $10Hz$. $Nb = 30$.

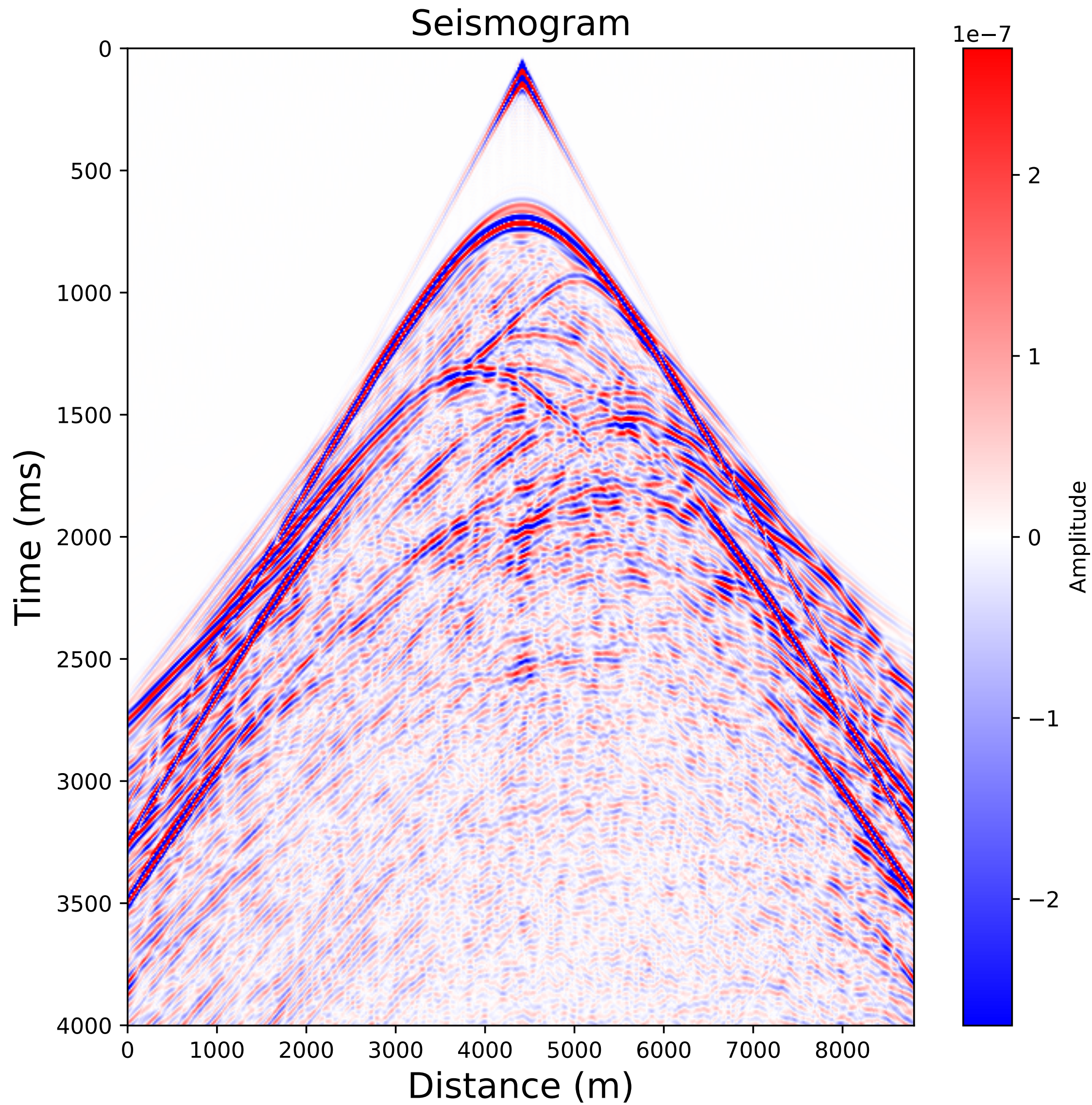


Figure 36: The Seismogram obtained from the simulation by placing a line of hydrophones just below the water-air interface. Here, the absorbing boundary area is cut from the seismogram

4.1.3 Standard Linear Solid vs Maxwell

Here, we will exemplify the effectiveness of the absorbing boundaries by employing and analyzing the Standard Linear Solid and Maxwell mechanisms.

Model Parameters

For all cases, we define a 401x401 mesh with a grid interval of $10m$. The time stepping interval is $0.5ms$ and the number of time steps is 3500 ($1.75s$ simulation time). The differentiator length is 8. The absorbing layer has a width of 30 grid points and it encapsulates the model. The model is homogeneous with a P wave velocity of $2000m/s$ and a density of $2000kg/m^3$. The quality factor is 10^5. We place a receiver at $(x = 3500m, y = 2000m)$.

Since the edge artifacts resulting from this method are small, we will show an amplified version of the results (both the trace recording and the wavefield have a zoom factor of around 100 compared

to the results shown in the previous sections).

We have shown that a SLS mechanism with a time-dependent density is equivalent to the C-PML boundary conditions. The amplified error resulting from this mechanism using a $15Hz$ Ricker Wavelet as a source is shown in Figure 37.

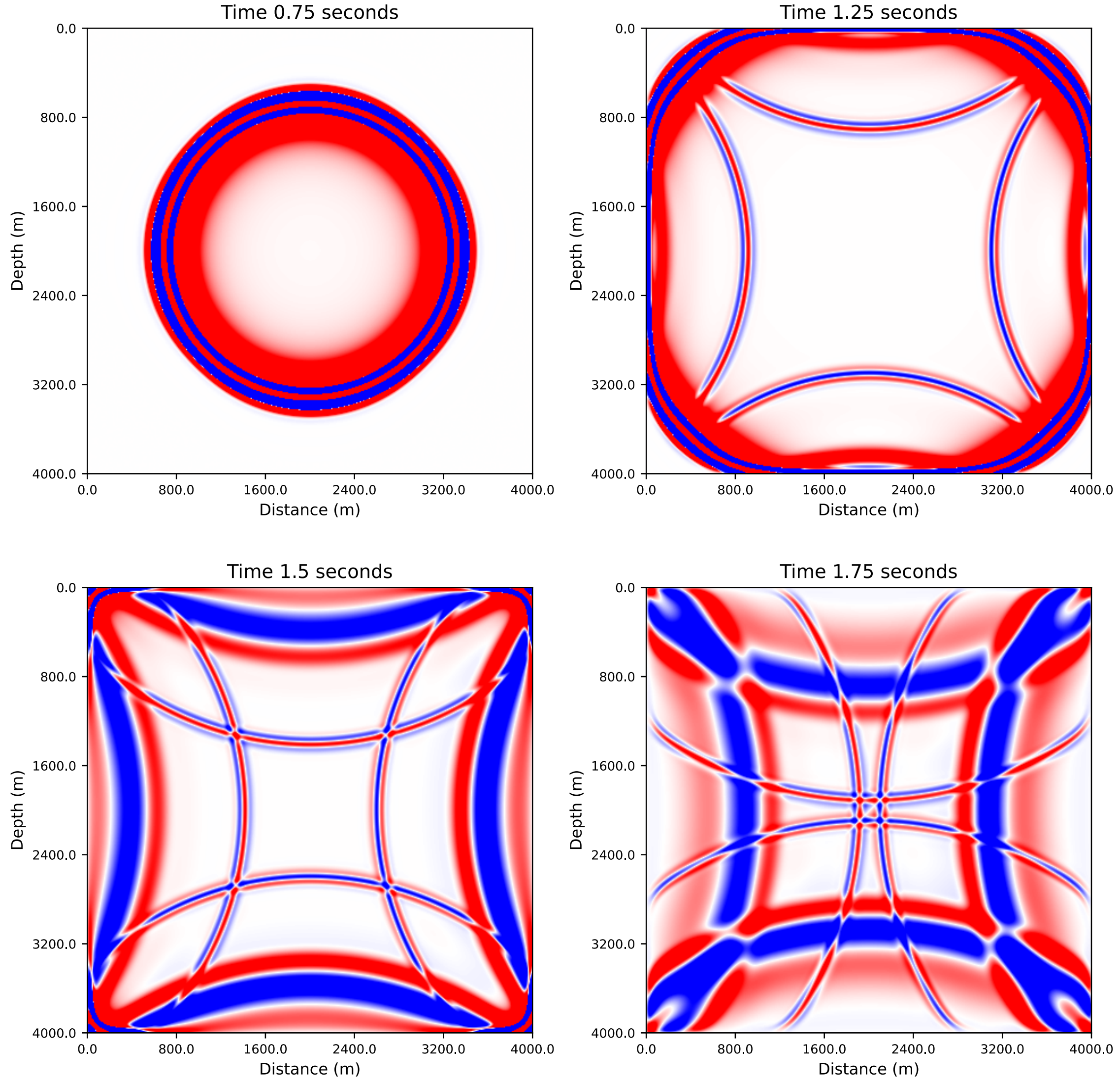


Figure 37: Edge Artifacts from the SLS mechanism. The source is a Ricker wavelet with a dominant frequency of $15Hz$. The absorbing boundary width is 30 grid points.

From the figure, we can distinguish two main types of edge artifacts. The first one is a reflection that appears as the wavefield enters the absorbing boundary. This can be seen clearly in the $1.25s$ snapshot. The second one is a reflection that appears as the wavefield hits the outer boundary of the absorbing layer. At the $1.5s$ snapshot, it can be seen how this second reflection develops.

In the following, we will show the behavior of these two errors when choosing different dominant frequencies for the source signal and different viscoacoustic kernels.

In Figure 38, the different errors are plotted for both the Standard Linear Solid and Maxwell mechanisms for a dominant frequency of 15Hz and 25Hz, respectively.

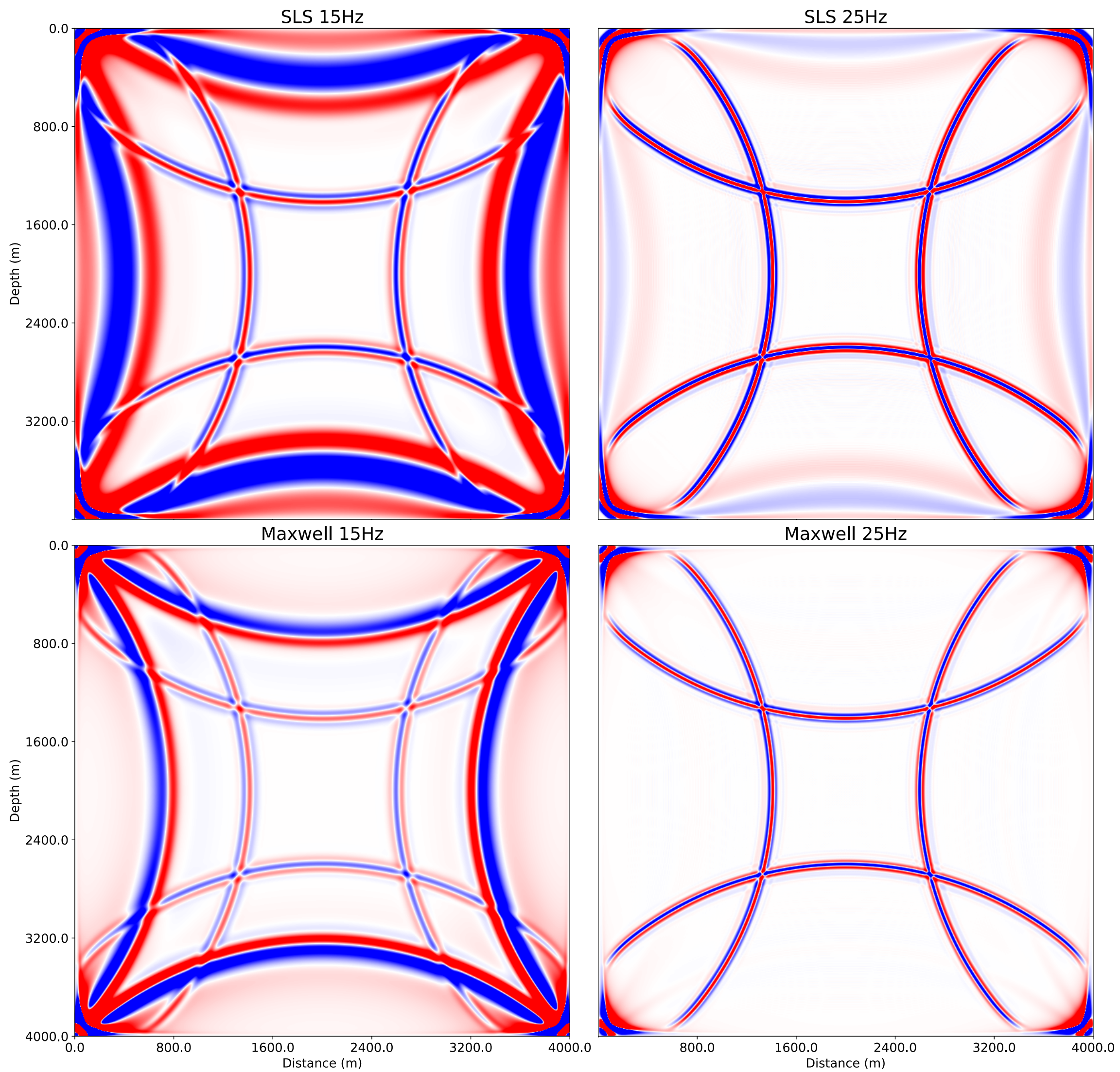


Figure 38: The reflection artifacts for the Standard Linear Solid mechanism (top) and Maxwell mechanism (bottom). For each case, we show the results for a dominant frequency of $15Hz$ (left) and $25Hz$ (right).

Firstly, from these results, we can say that the Maxwell model appears to perform better. In the $25Hz$ case, the second reflection, which comes from the outer edge of the absorbing region, is completely damped, whereas we can still see it in the SLS case. Overall, the Maxwell mechanism seems to provide lower amplitude reflections than its counterpart. Secondly, the frequency of the outer reflection produced by the Maxwell mechanism seems to be proportional to the original dominant frequency of the wavefield. Whereas, for the SLS mechanism, it can be seen that the reflection has a lower frequency than the original wavelet.

This behavior can be analyzed in more detail by looking at the trace recordings (Figure 39 and 40).

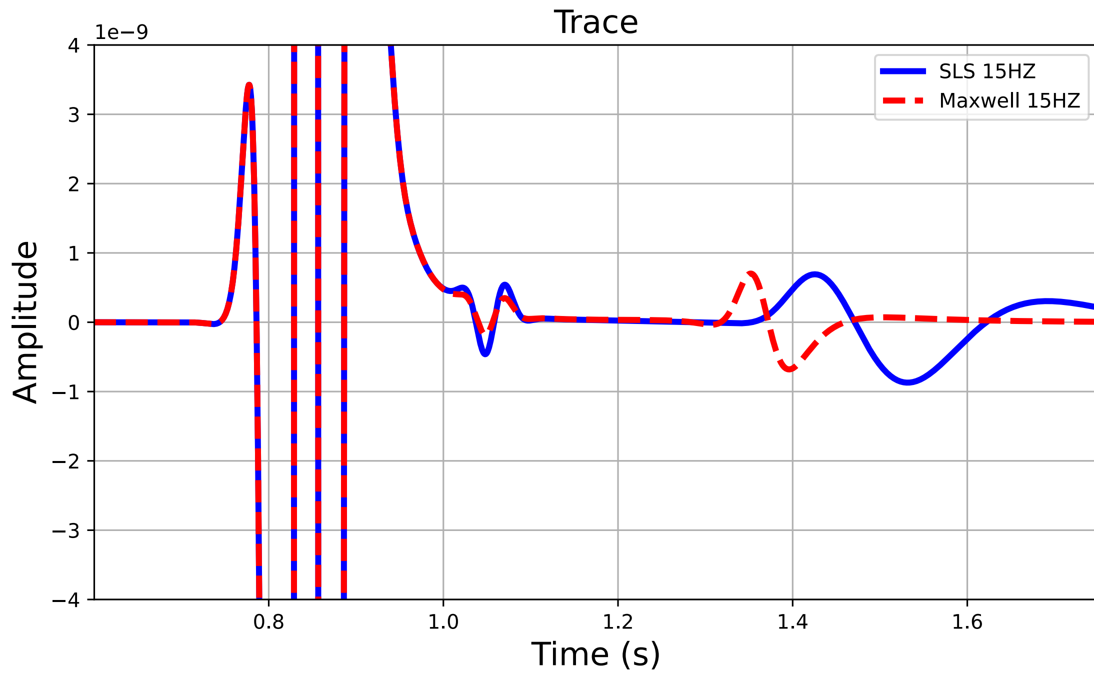


Figure 39: Trace comparison between the Maxwell and SLS mechanisms for a wavelet with a dominant frequency of 15Hz.

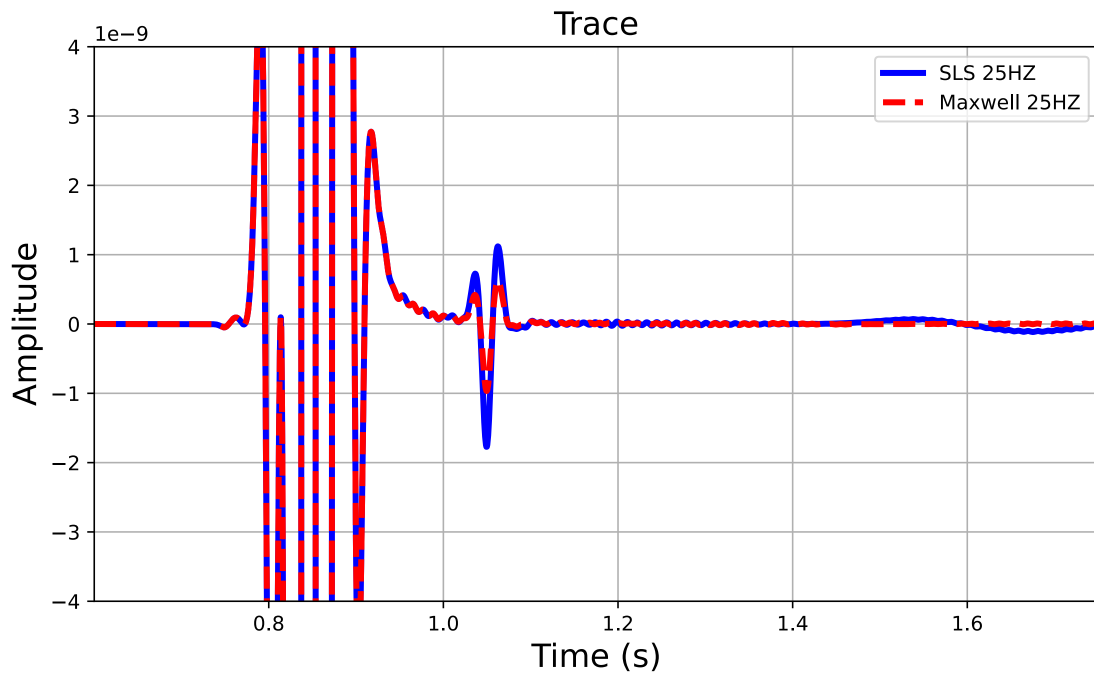


Figure 40: Trace comparison between the Maxwell and SLS mechanisms for a wavelet with a dominant frequency of 25Hz

Reflections due to abrupt change in viscosity

Again, we can see the reflections caused by the abrupt change in the viscoacoustic parameters as the wavefield enters the absorbing region, just after the $1s$ mark (for every case). This reflection is proportional to the dominant frequency of the original wavelet for both mechanisms. For the Maxwell model it has a lower amplitude.

This phenomenon is pointed out by Borcherdt 1982, which states that in anelastic media, abrupt variations in relaxation times can result in the emergence of reflected and refracted waves. In order to minimize the reflections caused by the change in viscosity, the taper function for the absorbing region could be modified in a way that allows for a smoother transition.

Reflections due to insufficient damping

From the figures, we can see that both mechanisms perform better when the dominant frequency of the source wavelet is higher. In the $25Hz$ case, the Maxwell mechanism manages to completely absorb the wavefield (no reflection observed after the $1.4s$ timestamp), while the SLS mechanism shows a very low frequency reflection with a relatively small amplitude. In the $15Hz$ case, both the Maxwell and SLS mechanisms are not able to completely absorb the wavefield. Again, the SLS reflection has a lower frequency than the original wavelet, while the Maxwell model gives a reflection with a frequency proportional to the original wavelet. Overall, the SLS error is higher than the Maxwell one.

In the low frequency case, the damping of the wavefield in both cases could be improved by increasing the width of the absorbing region (e.g. increase from 30 to 50 grid points).

From these results, we can say that the Maxwell mechanism appears to dampen all frequencies of the wavefield in the same manner, while the SLS mechanism shows a frequency dependent absorption behavior, as the higher frequencies are better damped while it struggles with the lower frequency components.

Plane Wave

In order to analyze this frequency dependent absorption behavior, we will make use of a plane wave. According to Ursin 1983, a downgoing wave in a homogeneous medium that has a unit amplitude is given by:

$$\exp(-ik_z|z|) \tag{115}$$

This equation describes how the amplitude of a wave changes as it travels through a medium, taking into consideration the depth/distance from the source ($|z|$) and the vertical wavenumber (k_z).

The vertical wavenumber (k_z) has the following expression:

$$k_z = \frac{\omega}{c} cos(\theta) \tag{116}$$

where θ represents the angle between the z-axis and the wave propagation direction.

For a viscoelastic medium, the bulk modulus and density can be written in the frequency domain as:

$$\begin{aligned} K(\omega) &= K_0 G(\omega), \\ \rho(\omega) &= \rho_0 G^-1(\omega). \end{aligned} \tag{117}$$

where G is the complex modulus. Thus, the wave velocity (c) can be written as:

$$c = \sqrt{\frac{K(\omega)}{\rho(\omega)}} = \sqrt{\frac{K_0}{\rho_0}} G(\omega) \tag{118}$$

Standard Linear Solid

For the SLS medium, G is given by Carcione and Casula 1992 as:

$$G(\omega) = \frac{1 + i\omega\tau_\epsilon}{1 + i\omega\tau_\sigma} \tag{119}$$

The wave velocity is then calculated as:

$$c = \frac{1 + i\omega\tau_\epsilon}{1 + i\omega\tau_\sigma} \sqrt{\frac{K_0}{\rho_0}} \tag{120}$$

For normal incidence, the vertical wavenumber k_z becomes:

$$k_z = \frac{\omega}{c_0} \frac{1 + i\omega\tau_\epsilon}{1 + i\omega\tau_\sigma} \tag{121}$$

Finally, the wave in the SLS medium can be described by the expression:

$$\exp\left[iz\left(\frac{\omega}{c_0}\right)\frac{1 + i\omega\tau_\epsilon}{1 + i\omega\tau_\sigma}\right] \tag{122}$$

which can be separated into real and imaginary parts:

$$\exp\left[iz\left(\frac{\omega}{c_0}\right)\frac{1 + \omega^2\tau_\epsilon\tau_\sigma}{1 + \omega^2\tau_\sigma^2}\right] \exp\left[\left(\frac{-z}{c_0}\right)\frac{\omega^2(\tau_\epsilon - \tau_\sigma)}{1 + \omega^2\tau_\sigma^2}\right] \tag{123}$$

The second part of this expression (the real part) governs the wave attenuation (amplitude attenuation) for SLS (C-PML). The relaxation times τ_ϵ and τ_σ depend on the dominant frequency (f_0) and the Quality factor (Q). Also, in this case, the absorption is frequency dependent, as we have the angular frequency (ω).

Maxwell

For the Maxwell medium, G is given by Carcione and Casula 1992 as:

$$G(\omega) = \left(1 - \frac{i}{\omega\tau}\right)^{-1} \tag{124}$$

The wave velocity is then calculated as:

$$c = \left(1 - \frac{i}{\omega\tau}\right)^{-1} \sqrt{\frac{K_0}{\rho_0}} \tag{125}$$

For normal incidence, the vertical wavenumber k_z becomes:

$$k_z = \frac{\omega}{c_0} - \frac{i}{c_0\tau} \tag{126}$$

Finally, the wave in the Maxwell medium can be described by the expression:

$$\exp\left(\frac{-i\omega z}{c_0}\right)\exp\left(\frac{-z}{c_0\tau}\right) \tag{127}$$

The plane wave attenuation is given by the second part of this expression (the real part). τ depends on the dominant frequency (f_0) and the Quality factor (Q). In this case, the absorption is frequency independent.

Based on the above equations, the frequency versus attenuation responses for the Maxwell and SLS mechanisms is shown in Figure 41.

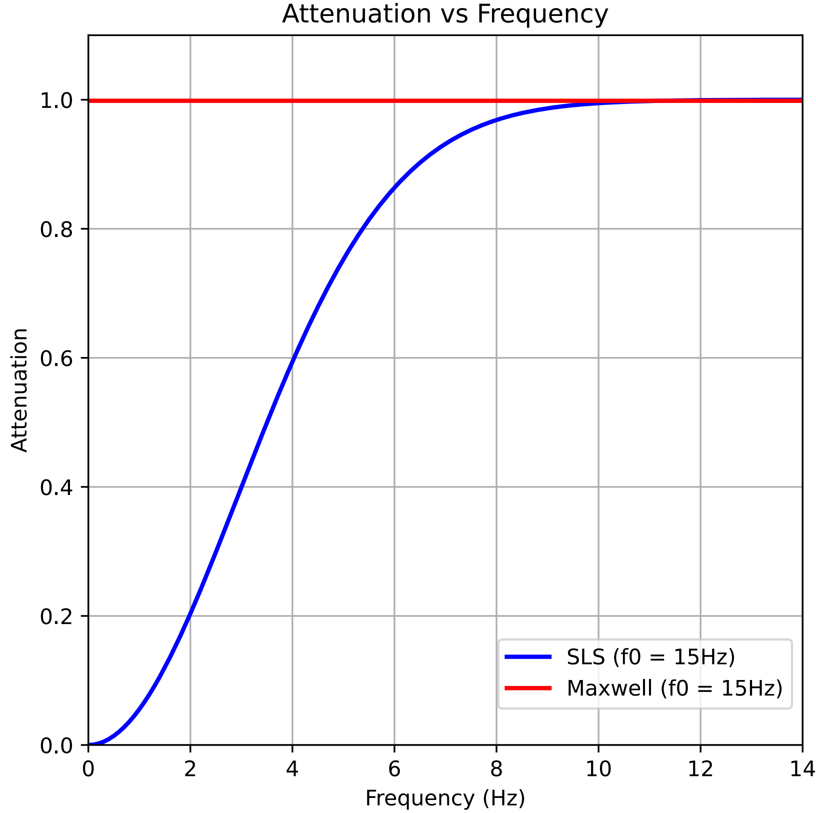


Figure 41: Frequency versus attenuation response for the Maxwell and SLS mechanisms. The dominant frequency of the propagating wavefield is $15Hz$ ($f_0 = 15Hz$). The Quality factor is 1.1, $c_0 = 2000m/s$ and $z = 150m$.

From the figure, it can be seen that if the absorbing boundary width is big enough, the Maxwell model manages to achieve complete attenuation for all frequencies, whereas the SLS model fails to dampen the lower frequencies. This behavior for the SLS mechanism is valid regardless of how much we increase the absorbing boundary width. This is the same behavior we had observed in our simulations.

4.2 Benchmarks

In this section, we will compare the performance of our different implementations for the finite-difference code. The code will be benchmarked in terms of runtime and relative speedup.

4.2.1 Performance Metrics

In order to test the performance of the different implementations, we will use the runtime and the relative speedup as metrics.

For each benchmark, we will compute the runtime for different model sizes: 100x100, 250x250, 500x500, 1000x1000, 2000x2000, 3000x3000, 4000x4000, 5000x5000, and 6000x6000. For all of them, we consider 1000 time steps, with $dt = 0.5ms$ and a simulation time of $0.5s$. The differentiator length is 8, $dx = 10m$, $vp = 2000m/s$, $\rho = 2000kg/m^3$, and $Q = 10^4$. The source is a Ricker wavelet with a central frequency of $25Hz$.

In the GPU implementation, we use a 2D grid containing 2D blocks. The block size used is 16x16.

For each test, the runtime is the arithmetic average of 3 different runs:

$$Runtime = \frac{1}{n}\sum_{i=1}^{n} t_i \tag{128}$$

where the runtime is measured in seconds and n is the total number of runs (3 in our case),

The speedup is measured as the ratio of a reference implementation time (in our case the Numba CPU serial implementation) to the time of the current tested implementation.

$$S = \frac{t_{\text{reference}}}{t_{\text{current}}} \tag{129}$$

It is essential to note that our measurements are limited to the computational time of the AC2D solver. In our calculations, we do not account for the time required for data preparation and saving the final results. In addition, for the Julia CUDA and Julia Metal M1 GPU implementation, we start the timing from the second time step iteration, as it is in the first iteration that the kernel functions are first accessed and compiled, thus adding significant latency.

4.2.2 Test Systems

All of the benchmarks, with the exception of one, are performed on a Dell Inc. Precision 7560 laptop. The operating system is a 64-bit ($x86_64$) Ubuntu 22.04.2 LTS Linux distribution. Only the Julia Metal M1 implementation (which is shown in Figure 45) will be carried on a 2020 MacBook Pro, which is the first laptop that uses the M1 chip. The hardware specifications of the laptops are shown in Table 2.

System	CPU	RAM	GPU
Dell Precision 7560	Intel Core i9-11950H 11th Gen @2.6GHz x 16	64 GB	Nvidia RTX A4000 Mobile
MacBook Pro 2020	Apple M1 8 cores	16GB	M1 integrated GPU 8 cores

Table 2: Hardware Characteristics

The Intel processor has 16 threads. The RAM memory operates at a frequency of 3200MHz. The Nvidia RTX A4000 mobile has 8 Gb of GDDR6 memory, a BUS width of 256 bit, 5120 CUDA cores, and operates at a base frequency of 1140 MHz.

For all the implementations that require C, we use the GCC 8.5.0 compiler with the following flags: -03, -ffast-math and -fopenmp (for the multithreaded code). For the Python implementation we use the 3.9.13 version and for Julia we use the 1.9.2 version.

The Nvidia CUDA GPU-based implementations have been compiled with the NVCC compiler, which is included in the CUDA software development toolkit. The NVCC compiles the CUDA host code using the GCC compiler of the system. The CUDA toolkit version is 12.0.

4.2.3 Saxpi

All the Saxpy benchmarks will not be showcased as they resemble the finite difference results. Thus, they can be found in the GitHub repository. Here, we only present Figure 42 that shows how the Julia CPU, Multithread, and GPU performance scales with the problem size.

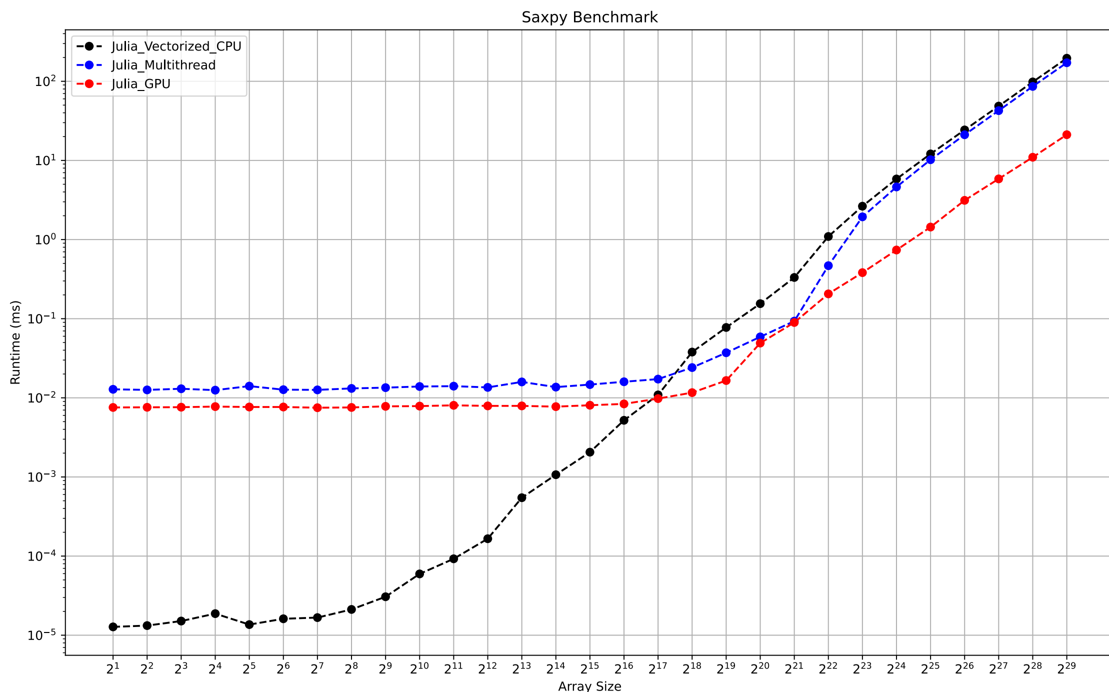


Figure 42: Julia serial CPU (black), Multithread (blue), and GPU (red) Saxpy implementations with varying problem sizes. The OY ans OX axis are logarithmic here. The runtime is given in milliseconds.

This visualization highlights an important observation: when considering the overall performance, GPU implementations emerge as the fastest methods, followed by Multithread and serial CPU implementations. However, it is essential to note that this trend holds true only for sufficiently large problem sizes. For smaller problems, the additional overhead of GPU and Multithread versions actually makes them slower compared to the CPU serial implementation. Consequently, when selecting the optimal approach for computing a task, careful consideration must be given to whether the problem size justifies the extra effort for GPU or Multiple cores optimization.

4.2.4 Viscoacoustic Modelling

The results presented in each of these tests are obtained from modeling a single shot using the viscoacoustic code. The parameters used for these models are detailed at the beginning of this chapter.

Figure 43 illustrates the runtimes associated with the serial CPU-based implementation.

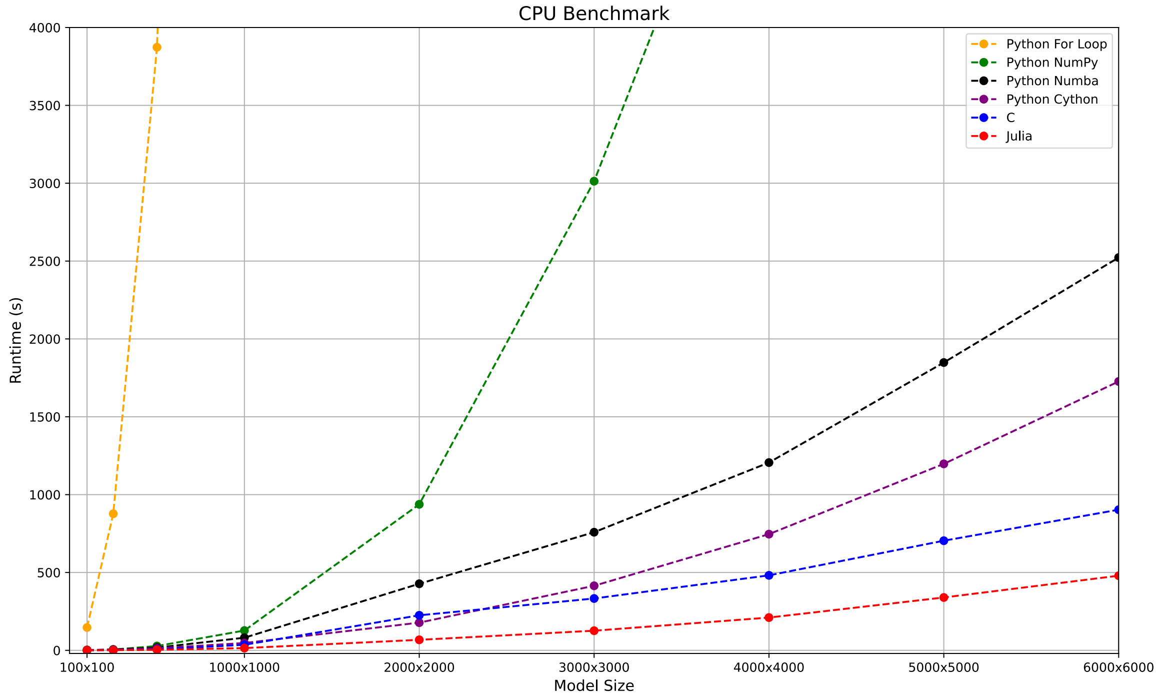


Figure 43: Serial CPU implementations for one shot using the viscoacoustic finite-difference method with varying problem size: Python For Loop (orange), NumPy (green), Numba (black), Cython (purple), C (blue) and Julia (red). The runtime is given in seconds.

There are several important observations that can be made:

1. The Python For Loop method, despite its simplicity, struggles with the complexity of the problem as the runtime rapidly escalates with the model size. The vectorized NumPy method performs adequately for smaller models, but its viability decreases as the problem size increases.
2. If the problem size is relatively small, most methods prove efficient, exhibiting relatively low runtimes.
3. Numba and Cython serve as viable alternatives to C, despite their slightly inferior performance, as they are more easy to implement and work with in comparison to C.
4. C has low runtimes and scales well with the model size, but it is also the method that requires the most lines of code and it poses more implementation challenges. The SWIG option, which was not included here, performs similarly to C, thus making it a good wrapper option.
5. Julia outperforms the other methods. Its implementation is relatively straightforward and similar to Python in certain aspects. The base version performs similarly to C. Here, we show the results from an optimized version that makes use of the Loop Vectorization package. The @turbo macro applied to each for loop enables automatic optimization, including loop unrolling, instruction-level parallelism, and efficient use of CPU cache. This macro attempts

to restructure the execution so that multiple iterations of the for loop can be performed at once using SIMD instructions.

Figure 44 presents the results of the multithreaded scripts. These indicate that all multithreaded implementations can achieve a speedup of 2-4 times compared to their serial CPU counterparts. Again, the C and Julia implementations deliver the best performance.

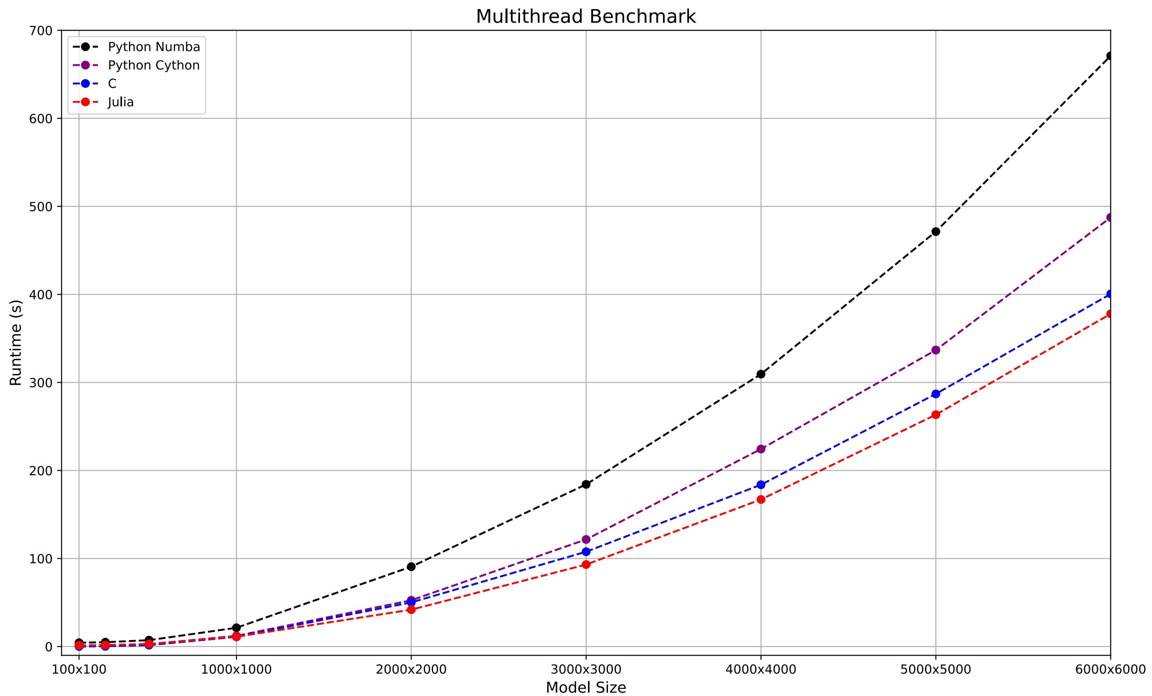


Figure 44: Multithread implementations for one shot using the viscoacoustic finite difference method with varying problem size: Numba (black), Cython (purple), C (blue) and Julia (red). The number of threads is set at 16. The runtime is given in seconds.

Finally, the GPU implementations are presented in Figure 45. These implementations exhibit a substantial reduction in runtimes, achieving speedups ranging from 10 to 20 times when compared to their serial CPU counterparts. The CUDA and Python Numba CuPy implementations demonstrate similar runtime trends as the model size increases. Once again, Julia is the fastest implementation.

Even though all of these methods rely on CUDA at their core, Julia outperforms the rest of them. This superior performance can be attributed to Julia's automatic optimization capabilities, which strive to optimize the given problem as efficiently as possible.

Another important aspect is that the Numba CuPy implementation, despite its relatively poorer performance with smaller problem sizes, matches the performance of CUDA C with larger problem sizes. Thus, high performance can be achieved with a more user-friendly option, which is easier to implement, and while remaining in the Python ecosystem.

Lastly, the Metal M1 implementation carried out on the MacBook Pro yields impressive results. For smaller problem sizes, its performance closely matches the other implementations (carried out on the Dell workstation), being only slightly slower when larger model sizes are handled. It's important to note that the MacBook, despite its smaller size and weight, still delivers high performance. Moreover, it has a lower price point and is much more portable.

Figure 46 provides a comparative analysis of the Numba serial CPU, Multithread, and GPU implementations. The large contrast in performance between these methods is evident. The CPU

version exhibits the least efficient scaling, while the GPU version demonstrates the most efficient scaling. Thus, when dealing with projects that require a large number of shots to be modeled, using the GPU can dramatically improve the runtime.

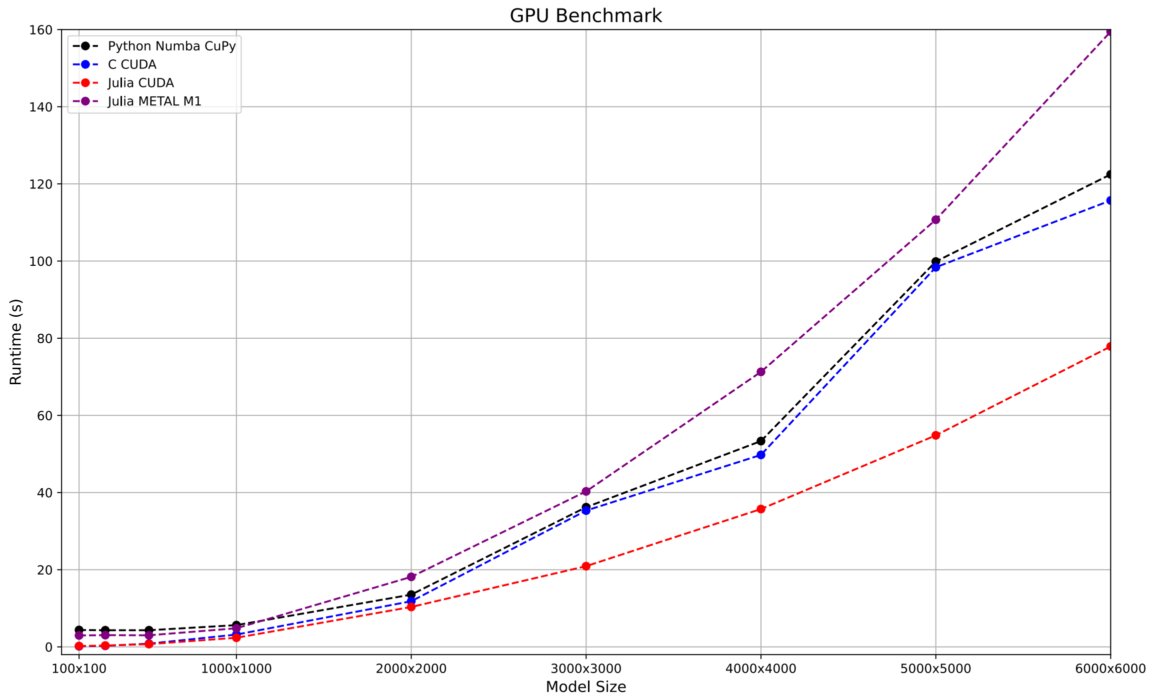


Figure 45: GPU implementations for one shot using the viscoacoustic finite difference method with varying problem size: Numba CuPy (black), CUDA C (blue), Julia CUDA (red) and Julia Metal M1 (purple). The blocksize is 16x16. The runtime is given in seconds.

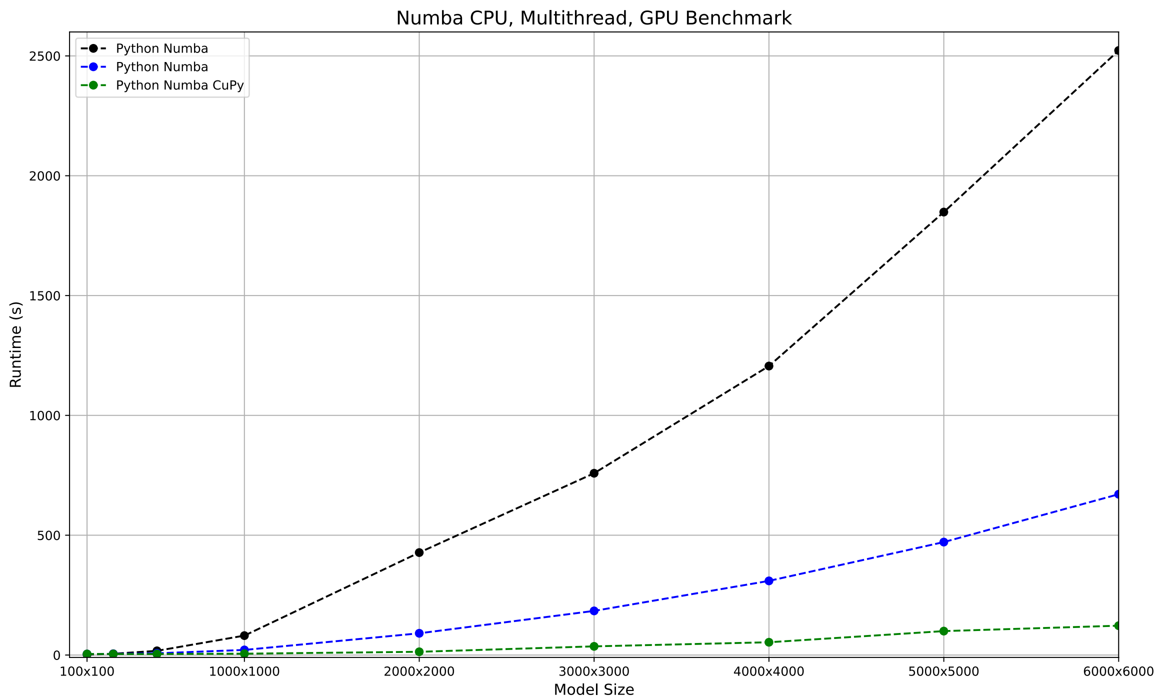


Figure 46: CPU, Multithread and GPU implementations using Numba: Python Numba serial CPU (black), Multithread Numba (blue) and Numba CuPy GPU (green).

Figure 47 shows the relative speedup of different implementations based on their runtime on the 6000x6000 model. The base reference here is considered the Python Numba serial CPU implementation.

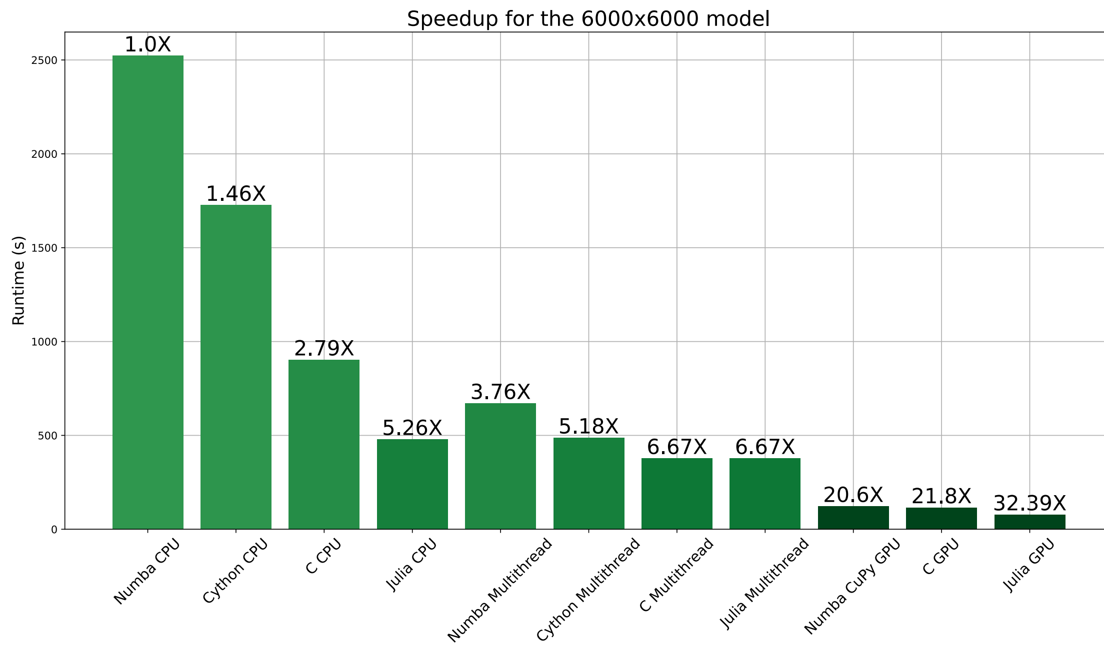


Figure 47: The relative speedup of the different implementations when compared to the Numba serial CPU version.

5 Conclusion and Discussion

This section summarizes the main takeaways from this thesis, evaluates whether the project's objectives were met, and discusses the limitations and future work.

In this thesis, we analyzed and implemented a viscoacoustic version of the wave equation. One of the primary goals was to explain the C-PML boundary conditions in terms of viscoelastic kernels and to express its principles in terms of relatable properties. We have shown that the C-PML boundary conditions can be described by a wave equation that incorporates a time-dependent density and bulk modulus. By using this approach, the same sets of equations can be used both inside and outside the absorbing zone. It is only the model's Quality factor that needs to be changed.

Different viscoelastic models can be implemented, such as the Standard Linear Solid or Maxwell mechanisms. The former was shown to better simulate realistic wave attenuation, while the latter provides frequency independent attenuation, which might be more desirable for an absorbing boundary. A potential area for future improvement could be researching and testing more viscoelastic models, which might provide better performance.

The second major aim of this thesis was to investigate whether the current personal laptops that are offered on the market are good enough for performing seismic modeling. To assess this, we employed a series of optimization strategies, testing implementations on a single core CPU, multiple cores, or the GPU using Python, C, and Julia.

From our benchmarks, we discovered that significant speed improvements can be achieved by applying the principles of heterogeneous computing, particularly by delegating the heavy computational tasks to the GPU. This analysis has shown that there are various strategies one can adopt to optimize their code.

Moreover, we found that Python, when equipped with the right modules and packages, can be a powerful tool for optimization, capable of rivaling C and CUDA in terms of performance. We also explored the capabilities of Julia, a high-level language known for its user-friendly interface, demonstrating its potential to even outperform C.

A key takeaway from this study is the potential of unified memory, a new paradigm that is likely to become the standard for future GPU hardware. We put this into practice by utilizing an Apple MacBook Pro equipped with an M1 chip. The benchmarking results revealed that by using this laptop, which is very portable and popular device among students, one can efficiently handle 2D seismic experiments in a reasonable time frame.

These experiments provide a foundation for further development into comprehensive schemes like Reverse Time Migration or Full Waveform Inversion. Thus, it demonstrates the increasingly accessible nature of seismic modeling, which is no longer confined to high-end, dedicated systems but is possible on personal, portable devices.

One limitation of the benchmarks is that they use a relatively simple quantity, namely, time as a performance metric. But, this does not tell the whole story, and does not offer any information regarding where bottlenecks in your computation are. It is essential to determine whether your computation is compute-bound or memory-bound, as this will dictate your optimization strategy.

In a compute-bound case, the speed of the calculation is limited by the processor's or GPU's speed. In other words, the majority of the program's time is spent performing calculations (such as the mathematical operations of the finite difference method). In this case, the faster the processor or the GPU is, the faster the computation will be.

On the other hand, if a computation is memory-bound, it indicates that its performance is primarily constrained by the speed of memory access. In other words, the program spends a significant time waiting for data to be fetched from memory rather than performing calculations.

An improved benchmarking model that takes advantage of this is the Roofline method. However, it is more challenging to implement. But it is for sure an important future work perspective, as it

allows you to better compare results between different workstations and explicitly tells you if the code performance can be further improved.

In our modeling tests, we are likely memory-bound. If we were handling a 3D model, where the number of computations for each grid point increases, the problem would most likely become compute-bound. In such a case, the speed of the GPU implementations compared to a CPU implementations would also increase significantly.

In summary, this thesis has achieved its proposed objectives and provided meaningful insights into the theory behind absorbing boundaries. It underscores the role of effective programming and optimization strategies, demonstrating that seismic modeling can indeed be executed on personal laptops, thereby making the field more accessible. However, there's always room for further improvement and advancement, keeping the domain exciting and continually evolving.

Bibliography

Igel, H. (2017). *Computational Seismology: A Practical Introduction.* Oxford University Press. ISBN: 9780198717409. URL: https://books.google.no/books?id=zslLDQAAQBAJ.

Igel, H. and M. Stupazzini (Aug. 2015). 'Simulation of Seismic Wave Propagation in Media with Complex Geometries'. In.

Devaney, A. J. and M. L. Oristaglio (1984). 'Geophysical diffraction tomography'. In: *SEG Technical Program Expanded Abstracts 1984*, pp. 330–333. DOI: 10.1190/1.1894018. eprint: https://library.seg.org/doi/pdf/10.1190/1.1894018. URL: https://library.seg.org/doi/abs/10.1190/1.1894018.

Wu, R. S. and M. N. Toksöz (1987). 'Diffraction tomography and multisource holography applied to seismic imaging'. In: *Geophysics* 52, pp. 11–25.

Pratt, R. G. and M. H. Worthington (1988). 'The application of diffraction tomography to cross-hole seismic data'. In: *GEOPHYSICS* 53.10, pp. 1284–1294. DOI: 10.1190/1.1442406. eprint: https://doi.org/10.1190/1.1442406. URL: https://doi.org/10.1190/1.1442406.

Gan, H., R. Ludwig and P. L. Levin (Feb. 1995). 'Nonlinear diffractive inverse scattering for multiple scattering in inhomogeneous acoustic background media'. In: *The Journal of the Acoustical Society of America* 97.2, pp. 764–776. ISSN: 0001-4966. DOI: 10.1121/1.412123. eprint: https://pubs.aip.org/asa/jasa/article-pdf/97/2/764/12210077/764_1_online.pdf. URL: https://doi.org/10.1121/1.412123.

Gelius, L.-J. (1995). 'Generalized acoustic diffraction tomogra phy1'. In: *Geophysical Prospecting* 43.1, pp. 3–29. DOI: https://doi.org/10.1111/j.1365-2478.1995.tb00122.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1365-2478.1995.tb00122.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2478.1995.tb00122.x.

Xie, Y., L. Li and Y. Yang (2018). '3D wavefront tomography: Part I 1 — NIP wavefront tomography'. In: *SEG Technical Program Expanded Abstracts 2018*, pp. 5178–5182. DOI: 10.1190/segam2018-2996300.1. eprint: https://library.seg.org/doi/pdf/10.1190/segam2018-2996300.1. URL: https://library.seg.org/doi/abs/10.1190/segam2018-2996300.1.

Baysal, E., D. Kosloff and J. Sherwood (Nov. 1983). 'Reverse-Time Migration'. In: *Geophysics* 48, pp. 1514–1524. DOI: 10.1190/1.1441434.

McMechan, G. (Apr. 1983). 'Migration by extrapolation of time-dependent boundary values'. In: *Geophysical Prospecting* 31, pp. 413–420. DOI: 10.1111/j.1365-2478.1983.tb01060.x.

Dai, W., X. Wang and G. T. Schuster (2011). 'Least-squares migration of multisource data with a deblurring filter'. In: *GEOPHYSICS* 76.5, R135–R146. DOI: 10.1190/geo2010-0159.1. eprint: https://doi.org/10.1190/geo2010-0159.1. URL: https://doi.org/10.1190/geo2010-0159.1.

Xu, s., Y. Zhang and B. Tang (Mar. 2011). '3D angle gathers from reverse time migration'. In: *Geophysics* 76. DOI: 10.1190/1.3536527.

Zhang, Y., L. Duan and Y. Xie (Sept. 2013). 'A stable and practical implementation of least-squares reverse time migration'. In: vol. 80, pp. 3716–3720. DOI: 10.1190/segam2013-0577.1.

Tarantola, A. (1984). 'Inversion of seismic reflection data in the acoustic approximation'. In: *GEOPHYSICS* 49.8, pp. 1259–1266. DOI: 10.1190/1.1441754. eprint: https://doi.org/10.1190/1.1441754. URL: https://doi.org/10.1190/1.1441754.

Pratt, R. G. and M. H. Worthington (1990). 'INVERSE THEORY APPLIED TO MULTI-SOURCE CROSS-HOLE TOMOGRAPHY.' In: *Geophysical Prospecting* 38.3, pp. 287–310. DOI: https://doi.org/10.1111/j.1365-2478.1990.tb01846.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1365-2478.1990.tb01846.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2478.1990.tb01846.x.

Sambridge, M. S., A. Tarantola and B. L. N. Kennett (1991). 'AN ALTERNATIVE STRATEGY FOR NON-LINEAR INVERSION OF SEISMIC WAVEFORMS1'. In: *Geophysical Prospecting* 39.6, pp. 723–736. DOI: https://doi.org/10.1111/j.1365-2478.1991.tb00341.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1365-2478.1991.tb00341.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-2478.1991.tb00341.x.

Virieux, J. and S. Operto (Nov. 2009). 'An overview of full-waveform inversion in exploration geophysics'. In: *Geophysics* 74, WCC1–WCC26. DOI: 10.1190/1.3238367.

Vigh, D. et al. (Jan. 2014). 'Elastic full-waveform inversion application using multicomponent measurements of seismic data collection'. In: *Geophysics* 79. DOI: 10.1190/geo2013-0055.1.

Li, Y. and L. Demanet (Jan. 2016). 'Full waveform inversion with extrapolated low frequency data'. In: *GEOPHYSICS* 81. DOI: 10.1190/geo2016-0038.1.

Clayton, R. and B. Engquist (1977). 'Absorbing boundary conditions for acoustic and elastic wave equations'. In: *Bulletin of the Seismological Society of America*.

Reynolds, A. C. (Oct. 1978). 'Boundary conditions for the numerical solution of wave propagation problems'. In: *Geophysics* 43.6, pp. 1099–1110. ISSN: 0016-8033. DOI: 10.1190/1.1440881. eprint: https://pubs.geoscienceworld.org/geophysics/article-pdf/43/6/1099/3157858/1099.pdf. URL: https://doi.org/10.1190/1.1440881.

Higdon, R. L. (1991). 'Absorbing boundary conditions for elastic waves'. In: *GEOPHYSICS* 56.2, pp. 231–241. DOI: 10.1190/1.1443035. eprint: https://doi.org/10.1190/1.1443035. URL: https://doi.org/10.1190/1.1443035.

Cerjan, C. et al. (Apr. 1985). 'A Nonreflecting boundary-condition for discrete acoustic and elastic wave-equations'. In: *Geophysics* 50, pp. 705–708. DOI: 10.1190/1.1441945.

Sochacki, J. S. et al. (1987). 'Absorbing boundary conditions and surface waves'. In: *Geophysics* 52, pp. 60–71.

Serón, F., J. Badal and F. Sm (Apr. 1996). 'A numerical laboratory for simulation and visualization of seismic wavefields1'. In: *Geophysical Prospecting* 44, pp. 603–642. DOI: 10.1111/j.1365-2478.1996.tb00168.x.

Berenger, J.-P. (1994). 'A perfectly matched layer for the absorption of electromagnetic waves'. In: *Journal of Computational Physics* 114.2, pp. 185–200. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.1994.1159. URL: https://www.sciencedirect.com/science/article/pii/S0021999184711594.

Francis, C. and C. Tsogka (Jan. 2001). 'Application of the Perfectly Matched Absorbing Layer Model to the Linear Elastodynamic Problem in Anisotropic Heterogeneous Media'. In: *Geophysics* 66, pp. 294–307. DOI: 10.1190/1.1444908.

Komatitsch, D. and J. Tromp (July 2003). 'A Perfectly Matched Layer absorbing boundary condition for the second-order seismic wave equation'. In: *Geophysical Journal International* 154, pp. 146–153. DOI: 10.1046/j.1365-246X.2003.01950.x.

Komatitsch, D. and R. Martin (Sept. 2007). 'An unsplit convolutional Perfectly Matched Layer improved at grazing incidence for the seismic wave equation'. In: *Geophysics* 72. DOI: 10.1190/1.2757586.

Zhang, W. and Y. Shen (July 2010). 'Unsplit complex frequency shifted PML implementation using auxiliary differential equation for seismic wave modeling'. In: *Geophysics* 75, T141–T154. DOI: 10.1190/1.3463431.

Drossaert, F. and A. Giannopoulos (Mar. 2007). 'A nonsplit complex frequency-shifted PML based on recursive integration for FDTD modeling of elastic waves'. In: *Geophysics* 72, T9–T17. DOI: 10.1190/1.2424888.

Carcione, J. and D. Kosloff (Jan. 2013). 'Representation of matched-layer kernels with viscoelastic mechanical models'. In: *International Journal of Numerical Analysis and Modeling* 10.

Oak Ridge National Laboratory (2023). *Frontier: America's Next Frontier for Exploration in Science and Technology*. Accessed: 2023-05-20. URL: https://www.olcf.ornl.gov/frontier/.

Ikelle, L. and L. Amundsen (2005). *Introduction to Petroleum Seismology*. Introduction to Petroleum Seismology v. 12. Society of Exploration Geophysicists. ISBN: 9781560801290. URL: https://books.google.no/books?id=o8IdyHUXpoMC.

Hudson, J. A. (1981). *The Excitation and Propagation of Elastic Waves. Cambridge Monographs on Mechanics and Applied Mathematics. Cambridge, Cambridge University Press 1980.* DOI: https://doi.org/10.1002/zamm.19810610716.

Alterman, Z. and J. Karal F. C. (Feb. 1968). 'Propagation of elastic waves in layered media by finite difference methods'. In: *Bulletin of the Seismological Society of America* 58.1, pp. 367–398. ISSN: 0037-1106. DOI: 10.1785/BSSA0580010367. eprint: https://pubs.geoscienceworld.org/ssa/bssa/article-pdf/58/1/367/5349940/bssa0580010367.pdf. URL: https://doi.org/10.1785/BSSA0580010367.

Boore, D. M. (1970). 'Love waves in nonuniform wave guides: Finite difference calculations'. In: *Journal of Geophysical Research (1896-1977)* 75.8, pp. 1512–1527. DOI: https://doi.org/10.1029/JB075i008p01512. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/JB075i008p01512. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB075i008p01512.

Alford, R. M., K. R. Kelly and D. M. Boore (1974). 'ACCURACY OF FINITE-DIFFERENCE MODELING OF THE ACOUSTIC WAVE EQUATION'. In: *GEOPHYSICS* 39.6, pp. 834–842. DOI: 10.1190/1.1440470. eprint: https://doi.org/10.1190/1.1440470. URL: https://doi.org/10.1190/1.1440470.

Kelly, K. R. et al. (1976). 'SYNTHETIC SEISMOGRAMS: A FINITE -DIFFERENCE APPROACH'. In: *GEOPHYSICS* 41.1, pp. 2–27. DOI: 10.1190/1.1440605. eprint: https://doi.org/10.1190/1.1440605. URL: https://doi.org/10.1190/1.1440605.

Madariaga, R. (June 1976). 'Dynamics of an Expanding Circular Fault'. In: *Bulletin of the Seismological Society of America* 66, pp. 639–666. DOI: 10.1785/BSSA0660030639.

Virieux, J. and R. Madariaga (Apr. 1982). 'Dynamic faulting studied by a finite difference method'. In: *Bulletin of the Seismological Society of America* 72.2, pp. 345–369. ISSN: 0037-1106. DOI: 10.1785/BSSA0720020345. eprint: https://pubs.geoscienceworld.org/ssa/bssa/article-pdf/72/2/345/5330336/bssa0720020345.pdf. URL: https://doi.org/10.1785/BSSA0720020345.

Virieux, J. (1984). 'SH-wave propagation in heterogeneous media: Velocity-stress finite-difference method'. In: *GEOPHYSICS* 49.11, pp. 1933–1942. DOI: 10.1190/1.1441605. eprint: https://doi.org/10.1190/1.1441605. URL: https://doi.org/10.1190/1.1441605.

— (Jan. 1986). 'P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method'. In: *Geophysics* 51, pp. 889–901. DOI: 10.1190/1.1442147.

Levander, A. (Nov. 1988). 'Fourth-order finite-difference P-S'. In: *Geophysics* 53, pp. 1425–1436. DOI: 10.1190/1.1442422.

Graves, R. W. (Aug. 1996). 'Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences'. In: *Bulletin of the Seismological Society of America* 86.4, pp. 1091–1106. ISSN: 0037-1106. DOI: 10.1785/BSSA0860041091. eprint: https://pubs.geoscienceworld.org/ssa/bssa/article-pdf/86/4/1091/5343090/bssa0860041091.pdf. URL: https://doi.org/10.1785/BSSA0860041091.

Holberg, O. (1987). 'Computational aspects of the choice of operator and sampling interval for numerical differentiation in large-scale simulation of wave phenomena'. In: *Geophysical P* 35, pp. 629–655.

Carcione, J. and G. Casula (Jan. 1992). 'Generalized mechanical model analogies of linear viscoelastic behaviour,l., 235-256.' In: *Bollettino di Geofisica Teorica ed Applicata* 34, pp. 235–256.

Landro, M. and L. Amundsen (Apr. 2018). *Introduction to Exploration Geophysics with Recent Advances*. ISBN: 978-82-303-3763-9.

Cheng, J., M. Grossman and T. McKercher (2014). *Professional CUDA C Programming*. EBL-Schweitzer. Wiley. ISBN: 9781118739327. URL: https://books.google.no/books?id=q3DvBQAAQBAJ.

Python Documentation (2023). *Our Documentation — Python.org*. Accessed: 2023-06-21. Python Software Foundation. URL: %5Curl%7Bhttps://www.python.org/doc/%7D.

NumPy Developers (2023). *NumPy documentation — NumPy v1.25 Manual*. Accessed: 2023-06-21. URL: https://numpy.org/doc/stable/.

Numba Documentation (2023). *Numba Documentation*. URL: https://numba.readthedocs.io/en/stable/index.html.

Cython Documentation (2023). *Cython: C-Extensions for Python*. Accessed: 2023-06-22. URL: https://cython.org/#documentation.

SWIG Documentation (2023). *SWIG Documentation*. Accessed: 2023-06-22. URL: https://www.swig.org/exec.html.

Julia Documentation (2023). *Julia Documentation*. Accessed on June 22, 2023. URL: https://docs.julialang.org/en/v1/.

OpenMP Documentation (2023). *OpenMP Documentation*. Accessed on June 23, 2023. URL: https://www.openmp.org/specifications/.

NVIDIA CUDA Documentation (2023). *CUDA Documentation*. Accessed: 2023-06-24. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

CuPy Documentation (2023). *CuPy Documentation*. Accessed: 2023-06-24. URL: https://cupy.dev.

Apple Developer Documentation (2023). *Apple Developer Documentation*. Accessed: 2023-06-23. URL: https://developer.apple.com/documentation/apple-silicon.

Julia Metal (2023). *Julia Metal Documentation*. URL: https://juliagpu.org/post/2022-06-24-metal/.

Borcherdt, R. D. (Sept. 1982). 'Reflection—refraction of general P-and type-I S-waves in elastic and anelastic solids'. In: *Geophysical Journal International* 70.3, pp. 621–638. ISSN: 0956-540X.

DOI: 10.1111/j.1365-246X.1982.tb05976.x. eprint: https://academic.oup.com/gji/article-pdf/70/3/621/1727743/70-3-621.pdf. URL: https://doi.org/10.1111/j.1365-246X.1982.tb05976.x.

Ursin, B. (1983). 'Review of elastic and electromagnetic wave propagation in horizontally layered media'. In: *GEOPHYSICS* 48.8, pp. 1063–1081. DOI: 10.1190/1.1441529. eprint: https://doi.org/10.1190/1.1441529. URL: https://doi.org/10.1190/1.1441529.

Appendix: GitGub Repository

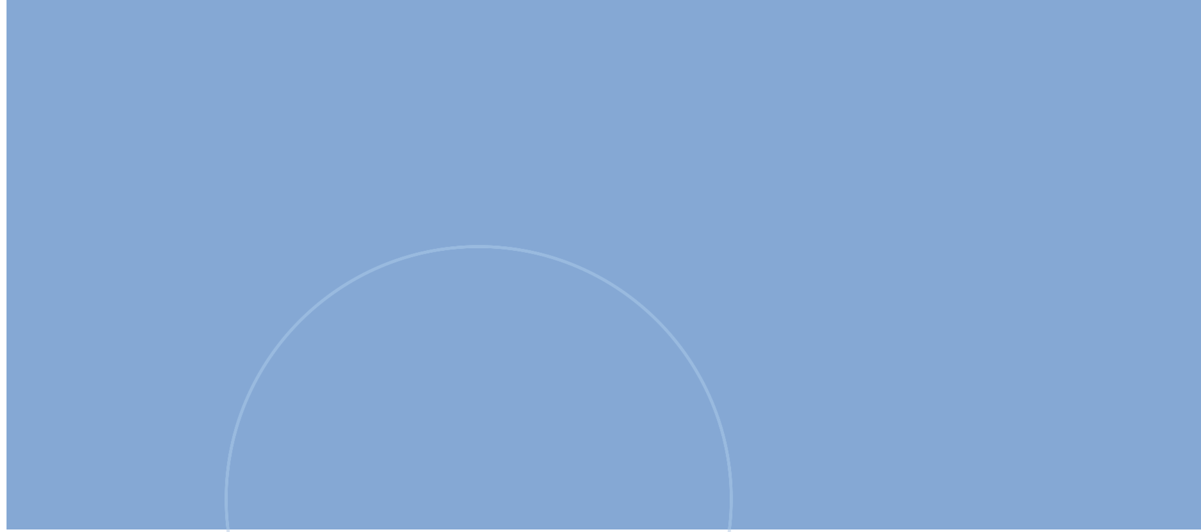
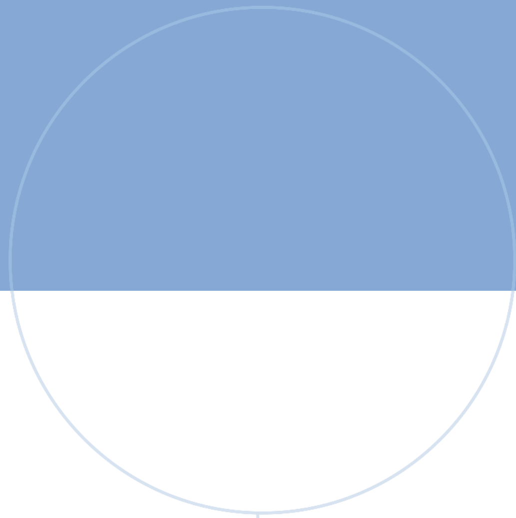
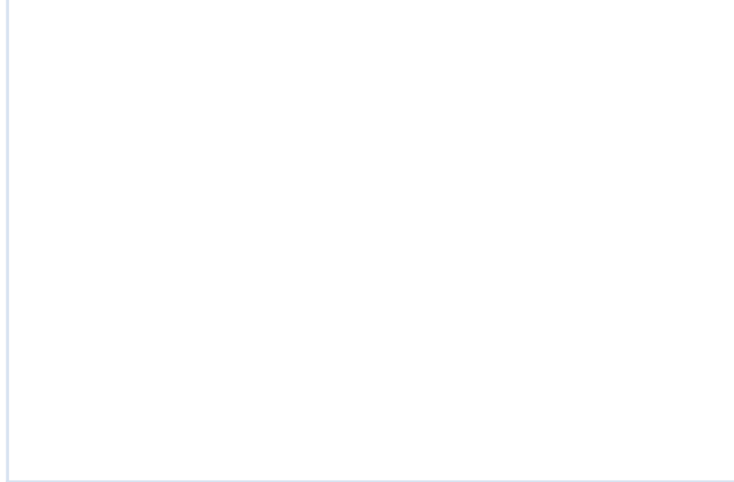
Since all the code implementations for the Saxpy function and 2D Viscoacoustic Finite Difference Modeling are extensive they have been uploaded to a GitHub repository.

This repository contains all the source code and all the benchmarks results used in this thesis. This can be accessed at the following link: https://github.com/StefanCatalinCraciun/Master-Thesis.git

Within the Repository there are two main folders: Saxpy and Viscoacoustic Modeling.

In the Saxpy folder we can find a jupyter notebook where all the benchmark times can be accessed and viewed using plots. The src file contains all the source code.

The Viscoacoustic Modeling folder also contains a jupyter notebook where all the benchmark times can be accessed and viewed using plots. For a detailed explanation of the code, the scripts within the Python_CPU_For_Loop folder can be viewed. Also, in the Visualization folder here you can view several plots and a *.mp4* video file that shows an animation of the propagating wavefield

Norwegian University of
Science and Technology